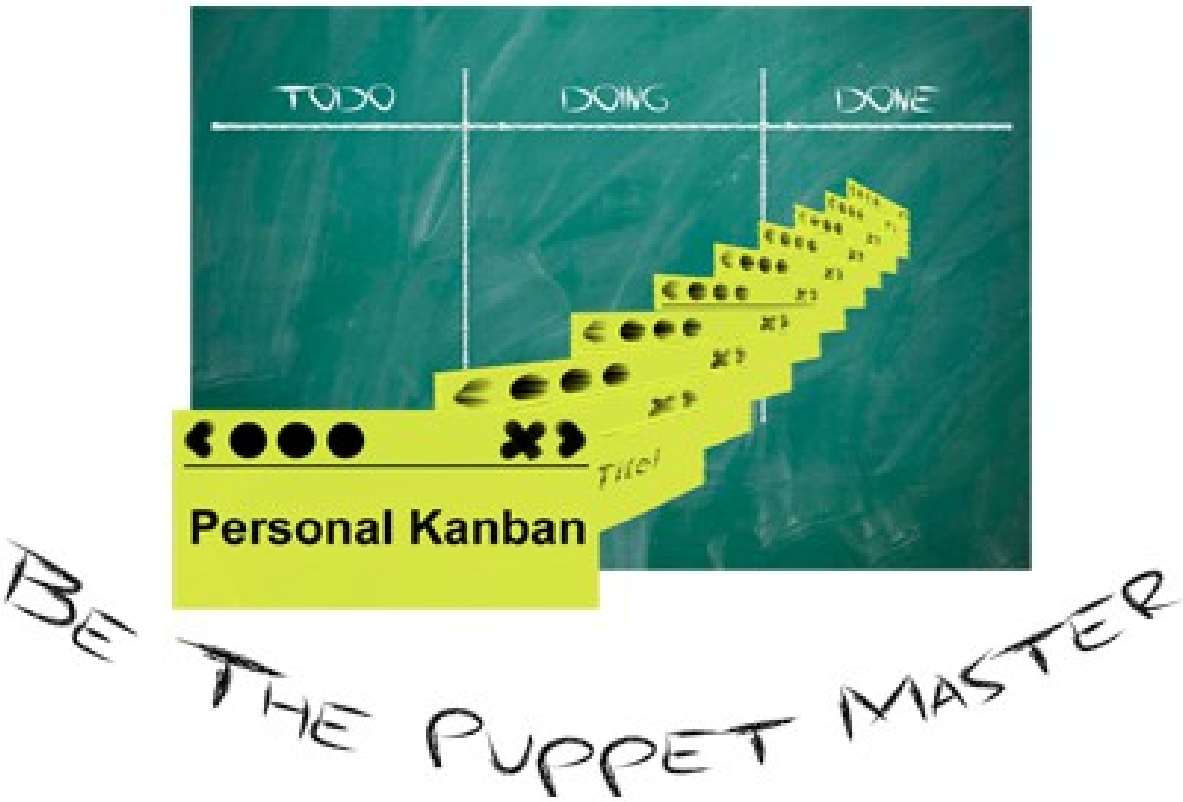


Personal Kanban

Mittelstufenprojekt im Fach Anwendungsentwicklung:

Auftraggeber:	Herr Heidemann
Klasse:	FA12B
Gruppenmitglieder:	Willi Sylvester
	Jan Zaydowicz
	Momme Raap

Inhaltsverzeichnis

Anhänge

1. Aufgabenstellung

1.1. Was ist Gegenstand des Projektauftrags?

Das Ziel unseres Projektes ist das Erstellen einer Webapplikation zur Verbesserung des Zeitmanagements von Personen und Personengruppen.
Bisher stellen wir uns dies auf Grundlage des Ansatzes von Canban vor, jedoch kann sich dieses im Verlauf des Projekts noch ändern, sofern andere Methoden/Ansätze zielführender erscheinen.

1.2. Warum wurde dieser Auftrag gewählt?

Das Zeitmanagement ist gerade in der heutigen Zeit ein immer wichtigeres Thema.
Dies gilt nicht nur im Beruf, sondern auch im privaten Bereich, insbesondere ist es wichtig offene Zeitressourcen zu erkennen und diese sinnvoll zu nutzen.
Da wir selber mitunter größere Probleme mit unserem Zeitmanagement haben, bzw. das Gefühl haben unsere Zeit ineffektiv zu nutzen, sehen wir in diesem Projekt außerdem einen persönlichen Nutzen.

2. Ist-Analyse

2.1 Zu lösendes Problem

Jeder hat einen Pool von Aufgaben, Terminen, Meetings etc., welche er zu bewältigen hat, sei es privat oder im beruflichen Umfeld. Wenn mehrere Personen daran beteiligt sind, steigt die Komplexität des Organisationsaufwandes noch weiter an.
Dabei kann leicht die Übersicht verloren werden, Aufgaben werden nicht rechtzeitig erledigt oder sogar vergessen. Die unstrukturierte Ausführung führt zu einem hohem Zeit-/Kostenaufwand, wodurch schnell Frust entsteht.
Unser Projekt beschäftigt sich also mit der Auflösung der Komplexität, mit Hilfe bestehender Elemente und Hilfsmittel der Canban-Praktiken.

2.2 Derzeitige Lösungen

Es gibt verschiedene Ansätze für die Lösung der beschriebenen Probleme im Zeitmanagement. Zwei dieser Lösungsansätze haben wir genauer betrachtet.

To-Do-Liste:

Die To-Do-Liste kann in verschiedenen Formen geschrieben werden z.B. Zettel, Flipchart, Tafel oder Applikation. Auf der To-Do-Liste werden die anstehenden Aufgaben, Verantwortlichkeiten und zeitlichen Fristen festgehalten. Erledigte Aufgaben werden abgehakt. Um Prioritäten zu definieren könnte die Eisenhower-Methode Abhilfe schaffen, bei der Aufgaben in einer Matrix 4 Typen zugeordnet werden mit der Spalte Dringlichkeit und der Zeile Wichtigkeit.
Ein Problem beider Methoden ist das Zustandsänderungen der Aufgaben (z.B. Zeitpunkt der Erledigung) nicht erfasst werden. Außerdem müssen wiederkehrende Aufgaben immer wieder neu aufgeschrieben werden, wenn sie abgehakt wurden.

Getting Things Done (GTD):

„Hauptprinzip der Methode ist, dass der Nutzer alle seine anstehenden Tätigkeiten in einem Verwaltungssystem notiert und dadurch den Kopf frei hat für Wichtigeres, nämlich die Erledigung der aktuellen Aufgabe, ohne befürchten zu müssen, andere Aufgaben zu vergessen. „ - Wikipedia

Bei GTD werden kontextbezogene Aufgabenlisten erstellt, unterschieden werden dabei die eigentlichen Aufgaben und Termine (Aufgaben mit Frist). Termine werden in einem Kalender notiert.
In einem festgelegten Rhythmus (z.B. wöchentlich) werden die Kontextlisten überprüft und angepasst. Neue

Aufgaben werden als Elemente im Eingang gesammelt und von dort in die Kontextlisten eingebunden.

