Konzeption und Umsetzung einer Zeiterfassungsapplikation mit integrierter Rechnungsgenerierung

für zeitbasiert arbeitende Solo-Selbstständige

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Yannick Schuchmann

2096818

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr. Andreas Plaß

Zweitprüfer: Prof. Dr. Edmund Weitz

Hamburg, 10.08.2016

Inhaltsverzeichnis

Abstract

Although the number of freelancers in Germany rises constantly there does not exist an accounting software, which solves the special requirements of hourly paid freelancers. A solution for this problem is combining features of time tracking and invoicing to a single application.
This paper documents the progress of creating this application including an analysis of requirements, conception and implementation. Because of different tax laws and currencies internationalisation and also the synchronisation of client timestamps and server timestamps became a challenge.

Zusammenfassung

Trotz ständig steigender Zahl von Solo-Selbstständigen in Deutschland existiert keine Buchhaltungsanwendung, die speziell auf die Anforderungen von zeitbasiert arbeitenden Solo-Selbstständigen zugeschnitten ist. Um dieses Problem zu lösen, wurde eine Anwendung entwickelt, die die Erfassung von Arbeitszeiten und die darauf basierten Rechnungserstellung zusammenführt.
Diese Arbeit dokumentiert die Anforderungsanalyse, Konzeption und Realisierung dieser Anwendung. Dabei stellten sich die Internationalisierung, aufgrund von gesetzlichen Steuerformalien und länderspezifischen Währungen, sowie der genaue synchronisierte Umgang mit Client- und Serverzeit als besondere Schwierigkeiten heraus.

1 Einleitung

Existenzgründung hat sich vor Allem für Einzelpersonen in den letzten Jahren immer mehr zum Trend entwickelt. So ist zwischen 2000 und 2011 die Zahl der Solo-Selbstständigen in Deutschland um 40% gestiegen, wobei gleichzeitig die Zahl der Selbstständigen mit eigenen Beschäftigen stagnierte. Motive sind, neben eines ansonsten fehlenden Arbeitsplatzes, hauptsächlich ein größerer Verdienst, als auch die Möglichkeit der eigene Chef und damit unabhängig zu sein.
Dabei kommen häufig neue Themen auf den Unternehmer zu, mit denen dieser sich zu diesem Zeitpunkt meist noch nicht auseinandergesetzt hat. Eines dieser Themen ist die Buchhaltung, die unter anderem die Rechnungserstellung umfasst.
Um diese Aufgaben auch für noch unerfahrene Unternehmer einfach zu gestalten, gibt es heute ein große Auswahl an Software. Diese Tools beziehen sich häufig auf Warenbestände und implementieren dafür komplexe Warenwirtschaftssysteme. Mit diesen lassen sich Rechnungen auf Basis von hinterlegten Artikeln, deren Verfügbarkeiten und fortlaufenden Kunden-, sowie Rechnungsnummern teilweise automatisch erstellen.
Viele Solo-Selbstständige können aus diesen Tools allerdings keinen allzu großen Nutzen ziehen, da sie häufig keine echte Ware, sondern Dienstleistungen in Form von aufgewändeter Arbeitszeit verkaufen. Sie können sich lediglich von Zeiterfassungssoftware die händische Buchführung über erbrachte Aufwände abnehmen lassen, bevor sie diese schließlich trotzdem manuell in Form einer Rechnung verfassen.
Der Autor entwickelte eine Software zur Vereinigung von Zeiterfassung und daraus resultierender Rechnungserstellung.
In dieser Arbeit wird der Leser chronologisch durch den Prozess der Umsetzung dieser Software geführt. Nach einer anfänglichen Anforderungsanalyse werden die Funktionalitäten der daraus entstandenen Anwendung im Detail beschrieben. Auf das Konzept der Benutzeroberfläche folgen die Kapitel zur Architektur der Software und schließlich auch der praktischen Realisierung. Abschließend wird ein Ausblick auf für die Zukunft geplante Aspekte der Anwendung gegeben.

2 Anforderungsanalyse

Ein essentieller Schritt zu Beginn der Konzeption ist die Analyse des Benutzers, sowie dessen Absichten und die daraus resultierenden Probleme und Fragestellungen. Hieraus lassen sich entsprechende Aufgaben ableiten, welche als Anforderungen an die Anwendung gestellt werden können.

2.1 Szenario

Im Folgenden ist ein fiktives Szenario aus dem Arbeitsleben eines zeitbasiert arbeitenden Solo-Selbstständigen dargestellt, dem die im nächsten Kapitel beschriebene Anwendung nicht vorliegt.

Eddie Example ist als freier Softwareentwickler selbstständig und wird hauptsächlich auf Projektbasis von Agenturen gebucht, wobei er generell auf Stundenbasis bezahlt wird. Daher führt er stetig sehr genau Buch über seine Arbeitszeiten und notiert jeweils zum Anfang und Ende jeder Arbeitssitzung die Uhrzeit zusammen mit dem Projekt. Dies macht er auf dem Papier, damit er immer und überall, egal ob im Büro oder im Fernzug, seine Zeiten sichern kann. Oft ergänzt Eddie seine Notizen auch um die Tätigkeit, die er im jeweiligen Zeitraum erbracht hat.
Nach Feierabend setzt er sich zu Hause an seinen Schreibtisch und berechnet für jedes Projekt, an dem er an diesem Tag gearbeitet hat, die Summe aller Arbeitsstunden und notiert sich diese.
Zu Beginn jedes neuen Monats stellt Eddie dann eine Rechnung an alle Agenturen für die er im vergangenen Monat tätig war. In diesen sind jeweils die summierten Arbeitsstunden als Rechnungspositionen aufgelistet.
Neben den regelmäßigen Tätigkeiten für Agenturen hat Eddie noch seinen eigenen Kundenstamm. Für diesen stellt er allerdings nicht monatsweise seine notierten Arbeitsstunden, sondern zum Abschluss des Projekts einen vorher ausgehandelten Fixpreis in Rechnung.
Eddie ist Einzelunternehmer in Deutschland. Daher muss er 19% Umsatzsteuer von seinem Umsatz abführen und landesspezifische Formalien zur Erstellung seiner Rechnungen einhalten.
Das manuelle Berechnen seiner täglichen Arbeitszeit, sowie diese monatlich in Rechnungsform zu bringen, kostet Eddie sehr viel Zeit, die er lieber in andere Tätigkeiten investieren würde.

2.2 Anforderungen

Aus dem definierten Szenario lassen sich nun spezifische Anforderungen an eine digitale Anwendung ableiten:

2.2.1 Zeiterfassung

Eddie notiert sich die Uhrzeit zum Anfang und Ende einer Tätigkeit hauptsächlich, um die Dauer daraus berechnen zu können. Daher muss es dem Nutzer der Anwendung möglich sein, Aufzeichnungen seiner Tätigkeiten zu starten, zu beenden und die Dauer direkt ablesen zu können, bestenfalls schon während der eigentlichen Aufzeichnung.
Diese Aufzeichnungen werden durch Projekte gruppiert.

2.2.2 Projektverwaltung

Damit der Nutzer eine Aufzeichnung einem Projekt zuweisen kann, muss die Anwendung eine Projektverwaltung beinhalten. Diese soll es dem Nutzer ermöglichen, Projekte anzulegen, zu bearbeiten und zu löschen.
Zusätzlich kann ein Projekt einem Kunden zugeteilt werden, sodass, basierend auf den Aufzeichnungen eines Projektes, Rechnungen erstellt werden können.

2.2.3 Kundenverwaltung

Analog zur Projektverwaltung müssen Kunden erstellt, bearbeitet und gelöscht werden können. Darüber hinaus muss der Nutzer jedem Kunden eine Währung und einen Stundensatz zuweisen können.

2.2.4 Rechnungserstellung

Durch die Angabe von Stundensatz und Währung können offene Arbeitszeiten, die sich aus allen noch nicht in Rechnung gestellten Aufzeichnungen ergeben, automatisch in Rechnungsform gebracht werden. Daher muss es dem Nutzer möglich sein, durch Knopfdruck alle Rechnungen für den vorangegangenen Monat zu erstellen.
Zusätzlich muss der Nutzer die Möglichkeit haben, jenen automatisch, sowie neu erstellten Rechnungen individuelle Rechnungspositionen hinzufügen zu können.
Ebenso müssen dort alle rechnungsrelevanten Unternehmensangaben zu sehen sein.

2.2.5 Unternehmensverwaltung

Für das Erzeugen der Rechnungen ist es wichtig, dass der Nutzer diverse Daten über sein Unternehmen hinterlegen kann. Darunter fallen neben dem Unternehmensnamen auch die Adresse, sowie Bank-, Steuer- und Kontaktdaten.

2.2.6 Internationalisierung

Damit Nutzer aus unterschiedlichen Ländern die Anwendung sinnvoll nutzen können, müssen verschiedene Punkte bei der Umsetzung beachtet werden. Darunter fallen die Angabe eines Steuersatzes, die Auswahl einer Währung pro Kunde, die Vorgabe eines internationalen Adress- und Zahlungsformats und eine länderspezifische Rechnungsvorlage.

2.2.7 Offline-Nutzbarkeit

Eddie notiert seine Arbeitszeiten unterwegs analog auf dem Papier. Damit ihm die Anwendung keine Nachteile einbringt und seinen Alltag nicht einschränkt, müssen die wichtigsten Funktionen auch offline nutzbar sein. Darunter fallen hauptsächlich die Zeiterfassung und Projektverwaltung.

2.3 Zielgruppe

Der gesamte Funktionsumfang zielt hauptsächlich auf zeitbasiert arbeitende Solo-Selbstständige ab. Im Normalfall möchten diese erfasste Arbeitszeiten, multipliziert mit dem jeweiligen Stundensatz, als Rechnungen für den vergangenen Monat generieren.
Das Anlegen von Kunden und Rechnungen ist nicht verpflichtend, wodurch sich eine sekundäre Zielgruppe bildet. Diese bezieht sich auf Nutzer, die lediglich die Zeiten beliebiger Tätigkeiten erfassen möchten.

3 Beschreibung der Anwendung

Nachdem die Anforderungen an die Anwendung gestellt wurden, wird in diesem Kapitel auf den generellen Aufbau dieser eingegangen und einzelne Bereiche, genau wie deren Zusammenhänge werden näher betrachtet. Danach wird kurz auf die Aufgaben der Authentifizierung eingegangen.

Grundsätzlich teilt die Anwendung sich in zwei große Bereiche auf: die Zeiterfassung und der Verwaltungsbereich, welcher sich aus Projekt-, Kunden-, Unternehmens- und Rechnungsverwaltung zusammensetzt.

3.1 Zeiterfassung

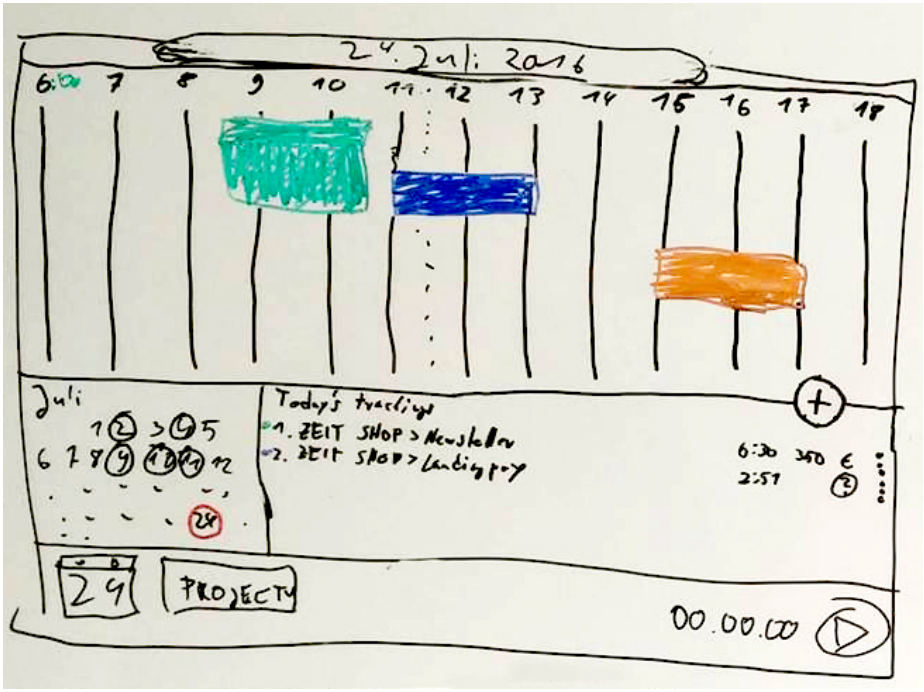

Abbildung 3.1: Skizzierung der Zeiterfassung am Whiteboard

Die Zeiterfassung wird als separater Bereich angesehen, da sie das Kernelement der Anwendung ist. Der Nutzer kann hier alle Aufzeichnungen verwalten, welche die Basisinformationen für fast alle weiteren Funktionalitäten bereitstellen.

Um die Auflistung aller bereits angelegten Aufzeichnungen übersichtlich zu halten, bezieht sich diese ausschließlich auf ein über einen Kalender auswählbares Datum, das bei Seitenaufruf zunächst dem aktuellen Datum entspricht.

Außerdem werden diese Aufzeichnungen zusätzlich nach Projekten gruppiert. Dabei wird auch die Summe des zeitlichen Aufwands pro Projekt dem Nutzer zugänglich gemacht.
Diese Auflistung wird in zwei unterschiedlichen Arten dargestellt. Zum Einen beinhaltet sie in Form einer klassischen Tabelle die Spalten Index, Projektname und Dauer. Dazu lassen sich die Listenzeilen aufklappen, um darunter alle unter dem Projekt gruppierten Aufzeichnungen anzuzeigen, wo sie auch bearbeitet beziehungsweise gelöscht werden können. Zum Anderen findet sich oberhalb der Tabelle auch eine visuelle Darstellung in Form eines horizontalen Zeitstrahls, in dem die Aufzeichnungen als Block über ihren entsprechenden zeitlichen Rahmen aufgespannt werden.

Damit in der Auflistung überhaupt Aufzeichnungen erscheinen, müssen diese erst erstellt werden. Dies ist über die Zeiterfassungsleiste möglich. Sie beinhaltet einen Button zum Starten einer Aufzeichnung und zum entsprechenden Stoppen dieser und verhält sich analog zu einer Stoppuhr. Einer auf diese Weise erstellten Aufzeichnung wird das aktuelle Datum und bei Klick auf den Start- beziehungsweise Stop-Button die aktuelle Uhrzeit als Beginn beziehungsweise Ende zugewiesen. Wichtig hierbei ist, dass es keine parallel laufende Aufzeichnungen geben kann. Ist eine Aufzeichnung gerade aktiv, wurde gestartet und noch nicht wieder gestoppt, so wird die aktuelle Dauer in der Zeiterfassungsleiste sekundengenau angezeigt.
Schließlich befindet sich in der Zeiterfassungsleiste noch ein weiterer Button, durch den sich ein Dialogfenster zur Auswahl eines Projektes öffnen lässt. Für diese Auswahl wird eine reduzierte Variante des Projektverwaltungsbereiches dargestellt, wodurch sich im Dialogfenster auch Projekte anlegen, bearbeiten, löschen und verschieben lassen. Erst wenn ein Projekt ausgewählt wurde, kann eine Aufzeichnung gestartet werden. Eine laufende Aufzeichnung wird durch das Navigieren in einen anderen Bereich nicht beeinflusst.

3.2 Verwaltungsbereich

3.2.1 Projektverwaltung

Die reduzierte Projektverwaltung basiert auf der eigentlichen Projektverwaltung aus dem Verwaltungsbereich.
Projekte haben einen Namen, beziehen sich optional auf einen Kunden und können beliebig viele Unterprojekte beinhalten. Da Unterprojekte die gleichen Eigenschaften besitzen und daher ebenfalls beliebig viele Unterprojekte beinhalten können, ist es sinnvoll, Projekte in Baumstruktur zu behandeln. Für eine übersichtliche Auflistung werden daher initial lediglich Projekte dargestellt, die kein Elternprojekt besitzen und zusammen die sogenannten *Top-Level-Projekte* sind. Ein Projekt lässt sich durch Anklicken aufklappen, wodurch dessen Unterprojekte erscheinen, die jenen Aufklappmechanismus ebenfalls bieten. Diese Art der Auflistung lässt sich mit der von Kategorien und deren Unterkategorien in einem Onlineshop vergleichen.
Zu jedem dargestellten Projekt gibt es ein Untermenü, durch das sich ein Projekt mit allen Unterprojekten löschen lässt oder der Nutzer das Projektformular zur Bearbeitung öffnen kann. Das Formular, welches sich auch über einen *Floating Action Button* (FAB)[1] zum Erstellen eine Projektes öffnen lässt, bietet die zwei Felder *Name* und *Elternelement*, wobei letzteres optional ist.
Zurück bei der Auflistung gibt es für jedes Projekt auch noch die Summe der aufgezeichneten Arbeitszeit, wobei diese jeweils auch die Zeiten der Unterprojekte einschließt.
Zuletzt findet sich für jedes Projekt noch ein Auswahlfeld, um dieses optional einem vorher angelegten Kunden zuweisen zu können.

3.2.2 Kundenverwaltung

Die Aufgaben der Kundenverwaltung sind lediglich das Auflisten der vorhandenen und das Erstellen neuer Kunden.
Die Auflistung ist alphabetisch aufsteigend sortiert und zeigt Index, Namen und Erstellungsdatum des Kunden.
Dazu ist ein Formular zum Erstellen eines neuen Kunden vorhanden. Dieses beinhaltet Felder für Namen, Kundennummer, Währung, Stundensatz und Adresse, welche anwendungsweit immer die Felder Adresszeile 1, Adresszeile 2, Stadt, Bundesland, PLZ und Land umfasst, wodurch sich auch internationale Adressen eintragen lassen.
Wird ein Kunde in der Auflistung selektiert, füllt sich das Formular mit dessen Datensatz und geht in den Bearbeitungsmodus über. In diesem wird der Kunde bei Absenden des Formulars gespeichert und nicht neu erstellt. Außerdem wird nach dem Absenden, sowie bei Deselektieren des Kunden das Formular zurückgesetzt und in den Erstellungsmodus überführt.

[1] Der Floating Action Button ist ein primäres Interaktionselement im Material Design. Näheres unter (Google Design).

3.2.3 Unternehmensverwaltung

Im Bereich der Unternehmensverwaltung können Daten zum eigenen Unternehmen hinterlegt und gepflegt werden. Er besteht aus einem einzigen Formular, welches sich in 3 Bereiche aufteilen lässt.
Zunächst lassen sich Unternehmensname, Web- und E-Mailadresse, Telefonnummer und Steuernummer, mit dazugehöriger prozentualer Angabe der abzutretenden Umsatzsteuer im Kontaktbereich angeben.
Darauf folgt der Adressbereich, der die Felder zur Adresse des Unternehmenssitzes beinhaltet.
Der Zahlungsbereich umfasst die Daten der Zahlungsadresse, die auf erstellten Rechnungen aufgeführt werden soll. Hier unterscheiden sich die Felder je nach gewählter Zahlungsmethode. Vorausgewählt ist IBAN, wodurch initial die Felder Kontoinhaber, IBAN und BIC (Swift-Code) vorhanden sind. Trotz des Namens sind IBAN Zahlungsadressen nicht weltweit, sondern nur innerhalb der europäischen Union zuverlässig nutzbar. Daher gibt es neben IBAN noch die Optionen PayPal, die das Feld PayPal-E-Mailadresse beinhaltet, sowie Kreditkarte, die die üblichen Felder Kartennummer, Prüfziffer und Ende der Gültigkeit umfassen.
Die hinterlegten Daten sind notwendig zur Erstellung von Rechnungen.