Aufgaben die aus mehr als einer Tätigkeit bestehen, werden dabei Projekte genannt. Um den Fluss des Projektes zu gewährleisten wird der nächste durchführbare Schritt formuliert.
Als Grundregel gilt das jede Tätigkeit die in weniger als 2 Minuten zu erledigen ist, sofort erledigt wird (2-Minuten-Regel).
Ein Nachteil dieses Systems ist die fehlende Priorisierung, es ist also nicht sofort erkennbar was als nächstes getan werden muss. Des Weiteren sind alle Aufgaben mit mehr als 2 Tätigkeiten Projekte, eine weitere Unterscheidung gibt es nicht. Das führt dazu das sowohl eine einfache Aufgabe wie abwaschen (spülen und abtrocknen) auf einer „Ebene“ stehen würde wie die Durchführung des Mittelstufenprojektes

Quellen: wikipedia.de - http://simplivest.de/wp-content/uploads/2008/11/eisenhower-methode-to-do-liste.jpg - http://www.todo-liste.de

2.3 Ziel des Projektes:

Das Ziel unseres Projektes ist es, durch die Verbesserung des Zeitmanagements, dem Nutzer mehr Freiräume für Sachen welche er gerne tut zu verschaffen. Außerdem soll durch eine sinnvolle Optische Aufbereitung schnell ersichtlich sein was dringend zu erledigen ist, um somit das vergessen wichtiger Aufgaben zu vermeiden.
Dies soll nicht nur für einzelne Nutzer, sondern auch für ganze Nutzergruppen möglich sein und somit die Koordination der Aufgaben in Gruppe vereinfachen.

3. Soll-Konzept

3.1. Lösungsansatz

Wir haben den Lösungsansatz gewählt, vorrangig Kanban-Praktiken einzusetzen. Es wird eine maximale Anzahl an zeitgleich zu bearbeitenden Aufgaben geben (WIP), visualisiert wir der Prozess auf einem Board mit Tickets die Aufgaben repräsentieren.
Vorerst werden wir eine festgelegte Anzahl von drei Spalten (Todo, Doing, Done) anbieten.
Die Priorisierung der Tickets wird durch verschiedene Ticketfarben realisiert. Terminierte Aufgaben gilt es besonders hervor zu heben, damit diese nicht vergessen werden.
Die Zustandsänderungen auf dem Board müssen auch nachträglich verfolgbar sein.
Besondere Features könnten individuelle Boardeinstellungen und ein einfaches Gruppenmanagement (Hinzufügen, Teilen, Bearbeiten eines gemeinsamen Boards) sein.

Generell ist es uns besonders wichtig so viele relevante Informationen wie möglich auf einen Blick zu repräsentieren.

3.2 Architektur

3.2.1 Client

Für die Strukturierung der Client-Programmierung, von Personal Canban, mit JavaScript haben wir uns für das MVC-Architekturmuster entschieden.
Für die Implementierung dieses Musters wollen wir das *Ember*[1] Framework nutzen.

Zum Grundverständnis einer Ember Web Applikation ist es wichtig ihren URL Aufbau zu verstehen. So eine URL könnte folgendermaßen aussehen:

http://www.emberapp.de/#ticket/123 |--
1--|---------- 2 ----------|----- 3 -----|

1 http://emberjs.com

1. Schema
2. Host
3. Fragment

   Das Fragment wird in den meisten Fällen zum Verweis auf Anker in HTML-Dateien verwendet, was zu keinem Server Request führt, solange der Rest der URL sich nicht verändert. Dies nutzt *Ember,* damit bei einem Klick auf einen Link die Seite nicht neu geladen wird, sondern einfach das Fragment ausgelesen werden kann um darauf hin den neuen Seiteninhalt per JavaScript nachzuladen.

Das **MVC** Muster wird in *Ember* wie folgt implementiert*:*

Model (*Model*):

EmberDataStorageModels (i. v. Model) sind JavaScript Objekte mit Eigenschaften und Funktionen – welche in diesem Zusammenhang für berechnete Eigenschaften verwendet werden. In unserem Fall werden diese die Tabellen unserer Datenbank widerspiegeln.

Außerdem enthalten sie Funktionen, welche das Speichern oder Löschen ihrer Inhalte ermöglichen. Dafür greifen sie auf den *EmberDataStorage* (i. v. Storage) zu, welcher für das Puffern und Persistieren der Model Daten zuständig ist. Er aktualisiert dann den Puffer und greift auf den für die Applikation bestimmten *DataStorageAdapter* – in unserem Fall den *RESTAdapter*, bzw. den *LocalStorageAdapter* – zu, um die Daten zu persistieren. Der Storage ist die Schnittstelle für das Erstellen, Lesen, Aktualisieren und Löschen von Model Einträgen.

View (*Präsentation*):

Die Implementierung der View findet bei *Ember* mit Hilfe der Template-Engine *Handlebars*[2] statt. *Handlebars* ist zum einen zuständig für die Templates, d. H. die Grundgerüste, der einzelnen Seiten der Applikation und zum anderen für die der einzelnen Komponenten (z . B. das Hauptmenü), welche auf mehreren verschiedenen Seiten wiederverwendet werden können.
Durch *Handlebars* ist es möglich auf von *Ember* übergebene Model Daten oder Controller Funktionen zu zugreifen und somit die Ausgabe dynamisch zu gestalten.
Außerdem ist es auch möglich z. B. die Ausgabe von Inhalten an Bedingungen zu knüpfen oder über eine Menge von Model Einträgen zu iterieren.

Für das Design der Seiten werden wir *LESS*[3] verwenden. Dabei handelt es sich um eine Erweiterung des *CSS* Standards. Es ist sehr gut dafür geeignet die sonst häufigeren Redundanzen in *CSS* Dateien zu vermindern (*Don't repeat yourself*). Durch *LESS* ist es z. B. möglich Variablen zu definieren oder auch Regeln zu verschachteln, jedoch muss es vor der Verwendung in reines *CSS* Kompiliert werden.

Controller (*Steuerung)*:

Ein wichtiger Teil der Steuerung ist der *EmberRouter*. Durch ihn werden der Name, der Pfad und die möglichen URL-Parameter der einzelnen *EmberRoutes* bestimmt.
Diese sind dann dafür zuständig der Anfrage den richtigen *EmberController* (i. v. Controller), das Model und das *Handlebars-Template* zu zuordnen.

Controller sind dafür zuständig den Templates zusätzlichen Funktionen zur Verfügung zu stellen, wie z. B. die Filterung aller zur Verfügung stehenden Model Einträge anhand einer bestimmten Eigenschaft.
Außerdem können in dem Controller auch Formulare ausgewertet und Informationen per AJAX-Request mit dem Server ausgetauscht werden.
EmberData stellt außerdem einen *RESTAdapter* und eine *LocalStorageAdapter* bereit.
Mit dem *RestAdapter ist* es möglich, über eine entsprechende *RESTAPI*, mit dem Server zu kommunizieren und so Model Daten abzufragen oder dem Server Änderungen mitzuteilen.
Der *LocalStorageAdapter* ermöglicht es die Model Daten im *LocalStorage,* im Browser des Clients, zu speichern.