3.2.4 Rechnungsverwaltung

Sobald Kunden und Unternehmensdaten hinterlegt sind, lassen sich Rechnungen erstellen. Hierfür kann man durch einen Klick auf den FAB innerhalb der Rechnungsverwaltung das Rechnungsformular öffnen.
In diesem lassen sich eine Rechnungsnummer, ein Rechnungsdatum, eine Zahlungsfrist, welche initial auf 14 Tage gesetzt ist und der dazugehörige Kunde festlegen. Darüber hinaus lassen sich der Rechnung beliebig viele Rechnungspositionen, bestehend aus Name, Menge und Einzelpreis hinzufügen. Der Preis einer einzelnen Rechnungsposition, sowie die Umsatzsteuer, der Brutto- und Nettogesamtpreis der Rechnung werden automatisch kalkuliert.
Sind zusätzlich noch offene Zeitaufzeichnungen angelegt, können daraus auch Rechnungen automatisch generiert werden. Dies ist handhabbar über das Generierungsformular, welches aus den Datumsfeldern *Von* und *Bis* besteht, die den Zeitraum festlegen, in dem die Aufzeichnungen stattgefunden haben. Nach Absenden des Formulares werden automatisch passende Rechnungen pro Kunde angelegt, deren Rechnungspositionen sich wie folgt zusammensetzen. Betrachtet werden alle Projekte des Kunden mit offenen Aufzeichnungen. Pro Projekt wird eine Rechnungsposition angelegt, wobei der Projektname als Name der Position und die summierte Arbeitszeit die Menge sind. Der Einzelpreis ergibt sich aus dem für den Kunden hinterlegten Stundensatz. Alle neu generierten Rechnungen werden als PDF gebündelt in einem ZIP Archiv dem Nutzer als automatischen Download zur Verfügung gestellt. Den genutzten offenen Aufzeichnungen werden nun als “in Rechnung gestellt” markiert und

gelten dementsprechend nicht mehr als offene Rechnungen.
Schließlich bietet die Rechnungsverwaltung noch eine Auflistung aller, egal ob manuell oder automatisch, angelegten Rechnungen. Diese ist nach Erstellungsdatum absteigend geordnet und bietet die Möglichkeiten zum Löschen, zum Öffnen als PDF und zum Bearbeiten. Soll eine Rechnung bearbeitet werden, öffnet sich das Rechnungsformular mit dem entsprechenden Datensatz im Bearbeitungsmodus.

3.3 Authentifizierung

Der Nutzer darf ausschließlich seine eigenen Datensätze verwalten und einsehen. Daher ist es notwendig, dass die Anwendung jedem Nutzer einen persönlichen Zugang bereitstellt. Da der Fokus dieser Arbeit auf den Bereichen liegt, die speziell für diese Anwendung sind und hier ein gängiges Modell zur Authentifizierung Verwendung findet, wird nur grob auf die einzelnen Aufgaben derer eingegangen.
Um Zugang zur Anwendung zu bekommen, muss sich der Nutzer mit seiner E-Mailadresse und einem Passwort registrieren, wodurch er eine E-Mail mit einem Bestätigungslink erhält. Erst nach Öffnen des Bestätigungslinks wird der Nutzer beziehungsweise dessen Zugang freigeschaltet.
Dazu hat er die Möglichkeit bei Vergessen des Passworts eine E-Mail anzufordern, die einen für ihn einzigartigen Link zur Passwortneuvergabe beinhaltet.
Ist der Nutzer bestätigt, kann dieser sich mit den vorher gewählten Zugangsdaten, E-Mail und Passwort, über den Login anmelden, wo er außerdem die Möglichkeit hat, sein Passwort zu ändern oder seinen Nutzeraccount zu löschen. Einem angemeldeten Nutzer stehen nun alle Funktionen der Anwendung zur Verfügung. Alle erstellten Datensätze sind entweder direkt oder transitiv mit dem Nutzer verknüpft. Gleichzeitig ist es diesem auch nur möglich Datensätze mit einer Verknüpfung zum eigenen Nutzer zu öffnen.

4 Benutzeroberfläche

Im vorherigen Kapitel ist die Anwendung mit all ihren Aufgaben und Komponenten beschrieben worden. Darauf basierend geht es in diesem Kapitel um die Überführung dieser in eine grafische Oberfläche mit Fokus auf eine möglichst gute Nutzererfahrung beim Bedienen der Anwendung. Dabei wird zuerst auf die vorher gesetzten Richtlinien und später auf die Konzepte der einzelnen Oberflächen mithilfe von Wireframes eingegangen.

4.1 Allgemeine Richtlinien

4.1.1 Designsprache Material Design

Um eine gute Bedienung der Anwendung zu gewährleisten, muss diese für den Nutzer über alle Oberflächen konsistent und möglichst intuitiv sein. Daher wird sich bei der Konzeption und Gestaltung der Designsprache Material Design von Google bedient.

Material Design basiert auf dem Gestaltungsstil Flat Design, der im Gegensatz zu Skeuomorphismus sich durch seine minimalistischen Eigenschaften, wie beispielsweise der Verzicht auf Verläufe und Texturen, auszeichnet. Material Design erweitert Flat Design um Animationen und Schatten, wodurch Elemente eine gewisse Tiefe bekommen und dadurch dem Nutzer interakionsfähig präsentiert werden können.

4.1.2 Farbpalette

Desweiteren sollen auch Farben konsistent über die gesamte Anwendung genutzt werden, weswegen eine Farbpalette passend für oben genannte Designsprache entwickelt worden ist.

Diese besteht aus folgenden Farben:

Tabelle 4.1: Farbpalette

Primärfarbe 1	
Primärfarbe 2	
Primärfarbe 3	
Akzentfarbe 1	
Akzentfarbe 2	
Akzentfarbe 3	
Textfarbe	
Textfarbe alternativ	
Hintergrundfarbe 1	
Hintergrundfarbe 2	
Hintergrundfarbe 3	
Rahmenfarbe	
Inaktivfarbe	

4.2 Wireframes

Im Folgenden werden nun Wireframes für jeden Bereich aufgeführt. Dazu wird auf das gewählte Layout und die Platzierung der Komponenten kurz eingegangen.

Grundlegend ist das Layout in eine Seitenleiste und einen Inhaltsbereich aufgeteilt. Die Seitenleiste beinhaltet zwei Buttons über die sich entweder die Zeiterfassung oder der Verwaltungsbereich betreten lassen. Der Inhaltsbereich gibt erwartungsgemäß den entsprechenden Inhalt des aktuellen Bereichs aus.

4.2.1 Zeiterfassungsbereich

Zeiterfassung

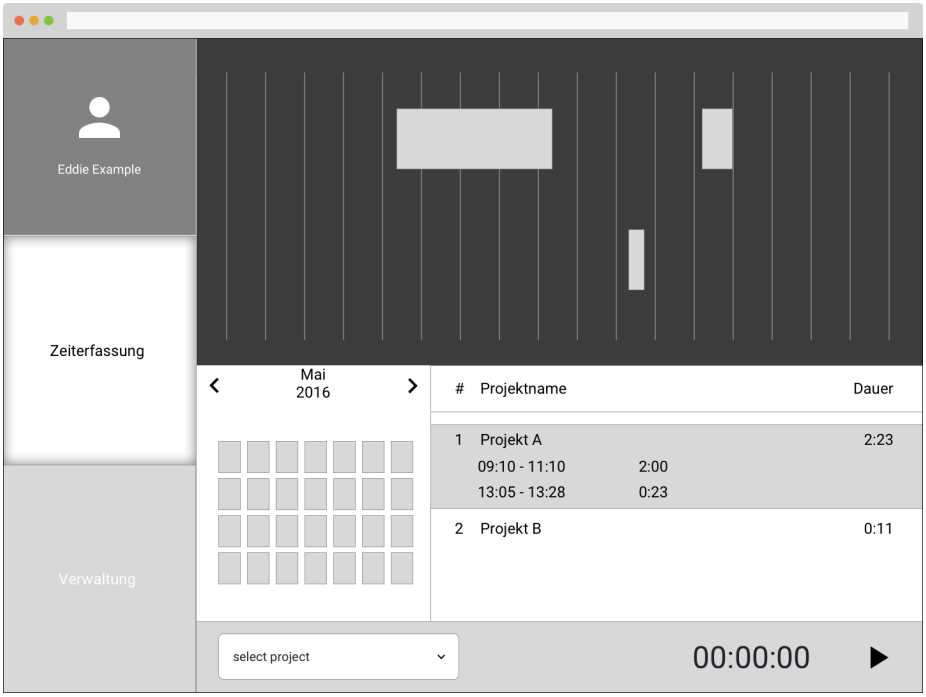

Abbildung 4.1: Wireframe Zeiterfassung

Vertikal in drei Reihen aufgeteilt umfasst die Zeiterfassung zuerst die grafische Darstellung der Aufzeichnungen, welche ein auf halbe Stunden basiertes Gitter im Hintergrund beinhaltet, danach einen Bereich, in dem sich horizontal benachbart der Kalender und die in Projekte gruppierte Auflistung aller Aufzeichnung befinden. Zuletzt erscheint die Zeiterfassungsleiste zum Starten und Stoppen einer Aufzeichnung.

Projektauswahl

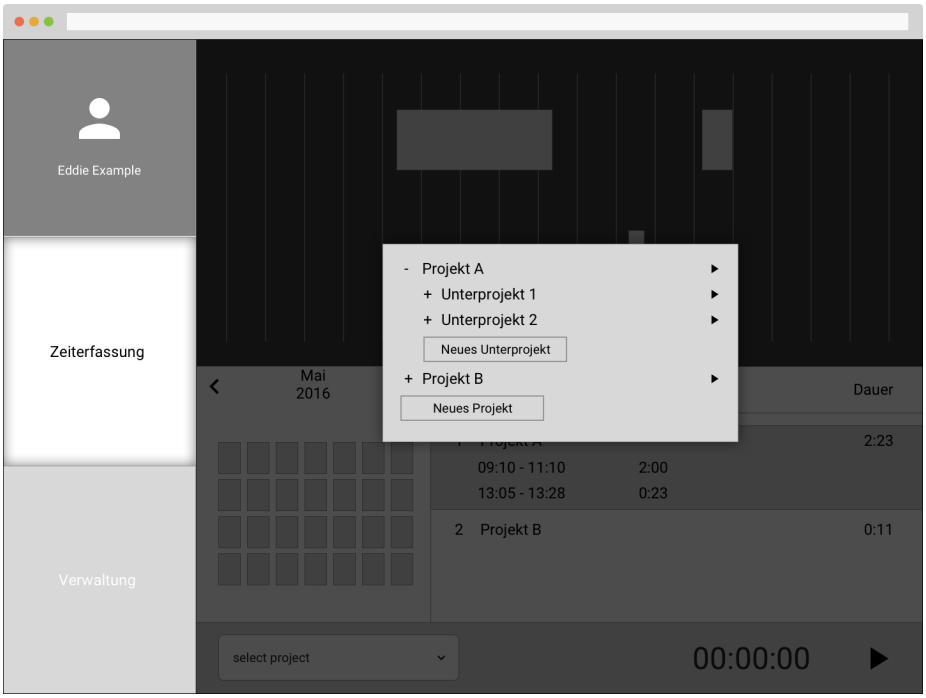

Abbildung 4.2: Wireframe Projektauswahl

Die Auswahl eines Projektes erfolgt über ein Dialogfenster, welches sich durch einen Button in der Zeiterfassungsleiste öffnen lässt. Darin findet der Nutzer alle Projekte in Baumstruktur vor, für welche er durch einen zugehörigen Play-Button eine Aufzeichnung direkt starten kann. Ebenfalls befindet sich in jeder Tiefe des Baumes ein Button zum Erstellen eines neuen Projekts oder Unterprojekts.

4.2.2 Verwaltungsbereich

Da der Verwaltungsbereich mehrere Unterseiten umfasst, ist dem dortigen Inhaltsbereich eine horizontale Navigationsleiste oben angeheftet worden. Diese beinhaltet einen Menüpunkt für die jeweilige Unterseite.

Kundenverwaltung

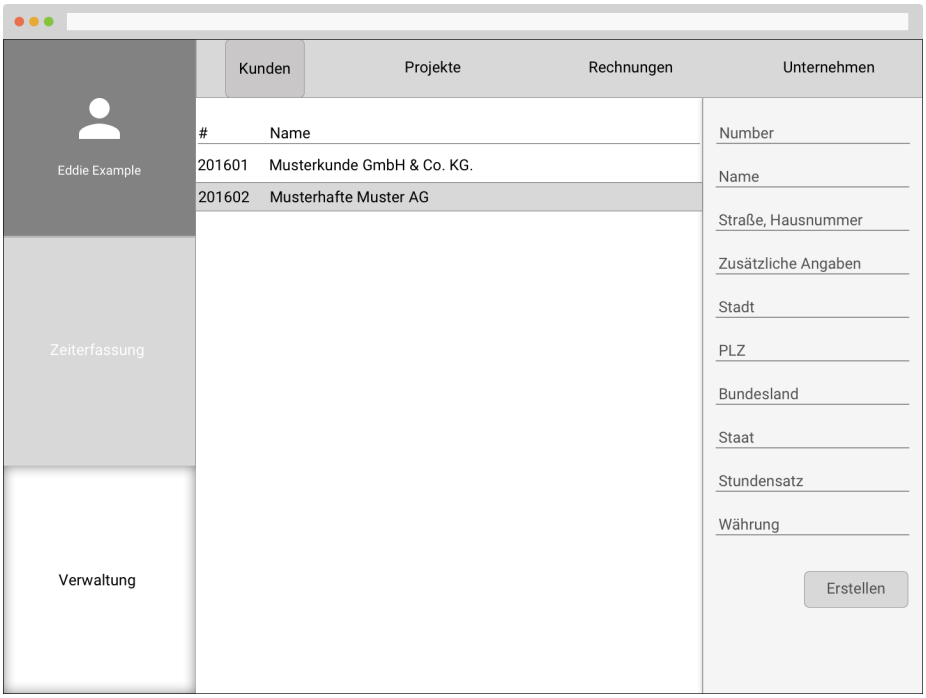

Abbildung 4.3: Wireframe Kundenverwaltung

Die Kundenverwaltung ist zweigeteilt. Hauptsächlich umfasst sie eine tabellarische Auflistung aller Kunden mit den Spalten Kundennummer und Name. Daneben ist ein Formular zur Erzeugung neuer und Bearbeiten existierender Kunden verfügbar.

Projektverwaltung

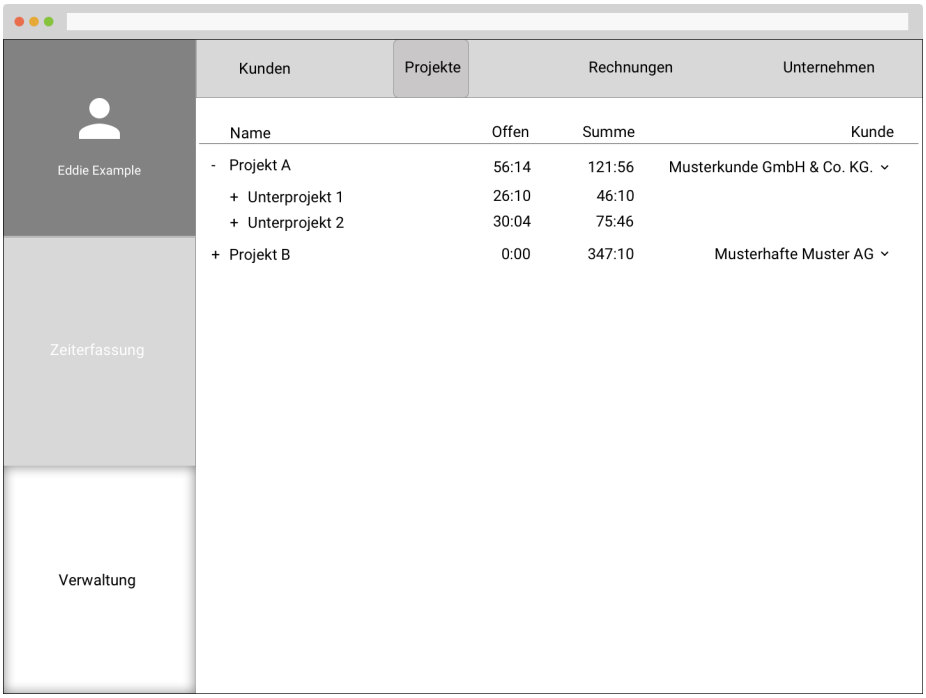

Abbildung 4.4: Wireframe Projektverwaltung

Bis auf das Formular ist der Aufbau der Projektverwaltung sehr ähnlich der Kundenverwaltung. Auch hier ist hauptsächlich eine Tabelle aller Projekte dargestellt.

Unternehmensverwaltung

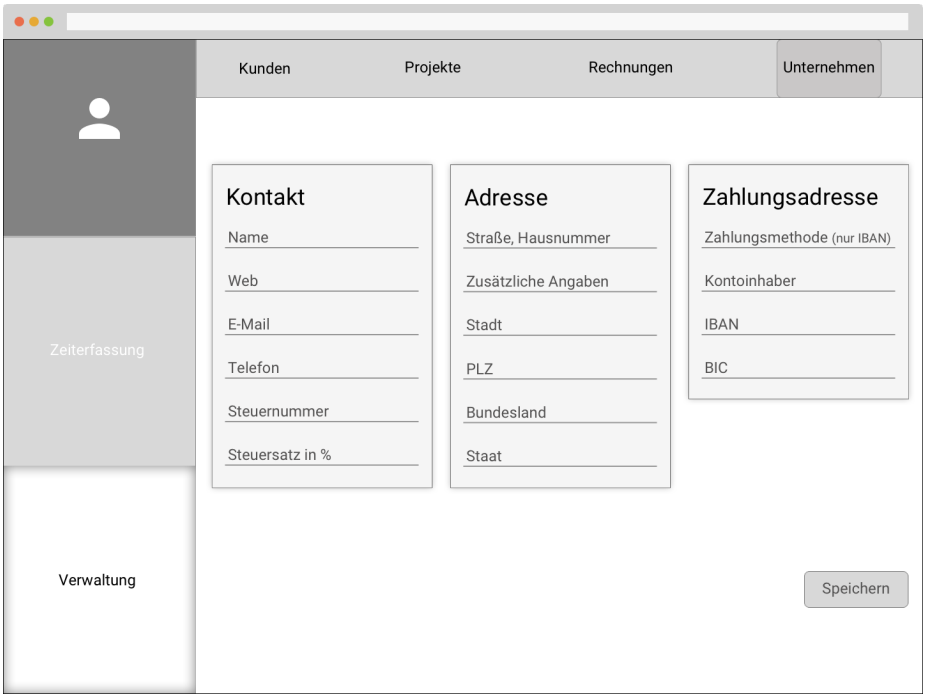

Abbildung 4.5: Wireframe Unternehmensverwaltung

Für die Verwaltung der Unternehmensdaten wird das zugehörige Formular in drei Abschnitte aufgeteilt. Ein Button zum Absenden des Formulars findet sich direkt darunter auf der rechten Seite.