2 http://handlebarsjs.com
3 http://lesscss.org

In der Standard Implementierung müssen alle *Handlebars-Templates* in eine Datei geschrieben werden. Damit wir dieses umgehen können haben wir uns dafür entschlossen *Grunt*[4] zu verwenden. *Grunt* ist ein *Nodejs*[5] plugin, welches uns erlaubt die Templates in einzelne Dateien zu splitten, da es die einzelnen

Templates mithilfe einer *Handlebars* Erweiterung in eine Datei kompiliert.
Weitere Vorteile welche wir durch *Grunt* erhalten sind folgende:

- "watch-Task": Automatische Ausführung aller Aufgaben sobald eine "beobachtete" Datei verändert wurde
- automatische *LESS* Kompilierung
- Minifizierung von *JavaScript* und *CSS* Dateien für Release Versionen
- jshint hinweise in der Windows Konsole

3.2.2 Architektur Server:

Als Backend-Solution-Stack benutzen wir WAMP (**W**indows, **A**pache, **M**ySQL und **P**HP).

Die Hauptaufgabe des PHP Backend ist die Behandlung von Requests, die Bereitstellung einer Schnittstelle zur Datenbank (REST-Operationen) und Response an den Client geben. Requestweiterleitungen fallen dabei in den Aufgabenbereich der Konfigurationsdatei .htaccess.

Für die Abstraktion unsere relationalen Datenbank werden wir ein bestehendes ActiveRecord-Framework[6] nutzen.

Besonderen Wert legen wir auf die Abkapselung des Backend und einen abstrakten, sowie modularen Aufbau, damit die Erweiterbarkeit und Anpassungsfähigkeit gegeben ist.

Um diesen Anforderungen gerecht werden zu können, werden wir uns verschiedener Designprinzipien bedienen. So werden wir unsere Klassen nach logischen Gesichtspunkten unterteilen (S*eperation of Concerns*), klare Verantwortlichkeiten zuweisen (*Single Responsibility Principle)* und eine möglichst lose Kopplung schaffen (*Dependency Injection*)

Zum Umsetzen der oben genannten Designprinzipien, bietet sich die Nutzung folgender Design Patterns an:

> Um mit Hilfe einer möglichst losen Kopplung andere Objekte benachrichtigen zu können (z.B. über die erfolgreiche Datenbankabfrage) wird das Observer-Subject-Pattern zum Einsatz kommen.
>
> Für das Erstellen unserer ActiveRecord-Modelklassen bietet sich das Factory-Pattern an, da die erstellende Klasse die zu erstellende nicht kennen darf (keine harten Abhängigkeiten)

Des Weiteren gibt es nur einen Einstiegspunkt ins Backend, die RESTAPI.

4 http://gruntjs.com
5 http://nodejs.org
6 http://www.phpactiverecord.org

3.3. Vor- und Nachteile (Architektur, Implementierung und Inhalt)

–Vorteile:
- alle relevanten Informationen auf einen Blick sehen
- keine wichtigen Aufgaben vergessen
- Architektur bietet ein sehr hohes Maß an Austauschbarkeit und Erweiterbarkeit
- Zeitplanung innerhalb einer Gruppe von Nutzern
- Überall erreichbar
- Zentrale Verwaltung aller Aufgaben und Termine
- Eine All-in-one Client-Architektur Lösung durch Emberjs

–Nachteile
- Handlebars Templates müssen in einer Datei stehen
- –Zusätzlicher Zeitaufwand für Implementierung von Grunt
- –Keine native SQL Unterstützung von Emberjs

4. Implementierung

4.1. Vorbereitung

Bevor wir mit der Implementierung beginnen konnten, galt es einige grundsätzliche Dinge zu klären. Um gemeinsam an unserem Projekt arbeiten zu können, haben wir uns für git als zentrales Versionsverwaltungssystem entschieden. Das hat den Vorteil, dass unsere Arbeit protokolliert und archiviert wird. Des Weiteren kann bei fehlerhaften commits eine ältere Version wiederhergestellt werden. Da unsere Software quelloffen ist, haben wir uns entschieden unser git-Repository bei github zu hosten, da es dort kostenlos möglich ist.

Als nächsten Schritt haben wir eine gemeinsame Entwicklungsumgebung geschaffen.
Dazu gehört, dass wir alle mit derselben IDE, in unserem Fall PhpStorm 7.1.2, arbeiten und dort die gleichen Einstellungen (Einrückungseinstellungen, Zeilenumbrüche usw.) haben. Das war in PhpStorm sehr einfach umsetzbar, da die Applikation die Möglichkeit bietet Einstellungen direkt zu exportieren. Die exportierte Datei haben wir dann ebenfalls in unserem git-Repository versioniert, damit, bei späteren Änderungen an den Einstellungen, die Gruppenmitglieder die aktualisierte Version direkt importieren können.

Die eigentliche Aufgabenteilung, während der Implementierung, haben wir anhand der geplanten Features vorgenommen.
So hatte jedes Gruppenmitglied die Verantwortung für festgelegte Features und hat diese in entsprechenden FeatureBranches bearbeitet.
Ein weiterer Vorteil der Plattform github ist das integrierte Issuetrackingsystem. Dort wurden die einzelnen Features als Milestones angelegt. Für die Aufteilung in Unteraufgaben(Issues) war das entsprechende Gruppenmitglied selbst verantwortlich.
In regelmäßigen Abständen haben wir im Team den Status der Issues besprochen und Fertigstellungstermine festgelegt. Die Termine waren dabei an unseren Projektablaufplan angelehnt, mussten aber häufiger angepasst werden, da die geplanten Termine nicht eingehalten werden konnten.
Als Diskussionsplattform diente uns hauptsächlich das Issuetrackingsystem in github. Bei zeitkritischen Angelegenheiten gab es ebenfalls telefonischen bzw. E-Mail-Kontakt.
Die Berufsschulstunden wurden überwiegend genutzt um Aufgaben zu verteilen, Probleme zu besprechen und den aktuellen Stand zu vergleichen.

4.2 Kommunikation zwischen Server und Client

Wir halten wir uns bei der Kommunikation an das REST Paradigma. Dieses umfasst folgende 5 Prinzipien:

1. Adressierbarkeit:
   Jeder Dienst muss über eine eindeutige Adresse (URL) verfügen.

2. Unterschiedliche Repräsentationen:
   Dienste können unterschiedliche Darstellungsformen haben. Der REST-Server muss dabei verschiedene Repräsentationen (zum Beispiel XML oder JSON) ausliefern können.

3. Zustandslosigkeit:
   Das Client-Server-Protokoll muss zustandslos sein (zum Beispiel HTTP).

4. Operationen:
   REST-konforme Dienste müssen einige Operationen vorsehen, die auf alle Ressourcen angewendet werden können. (siehe HTTP-Verbs)

5. Verwendung von Hypermedia:
   Für Anwendungsinformationen und Zustandsänderungen werden Hypermedia verwendet.

Als Anwendungsschicht-Protokoll haben wir uns für *HTTP/1.1*[7] entschieden. Dabei nutzen wir folgenden Befehle (HTTP-Verbs):

PUT:
   Die angegebene Ressource wird angelegt, existiert sie schon wird sie geändert

GET:
   Die angegebene Ressource wird vom Server angefordert. GET gilt als sicher, es führt also zu keiner Zustandsänderung.

POST:
   Fügt eine neue Ressource ein.

DELETE:
   Löscht eine angegebene Ressource

Clientseitig nutzen wir den von uns erweiterten, in Ember.js integrierten, REST Adapter (`DS.RESTAdapter`), um REST-konforme Requests an den Server zu schicken.
Diesem haben wir zusätzliche Funktionen hinzugefügt, um parallel zum REST-Request in den LocalStorage zu schreiben bzw. aus ihm zu lesen.
Der REST Adapter erzeugt, durch den Aufruf einer seiner *Actions*[8], einen `XMLHttpRequest` um mit dem Server zu kommunizieren.

---

7 HTTP = Hypertext Transfer Protokoll
8 Siehe Anhang B Tabellenverzeichnis:Tabelle 2 REST Adapter Actions

Der konkrete Ablauf des Datenaustausches zwischen Client und Server ist im folgenden skizziert:

1. Ein User loggt sich ein, es wird ein GET-Request an den Server geschickt, dieser beinhaltet seine Anmeldeinformationen. Diese werden vom Server mit der Datenbank abgeglichen. Sind die Informationen korrekt, so wird dem User ein Token zugewiesen.
Dieser Token wird bei jedem folgenden Request über den `HTTP-Header` mit übergeben.
Anhand dieses Tokens wird überprüft, ob der Request von einem eingeloggten User stammt.

2. Wenn der Login erfolgreich durchgeführt wurde, dann werden die, dem Nutzer zugehörigen, Boards und Tickets von dem Server abgefragt. Dabei wird das aktuelle Datum im LocalStorage gespeichert, um im Folgenden nur noch Daten abzufragen, welche sich nach diesem Datum geändert haben.

3. Da der LocalStorage des Users jetzt alle aktuellen Informationen beinhaltet, können wir uns zum Nachladen von Inhalten ausschließlich aus diesem bedienen. Das hat den großen Vorteil das, nach der initialen, längeren Ladezeit beim Einloggen, die Ladezeiten nachfolgend ziemlich gering sind.

4. Nimmt der User jetzt Änderungen vor (zum Beispiel erstellt der Nutzer ein neues Ticket), werden diese Änderungen in den LocalStorage geschrieben und mit dem Server persistiert. Also werden nach dem initialen GET-Request ausschließlich DELETE, POST oder PUT-Requests zum Persistieren an den Server geschickt, da nur diese Operationen Änderungen an den Daten bedeuten. Sollte es aufgrund zu großer Datenmengen zu extremen initialen Ladezeiten kommen, wäre auch ein modulares Nachladen denkbar, also nur dann die Inhalte serverseitig anzufragen wenn sie gebraucht werden. Vorerst ist dies allerdings nicht vorgesehen.

4.3. Server

Serverseitig haben wir als erstes die gewählte ORM-Library, das ActiveRecord-Framework in der stable-version 1.0, implementiert und die Datenbank erstellt.
Da uns keine zentrale MySQL-Datenbank zur Verfügung stand, haben wir unser Datenbankschema versioniert. Bei Änderungen an der Datenbank wurde somit nur die neue Version commited und die anderen Gruppenmitglieder konnten entsprechend ihre MySQL-Datenbank aktualisieren.
Ein besonderes Problem trat im Zusammenhang mit den Zugangsdaten der Datenbank auf. Wir wollten aus Sicherheitsgründen die Zugangsdaten nicht öffentlich zugänglich versionieren, als einfache Lösung haben wir uns dafür entschieden, die Zugangsdaten aus einer nicht versionierten xml-Datei zu laden, die jedes Gruppenmitglied selbstständig anlegen musste. Für das Laden der entsprechenden Datei haben wir uns im Backend eine Hilfsklasse (`CredentialsReader`) geschrieben.

REST-API:

Im nächsten Schritt haben wir uns mit der Behandlung von Requests auseinandergesetzt. Zuerst musste die Konfigurationdatei unseres apache-Webservers, `.htaccess`, angepasst werden. Dieses konnten wir dank unserer Vorkenntnisse im Umgang mit Regular Expressions einfach lösen. Es werden dabei alle Requests unserer Applikation vom Webserver intern auf eine einzige Datei (`index.php`) weitergeleitet, die den zentralen Einstiegspunkt zum Server darstellt. Die übergebenen Request-Parameter werden dabei übernommen, damit diese weiter verarbeitet werden können. Der User bekommt von dieser Weiterleitung nichts mit. Für die Verarbeitung und Aufbereitung der Parameter ist unser `RequestHandler` zuständig.

Die Verarbeitung umfasst dabei das Auslesen der angefragten REST-Operation, des affektierten Modelnamens, die Dekodierung der übermittelten JSON-Daten und, wenn vorhanden, der mitgelieferten Identifikationsnummer. Die aufbereiteten Parameter werden dann dynamisch in die `ModelFactory` injiziert.
In der `ModelFactory` wird zuerst die Verbindung zur Datenbank initialisiert, die Credentials werden dabei mit Hilfe der oben beschriebenen Hilfsklasse geladen. Anhand des übermittelten Modelnamens kann die zugehörige Modelklasse dynamisch instantiiert werden. Die übergebene REST-Operation wird dabei, in der entsprechenden Modelklasse, in Form einer Funktion repräsentiert. Wird zum Beispiel für das Model `user` die Operation `create` angefragt, muss in der Modelklasse `User` eine entsprechende Funktion `createUser` implementiert sein. Die JSON-Daten werden als Array, bereinigt von NULL-Werten, in die Modelfunktion injiziert und dort weiter verarbeitet.

Die Modelklassen repräsentieren unsere Datenbanktabellen und erben von der ActiveRecord/Model-Klasse. So gibt es für jede Tabelle auch eine entsprechende Klasse. In diesen Klassen wird zu der angefragten REST-Operation die entsprechende CRUD[9]-Datenbankoperation durchgeführt. Das Ergebnis dieser Abfrage wird dann, in Form von ActiveRecord-Modelobjekten, zurückgeliefert und durch Nutzung des Observer-Subject-Pattern als Notification an den Observer (`RequestHandler`) weitergegeben. Im `RequestHandler` wird das Ergebnis dann an die `ResponseFactory` übertragen. Die `ResponseFactory` ist dann für die Rückgabe des Ergebnisses an den Client zuständig. Dazu wird das Ergebnis wieder in das JSON-Format enkodiert. Dabei gab es ein Problem, wenn das Ergebnis mehrere Objekte umfasst (zum Beispiel alle Tickets eines Users), da der enkodierte JSON-String für den Client nicht im korrekten Format vorlag. Als Lösung haben wir eine zusätzliche `to_array`-Funktion in die ActiveRecord/Model-Library implementiert. Damit konnte dieses Problem zufriedenstellend gelöst werden. Außerdem hat die `ResponseFactory` die Aufgabe im Falle eines Serverfehlers, diesen aufzubereiten, sodass Clientseitig darauf reagiert werden kann (zum Beispiel in Form einer angepassten, userfreundlichen Fehlerbenachrichtigung).
Die `ResponseFactory` hat auch die Verantwortlichkeit die korrekten HTTP-Header für die Antwort zu setzten, abhängig davon ob das Ergebnis erfolgreich war oder ein Fehler aufgetreten ist (und welcher).Generell behandeln wir serverseitig Exceptions und leiten die Fehlermeldungen, als solche gekennzeichnet, an den Client weiter, damit dort auf den entsprechenden Fehlerfall reagiert werden kann und der User keine Serverfehler angezeigt bekommt. Des Weiteren bietet dies den Vorteil, dass wir durch die Fehlermeldungen clientseitig besser debuggen können.

9 CRUD = Create, Retrieve, Update and Delete

4.4 Client

Wir haben uns dafür entschieden diese Funktionalitäten zu implementieren:

- Personal Canban Route
- Login und Registrierung
- Boardansicht
- Ticket Komponente
- (User-)Profilbereich
- Boardmanagement

Im Folgenden gehen wir näher auf diese Funktionalitäten ein und wie wir diese umgesetzt haben.