Rechnungsverwaltung

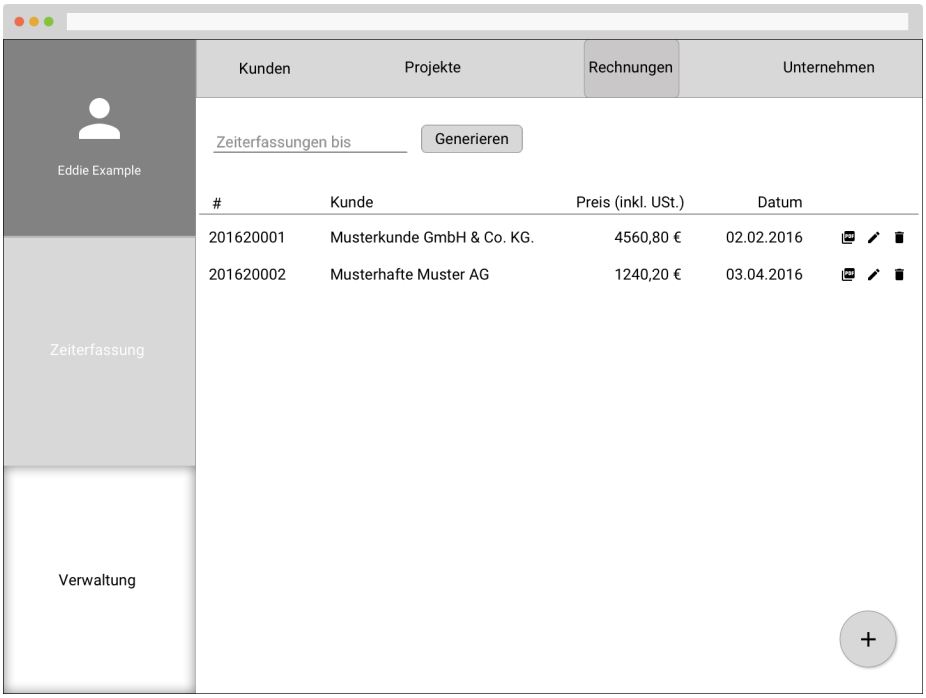

Abbildung 4.6: Wireframe Rechnungsverwaltung

Ebenfalls findet sich die aus der Kunden- und Projektverwaltung bekannte Tabelle bei der Auflistung der Rechnungen wieder. Zusätzlich werden pro Zeile die Icon-Buttons zur Ausgabe als PDF, sowie zum Bearbeiten und Löschen aufgeführt. Direkt darüber können über ein kleines Formular mit nur einem Feld Rechnungen generiert werden. Zur manuellen Erstellung einer Rechnung lässt sich der Floating Action Button verwenden, der das Rechnungsformular öffnet.

Rechnungsformular

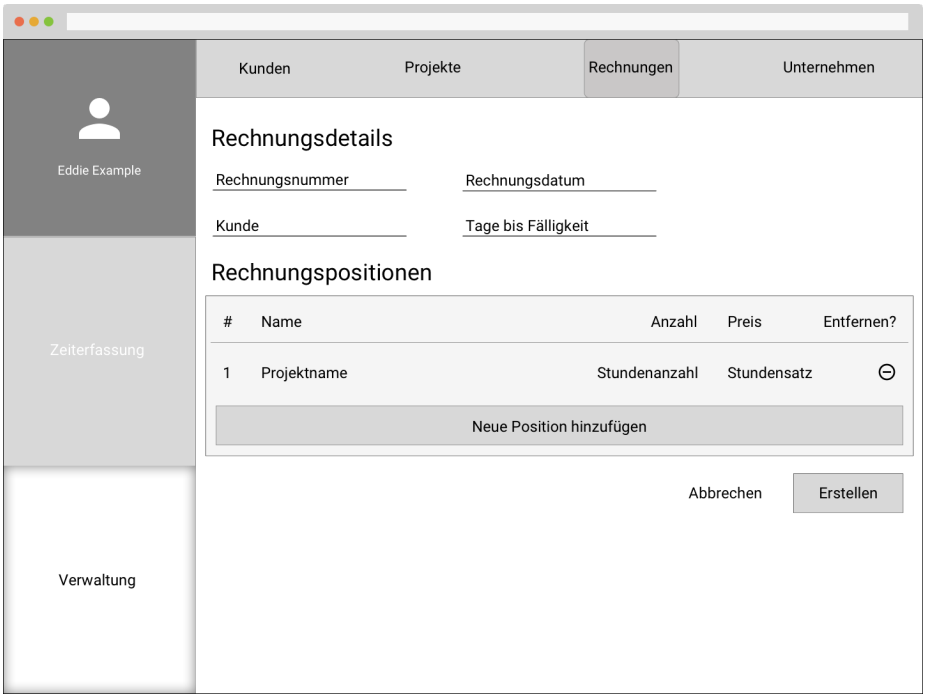

Abbildung 4.7: Wireframe Rechnungsformular

Das Rechnungsformular erstreckt sich über den gesamten Inhaltsbereich mit Ausnahme der Navigationsleiste. Es ist aufgeteilt in Rechnungsdetails, welches aus vier Formularfeldern besteht und den Rechnungspositionen. Letztere werden tabellarisch darstellt. Über einen darunter angeordneten Button lassen sich neue Positionen hinzufügen. Darüber hinaus verfügt jede Zeile, also jede Rechnungsposition, über einen Button zum Entfernen. Abschließend finden sich darunter ein Button zum Abbrechen des Vorgangs, sowie zum Speichern der Rechnung. Das Rechnungsformular wird ebenfalls auch zur Bearbeitung von Rechnungen verwendet.

5 Softwarearchitektur

Nachdem auf den vorherigen Seiten auf die Anforderungen und das Oberflächenkonzept eingegangen wurde, widmet sich dieses Kapitel dem technologischen Aufbau der Anwendung. Zuerst wird der generelle Aufbau der Architektur aufgezeigt, worauf die Beschreibung des Datenmodells, sowie die des Datenflusses folgen.

5.1 Genereller Aufbau

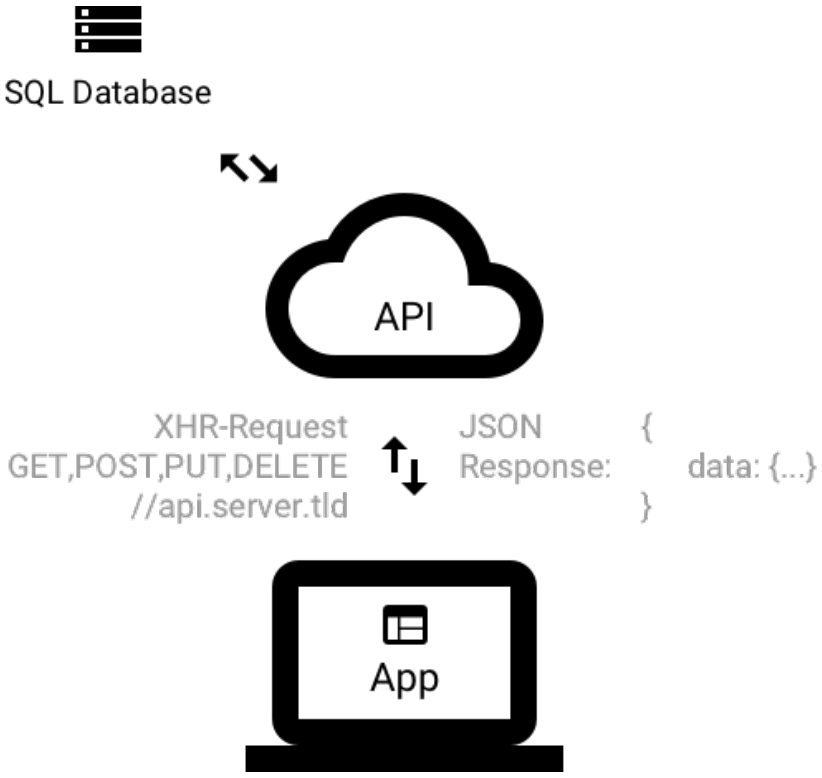

Abbildung 5.1: Aufbau der Anwendung

Um plattformunabhängig zu sein und somit von vornherein möglichst keine Nutzungseinschränkungen zu haben, wird die Anwendung dem Nutzer als clientseitige Webappikation, die innerhalb eines Browsers läuft, zugänglich gemacht. Um Informationen, wie beispielsweise die Liste aller vom Nutzer erstellten Projekte, zu erfragen, kann sie bei Bedarf, etwa nach Benutzerinteraktion oder beim Seitenaufruf, Anfragen

an einen Server schicken. Dieser antwortet mit dem entsprechenden Datensatz.
Damit der Nutzer seine erstellten Daten von einem beliebigen Browser auf einem beliebigen Computer abrufen kann, werden diese in einer Datenbank eines zentralen Servers von der Webapplikation gespeichert. Hierbei kommuniziert sie jedoch nicht direkt mit der Datenbank, sondern ausschließlich mit dem Application Programming Interface (API). Die API ist eine serverseitig laufende Anwendung, die Anfragen des Browsers entgegen nimmt und mit diesen nach entsprechenden Regeln, der eigentlichen Logik der gesamten Anwendung, umgeht. Das umfasst unter anderem das Authentifizieren des Nutzers, das Erstellen, Bearbeiten und Speichern von Datensätze, sowie das Exportieren von Datensätzen im PDF Format.
Dabei antwortet die API nicht wie herkömmliche Serveranwendungen in Form von fertig gerendertem HTML-Text, sondern im Javascript Object Notation (JSON) Format, welches die erfragten Informationen als reinen Datensatz darstellt. Die Webapplikation selbst kann daraus dann dem Nutzer ein entsprechendes User Interface (UI) in Form von HTML erstellen.
Der große Vorteil der gesamten Architektur ist die nahtlose Integration weiterer Clientapplikationen in das System, da diese ebenfalls die reinen Daten der API abfragen und über ihr eigenes UI-System, was nicht auf HTML basieren muss, darstellen können.

5.2 Datenmodell

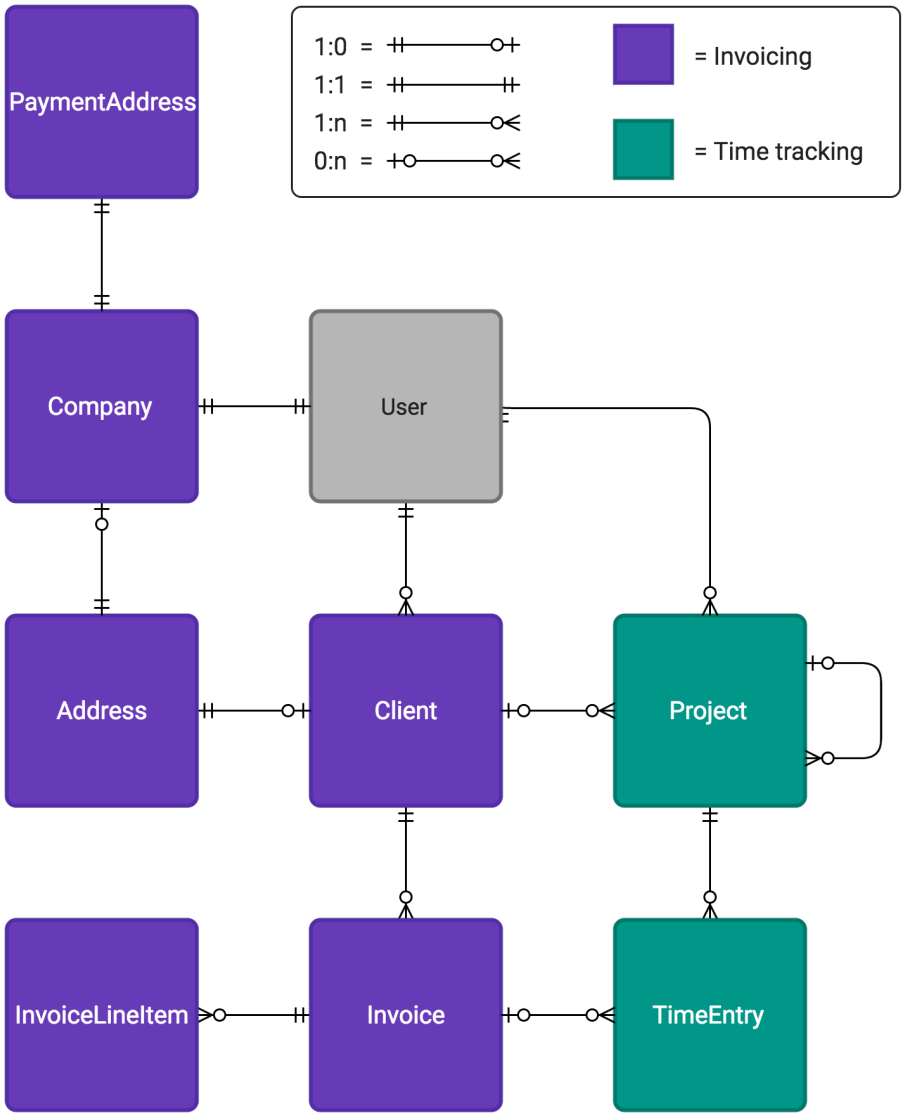

Abbildung 5.2: Entity-Relationship-Diagram

Die API ist nach dem Model-View-Controller Entwurfsmuster aufgebaut. Anhand eines Entity-Relationship-Diagrams in Abbildung 5.2 wird auf die einzelnen Models des Datenmodells eingegangen. Dabei werden auch die einzelnen Attribute und die jeweiligen Zusammenhänge dieser aufgezählt. Es ist anzumerken, dass jedes Model zusätzlich über die zwei Felder created_at und updated_at verfügt, welche jeweils einen Zeitstempel zur Erstellung sowie zur letzten Änderung bereitstellen. Letzteres wird nach jedem Speichern eines Datensatzes an die aktuelle Zeit angepasst.

User

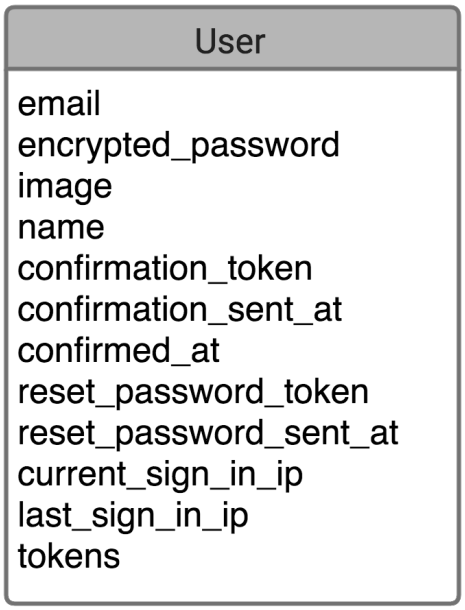

Abbildung 5.3: Attribute des User Models

Das User Model repräsentiert einen registrierten Nutzer und ist der Ausgangspunkt für alle vom Nutzer erstellten Einträge. Ein User kann beliebig viele Projects, TimeEntries, Clients und Invoices, sowie genau eine Company besitzen. Über die Company besitzt er transitiv auch genau eine PaymentAddress. Nach Erstellen eines Users, was bei erfolgreicher Registrierung geschieht, wird automatisch eine Instanz der Company erstellt und ihm zugewiesen.

Company

Abbildung 5.4: Attribute des Company Models

Das Company Model existiert pro Nutzer nur ein einziges Mal und beinhaltet die Unternehmensdaten dessen. Dazu bezieht es sich über drei Fremdschlüssel auf das User- , Address- und PaymentAddress Model.

Address

Abbildung 5.5: Attribute des Address Models

Das Address Model umfasst die Daten eines internationalen Adressformats. Es wird vom Company-, sowie vom Client Model verwendet, welche jeweils einen Fremdschlüssel für eine Address besitzen. In der Theorie ist es daher möglich, dass eine Instanz des Address Models beliebig viele Companies beziehungsweise Clients besitzt, allerdings wird die Beziehung zwischen diesen als 1:1-Relation behandelt.

PaymentAddress

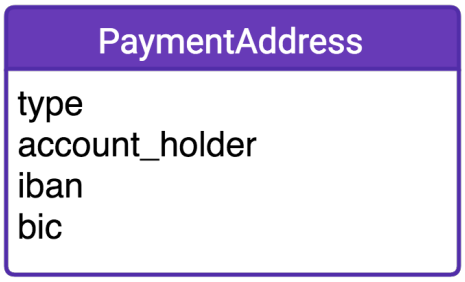

Abbildung 5.6: Attribute des PaymentAddress Models

Analog zum Address Model bietet das PaymentAddress Model ein Interface für ein internationales Zahlungsformat. Bisher ist lediglich "iban" als type unterstützt. Das PaymentAddress Model ist nur vom Company Model direkt referenziert, über welches es vom User Model aus transitiv durch eine *has-many-through* Relation[1] dennoch erreichbar ist.

Client

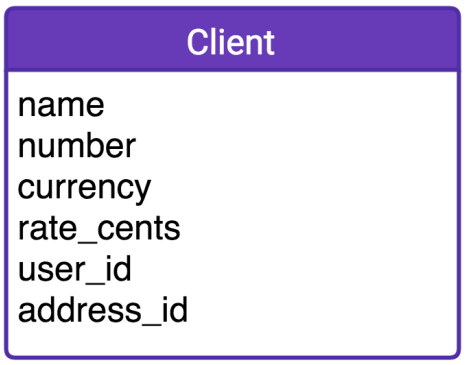

Abbildung 5.7: Attribute des Client Models

Das Client Model spiegelt einen Kunden wieder. Neben Basisdaten wie Name, Kundennummer und Adresse umfasst es auch die Währung und den Stundensatz, die für die Rechnungsstellung wichtig sind. Ein Client bezieht sich immer auf einen User und kann beliebig viele Projects und Invoices beinhalten.

[1] Näheres dazu in den (Ruby on Rails Guides)

Invoice

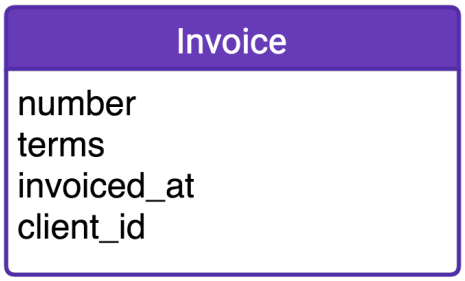

Abbildung 5.8: Attribute des Invoice Models

Das Invoice Model repräsentiert eine Rechnung. Dazu gehören neben der Felder Rechnungsnummer und Rechnungsfrist auch das Rechnungsdatum. Durch einen Fremdschlüssel bezieht es sich einen bestimmten Client. Der Rechnungsinhalt ergibt sich aus den InvoiceLineItems zu denen eine 1:n Beziehung besteht. Über diese kann eine Invoice optional transitiv mit Projects verknüpft sein.
Außerdem beinhaltet das Invoice Model noch gewisse Hilfsmethoden, um den Gesamtpreis mit und ohne Umsatzsteuer, sowie die Umsatzsteuer selbst zu erhalten.

InvoiceLineItem

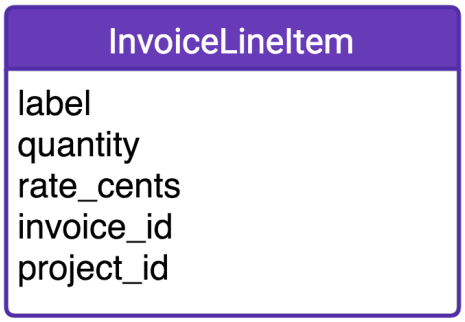

Abbildung 5.9: Attribute des InvoiceLineItem Models

Der Inhalt einer Rechnung ergibt sich aus den Rechnungspositionen, welche durch das InvoiceLineItem Model umgesetzt und durch einem Fremdschlüssel mit der Invoice verknüpft werden. Bei der Generierung von Rechnungspositionen aus offenen Zeiterfassungen werden InvoiceLineItems auch mit dem entsprechenden Project der Zeiterfassung verknüft. Dadurch ist es möglich, bereits in Rechnung gestellte Zeiterfassungen durch das Löschen von so verknüpften InvoiceLineItems wieder als offen zu kennzeichnen.

Project

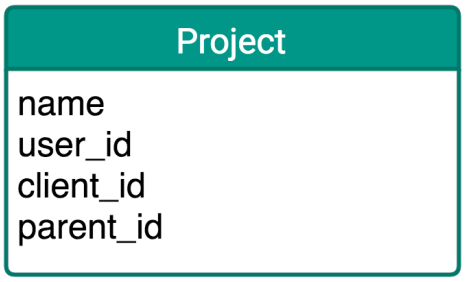

Abbildung 5.10: Attribute des Project Models

Zum Gruppieren und Zuordnen von Zeiterfassungen dienen Projekte, welche durch das Project Model repräsentiert werden. Die Beziehung zu einem Client ist im Gegensatz zur Beziehung zum User optional. Um die erwähnte baumartige Struktur umzusetzen, kann ein Project sich ebenfalls optional über das parent_id Feld auf ein Elternprojekt beziehen. Daher hat das Project Model eine 1:n Beziehung auf sich selbst.
Es besteht ebenso eine 1:n Beziehung zu TimeEntries und InvoiceLineItems. Dazu bietet das Model eine Hilfsmethode zum Zählen aller Projektkinder, sprich die Projects, die sich über die parent_id auf das Projekt oder dessen Unterprojekte beziehen. Desweiteren beinhaltet es eine Methode zum Summieren der Dauer aller nicht laufenden TimeEntries des Projects selbst und dessen Unterprojekte.
Bei Unterprojekten sind auch die Unterprojekte jedes Einzelnen eingeschlossen. Dies kann sich in beliebige Tiefe wiederholen und wird daher rekursiv gelöst.