Personal Canban Route

Diese Route wird geladen sobald ein User http://canban.6te.net aufruft.
Hier wird überprüft ob sich ein User *erfolgreich eingeloggt hat*[10].
Ist dies der Fall, dann wird der Benutzer auf sein Mainboard weitergeleitet, nachdem alle neuen Änderungen per GET-Request von dem Server angefordert wurden.
Das Mainboard gibt dem Nutzer eine Übersicht über alle Tickets, auf allen anderen Boards.
Ist noch kein Nutzer angemeldet, wird er auf die `LoginRoute` weitergeleitet.

Login

Der Login wird über die Route `/login` angesprochen. Beim Aufrufen dieser wird entsprechend der `LoginController` und das zugehörige `login_template` geladen. Das `login_template` besteht dabei aus einem Formular mit Inputfeldern, die mit der `ValidationView` und darüber mit dem `LoginController` verbunden sind, die `ValidationView` ist für die on-the-fly Validation der Usereingaben zuständig. Werden die validen Eingaben durch Bestätigung (Klick auf Login) übertragen, wird im Controller die `save`-Methode des `EmberStores` aufgerufen.
Wird der User in der Datenbank gefunden, wird überprüft ob sein eingegebenes Passwort mit dem Vorhandenem übereinstimmt. Die Response die zurück an den Client geliefert wird, enthält das Usermodel oder im Fehlerfall eine Fehlermeldung, die aufgearbeitet dem Nutzer angezeigt wird. Am Ende des erfolgreichen Logins werden die Daten mit dem LocalStorage persistiert und der Nutzer wird zur `PersonalCanbanRoute` zurück geleitet.
Die Route der Registrierung ist `/registration`, das Prinzip dahinter ist dem des Logins sehr ähnlich. Den einzigen Unterschied gibt es in der Kommunikation mit dem Server, es wird diesmal ein POST-Request an den Server gesendet und der User wird neu in der Datenbank angelegt.

10 Siehe 4.2 Kommunikation zwischen Server und Client

Boardansicht

Die Boardansicht wird über die Route `/board/:board_name` aufgerufen. Dabei wird mit dem „:“ angegeben, dass der nachfolgende Teil der Route variable ist. Das `board_name` wird zu einer Variablen geparst auf dessen Inhalt in der `BoardRoute` zugegriffen werden kann. Wird diese Route aufgerufen, so wird in dem model-hook des Controllers eine Anfrage an den `EmberStore` geschickt, welche nach einem Board sucht dessen Name dem Inhalt des variablen Teils entspricht.

Gibt es ein Board mit dem eingegeben Namen, so wird es dem User angezeigt. Hier ist es ihm möglich seine vorhanden Tickets zu bearbeiten oder zu löschen. Die Tickets werden mit Hilfe der `TicketComponent` angezeigt.
Des Weiteren kann das `BoardMenu` genutzt werden um sich Informationen zum Board anzeigen zu lassen, Einstellungen des Boards zu ändern oder neue Tickets auf dem Board anzulegen. In den ersten beiden Fällen wird der User auf die entsprechenden Bereiche im Boardmanagement weitergeleitet.

Ticket Komponente

Die `TicketComponent` ist für die Darstellung von Tickets und Verarbeitung von Usereingaben zuständig. Dafür stellt sie verschiedene Ansichten bereit:

`basic`:
: Es wird nur der Titel angezeigt und ein Menü, um zu den anderen Ansichten zu wechseln.

`details`:
: In dieser Ansicht sieht der User zusätzlich zu dem Titel auch noch den Inhalt des Tickets. Diese und die folgenden Ansichten werden in Form eines jQuery-Dialogs angezeigt.

`edit`:
: Hier werden sowohl der Titel als auch der Inhalt des Tickets in Form von Inputfeldern angezeigt. Klickt der User auf den Save-Button, dann werden die geänderten Daten an den `EmberStore` übergeben und werden per PUT-Request vom Server persistiert. Klickt er hingegen auf den Cancel-Button, werden die Änderungen rückgängig gemacht und die Ansicht wechselt in den `basic`-Modus zurück.

`delete`:
: Wenn der Nutzer in diese Ansicht wechselt, sieht er nochmal den Titel und Inhalt des Tickets und kann sich für das Klicken auf Delete oder Cancel entscheiden. Klickt er auf Delete wird ein DELETE-Request an den Server geschickt und es gibt keine Möglichkeit dies wieder rückgängig zu machen.

Profile

Im Profile-Bereich, Route `/profile`, hat der User die Möglichkeit seine persönlichen Angaben zu ändern, wie zum Beispiel den Namen oder das Passwort. Der Bereich besteht aus einem Formular mit Inputfeldern die validiert werden. Die Felder sind mit den Userdaten vorausgefüllt, damit der User nicht alles eingeben muss, wenn er nur ein Feld ändern möchte.

Es besteht die Möglichkeit von der Profile-Seite zurück zur `PersonalCanbanRoute` zu wechseln, seine veränderten Daten zu speichern, den Vorgang mit Cancel abzubrechen und die vorherigen Daten wiederherzustellen und sich selbst zu löschen. Beim Löschen erscheint ein Sicherheits-Popup das bestätigt werden muss, geschieht dieses wird der User endgültig gelöscht und zurück zur Registrierungs-Seite geleitet.

Boardmanagement

Über das Boardmanagement, Route `/boards/management`, ist es dem User möglich, eine Auflistung aller seiner Boards zu sehen, sowie die Gesamtanzahl seiner Boards und Tickets. Des Weiteren ist es möglich, sich zu jedem Board Informationen, wie z. B. sein Erstellungsdatum, Parentboard, Boardnamen und den eingestellten WIP, anzeigen lassen oder diese, abgesehen vom Erstellungsdatum, auch bearbeiten. Diese Option steht nicht für das Mainboard zu Verfügung, da dieses eine einfache Übersicht über alle vorhandenen Tickets ist und somit nicht über Einstellungen wie einen WIP verfügt.
Außerdem kann jederzeit zu einem beliebigen Board gewechselt werden.

5. Testing

Beim Testen unserer Applikation haben wir uns statischer Software-Testverfahren bedient, insbesondere das kontinuierliche, gegenseitige durchführen von Reviews hat uns geholfen die selbst festgelegte Qualität zu sichern.
Des Weiteren, da jedes Gruppenmitglied für seine *Features selbst verantwortlich war*[11], musste vor dem mergen des entsprechenden FeatureBranches ein Komponententest durchgeführt werden, in dem die Funktionalität ausschließlich dieses Features getestet wurde. Erst danach durften die Änderungen in unseren Master-Branch gemerged werden.
Um unsere REST-API applikationsunabhängig zu testen hat uns ein Firefox-Addon geholfen, da das testen ohne implementierte Funktionalitäten nicht möglich war. Das korrekte Zusammenspiel zwischen Server und Client wurde danach mit Hilfe eines Integrationstests sichergestellt. Ein abschließender Systemtest mit Testdaten wurde, vor Abgabe des Projektes an den Auftraggeber, mit dem Blackbox-Verfahren durchgeführt, um die Gesamtfunktionalitäten der Applikation aus Usersicht zu überprüfen. Insbesondere wurde dabei der Blick auf unsere selbst gestellten Anforderungen gelegt und ob diese zufriedenstellend erfüllt worden sind. Ob der Auftraggeber ebenfalls mit der Umsetzung seiner gestellten Anforderungen zufrieden ist, wird dieser in einem finalen Abnahmetest erörtern.
Leider konnten wir es auch zeitlichen Gründen nicht schaffen testgetrieben mit Unittests zu entwickeln und weitere Testverfahren, wie zum Beispiel Lasttests oder Sicherheitstests, durchzuführen. Ebenfalls hatten wir auch keinen allgemeingültigen Testplan, sodass die Gruppenmitglieder sich selbst um den Umfang und Ablauf der Tests kümmern mussten.

11 In Kapitel 4.1. (Vorbereitung) beschrieben

6. Fazit

6.1. Soll-Ist-Abgleich

Im Folgenden wollen wir näher darauf eingehen, welche Features aus unserer Planung umgesetzt werden konnten und welche nicht. Des Weiteren werden wir erläutern, warum geplante Funktionalitäten nicht umgesetzt wurden.
Nach Vorbild des Kanban-Zeitmanagement-Systems besteht der Hauptbereich unserer Applikation aus dem persönlichen Board und Tickets. Die Ticketfunktionalitäten sind dabei durch Icons dargestellt und in der Hauptansicht ist nur der Titel des Tickets erkennbar. Priorisierungen werden durch verschiedenfarbige Tickets dargestellt. Durch diese Wahl der Darstellung können wir ein hohes Maß an Übersichtlichkeit gewährleisten, dass uns schon während der Planungsphase wichtig war.
Das geplante Boardmanagement konnten wir ebenfalls umsetzten, allerdings ist dort sicher noch viel Potential, so wären zum Beispiel zusätzliche Individualisierungsmöglichkeiten der Boards denkbar.
Bei der Implementierung haben wir die vorgesehenen Designprinzipien wie Seperation of Concerns, Single-Responsibility-Principle und Dependency Injection umgesetzt. Da wir uns an das Ember.js-Coreconcept gehalten haben und diese Prinzipien dort Standard sind. Lediglich in unseren PHP-Klassen mussten wir diese Prinzipien aktiv implementieren und haben dazu die angedachten Design Pattern verwendet.
Das Observer-Subject-Pattern kommt in unserer aktuellen Implementation ebenfalls zum Einsatz. Wir konnten damit die Kopplung zwischen den Klassen im Backend noch loser gestalten, dadurch ist unser Backend recht schmal aufgebaut.
Durch die Nutzung der besagten Prinzipien ist unsere Implementation durch ein hohes Maß an Austauschbarkeit und Erweiterbarkeit gekennzeichnet.
Aus zeitlichen Gründen konnten wir es nicht schaffen das Terminieren von Tickets einzubinden. Auch statistische Auswertungen zum Durchsatz und Engpässen, sowie nachträglich einsehbare Zustandsveränderungen konnten nicht mehr umgesetzt werden.
Leider war es uns ebenfalls nicht möglich ein Gruppenmanagement zu entwickeln, da die zusätzliche Komplexität, die dieses Themas mit sich gebracht hätte, den Rahmen des Projektes gesprengt hätte.

6.2 Ausblick

In dem Projekt sehen wir sehr viel Potential, das wir gerne weiter ausgeschöpft hätten, wenn mehr Zeit für das Projekt gegeben wäre. Dazu gehört insbesondere die Umsetzung der in 6.1. Soll-Ist-Abgleich beschriebenen Features, die bisher nicht realisiert werden konnten. So würden wir gerne das Boardmanagement um weitere Individualisierungsmöglichkeiten, bezüglich der Darstellung und Funktionalität des Boards und der Tickets, erweitern. Ein gutes Beispiel hierfür wäre die Möglichkeit, dass der User die Anzahl und Benennung seiner Boardspalten selbst vor nimmt. Außerdem würde,es uns reizen ein vollwertiges Gruppenmanagement zu implementieren.
Um eine Schnittstelle zwischen den beiden Teilen des Mittelstufenprojekts, unser Marketingkonzept und unsere Applikation, zu schaffen, wäre ein Refactoring nach *SEO*[12]-Gesichtspunkten wünschenswert.

12 SEO = Search Engine Optimization (Suchmaschinenoptimierung)

6.3 Schlusswort

Das Projekt hat uns sehr viel Spaß bereitet und den Lerneffekt schätzen wir sehr hoch ein. Wir haben sowohl technische Fertigkeiten im Umgang mit dem Ember.js-Framework erworben, als auch generell in der Programmierung mit JavaScript und PHP. Für uns völlig neue Technologien wie Grunt und LESS runden den sehr guten Gesamteindruck ab. Des Weiteren haben wir uns intensiv mit verschiedenen Designprinzipien und dem REST-Paradigma auseinandergesetzt.
Generell sehen wir diese Form der Web-Applikationen als zukunftsweisend an und werden uns definitiv im Arbeitsumfeld oder privat weiterhin mit diesen Themen beschäftigen. Die Fortführung des Projektes im Privatbereich ist auch sehr gut denkbar, allerdings würden wir dann testgetrieben mit Unittests entwickeln, um unsere selbst festgelegte Qualität sicher zu stellen.

A Begriffserläuterungen

4. Implementierung

4.1. Vorbereitung

Begriff	Version vom	Erläuterung/Link
Commit	07.05.2014	http://de.wikipedia.org/wiki/Commit
Git	07.05.2014	http://git-scm.com/
Repository	07.05.2014 29.05.2014	http://de.wikipedia.org/wiki/Repository **https://github.com/Canbanizers/Canban** (unser Repository)
IDE	07.05.2014	http://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung

4.2. Kommunikation zwischen Server und Client

Begriff	Version vom	Erläuterung/Link
Representational State Transfer: Prinzipien	09.05.2014	http://de.wikipedia.org/wiki/Representational_State_Transfer#Prinzipien
Hypermedia	09.05.2014	http://de.wikipedia.org/wiki/Hypermedia
HTTTP/1.1: Method Definitions	09.05.2014	http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
Ember.js: The Rest Adapter	12.05.2014	http://emberjs.com/guides/models/the-rest-adapter/
JSON	09.06.2014	http://de.wikipedia.org/wiki/JavaScript_Object_Notation

4.3. Server

Begriff	Version vom	Erläuterung/Link
Regular Expression	12.05.2014	http://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck
Feature Branch	08.06.2014	http://martinfowler.com/bliki/FeatureBranch.html
Github: Features	07.05.2014	https://github.com/features

5. Testing

Begriff	Version vom	Erläuterung/Link
Statisches Software-Testverfahren	12.05.2014	http://de.wikipedia.org/wiki/Statisches_Software-Testverfahren
Modultest	08.06.2014	http://de.wikipedia.org/wiki/Modultest
Systemtest	12.05.2014	http://de.wikipedia.org/wiki/Softwaretest
Black-Box-Test	12.05.2014	http://de.wikipedia.org/wiki/Black-Box-Test
Integrationstest	07.05.2014	http://de.wikipedia.org/wiki/Integrationstest

B Tabellenverzeichnis

Tabelle 1 Clientseitige Bibliotheksversionen

Name (Version)	Verwendungszweck
ember (1.4.0)	Hauptframework der Applikation
ember-data (1.0.0b7)	Persistieren von Applikationsdaten
handlebars (1.3.0)	Templateengine
Moment.js (2.5.1)	Datumsformatierung
bootstrap (3.1.1)	Design und DOM-Manipulation
jquery (1.10.2)	
jquery-ui (1.10.4.custom)	