TimeEntry

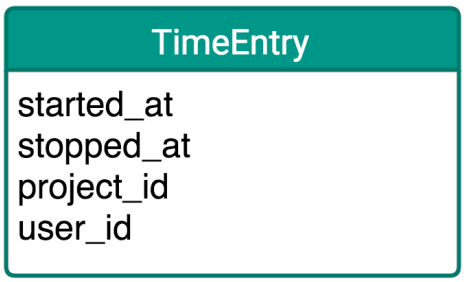

Abbildung 5.11: Attribute des TimeEntry Models

Um eine Zeiterfassung wieder zu geben, gibt es das TimeEntry Model, welches aus einer Startzeit und einer Endzeit besteht. Außerdem bezieht es sich auf ein Project, sowie einen User. Die Beziehung zu einem User ist nicht transitiv über das Project,

sondern direkt über einen Fremdschlüssel realisiert, da es eventuell als zukünftiges Merkmal der Anwendung möglich sein soll, dass mehrere User am selben Project arbeiten und dementsprechend auch Zeit für dieses erfassen können. Beim Starten einer neuen Zeiterfassung stoppt das TimeEntry Model eine andere bis dahin laufende TimeEntry Instanz automatisch.
Es sind auch hier Hilfsmethoden verfügbar. Beispielsweise um die Dauer einer Zeiterfassung oder den Status, ob ein TimeEntry gerade läuft, zu erhalten. Auch werden hier beim Speichern einer TimeEntry Instanz die Start- und Endzeit auf Minuten gerundet.

5.3 Datenfluss

Im vorherigen Kapitel wurde das Datenmodell samt Models und deren Zusammenhänge aufgezeigt. Wie und wann Daten aus jenem Modell akquiriert werden, soll dieses Kapitel aufzeigen. Daher werden nun mögliche Anfragen der Webapp an die API näher betrachtet.

5.3.1 HTTP-Methoden

In der beschriebenen Architektur werden fast ausschließlich die folgenden vier HTTP-Anfrage Methoden genutzt. Um Datensätze abzufragen, sendet die Webapplikation eine GET-Anfrage. Zum Bearbeiten eines bestehenden Datensatzes wird eine PUT-Anfrage mit den zu ändernden Werten gewählt. Soll etwas erstellt werden, so schickt sie eine POST-Anfrage mit den Werten des neuen Datensatzes und zum Löschen wird eine DELETE-Anfrage gestellt.
Alle Anfragen an die API werden vom auf einem Server laufenden Webserver angenommen und an die API-Anwendung weitergegeben. Diese stellt eine Router-Klasse zur Verfügung, welche mit den in der Webapplikation hinterlegten Anfrage-Pfaden umgehen kann. Dafür sind gewisse Routing-Regeln innerhalb der Klasse hinterlegt, die einen bestimmten Anfrage-Pfad an einen bestimmten Controller und dessen Action-Methode weiterleiten. Dort wird die Anfrage verarbeitet, notwendige Daten akquiriert, berechnet und als Antwort an die Webapp wieder zurückgesendet.

Für die Models Client, Project, TimeEntry, Invoice und Company bietet die API jeweils einen Controller, der pluralisiert nach dem sich beziehenden Model benannt ist. Bis auf den CompaniesController sind alle nach dem CRUD-Prinzip[2] aufgebaut und stellen im gleichen Zug auch deren Endpunkte der API bereit.

[2]Bietet die häufig genutzten Datenbankoperationen Create, Read, Update und Destroy.

5.3.2 CRUD-Controller

Mithilfe des ProjectsControllers wird ein auf CRUD basierender Controller exemplarisch erläutert.
Zum Erstellen eines neuen Projekts (Create) muss eine POST-Anfrage mit dem Projektnamen an den Endpunkt /projects gesendet werden. Diese wird innerhalb der API an die create-Methode des ProjectsControllers weitergegeben, wo ein neues Projekt erstellt und in der Datenbank persistiert wird. Wenn das erfolgreich ist, wird das neu erstellte Projekt als Antwort auf die POST-Anfrage zurückgesendet.
Um alle Projekte geliefert zu bekommen, ist es notwendig, eine GET-Anfrage an /projects zu senden, welche letztlich zur index-Methode des ProjectsControllers durchgereicht wird. Dort werden nun alle Projekte aus der Datenbank abgefragt und als Antwort an den Browser zurückgeschickt.
Jeder Datensatz trägt eine Identifikationsnummer (ID), welche man an den Pfad der Anfrage anhängt, um bestimmte Datensätze zu ändern. Beispielsweise kann der Name des Projekts mit der ID 7 geändert werden, indem eine PUT-Anfrage mit dem neuen Namen an /projects/7 gesendet wird. Die update-Methode führt den entsprechenden Befehl zum Ändern des Datensatzes aus.
Analog dazu kann durch eine DELETE-Anfrage an /projects/7 das Projekt 7 gelöscht werden. Diese wird an die destroy-Methode durch gereicht, die den Datensatz aus der Datenbank entfernt.

Im Sequenzdiagramm 5.12 werden CRUD-Anfragen der Webapplikation an die API dargestellt.

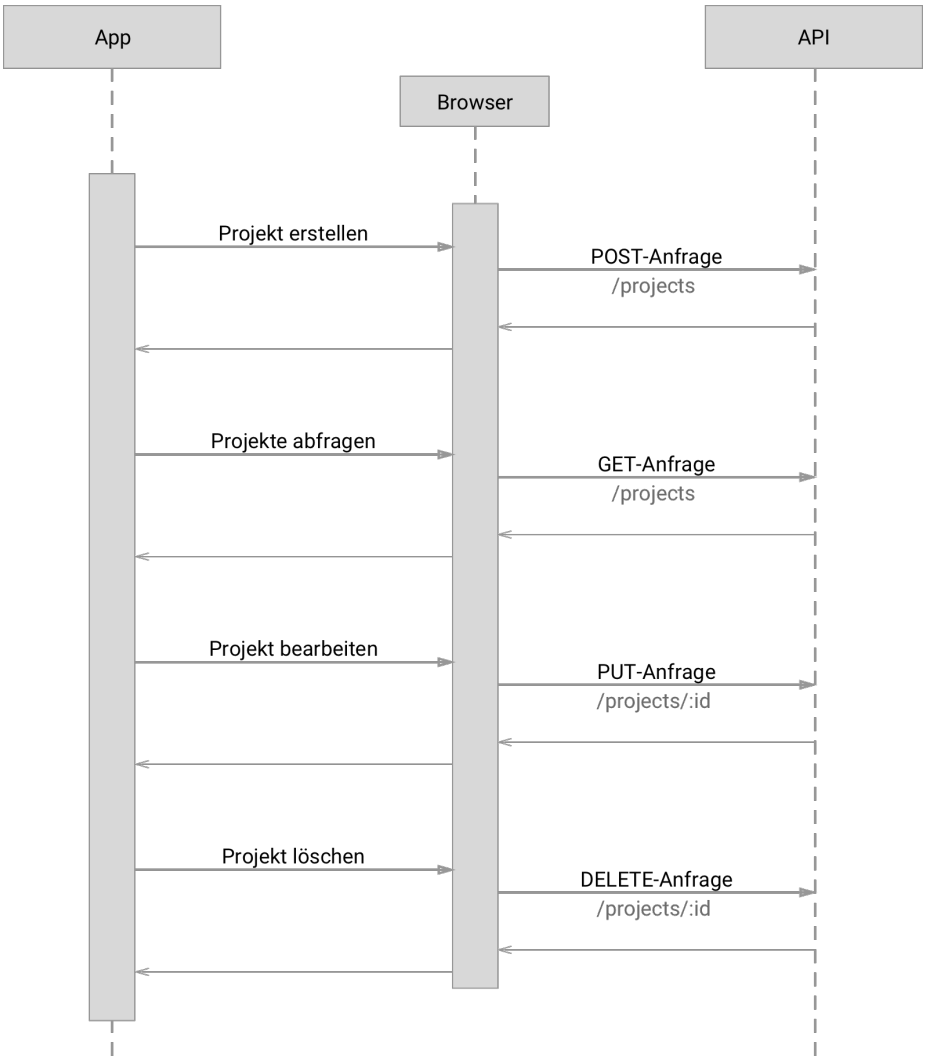

Abbildung 5.12: Kommunikation der Webapplikation mit der API am Beispiel des ProjectsControllers

6 Realisierung

Dieses Kapitel beschreibt, wie der theoretische Teil der Softwarearchitektur in eine benutzbare Anwendung umgesetzt wird.

6.1 API

Die API wird mithilfe des Frameworks Ruby on Rails (im Folgenden lediglich als Rails bezeichnet) realisiert. Zunächst wird es über dieses einen kurzen Überblick geben, bevor auf die Übertragung des genannten Datenmodells in die Datenstruktur von Rails eingegangen wird. Nach der Darstellung der API Endpunkte wird auf die Umsetzung der Authentifizierung und schließlich auf anwendungsspezifische Schwierigkeiten eingegangen.

6.1.1 Ruby on Rails

Zur Umsetzung der API wird das open-source Webframework Ruby on Rails in Version 5.0.0.beta3 genutzt. Die erste Version von Rails wurde von David Heinemeier Hansson 2003 entwickelt, indem er es aus der damaligen Version von Basecamp, einer Software für Projektmanagement, extrahierte. Seit dem ist Rails open-source und wird communitygetrieben nach dem gleichen Prinzip wie Ruby, *make the programmer happy*, weiterentwickelt. Daher folgt Rails zum einen dem Entwicklungsmuster *Don't repeat yourself* (DRY), nach dem Redundanzen durch diverse Abstrahierungen vermieden werden sollen. Zum Anderen gilt *Convention over Configuration*, wodurch sich für viele Problemstellungen vom Framework vorgegebene Lösungswege ergeben. Diese werden von der Community auch *Rails Way* genannt, sind bei bestimmten Grenzfälle jedoch nicht immer angemessen[1].
Ein gutes Beispiel für einen positiven Effekt von Konventionen ist die Namenskonvention von Modelrelationen. Rails pluralisiert beziehungsweise singularisiert Klassennamen, um eine höhere Semantik und dadurch ein verständlicheres Lesen innerhalb des Programmiercodes zu ermöglichen. So ist in der Modelklasse Project lediglich durch `has_many time_entries` eine 1:n Relation zur Modelklasse TimeEntry festgelegt. Darum hat jede Instanz der Klasse Project nun eine Hilfsmethode *time_entries*, welche alle TimeEntry Instanzen des Projekts zurückliefert.

[1] http://andrzejonsoftware.blogspot.de/2014/04/be-careful-with-rails-way.html

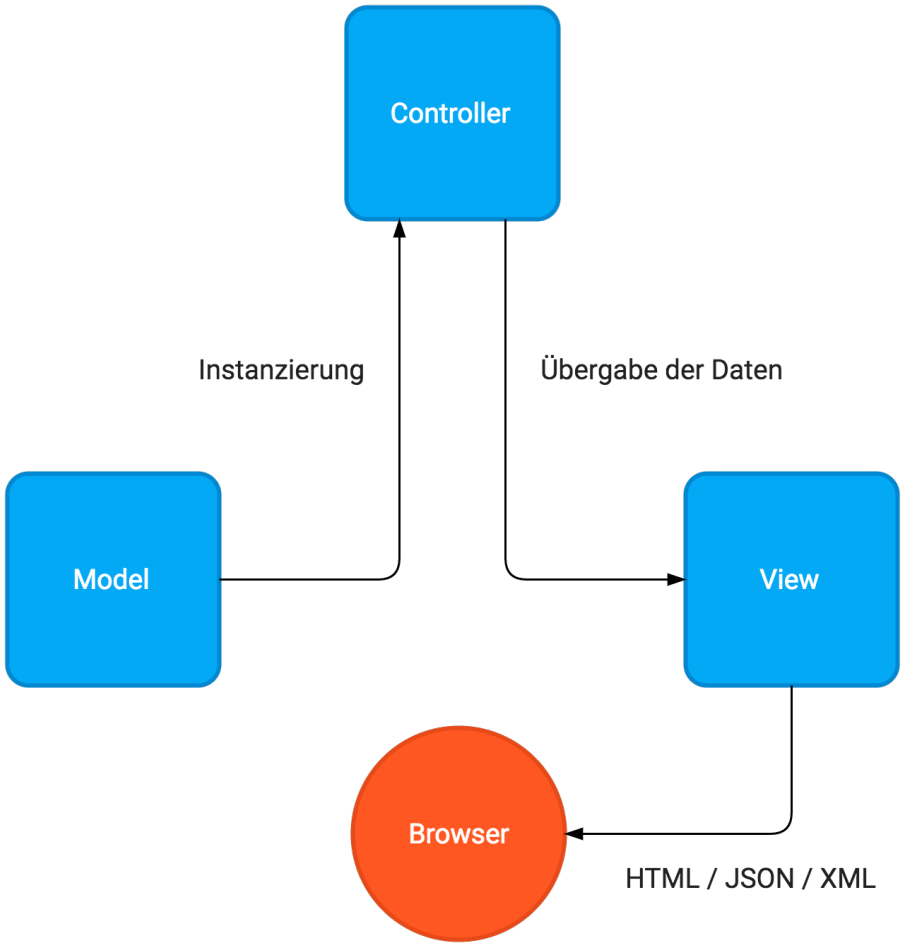

Abbildung 6.1: Datenfluss Model-View-Controller

Railsanwendungen sind nach dem Model-View-Controller (MVC) Muster aufgebaut. MVC ist ein beliebtes Entwurfsmuster in der Softwaretechnik, vor Allem im Webumfeld, welches eine Anwendung grundlegend in drei Bereiche gruppiert.

Model

Die Logik eines Datenmodells wird in Models aufgeteilt, hier sind häufig Funktionalitäten zur Behandlung von Relationen untereinander oder das Persistieren in Datenbanken implementiert.

Controller

Controller sind für die Steuerung der Anwendung zuständig. Sie instanzieren Models, um so Daten, beispielsweise aus einer Datenbank, zu erhalten und geben diese letztlich an die dritte Gruppe, die Views, weiter.

Views

Diese sorgen für die Darstellungsform der vom Controller bereitgestellten Daten. In herkömmlichen Webanwendungen übergeben Views als HTML-Text die Daten an den Browser, wo dieser als grafische Webseite angezeigt wird.
Da diese Anwendung als API genutzt werden soll, werden die aus der Datenbank bezogen Daten nicht in Form von HTML, sondern in *Javascript Object Notation* (JSON) an den Browser geschickt. JSON ist ein neutrales Datenformat, welches sich ausschließlich Daten präsentiert und keine für die grafische Darstellung notwendigen Informationen beinhaltet. Dadurch ist eine an das API verbundene Clientanwendung uneingeschränkt in der Nutzung der empfangenen Daten und kann bespielsweise eine komplett andere grafische Oberfläche implementieren, als eine andere Clientanwendung, die ebenfalls mit der API verbunden ist.

rails-api

Seit 2012 gibt es eine Abwandlung von Ruby on Rails namens *rails-api*[2], welche eine Railsanwendung aufsetzt, die vorkonfiguriert ist für die Nutzung als API. Durch die Vorkonfiguration zur Ausgabe als JSON muss keines der Module zum Rendern von HTML-Views integriert und geladen werden. Zusätzlich muss eine API-Anwendung weder *Cascading Stylesheets* (CSS), noch Javascript Dateien komprimieren und ausliefern. Dies geschieht in einer normalen Rails Anwendung über die *Asset-Pipeline*. Diese wird bei einer API nicht benötigt und ist daher ebenfalls nicht integriert. Seit Version 5 ist die rails-api Variante in Rails direkt integriert[3] und wird in dieser Anwendung auch genutzt.

Scaffolding

Durch eine konventionsgetriebene Entwicklung lassen sich mit Rails sehr schnell Prototypen entwerfen. Außerdem bietet Rails mit dem Scaffolding ein weiteres Tool, welches den Entwicklungsprozess vereinfachen und beschleunigen kann.
Rails Scaffolding ist ein auf der Kommandozeile basierender Generator, der verschiedene Methoden bereitstellt, um unter Anderem grundlegende Models und Controller automatisch zu erzeugen. Gerade für die Erstellung von Models eignet sich dieser Generator, der neben der eigentlichen Modelklasse auch eine Migration[4] anlegt.

[2] https://github.com/rails-api/rails-api
[3] Für Details siehe folgenden Pull Request `https://github.com/rails/rails/pull/19832`
[4] Siehe (Ruby on Rails Guides)

Migrations

Migrations beinhalten Änderungsvorschriften für eine Datenbank. Beispielsweise beinhaltet eine durch Modelscaffolding erzeugte Migration den Befehl zum Anlegen einer Tabelle mit den beim Generieren angegebenen Feldern. Änderungen an der Datenbankstruktur werden in Rails ausschließlich über Migrations gehandhabt. Bei Aufsetzen der Anwendung auf einem anderen Rechner ist es dann nur notwendig alle Migrations einmalig auszuführen und schon ist die anwendungsspezifische Datenbankstruktur dort ebenfalls vorhanden. Daher sind Migrations sehr geeignet zum Synchronisieren von Datenbankstrukturen innerhalb eines Entwicklungsteams.

Viele der genannten Eigenschaften bieten auch andere Webframeworks wie *Django*[5] oder *Phoenix*[6]. Aber aufgrund des Bestehens persönlicher Expertise und der erst vor kurzem integrierten rails-api Variante wurde sich für Ruby on Rails entschieden.

6.1.2 Übertragung Datenmodell nach Rails

Object-Relationship-Mapping

Für die Umsetzung des oben genannten Datenmodells wird das in Rails integrierte *Object-Relationship-Mapping* (ORM) Framework *ActiveRecord* genutzt. In der Datenbank sind bestimmte Tabellen über Fremdschlüssel verknüpft. ORM beschreibt ein Prinzip, um die Relationen einer Datenbank auf objekt-orientierte Klassen abzubilden. Eine Umsetzung dieses Prinzips abstrahiert alle trivialen Datenbankanfragen zu Klassenmethoden und fungiert somit als Schnittstelle zwischen Datenbank und Programmierung.
Beispielsweise bietet die ActiveRecord *Base* Klasse, von der alle Models erben, die Methode *all*. Diese fragt alle Einträge des Models aus der Datenbank ab, instanziert mit deren Daten jeweils ein Objekt der Modelklasse und liefert diese als Array zurück. Die *all* Methode selbst übernimmt also den notwendigen *Structured-Query-Language* (SQL) Aufruf an die Datenbank, um alle Einträge auszulesen. Dadurch wird die Arbeit des Entwicklers nicht nur erleichtert, sondern auch sicherer. Vor der Ausführung eines SQL Aufrufs wird dieser nämlich gefiltert, um *SQL Injections*[7] zu verhindern.

[5] https://www.djangoproject.com/
[6] http://www.phoenixframework.org/
[7] Bezeichnet das Einschleusen von nicht vorgesehenen sicherheitsrelevanten Zeichenketten in einen SQL-Aufruf. Näheres unter http://www.derkeiler.com/Mailing-Lists/securityfocus/secprog/2001-07/0001.html

Relationen

Relationen lassen sich mit ActiveRecord über die Hilfsmethoden *belongs_to*, *has_one*, *has_many* und *has_many through* definieren[8]. Diese implementieren im Hintergrund die Verknüpfung zweier Tabellen in der Datenbank mit Fremdschlüsseln und bieten ebenfalls Hilfsmethoden zur vereinfachten Abfrage. Beispielsweise wird durch `belongs_to :user` innerhalb des Project Models eine *user* Hilfsmethode definiert. Bei Aufruf dieser wird die, durch den Fremdschlüssel *user_id* referenzierte, Instanz des User Models zurückgeliefert.

Validierung

Eine letzte wichtige Eigenschaft von ActiveRecord ist die integrierte Validierung direkt vor dem Speichern einer Modelinstanz. Es lassen sich diverse Voraussetzungen definieren, damit diese erfolgreich ist. Nur wenn dies der Fall ist, wird die Modelinstanz tatsächlich gespeichert. So lassen sich zum Beispiel Pflichtfelder eines Models definieren oder prüfen, ob der Inhalt eines Feldes in einem bestimmten Format vorliegt. Diese Prüfmechanismen sind von ActiveRecord bereits vorgegeben. Es lassen sich aber auch eigene Kriterien definieren. So kann man für die Validierung des TimeEntry Models definieren, dass der Wert für *started_at* zeitlich vor dem von *stopped_at* liegen muss, falls letzterer existiert.

6.1.3 Endpunkte

Durch den von Rails bereitgestellten Router werden die von Clients ansteuerbaren Endpunkte beschrieben. Diese ergeben sich aus Routes, welche Verweisungen von URLs auf Controllermethoden sind. Diese Methoden, die durch den Aufruf einer Route ausgeführt werden, heißen Actions und sind in einem bereits genannten CRUD Controller die Methoden *create*, *index*, *update* und *destroy*.

Definieren von Routes

Für das Hinterlegen einer Route sind eine HTTP-Methode, ein Pfad und eine Action notwendig. Dabei können im Pfad auch beliebig viele Platzhalter für Parameter, sowie *Wildcards* hinterlegt werden[9]. Zum Beispiel legt `get "time_entries/:id" => "time_entries#show"` fest, dass beim Öffnen des Pfades "/time_entries/42" die Action *show* des *TimeEntriesControllers* aufgerufen und dort als Feld *id* im Hash *params* 42 mitgegeben werden soll. Mit diesen Informationen ist es dem Controller beispielsweise möglich, relevante Daten für den TimeEntry mit der ID 42 aus der Datenbank auszulesen und an den Browser zu senden.

[8] Siehe (Ruby on Rails Guides)
[9] Siehe (Ruby on Rails Guides)

Resources

Da für einen CRUD-Controller jeweils fünf Routes auf jene Weise definiert werden müssten, lassen sich diese, um Redundanz zu vermeiden, mit sogenannten Resource Routes vereinfacht festlegen. Der Befehl `resources :time_entries` stellt nach Konvention die Routes zum Lesen aller und zum Lesen, Erstellen, Bearbeiten und Löschen eines bestimmten TimeEntries bereit.

Namespaces

Um Routes zu strukturieren bietet Rails Namespaces an. Diese erlauben es, beliebige Routes zu gruppieren und unter einem festgelegten Pfadprefix bereitzustellen. Zusätzlich erwarten diese nach Konvention, dass die zugeordneten Controller sich in einem Ruby Module befinden, das den gleichen Namen wie der Namespace trägt.
In dieser Anwendung ist ein Namespace "api" definiert. Daher befinden sich alle Controller innerhalb diesen Ordners beziehungsweise des API Modules.
Darüber werden Namespaces in dieser Anwendung auch zur Versionierung genutzt. Damit bestehende Clientanwendungen trotz Entwickelung von neuen Versionen der API diese weiterhin nutzen können, ist jede Version in ihrem eigenen Namespace gekapselt. So ist der Projects-Endpunkt von Version 1 unter */v1/projects* erreichbar.
Anzumerken ist, dass der Namespace *api* nicht als Pfadprefix, sondern als Subdomain konfiguriert ist. Außerdem sind alle Namespaces zur Versionierung diesem untergeordnet. Wodurch sich folgendes URL Pattern für Version 1 der API ergibt: *https://api.domain.tld/v1/*
Die oben erwähnte Route zum Lesen eines einzelnen TimeEntries mit der ID 42 wird nun auf die URL *https://api.domain.tld/v1/time_entries/42* abgebildet.

6.1.4 Authentifizierung

Die Authentifizierung ist mithilfe des Gems[10] devise[11] und devise_token_auth[12] realisiert.

Devise

Devise ist eine Komplettlösung zum Authentifizieren eines Nutzers für Ruby on Rails, welche beliebige Models zu authentifizierbaren Models samt Registierung, Login und weiteren Eigenschaften erweitern kann. Allerdings baut es auf das von Rails normalerweise bereitgestellte HTML-View System auf, welches in dieser API nicht integriert ist, siehe Kapitel *rails-api*.

[10] Ein einzelnes Paket des Ruby Paketsystems RubyGems
[11] https://github.com/plataformatec/devise
[12] https://github.com/lynndylanhurley/devise_token_auth

Token Auth

Damit Clientanwendungen nicht abhängig von HTML-Views des devise Gems sind, ist es notwendig alle Funktionalitäten der Authentifizierung ebenfalls als JSON Schnittstelle zugänglich zu machen, weshalb das Gem *devise_token_auth* integriert worden ist. Durch dieses kann jede Clientanwendung eine eigene Handhabung zur Darstellung implementieren und sich mittels HTTP-Anfragen authentifizieren.

Das Prinzip von Token Auth setzt voraus, dass bei jeder Anfrage eines Clients an die API die Informationen *access-token*, *token-type*, *client*, *expiry* und *uid* im Header mitgeführt werden. Diese kann die API mit der Datenbank abgleichen und bei Gültigkeit zuerst den Benutzer feststellen und danach die erfragten Benutzerdaten preisgeben. Dabei wird neben dem Inhalt der Antwort auch ein neu erstellter Token mitgeschickt, welcher für die kommende Anfrage des Nutzers gültig ist. Das bedeutet, dass die Clientanwendung diese Daten persistieren und dort den alten Token nach jeder Anfrage mit dem Neuen ersetzen muss. Nach 2 Wochen ist ein Token abgelaufen, spätestens dann muss der Nutzer neu authentifiziert werden.
In Abbildung 6.2 ist das Verfahren illustriert.

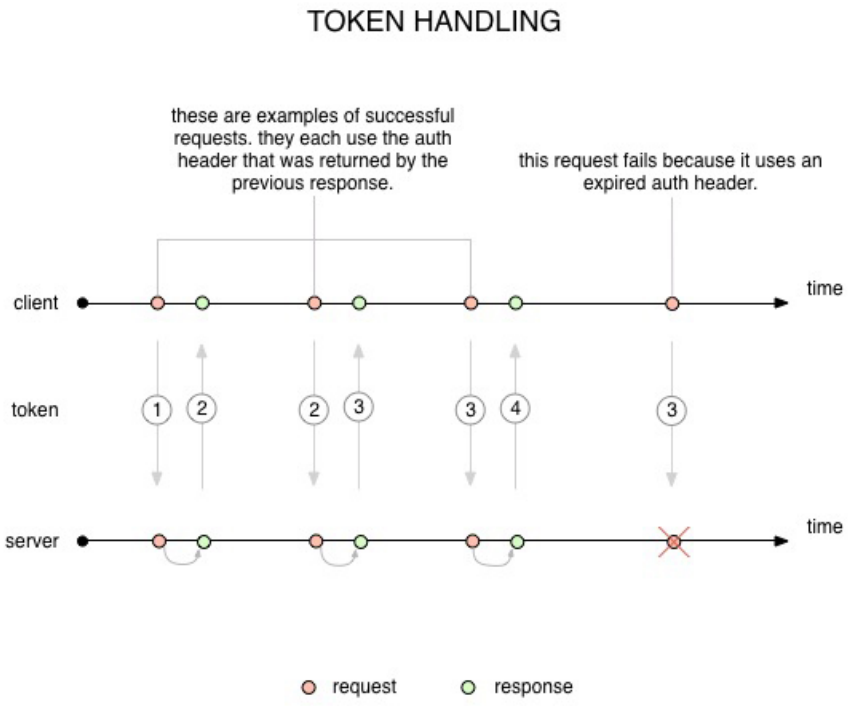

Abbildung 6.2: Skizzierung zu Token-Handling aus der *devise_token_auth* Dokumentation (devise-token-auth)

Im Folgenden wird nun auf einzelne Eigenschaften des bereitgestellten Systems zur Authentifizierung eingegangen.

Token Validierung

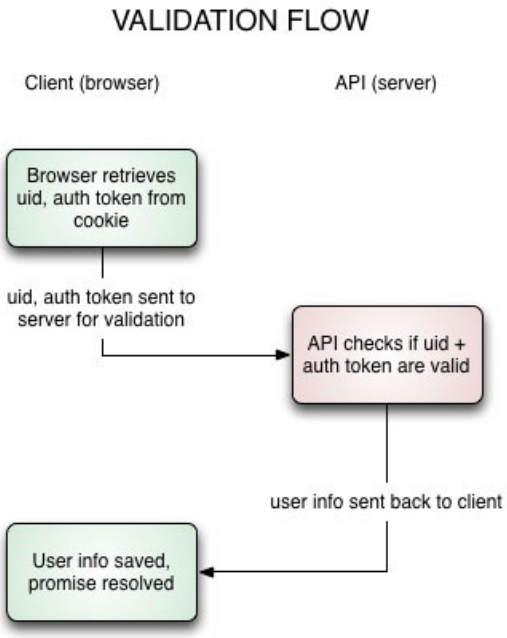

Abbildung 6.3: Skizzierung zur Token-Validierung aus der *devise_token_auth* Dokumentation (devise-token-auth)

Der Ablauf der Token Validierung ist in Abbildung 6.3 dargestellt. Um zu Prüfen, ob der Nutzer bereits authentifiziert ist, sendet die Clientanwendung ihre bestehenden Informationen über den Token und die uid an den *Token Validation* Endpunkt, der unter */api/v1/auth/validate_token* erreichbar ist. Bei gültigen Daten antwortet die API mit den Informationen des erfragten Users. Diese Anfrage wird beispielsweise beim Seitenaufruf beziehungsweise beim Initialisieren der Webapplikation ausgeführt.

Registrierung

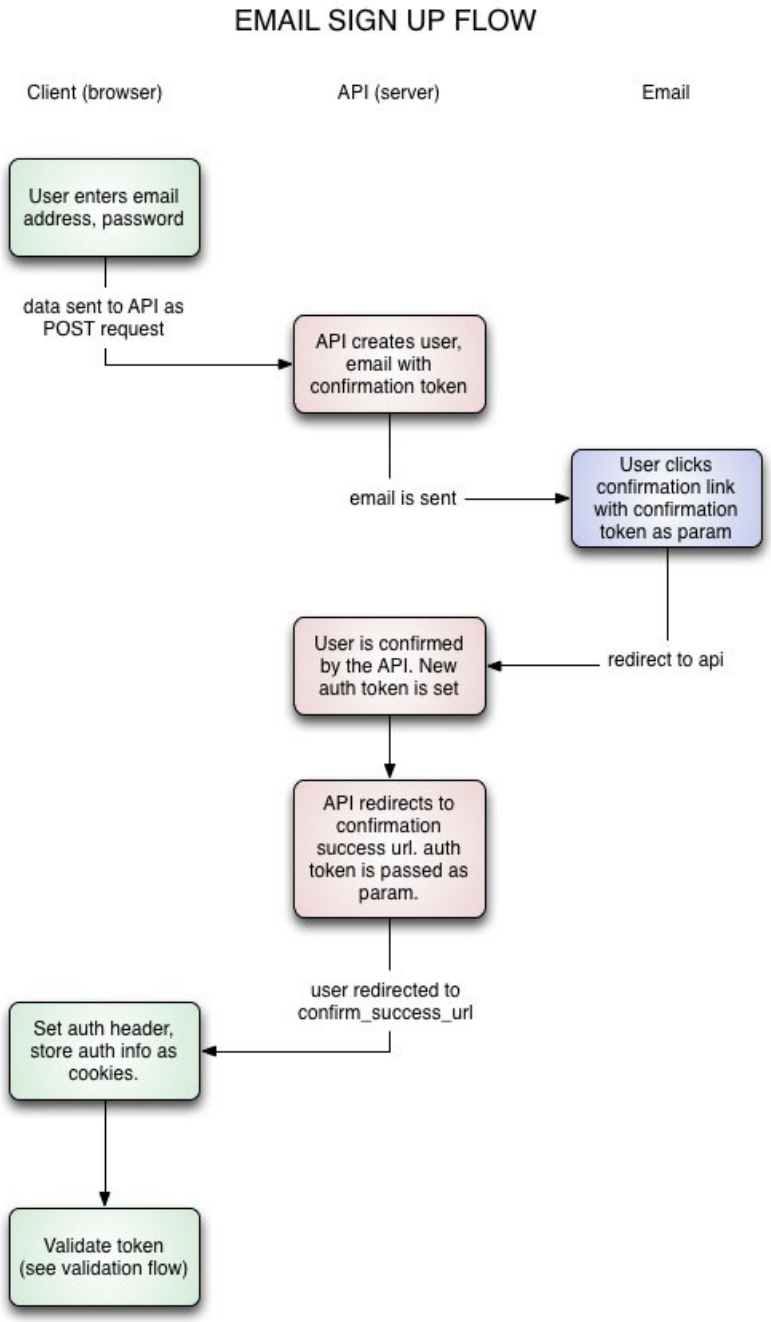

Abbildung 6.4: Skizzierung zur E-Mail Registrierung aus der *devise_token_auth* Dokumentation (devise-token-auth)

Damit ein neuer User registriert wird, sendet der Client eine POST-Anfrage mit der Mailadresse, dem gewählten Passwort, sowie der Passwortbestätigung an den *Sign Up* Endpunkt an */api/v1/auth*. Darauf legt die API einen neuen Nutzer an und sendet eine E-Mail an die übermittelte Adresse mit einem Bestätigungslink. Dieser beinhaltet einen einzigartigen *Confirmation Token*, der bei Aufruf des Links an die API übergeben und von dieser mit dem zu bestätigenden User abgeglichen wird. Stimmen beide Werte überein, ist die Bestätigung vertrauenswürdig und der User ist zur uneingeschränkten Nutzung der Anwendung ermächtigt. Daraufhin leitet die API den Nutzer zur Webapplikation weiter. Dort werden die mitgelieferten Token Auth Informationen im LocalStorage[13] gespeichert und können zur Token Validierung genutzt werden (Siehe Abbildung 6.4).

Login

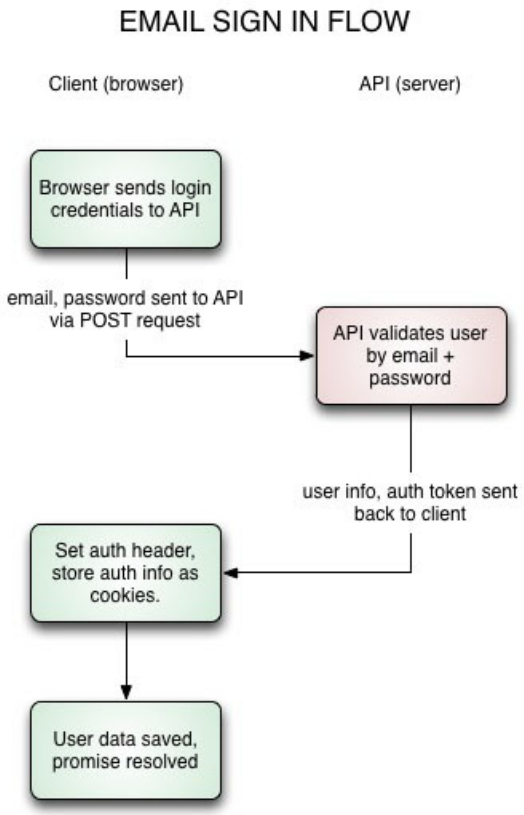

Abbildung 6.5: Skizzierung zum E-Mail Login aus der *devise_token_auth* Dokumentation (devise-token-auth)

[13] https://developer.mozilla.org/de/docs/Web/API/Window/localStorage

Ist der Token einmal abgelaufen oder der Nutzer möchte sich von einem anderen Gerät aus authentifizieren, muss er sich über den *Sign In* Endpunkt einloggen. Wie in Abbildung 6.5 gezeigt, schickt die Clientanwendung dafür eine POST Anfrage mit Mailadresse und Passwort an */api/v1/auth/sign_in*, worauf sie neu erstellte Token Auth Informationen und die Daten des zugehörigen Users zurückgeliefert bekommt.

Passwort vergessen

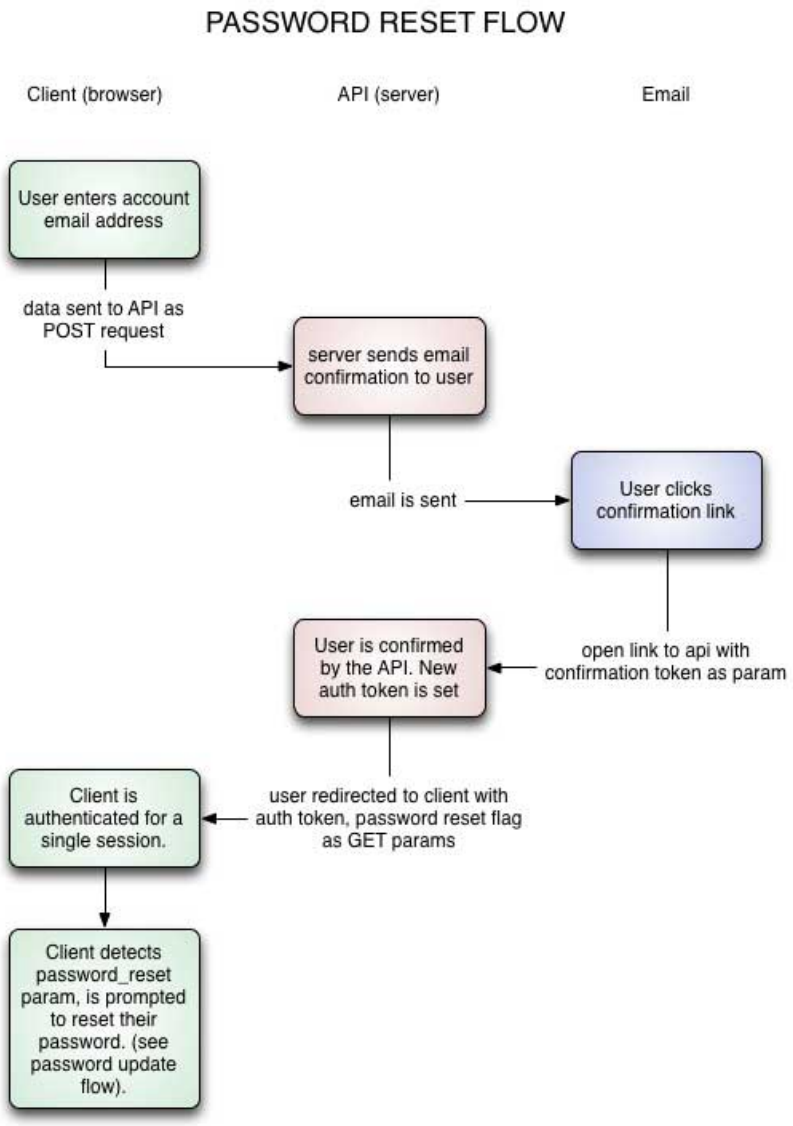

Abbildung 6.6: Skizzierung zur Passwort Zurücksetzung aus der *devise_token_auth* Dokumentation (devise-token-auth)

Falls ein Nutzer sein Passwort vergisst, kann er sich über einen Bestätigungslink einmalig authentifizieren, um ein neues Passwort zu setzen. Dafür wird, wie in Abbildung 6.6 gezeigt, die Mailadresse über eine POST Anfrage an */api/v1/auth/password* gesendet, woraufhin die API eine Passwort-Zurücksetzen-Mail an diese sendet. Die E-Mail beinhaltet analog zur Bestätigungsmail einen Bestätigungslink. Wird dieser vertrauenswürdig aufgerufen, leitet die API den Nutzer mit einem einmalig gültigen Token auf die Webapplikation weiter. Dort hat der Nutzer die Möglichkeit sein Passwort zu ändern.