Tabelle 2 REST Adapter Actions

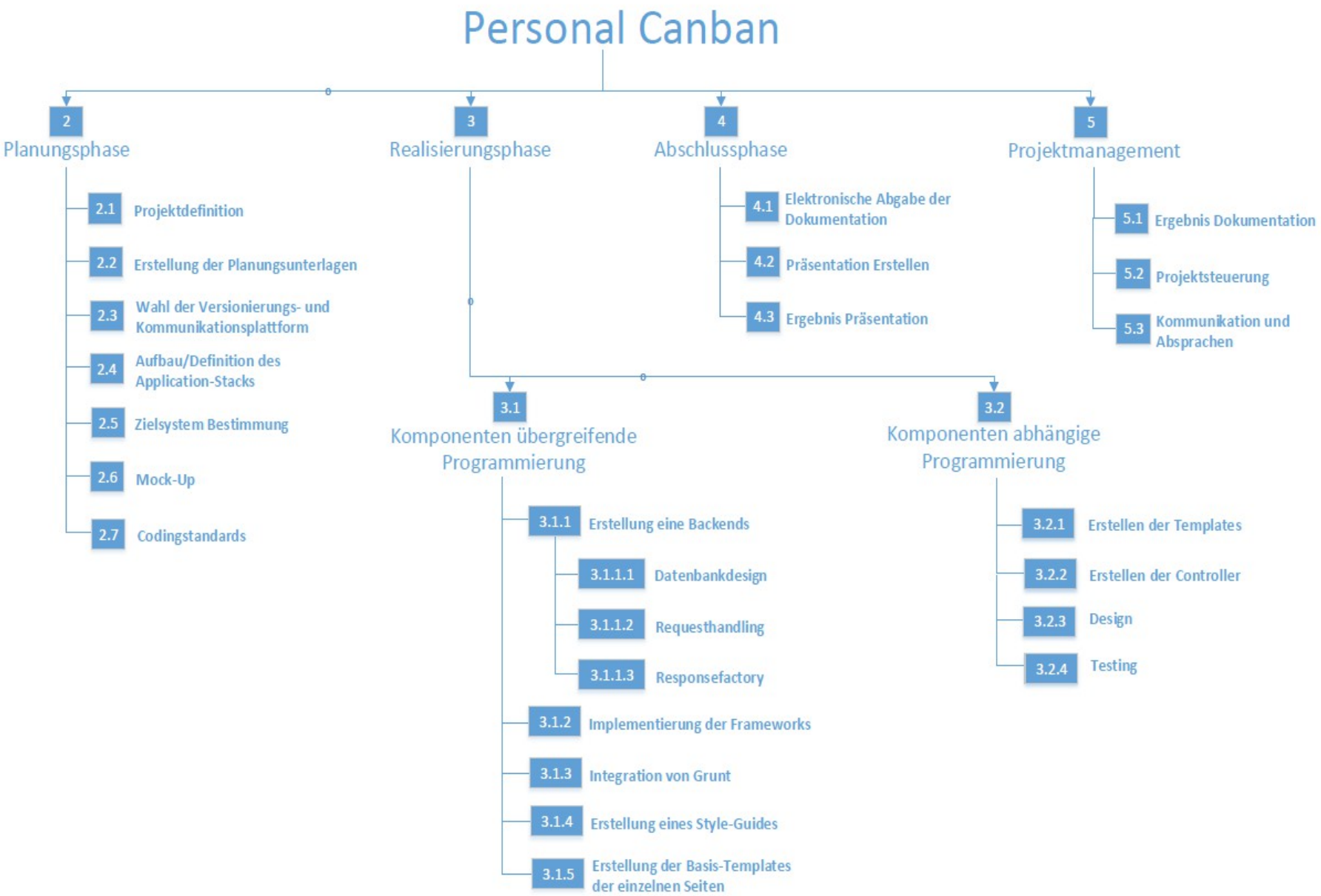
Action	**HTTP Verb**	**URL**
Find	GET	/people/123
Find All	GET	/people
Update	PUT	/people/123
Create	POST	/people
Delete	DELETE	/people/123