6.1.5 Besondere Schwierigkeiten

Im Laufe der Realisierung der API ergaben sich einige besondere Schwierigkeiten, die für die Anwendung speziell sind und in diesem Kapitel hervorgehoben werden sollen.

Zeitaufzeichnung

Synchronisierung Client- und Serverzeit Bei der Synchronisierung von Zeit zwischen Client und API ergaben sich durch unterschiedliche Zeitzonen Differenzen. Im Browser wird standardmäßig die lokale Zeit des Anwenders genutzt. Dieser kann sich aber in einer anderen Zeitzone als die des Servers der API befinden. Ein möglicher Lösungsansatz wäre die zu nutzende Zeitzone vom Anwender konfigurieren zu lassen, allerdings müsste diese bei Betreten einer anderen Zeitzone jedes Mal aktualisiert werden.
Um dem vorzubeugen wird Zeit innerhalb der Anwendung immer in Form der koordinierten Weltzeit UTC[14] verwendet. Damit lassen sich Werte für Zeiten auf Client-, sowie auf API-Seite gleich behandeln.

Rundung von started_at und stopped_at Obwohl in der Webapplikation die Stoppuhr der Zeiterfassung sekundengenau dargestellt wird, dient dies lediglich, um dem Nutzer ein Gefühl von Aktivität und Fortschritt zu vermitteln. Für die eigentliche Auswertung von Zeiterfassungen, wie sie zur Rechnungsgenerierung letztendlich genutzt wird, soll die Dauer jeweils nur minutengenau sein. Daher ist es notwendig, dass vor dem Speichern einer Zeiterfassung die Zeitstempel für *started_at* und *stopped_at* auf Minuten gerundet werden. Dabei gibt es eine Sonderregelung. Sind die Werte nach der Rundung identisch, wird der Wert für *stopped_at* exakt um eine Minute erhöht. So ergibt sich auch nach einer Zeiterfassung von einer eigentlichen Dauer von 20 Sekunden ein Eintrag über eine Minute. Daraus folgt die Mindestlänge einer Zeiterfassung von einer Minute.

[14] englisch *Coordinated Universal Time (CUT)*, französisch *Temps Universel Coordonné (TUC)*, Kompromiss *Universal Time Coordinated (UTC)*

Rechnungsgenerierung

Zeitraum- und Kundenselektion Da es bislang keine vergleichbare Anwendung zur Generierung von Rechnungen auf Zeitbasis gibt, war zu Beginn ungewiss, welche Informationen man neben den erfassten Zeitaufzeichnungen benötigt. Nach Rücksprache mit einigen Selbstständigen erwies sich, dass hauptsächlich zu Monatsanfang eine Rechnung für den vergangenen Monat erstellt wird. Daraus folgt, dass der Nutzer einen Zeitraum bestimmen muss, um abzuleiten welche bislang offenen Zeiterfassungen in die Rechnung aufgenommen werden sollen. Ebenfalls ergab sich, dass neben dem Zeitraum eine Kundenselektion notwendig ist, da oft nicht für alle, sondern für ausgewählte Kunden Rechnungen erstellt werden sollen.

Individuelle Rechnungspositionen Da sich neben den aus Zeiterfassung ergebenden Rechnungspositionen ebenso auch Individuelle einer Rechnung hinzufügen lassen, muss das InvoiceLineItem Model auf beide Fälle anwendbar zu sein. Deshalb ist der Wert für das Feld *label* bei zeitbezogenen Positionen der entsprechende Projektname und bei individuellen Positionen ein vom Nutzer gewählter Text. Dieser Kompromiss wurde getroffen, um kein zusätzliches Model für zeitbezogene Rechnungspositionen anlegen zu müssen, sondern beide Fälle mit einem Model abzudecken. Dies dient ebenfalls einer besseren Verständlichkeit des Datenmodells, zusätzlich oder gerade deswegen erleichtert sich die weitere Entwicklung.

Status von Zeitaufzeichnungen Eine Zeitaufzeichnungen befindet sich immer in einem der beiden Zustände offen oder in Rechnung gestellt, um zu markieren ob ein Eintrag bereits zur Rechnungsgenerierung genutzt wurde. Zunächst wurde dafür dem TimeEntry Model ein boolesches Feld *invoiced* gegeben. War dieses *false* wurde der entsprechende Eintrag in die Rechnungsgenerierung einbezogen und danach auf *true* gesetzt. Jedoch ergab sich später, dass dadurch rückwirkend Einträge von TimeEntry nicht mit deren Rechnung in Verbindung gebracht werden konnten.
Um das zu gewährleisten, wurde das Feld *invoiced* mit einem optionalen Fremdschlüssel *invoice_id* ersetzt.
Zeiterfassungen werden als offen angesehen und zur Rechnungsgenerierung genutzt, wenn keine *invoice_id* hinterlegt ist. Danach wird die ID der erstellen Rechnung allerdings dort hinterlegt, wodurch der Eintrag in den Zustand *in Rechnung gestellt* übergeht. Diese Lösung ermöglicht es, nach Löschen einer Rechnung alle umfassenden Zeitaufzeichnungen wieder zurück in den offenen Status zu überführen und daraus beispielsweise eine Rechnung über einen anderen Zeitraum zu erstellen.

Historie an Rechnungsdaten Ein weiteres Problem stellt das Ändern von vorher genutzten Datensätze dar, was durch das folgende Szenario beschrieben werden soll.

Ein Nutzer hat seine Arbeit für Kunde A in Rechnungen gestellt. Durch einen Umzug bekommt er eine neue Steuernummer zugeteilt. Am Jahresende lädt er aus steuerlichen Gründen alle über das Jahr verteilt monatlich erstellten Rechnungen nochmals herunter. Er erwartet auf den Rechnungen vor seinem Umzug seine alte Steuernummer, findet jedoch seine Neue.

Änderungen am Company Eintrag müssten historisch versioniert werden, um die Erwartungen des Nutzers im oben aufgeführten Szenario zu erfüllen. Dies ist aus zeitlichen Gründen vorerst nicht für die Anwendung vorgesehen und wird in dieser Arbeit daher auch vernachlässigt.

6.2 Webapplikation

6.2.1 React

Facebook Inc. stellte Mitte 2013 die Javascript Bibliothek React[15] vor. Im Vergleich zu anderen Javascript Frameworks wie AngularJS oder EmberJS ist sie keine Komplettlösung zur Implementierung einer clientseitigen Webapplikation. Im Sinne von MVC steht sie lediglich für das V, die View, da sie aussschließlich eine Bibliothek zur Umsetzung von Benutzeroberflächen ist.

Komponenten

React vereint Logiken und HTML-Markup in Form von Komponenten. Diese können weitere Komponenten beinhalten und damit, ähnlich der gewohnten HTML-Struktur, verschachtelt werden. Dabei können Informationen über beliebige Parameter an diese übergeben werden. Zusätzlich haben Komponenten ein *internen State*, der den eigenen Datenspeicher repräsentiert. Über die *render* Methode wird die Darstellung der Komponente zurückgeliefert. Diese wird jedesmal aufgerufen, wenn neue Parameter übergeben werden oder der interne State sich ändert. Dadurch bleiben Daten und Oberfläche immer synchronisiert.

[15] http://facebook.github.io/react/

Virtual DOM

Direkte DOM[16] Änderungen, wie man sie von anderen View Bibliotheken, wie beispielsweise jQuery, kennt, sind sehr teuer und vergleichsweise langsam. Deswegen bildet React ein Abbild des neuen DOM Baums, dem *virtual DOM*, mit allen geänderten Komponenten und vergleicht diesen mit einer Kopie des Aktuellen. Durch einen heuristischen Vergleichsalgorithmus wird festgestellt, welche Zweige sich tatsächlich geändert haben. Nur geänderte Zweige werden in den echten DOM Baum der Seite eingearbeitet.

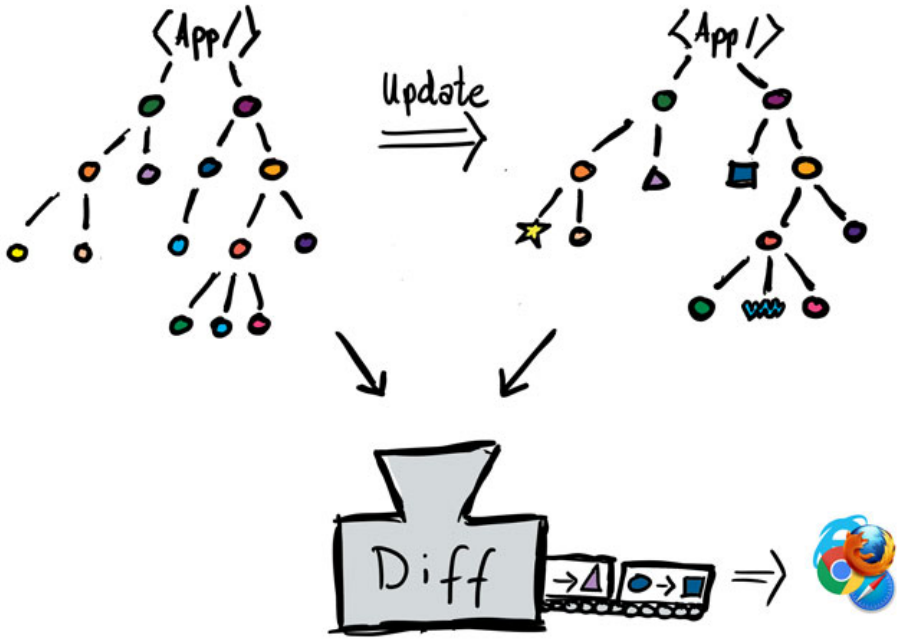

Abbildung 6.7: Illustration aus der react-vdom Dokumentation (react-vdom)

Dieses Verfahren, illustriert in Abbildung 6.7, ist eine äußerst performante Strategie zum Rendern von Oberflächen, die es React erlaubt bei jeder einzelnen Änderung ein komplettes Rendering auszuführen. Der Programmierer muss sich daher im Allgemeinen nicht um die Performanz der View kümmern und kann sich auf andere Bereiche der Anwendung fokusieren.

[16] steht für Document Object Model und ist ein Interface zum, im Browser dargestellten, HTML Dokument

6.2.2 Flux / Redux

Zur Steuerung des Datenflusses gibt Facebook zwar das Flux-Prinzip vor, allerdings ist dieses nicht zwingend umzusetzen, weshalb viele verschiedene Implementierungen dieser Architektur existieren. Zunächst wird auf die von Facebook selbst vorgestellte Flux Variante eingegangen. Daraufhin folgt ein Vergleich zur, in dieser Webapplikation genutzten, Variante *Redux* und deren Vorteile.

Flux

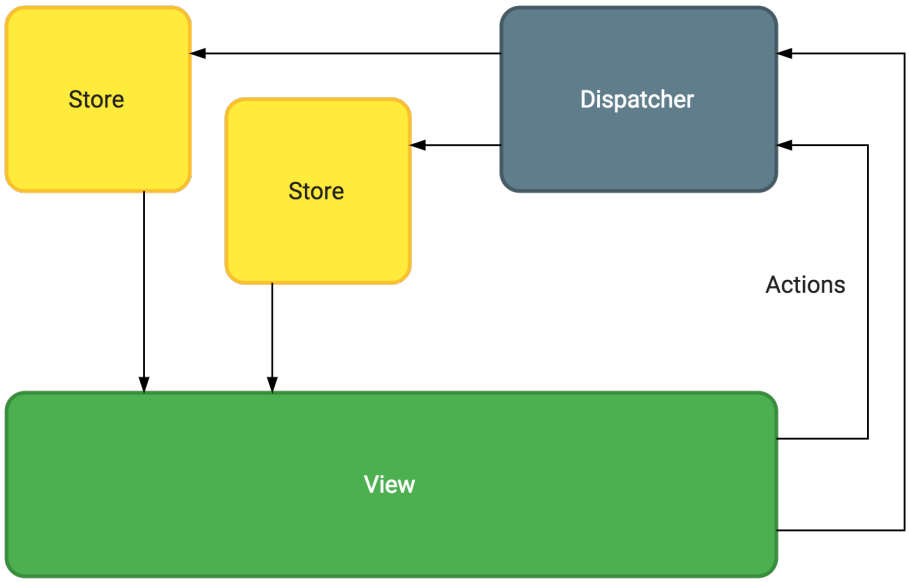

Abbildung 6.8: Datenfluss Flux

Flux beschreibt einen uni-direktionalen Datenfluss. Die Informationen einer Anwendung sind in *Stores* gespeichert. Views lauschen auf Änderungen innerhalb eines Stores und können bei Bedarf ihren internen State aktualisieren, was gleichzeitig zu einem neuen Rendering der Komponente führt. Damit sich ein Store ändert, können über *ActionCreator* verschiedene *Actions* von einer View erzeugt werden, welche über einen *Dispatcher* an den entsprechenden Store weitergereicht werden. Eine Action besteht aus einem Namen und optionalen Daten. Im Store selbst ist hinterlegt, wie mit einer Action umgegangen werden soll.
Als Beispiel müsste eine Action mit dem Namen ERSTELLE_KUNDE die Informationen über den neuen Kunden beinhalten, worauf der Store diese dann der Liste aller Kunden hinzufügen und der View die Änderung mitteilen kann.
Zu Beachten ist, dass eine auf Flux basierte Anwendung auf mehrere Stores aufgeteilt ist, wodurch sich übergreifende Logiken nur schwer teilen lassen und oft redundant realisiert werden müssen.

Redux

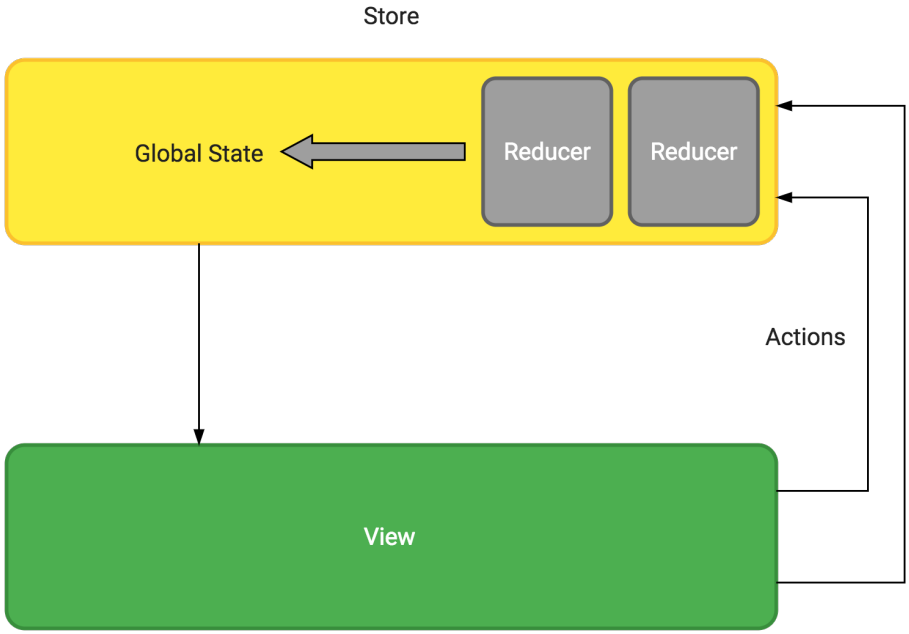

Abbildung 6.9: Datenfluss Redux

Dagegen besagt Redux, dass nur ein einziger Store existieren soll, dessen Informationen durch den *Global State* repräsentiert werden. Actions sind identisch zu Flux implementiert, jedoch werden sie direkt an den Store zu den *Reducern* übergeben. Ein Reducer ist eine Funktion, welcher der aktuelle Global State und die aufgerufene Action übergeben werden. Diese "reduziert" den übergebenen State und erzeugt aus den Änderungen einen Neuen, welcher dem nächsten Reducer übergeben wird. Sind alle Reducer durchlaufen, wird der zuletzt erzeugte State als neuer Global State gesetzt.
Ein weiterer Unterschied ist, dass Komponenten nicht direkt auf Änderungen des Stores lauschen. Sie sind alle entweder direkt oder transitiv der Elternkomponente *Provider* untergeordnet. Dieser werden Änderungen des Stores mitgeteilt, welche sie dann über Parameter an alle Unterkomponenten weitergibt.

Das globale Behandeln des Zustands der Anwendung vereinfacht und ermöglicht die Entwicklung von Funktionalitäten, die im Kontext der gesamten Anwendung stehen. Beispielsweise ist in dieser Anwendung ein sich über die komplette Webapplikation erstreckendes Benachrichtigungssystem für den Nutzer ohne großen Aufwand durch einen *NotificationReducer* implementiert.
Ein weiteres Beispiel ist das Entwicklerwerkzeug *Redux Devtools*. Dieses hängt sich in den zentralen Datenfluss und kann somit alle aufgerufenen Actions kontrollieren.

Redux Devtools ist eine Browsererweiterung für Chrome[17]. Sie zeigt eine Historie über alle aufgerufenen Actions und dokumentiert alle dadurch hervorgerufenen Änderungen am Global State. Außerdem ist es möglich innerhalb dieser Historie vorbeziehungsweise zurückzuspulen, wodurch eine Fehlersuche erleichtert werden kann. Darüber hinaus ermöglicht Redux *Hot reloading* einzelner Komponenten. Hot reloading wird das Integrieren von Codeänderunguen einzelner Softwarekomponenten genannt, bei dem die Anwendung nicht gestoppt werden muss, sondern an genau der gleichen Stelle mit gleichbleibenden Speicherstand weiter ausgeführt wird.

6.2.3 Entwicklungshilfen

Für die Entwicklung einer moderne Webapplikation sind einige Tasks und Präprozessoren notwendig. Daher soll hier auf den Aufbau der Entwicklungsumgebung eingegangen werden. Auf eine spezifische Nennung des Texteditor wird verzichtet.

Webpack

[http://webpack.github.io/assets/what-is-webpack.png] Genutzte Bibliotheken und Abhängigkeiten werden alle über das *NodeJS*[18] Paketsystem *NPM* bezogen. Gegenüber anderen Sprachen wie Java oder Ruby ist es für im Browser ausgeführtes Javascript nicht möglich, Bibliotheken bei Laufzeit und Bedarf zu integrieren. Daher ist es immernoch praktikabel Bibliotheken und den Code der eigentlichen Anwendung zu einer einzelnen Javascript Datei zusammen zu fassen. Zusätzlich werden hierdurch HTTP Anfragen gespart, die normalerweise hintereinander ausgeführt werden. Lädt man jede einzelne Bibliothek jeweils durch eine eigene HTTP Anfrage herunter, würde der Seitenaufbau sich stark verzögern.
Da nach Zusammenfügen des gesamten Codes zu einer einzelnen Datei jegliche Strukturen und Übersicht der Anwendung verloren gehen, wird für die Entwicklungsphase auf das Modulverwaltungssystem *Webpack* zurückgegriffen. Dieses ermöglicht es, jede Komponente der Anwendung in eigene Dateien auszulagern, wodurch sich eine in Komponenten unterteilte Ordnerstruktur anlegen lässt. Webpack läuft dabei parallel als Kommandozeilenprogramm und ist in mehrere *Tasks* aufgeteilt.

[17] https://github.com/zalmoxisus/redux-devtools-extension
[18] serverseitige Javascript Umgebung

Build

Sobald Webpack gestartet ist, werden alle Dateien der Anwendung auf Änderungen beobachtet. Bei jeder Änderung wird der Build Task ausgeführt. Dieser wandelt alle Stylinganweisungen von *SASS*[19] in CSS um, ebenfalls werden Bilder für eine bessere Verwendung im Web komprimiert.
Der Javascript Code wird zunächst über ESLint[20] auf Syntaxfehler überprüft und bei Richtigkeit weiterbehandelt. Die Anwendung ist unter Berücksichtigung von *ECMAScript 6* geschrieben, einer weiterentwickelten Spezifikation von Javascript. Diese wird jedoch noch nicht vollständig in modernen Browsern unterstützt, weswegen der komplette Code über das Modul *Babel* in browserverständliches Javascript konvertiert wird. Im Entwicklungsmodus werden zusätzlich noch *Sourcemaps* für Javascript und SASS generiert, die die Fehlersuche erleichtern können. Für einen Build im Produktionsmodus werden alle generierten Dateien noch komprimiert.

Entwicklungsmodus

Führt man Webpack im Entwicklungsmodus aus, startet sich nebenher ein lokaler Webserver. Dieser stellt die Anwendung unter der Adresse `http://localhost:4000` bereit. Ebenfalls startet sich das oben bereits beschriebene *File watching*, welches bei Änderungen am Code jedes Mal den Build-Task startet und diese über Hot reloading integriert.

Deploy

Führt man den Build-Task im Produktionsmodus aus, werden die erstellten Dateien nicht über einen lokalen Webserver bereitgestellt, sondern im Ordner */dist* hinterlegt. Über den Deploy-Task wird dieser Ordner über *Rsync* mit dem Server synchronisiert.

[19] einer syntaktischen Erweitung von CSS
[20] `http://www.eslint.org`

6.2.4 Aufbau

Nachdem die genutzte theoretische Architektur, sowie die Entwicklungsumgebung bekannt sind, wird in diesem Kapitel der konkrete Aufbau der Webapplikation behandelt.

Containers

Als *Container* werden Komponenten bezeichnet, die in direkter Verbindung mit der genannten Provider-Komponente und somit auch dem Store stehen. Sie beschreiben einen ganzen Funktionsbereich und beinhalten dessen Unterkomponenten. Grundsätzlich lassen sich Container in zwei Gruppen aufteilen.
Die *Site-Container* sind alle Container, die sich auf die eigentliche Website der Anwendung beziehen. Diese umfassen inhaltliche Seiten, wie Startseite, Produkterklärungen und Impressum. Aber auch die Seiten zur Anmeldung und Registrierung gehören zu den Site-Containern.
Dem gegenüber stehen die App-Container, die sich auf anwendungsspezifische Inhalte beziehen und für authentifizierte Nutzer erreichbar sind. Darunter fallen die Container zur Zeiterfassung und zur Verwaltung von Projekten, Kunden, Rechnungen und Unternehmensdaten.

Components

Components sind einzelne wiederverwendbare Bausteine, die die eigentliche Funktionalitäten eines Bereiches implementieren. Sie rufen keine Actions direkt auf, sondern von der Elternkomponente mitgegebene *Callbackfunctions*. Beispielsweise liegt eine Form-Komponente im Company-Container, der die Unternehmensdaten verwaltet. Der Container übergibt an die Form-Komponente als *onSubmit*-Parameter eine Funktion, die bei Aufruf die Action zum Speichern der Unternehmensdaten ausführt. Nun kann die Form-Komponente entscheiden wann und wodurch diese onSubmit-Funktion aufgerufen wird, sinnvollerweise beim Bestätigen der Enter-Taste oder des Speichern-Buttons.
Dazu hat sich bei der Entwicklung die Aufteilung in Listing- und Item-Komponenten bewährt. So gibt es im Project-Container, der für die Projektverwaltung verantwortlich ist, eine Auflistung aller existierenden Projekte über die ProjectListing-Komponente. Diese erwartet ein Array aller Projekte als Parameter vom Container übergeben zu bekommen. Daraufhin instanziert sie für jedes Element dieses Arrays eine ProjectListingItem-Komponente und gibt diese über ihre Render-Methode aus. Nach diesem, in Abbildung 6.10 dargestellten, Schema werden Kunden, Zeiteinträge und Rechnungen ebenfalls strukturiert. Dazu akzeptiert die ProjectListing-Komponente einen advanced-Parameter, welcher die Komplexität der Auflistung definiert. Im Zeiterfassungscontainer wird nämlich ein ProjectListing in reduzierter Form benötigt. Dort sind Funktionalitäten zur Zuweisung eines Kunden zum Projekt beispielsweise nicht notwendig.

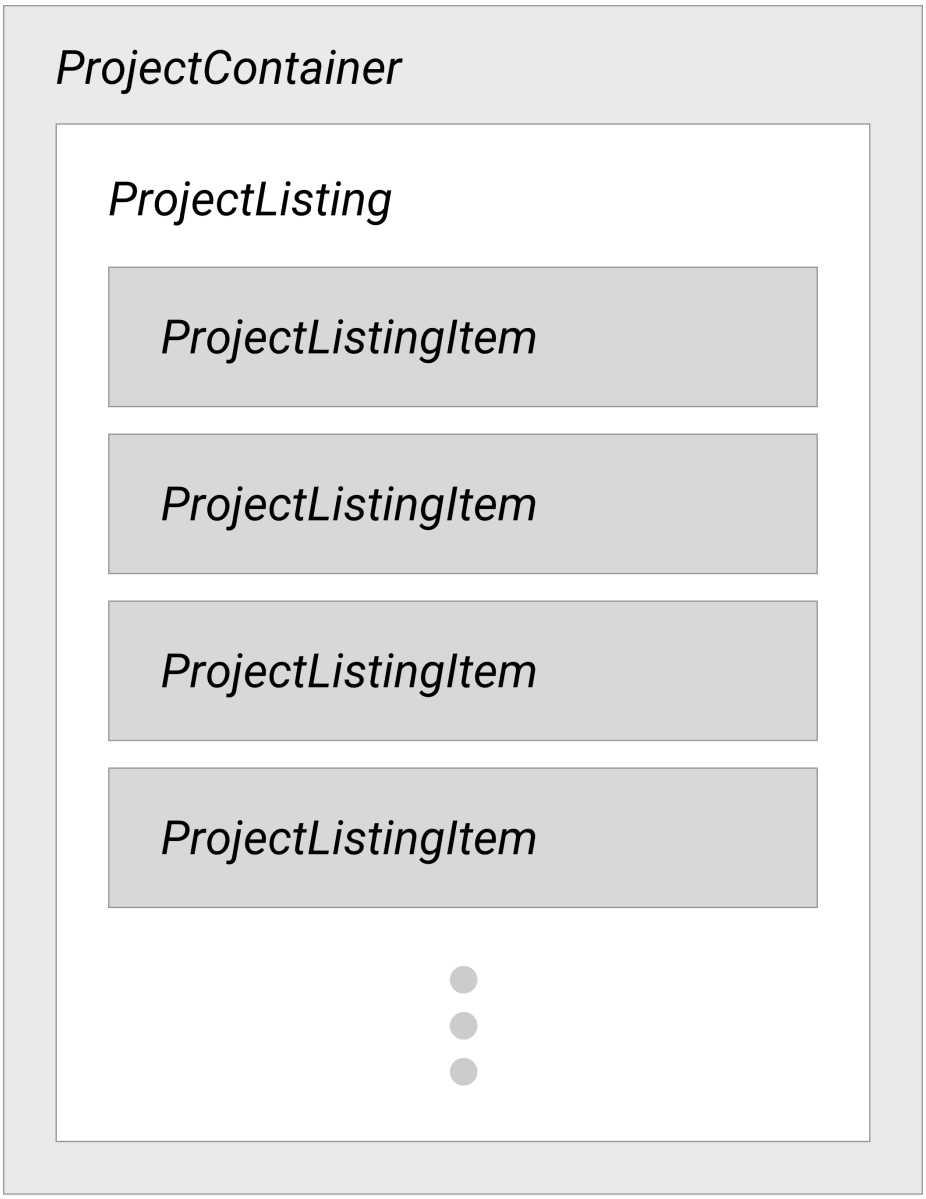

Abbildung 6.10: Schachtelung der Komponenten des Project-Containers

Actions

Grundlegend kann man zwischen zwei Arten von Actions unterscheiden. Synchrone Actions werden direkt ausgeführt und beziehen sich auf den clientseitigen Informationskontext der Anwendung. Ein Beispiel hierfür ist die Selektierung des Projekts für das Zeit erfasst werden soll. Dabei wird beim Klick auf ein Projekt die Action *SELECT_PROJECT* mit der ID des Projekts aufgerufen, woraufhin das Projekt im Global State als selektiert markiert wird. Beim Klick auf den Startbutton zur Zeiterfassung wird die ID dieses vorher selektierten Projekts gewählt und innerhalb einer asynchronen Action *POST_TIME_ENTRY* aufgerufen.
Asynchrone Actions werden genutzt, um mit der API zu kommunizieren und somit Informationen permanent zu ändern. Dabei umfassen sie jeweils drei einzelne synchrone Actions. Exemplarisch wird das an der *POST_TIME_ENTRY* Action verdeutlicht. Beim Aufruf der asynchronen Action wird direkt eine synchrone Action *POST_TIME_ENTRY_REQUEST* ausgeführt. Durch diese kann die Anwendung in einen Ladezustand gebracht werden und dem Nutzer somit visualisieren, dass ein Prozess im Gange ist, auf den derzeit gewartet wird. Gleichzeit wird im Hintergrund der eigentliche asynchrone Prozess, das Senden der Daten an die API, ausgeführt. Ist dieser erfolgreich abgeschlossen, folgt automatisch der Aufruf der Action *POST_TIME_ENTRY_OK*. Bei einem Fehler des asynchronen Prozesses wird *POST_TIME_ENTRY_ERROR* aufgerufen.
Bei letzteren beiden Actions kann der Ladezustand wieder aufgehoben werden und mit den entsprechenden Antwortdaten der API weiterverfahren werden.

Reducer

Genanntes Weiterverfahren einer Action wird von den hintereinander geschalteten Reducern durchgeführt. So ist im Reducer fest gelegt, dass bei einer *ACTION_NAME_REQUEST* Action ein *isLoading* Status im Global State auf *true* gesetzt wird, beziehungsweise später wieder aufgehoben wird. Zur besseren Übersicht sollen Reducer sehr schlank gehalten werden und lediglich neue Werte des Global States setzen. Die Logiken zur Berechnung dieser Werte sind thematisch in diverse *Helper* ausgelagert.

Helper

Alle Hilfsmethoden zum Umgang mit den Anwendungsdaten sind gruppiert in einzelne Helper-Dateien. Diese Methoden kümmern sich ausschließlich um Berechnungen oder Umstrukturierungen und haben keinerlei Zugriff auf den Anwendungskontext. Daher bekommen sie nur Daten übergeben und liefert neue Daten zurück, halten und persistieren somit keinerlei Daten selbst.

CalendarHelper Formatierungen von Zeitangaben erstrecken sich über die gesamte Applikation. Daher wurden diese in den CalendarHelper ausgelagert. Dieser beinhaltet die Methode zur Umwandlung von Sekunden in ein von Stoppuhren bekanntes Format *hh:mm:ss*, wobei hh für Stunden, mm für Minuten und ss für Sekunden stehen. Dieses Format wird hauptsächlich in der Zeiterfassungsleiste zur Darstellung einer laufenden Zeiterfassung genutzt.
Daneben lassen sich Minuten auch in das Format *hh:mm* umwandeln, welches die Dauer von erfassten Zeiteinträgen lesbar wiedergibt.
Entsprechend seinem Namen beinhaltet der CalendarHelper aber hauptsächlich Methoden zum Erzeugen eines Datensatzes für einen Monat, der von der *Calender*-Komponente genutzt wird. Der *createCalendar* Methode werden Jahr, Monat, Zeiteinträge und ein Offset zum Definieren des Wochenbeginns, welcher international unterschiedlich gehandhabt wird, übergeben. Damit werden zunächst der erste Tag der ersten Woche, sowie der letzte Tag der letzten Woche des Monats festgelegt. Auf dieser Basis werden die Tage iteriert und in Wochen gefasst. Dabei wird einem Tag angehängt, ob dieser Zeiterfassungen beinhaltet, ob er der aktuelle Tag ist und ob er außerhalb des anzuzeigenden Monats liegt. Letzteres beschreibt Tage des vorangegangenen Monats innerhalb der ersten Woche oder Tage des folgenden Monats innerhalb der letzten Woche.

TreeHelper Von der API werden Projekte als Array an die Webapplikation übergehen. Jedes Projekt beinhaltet dabei eine *parent_id*, welche der ID des Elternprojekts entspricht. Um diese Projekte vom Array in eine Baumstruktur zu verwandeln, wie sie im *ProjectListing* Verwendung findet, wird die Hilfsmethode *unflattenEntities* des TreeHelpers genutzt.
Diese wird für jedes Projekt aufgerufen, welches keine *parent_id* besitzt. Neben dem Projekt selbst werden auch alle weiteren Projekte an die Methode übergeben.
Zuerst wird die *parent_id* jedes einzelnen Projekts mit der ID des übergebenen, potentiellen Elternprojekts verglichen und bei Übereinstimmung in Form eines *children* Arrays diesem zugeordnet. Nun ruft sich die Methode selbst für jedes einzelne Projekt des *children* Arrays als potentielles Elternelement auf und ordnet die dabei erkannten *children* diesem zu.
Daraus entsteht rekursiv ein Projektbaum, der, sobald alle Unterprojekte durchlaufen sind, zurückgeliefert wird.

TimeEntryHelper Zeiterfassungen werden immer im Kontext des ausgewählten Tages angezeigt. Es ergibt sich somit nur ein Zeitfenster von 0 bis 24 Uhr für die Visualisierung. Daher entsteht eine Überlappung von Zeiteinträge, die sich über Mitternacht auf den Folgetag beziehungsweise über mehrere Tage erstrecken. Um diese darzustellen werden überlappende Zeiterfassungen direkt nach Empfangen von der API im Reducer aufgeteilt. Für jeden Tag über den sich ein Eintrag erstreckt, wird ein eigener Eintrag angelegt. So ergeben sich aus einer Erfassung von 22 Uhr abends bis

2 Uhr nachts zwei Einträge. Der Erste umfasst als Startzeit 22 und als Endzeit 24 Uhr. Der Zweite beginnt um 0 und endet um 2 Uhr. Öffnet der Nutzer allerdings das Formular einer Zeiterfassung zur Bearbeitung, wird ihm die eigentliche Start- beziehungsweise Endzeit angezeigt. Die genannte Aufteilung wird nämlich nur intern von der Applikation zur Visualisierung genutzt. Zum Umgang mit den Informationen einer Zeiterfassung wird dieser weiterhin als einzelner Eintrag behandelt.
Der Reducer bedient sich zum Behandeln der Überlappungen den Hilfsmethoden des TimeEntryHelpers. Diese berechnen über die Dauer einer Erfassung, ob eine Aufteilung notwendig ist und setzen entsprechend gesonderte Start- und Endzeiten.

6.2.5 Offline-Funktionalität

Zum Bereitstellen der wichtigsten Funktionalitäten im Betrieb ohne aktive Internetverbindung sind diverse Mechanismen notwendig. Der Austausch der Informationen zwischen Webapplikation und API muss in dieser Phase überbrückt werden. Offline-Funktionalitäten sind nur verfügbar, wenn der Nutzer davor bereits authentifiziert war. Da ein hinterlegter Access-Token ohne Verbindung zur API nicht validiert werden kann, wird im Offline-Betrieb von dessen Gültigkeit ausgegangen.

Service Workers

2015 wurden *Service Worker* vorgestellt, die derzeit nur in aktuellen Versionen des Chrome und Firefox Browsers verfügbar sind. Diese können von einer Webapplikation registriert werden. Von da an laufen sie asynchron und können Aufgaben unabhängig von der eigentlichen Applikation lösen. Service Worker sind hauptsächlich zur Implementierung von individuellen Caching Strategien erschafft worden. Obwohl die Spezifikation von Service Workern aktuell noch nicht abgeschlossen ist, sind schon einige Funktionalitäten in neuen Browsern integriert worden.

Request Cache

Derzeit lassen sich schon URL-Pfade im Cache hinterlegen. Diese umfassen in dieser Anwendung die Pfade */index.html*, */styles.css* und */bundle.js*. Damit lässt sich schon die gesamte Webapplikation offline im Browser öffnen.

FetchEvent

Daneben werden mehrere Events bereitgestellt, für die entsprechende *EventListener* registriert werden können. Eines davon ist das *FetchEvent*. Dieses wird bei ausgehenden Anfragen gefeuert, wodurch man diese abfangen und manipulieren kann. Allerdings gilt das aktuell nur für GET-Anfragen.

SyncEvent

Das *SyncEvent* wird gefeuert, sobald der Browser wieder über eine aktive Internetverbindung verfügt. Damit lassen sich zum Beispiel im Offline-Betrieb angefallene Daten mit der API synchronisieren. Service Worker laufen, auch wenn die zugehörige Applikation geschlossen wurde, im Hintergrund weiter. Das SyncEvent wird daher auch bei geschlossener Applikation gefeuert und die entsprechenden Daten werden synchronisiert.

Im weiteren Teil dieses Kapitels wird die angedachte Offline-Strategie visualisiert und näher erläutert. Dafür wird ein Service Worker auf Clientseite registriert.

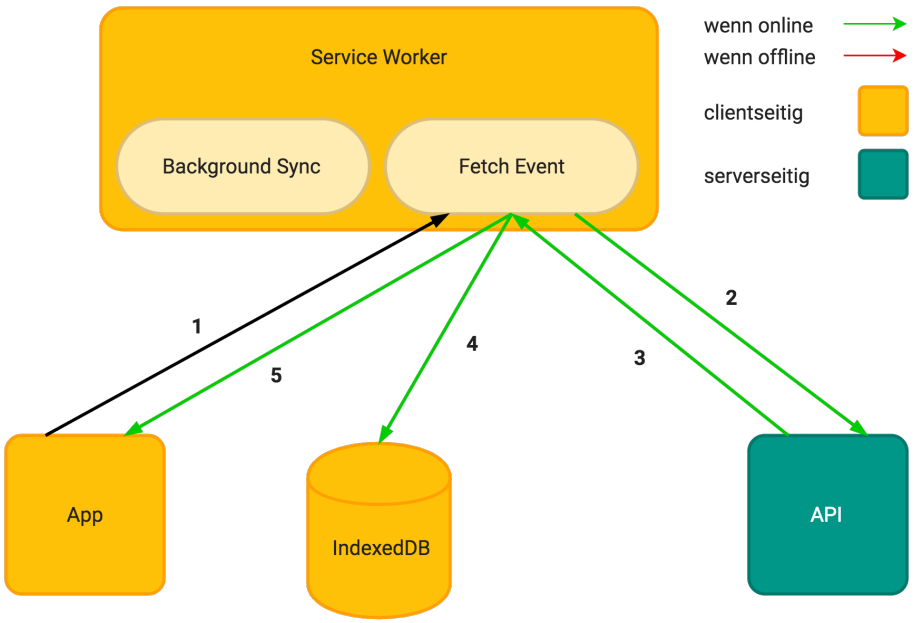

Abbildung 6.11: Datenfluss GET-Anfragen im Online Modus

Bei GET-Anfragen an die API hängt sich der Service Worker durch das FetchEvent dazwischen und prüft den Verbindungsstatus des Browsers. Im Online-Modus wird die GET-Anfrage weitergeleitet an die API. Die empfangenen Daten werden dann, bevor sie an die Anwendung zurückgegeben werden, zusammen mit der Anfrage-Adresse in der indexbasierten Browserdatenbank *IndexedDB* gespeichert. Diese Datenbank dient als Cache für die API-Daten.

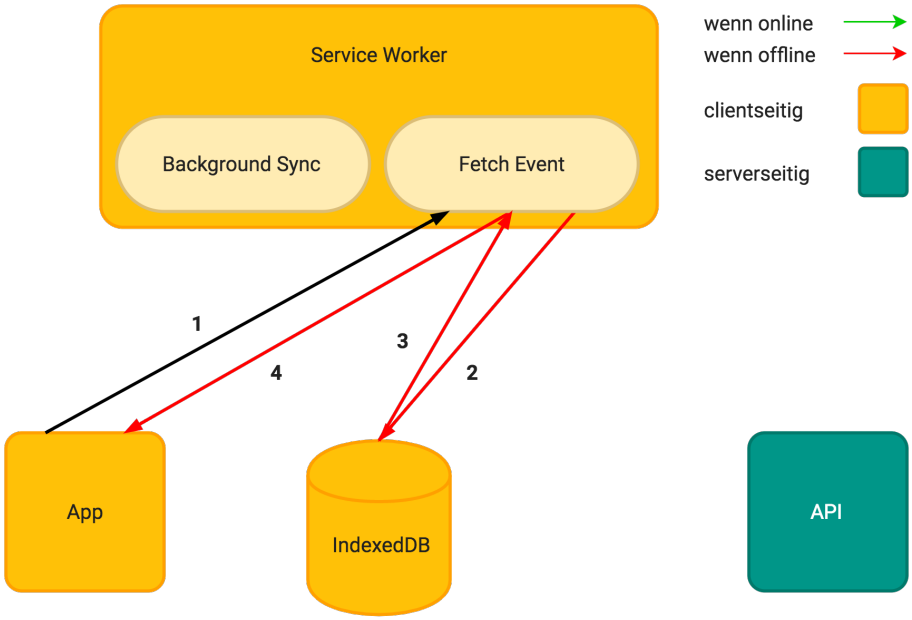

Abbildung 6.12: Datenfluss GET-Anfragen im Offline Modus

Befindet sich der Browser jedoch im Offline-Modus, liefert der Service Worker die Daten der IndexedDB an die Applikation zurück.

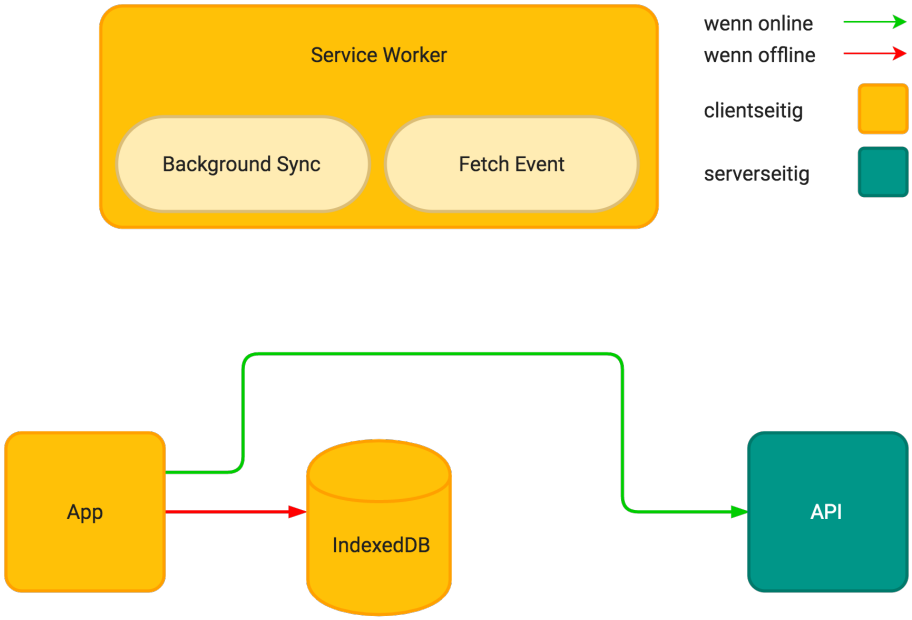

Abbildung 6.13: Datenfluss POST-Anfragen

POST-, PUT- und DELETE-Anfragen verhalten sich dabei anders, da Service Worker noch keine Möglichkeit besitzen, mit diesen umzugehen. Im Online-Modus werden sie direkt an die API gesendet. Im Offline-Modus werden die geänderten, gelöschten oder erstellten Datensätze mit den Informationen *isChanged*, *isDeleted* und *isNew* versehen und in der IndexedDB direkt gespeichert.
Bei Neuaufruf der Applikation im Offline-Modus sind alle ausstehenden Datenänderungen durch den Fallback auf die IndexedDB weiterhin verfügbar.

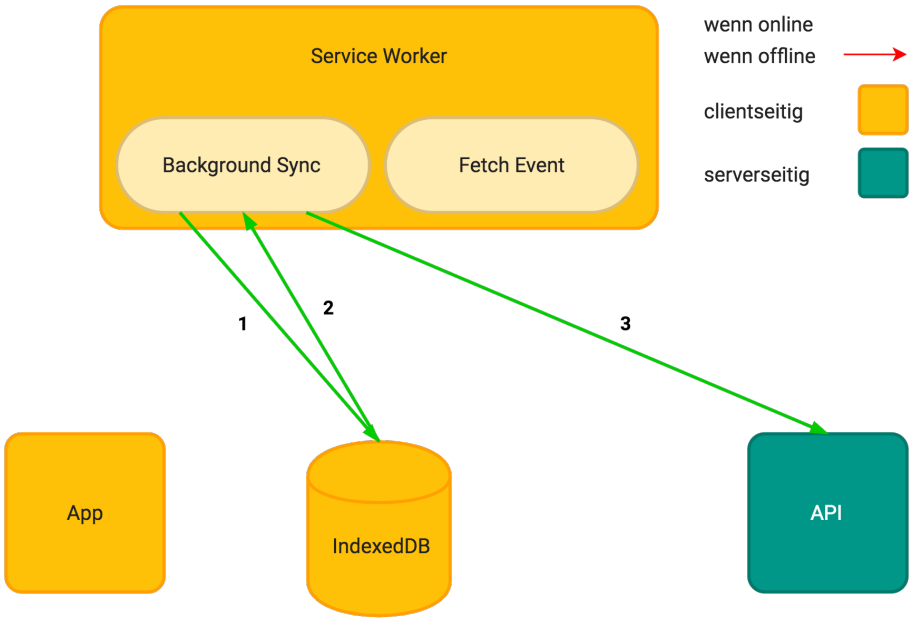

Abbildung 6.14: Datenfluss bei Wechsel in den Online-Modus

Sobald der Browser in den Online-Modus wechselt, werden durch das SyncEvent diese ausstehenden Datenänderungen einzeln über entsprechende POST-, PUT- oder DELETE-Anfragen an die API gesendet. Schließlich sind die Daten der Webapplikation und der API wieder synchronisiert.

6.3 Testing

Auf Serverseite sind Teile der API mit Tests abgedeckt. Der Fokus lag dabei auf den wichtigsten Aspekten der Anwendung. Rails bietet ein komplettes Testing Framework, welches Kommandozeilenanwendungen zum Ausführen von Tests beinhaltet. Diese erzeugen eine spezielle Test Datenbank, die mit *Fixtures*, vorher händisch definierten Testdaten, vor jedem Testlauf gefüllt wird. Darüberhinaus bietet Rails zwei Testklassen, deren Nutzung im Folgenden beschrieben wird.

6.3.1 Unit Tests

Um ein geplantes Verhalten von Modelklassen zu gewährleisten, wird für jedes Model ein *Unit Test* erstellt. In diesem können einzelne Testfälle definiert werden.
Für TimeEntry Instanzen ist es wichtig, dass die Startzeit immer vor der Endzeit liegt. Daher wurde hierfür ein Testfall geschrieben, welcher dies für die hinterlegten Fixtures gegenprüft.
Ein anderer Unit Test ist beispielsweise für das Prüfen von Relationen zwischen Projekten und ihren Elternprojekten erstellt worden.

6.3.2 Integration Tests

Etwas umfangreicher sind *Integration Tests*. Diese beziehen sich meist auf Controller und testen daher nicht nur eine einzelne Funktion, sondern ein ganzes System. Beispielsweise könnte ein Integration Test das Authentifizieren eines Nutzers mit darauf folgendem Anlegen eines Projektes und dem Erstellen eines TimeEntrys umfassen.
In der API sind hauptsächlich reine CRUD-Controller hinterlegt, für die jeweils ein Integration Test hinterlegt ist, der den Aufruf des Create-, Read-, Update- und Delete-Endpunktes testet.

Um zu gewährleisten, dass eine Codeänderung keine unvorhersehbaren, negativen Seiteneffekte auslöst, wird ein *Deploy*[21] erst durchgeführt, wenn alle Tests erfolgreich durchlaufen sind.

[21] Veröffentlichung des Codes auf die Produktionsumgebung

6.4 Continuous Integration

Als *Continuous Integration* (CI) wird der regelmäßig durchlaufende Zyklus von Entwicklung bis zur Integration in die Live-Anwendung genannt.

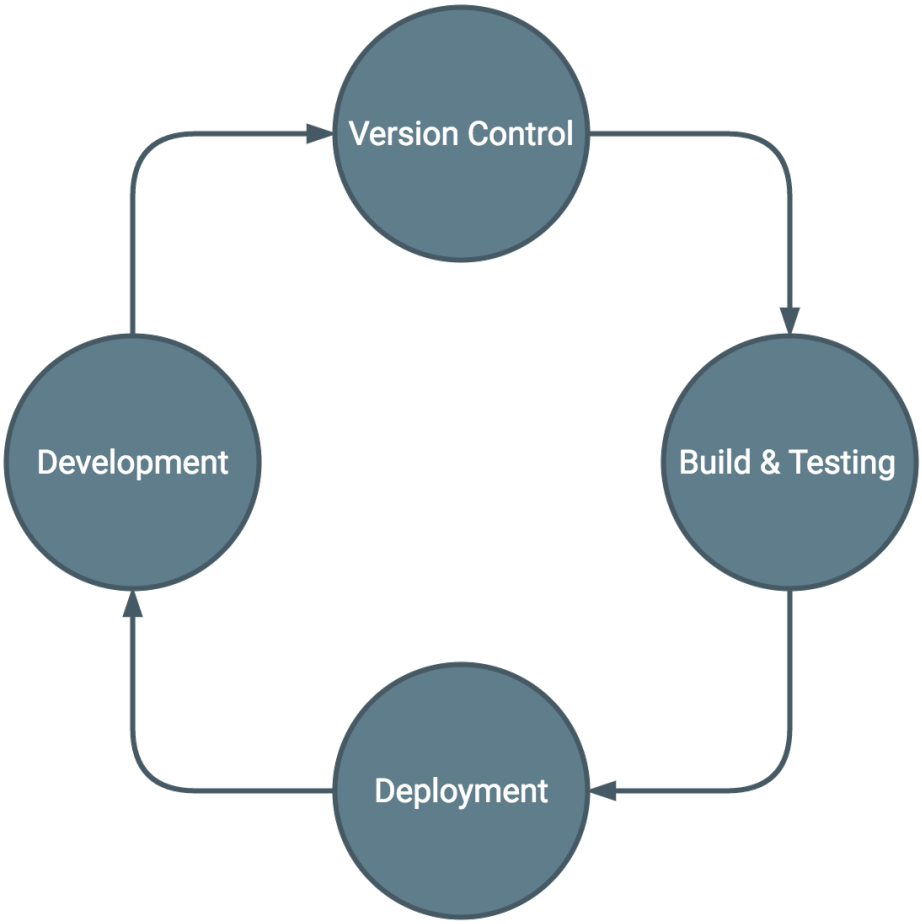

Abbildung 6.15: Fluss eines CI getriebenen Entwicklungsprozesses

Dabei wird nach dem Commit eines Entwicklers vom *Version Control System* ein dort konfigurierter Webhook aufgerufen. Dieser geht an Travis CI[22], einem CI-Tool zum Erstellen eines Builds. Dort werden alle Tests mit dem erzeugten Build ausgeführt. Treten hierbei Fehler auf, wird der Verfasser des Commits per E-Mail benachrichtigt. Nach erfolgreichem Abschluss aller Tests wird die neue Version automatisch auf dem Produktionsserver in die Anwendung integriert.

[22] http://www.travis-ci.com

6.4.1 Vorteile

Oft werden mehrere neue Funktionalitäten in Meilensteine zusammengefasst und nur gemeinsam als ganzer Meilenstein integriert. Das kann dazu führen, dass neue Funktionen erst nach einer sehr langen Zeitspanne im Livebetrieb getestet werden können. Fehler und Probleme können sich bis zur Integration häufen, werden oft aber dann erst erkannt.
Dem möchte Continuous Integration entgegenwirken. Durch den kurzen Zyklus von der Entwicklung bis zur Inbetriebnahme einer neuen Funktionalität, lassen sich Probleme schnell identifizieren. Ebenfalls werden Entwickler durch ständiges Testing früh auf entstehende Fehler hingewiesen.

6.4.2 Deployment

Für das *Deployment* wird das Gem *Capistrano*[23] verwendet, welches mit den Daten des Produktionsservers konfiguriert ist. Beim Deploy transferiert es via *Secure Copy Protocol* (SCP) den Code des aktuellen Commits in einen neuen, nach dem Commithash benannten Ordner auf den Server. Nach erfolgreicher Übertragung wird der dort laufende Webserver auf die neue Version umgestellt.
Auf dem Produktionsserver sind gleichzeitig immer die letzten fünf Versionen verfügbar. Tritt ein Fehler in einer neuen Version auf, kann einfach und schnell durch einen *Rollback* auf eine der vorherigen Versionen zurückgestellt werden, wodurch ein zeitkritischer *Hotfix* nicht zwingend erforderlich ist.

[23] https://github.com/capistrano/capistrano

7 Ausblick

Mit der bisher beschriebenen Umsetzung ist eine gute und funktionierende Basis für die Anwendung geschaffen worden. Allerdings kann man den bisherigen Stand eher als *Proof-of-Concept* verstehen. Ziel ist es ein vollständiges *Software-as-a-Service* Produkt zu erstellen. Die hierfür zu entwickelnden, notwendigen Merkmale werden im vorliegenden Kapitel genannt, jedoch in der vorliegenden Arbeit nicht mehr genau betrachtet.

7.1 Versionierung inhaltlicher Änderungen

Damit auch nach späterer inhaltlicher Änderung von Datensätzen die bis dahin erstellten Rechnungen unbeeinflusst bleiben, ist eine Historie beziehungsweise Versionierung notwendig. Um diese umzusetzen, könnte man eine dokumentenorientierte NoSQL-Datenbank anbinden und den aktuellen Datensatz vor einer Änderung als Revision dort persistieren.

7.2 Internationalisierung

Für die Anwendung ist eine internationale Verwendung vorgesehen. Bisher wurde dies bei der Umsetzung von Währung, Steuersatz, Adressen und Zahlungsadressen berücksichtigt. Darüber hinaus müssen neben der Möglichkeit zur Wahl der Sprache auch inhaltliche Übersetzungen in die Webapplikation integriert werden.
Darauf aufbauend muss das Rechnungstemplate zur Generierung der PDFs die gewählte Sprache ebenfalls berücksichtigen.

7.3 Corporate Page

Zur Veröffentlichung der Anwendung ist eine Produktpräsentation beziehungsweise -beschreibung in Form einer Webpräsenz sinnvoll. Die entsprechenden Container für eine Corporate Page sind in der Webapplikation bereits hinterlegt, jedoch besteht für diese weder ein Design noch ein Konzept.

7.4 Zahlungsanbindung

Der nächste Schritt nach der Realisierung einer Corporate Page und der Entwicklung eines vernünftigen Geschäftskonzepts ist die Umsetzung des vorgesehenen Zahlungsmodells. Eine gute Lösung hierfür ist die Anbindung eines Zahlungsanbieters wie FastSpring[1]. FastSpring übernimmt den gesamten Zahlungsprozess mit etablierten Zahlungsmethoden wie PayPal[2], Kreditkarte oder SOFORTüberweisung[3]. Über die FastSpring-API kann die Anwendung den Bezahlstatus eines Users abfragen und den entsprechenden Zugang zur Webapplikation freigeben oder zur Zahlung auffordern.

[1] http://www.fastspring.com/
[2] http://www.paypal.com
[3] http://www.sofort.com

8 Fazit

Es erscheinen immer mehr Tools, um aufwändige Buchhaltungsprozesse von Selbstständigen zu vereinfachen. Jedoch berücksichtigen diese selten die Ansprüche von zeitbasiert arbeitenden Solo-Selbstständigen. Gleichzeitig existieren unzählige Anwendungen zur Zeiterfassung, welche mit einer digitalen Stechuhr vergleichbar sind. Diese verfügen allerdings nicht über Funktionalitäten zur Unterstützung der Buchhaltung.
Eine Lösung, die die Alltagsprozesse eines zeitbasiert arbeitenden Solo-Selbstständigen deutlich erleichtern, ist das Zusammenführen von Zeiterfassung und Rechnungserstellung. Aus diesem Ansatz entstand im Rahmen der vorliegenden Arbeit eine Anwendung, die neben der genannten Zeiterfassung, der Verwaltung von Projekten, Kunden und Unternehmensdaten auch eine Rechnungsgenerierung auf Basis der gesammelten Daten umfasst.
Dabei stellten sich einige besondere Schwierigkeiten heraus. Zum einen ereignete sich die Internationalisierung durch länderspezifische Währungen, Adressen, Rechnungsformate und Steuerformalien umfangreicher als ursprünglich angenommen. Zum anderen war ein sehr sauberer und konsistenter Umgang mit Zeitwerten zur Synchronisation zwischen Client und Server von Nöten. Dazu war auch die Unterbringung der einzelnen Komponenten in ein geeignetes, intuitiv bedienbares Layout eine Herausforderung.
Letztlich ist jedoch eine Anwendung entstanden, die uneingeschränkt nutzbar ist und im täglichen Arbeitsleben ohne Bedenken eingesetzt werden kann.
Zukünftige Meilensteine sind unter Anderem die iterative Verbesserung der Benutzeroberfläche, sowie der Erweiterung der Internationalisierung. Wünschenswert ist, dass sich das Projekt zu gegebener Zeit finanziell selbst tragen kann. Dafür sind die Ausarbeitung eines Geschäftsmodells, sowie die technische Anbindung an einen Zahlungsanbieter notwendig.
Zusammengefasst kann gesagt werden, dass die hier beschriebene Anwendung ein positives Ergebnis darstellt, mit dem die Ansprüche von zeitbasiert arbeitenden Solo-Selbstständigen gut bedient werden.

Abbildungsverzeichnis

Literaturverzeichnis

[Ruby on Rails Guides] Ruby on Rails Guides *Active Record Associations*
http://guides.rubyonrails.org/association_basics.html#the-has-many-through-association
Letzter Zugriff: 04.06.2016

[Ruby on Rails Guides] Ruby on Rails Guides *Active Record Migrations*
http://guides.rubyonrails.org/active_record_migrations.html
Letzter Zugriff: 04.06.2016

[Ruby on Rails Guides] Ruby on Rails Guides *Routing*
http://guides.rubyonrails.org/routing.html#route-globbing-and-wildcard-segments
Letzter Zugriff: 04.06.2016

[Google Design] Material Design Specification *Components Buttons: Floating Action Button*
https://www.google.com/design/spec/components/buttons-floating-action-button.html
Letzter Zugriff: 04.06.2016

[devise-token-auth] devise-token-auth *Token Handling*
https://raw.githubusercontent.com/lynndylanhurley/ng-token-auth/master/test/app/images/flow/token-update-detail.jpg
Letzter Zugriff: 06.08.2016

[devise-token-auth] devise-token-auth *Token Validation*
https://raw.githubusercontent.com/lynndylanhurley/ng-token-auth/master/test/app/images/flow/validation-flow.jpg
Letzter Zugriff: 06.08.2016

[devise-token-auth] devise-token-auth *Email Sign Up*
https://raw.githubusercontent.com/lynndylanhurley/ng-token-auth/master/test/app/images/flow/email-registration-flow.jpg
Letzter Zugriff: 06.08.2016

[devise-token-auth] devise-token-auth *Email Sign In*
https://raw.githubusercontent.com/lynndylanhurley/ng-token-auth/master/test/app/images/flow/email-sign-in-flow.jpg
Letzter Zugriff: 06.08.2016

[devise-token-auth] devise-token-auth *Password Recovery*
https://raw.githubusercontent.com/lynndylanhurley/ng-token-auth/master/test/app/images/flow/password-reset-flow.jpg
Letzter Zugriff: 06.08.2016

[react-vdom] react-vdom *Virtual DOM Illustration*
http://maerch.github.io/img/react-vdom/vdom.jpg
Letzter Zugriff: 06.08.2016

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Yannick Schuchmann