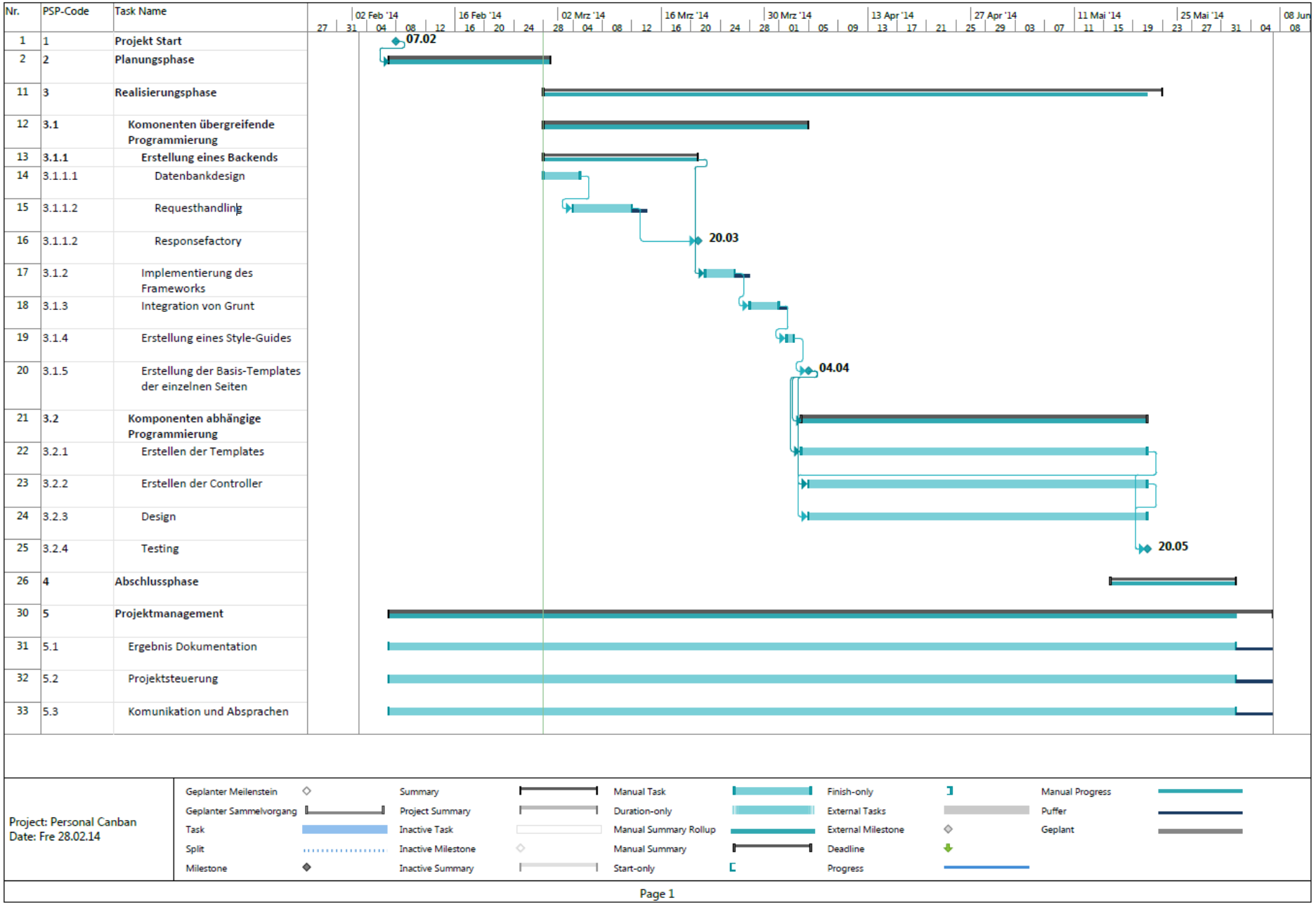
C Planungsunterlagen

Projektantrag

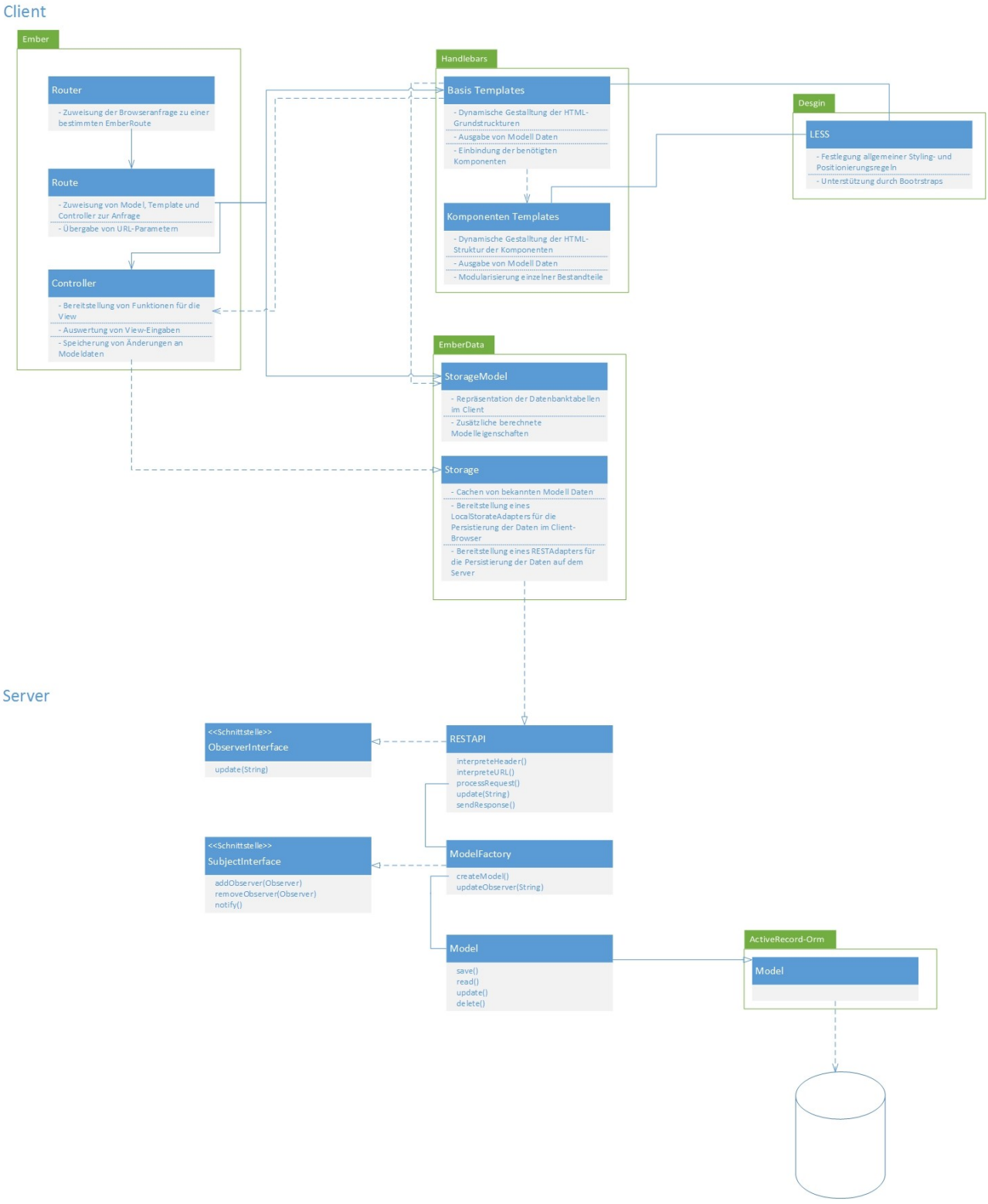
Das Ziel unseres Projektes ist das Erstellen einer Webapplikation zur Verbesserung des Zeitmanagements von Privatpersonen und Personengruppen.
Bisher stellen wir uns dies auf Grundlage des Ansatzes von Canban vor, jedoch kann sich dieses im Verlauf des Projekts noch ändern, sofern andere Methoden/Ansätze zielführender erscheinen.

Gruppenmitglieder:
Willi Sylvester, Jan Zaydowicz, Momme Raap, Christian Ryf


Personal Canban
2
Planungsphase
2.1
Projektdefinition
2.2
Erstellung der Planungsunterlagen
2.3
Wahl der Versionierungs- und Kommunikationsplattform
2.4
Aufbau/Definition des Application-Stacks
2.5
Zielsystem Bestimmung
2.6
Mock-Up
2.7
Codingstandards
3
Realisierungsphase
3.1
Komponenten übergreifende Programmierung
3.1.1
Erstellung eine Backends
3.1.1.1
Datenbankdesign
3.1.1.2
Requesthandling
3.1.1.3
Responsefactory
3.1.2
Implementierung der Frameworks
3.1.3
Integration von Grunt
3.1.4
Erstellung eines Style-Guides
3.1.5
Erstellung der Basis-Templates der einzelnen Seiten
3.2
Komponenten abhängige Programmierung
3.2.1
Erstellen der Templates
3.2.2
Erstellen der Controller
3.2.3
Design
3.2.4
Testing
4
Abschlussphase
4.1
Elektronische Abgabe der Dokumentation
4.2
Präsentation Erstellen
4.3
Ergebnis Präsentation
5
Projektmanagement
5.1
Ergebnis Dokumentation
5.2
Projektsteuerung
5.3
Kommunikation und Absprachen


Nr.
PSP-Code
Task Name
02 Feb '14
16 Feb '14
02 Mrz '14
16 Mrz '14
30 Mrz '14
13 Apr '14
27 Apr '14
11 Mai '14
25 Mai '14
08 Jun
1 1 Projekt Start
2 2 Planungsphase
11 3 Realisierungsphase
12 3.1 Komponenten übergreifende Programmierung
13 3.1.1 Erstellung eines Backends
14 3.1.1.1 Datenbankdesign
15 3.1.1.2 Requesthandling
16 3.1.1.2 Responsefactory
17 3.1.2 Implementierung des Frameworks
18 3.1.3 Integration von Grunt
19 3.1.4 Erstellung eines Style-Guides
20 3.1.5 Erstellung der Basis-Templates der einzelnen Seiten
21 3.2 Komponenten abhängige Programmierung
22 3.2.1 Erstellen der Templates
23 3.2.2 Erstellen der Controller
24 3.2.3 Design
25 3.2.4 Testing
26 4 Abschlussphase
30 5 Projektmanagement
31 5.1 Ergebnis Dokumentation
32 5.2 Projektsteuerung
33 5.3 Komunikation und Absprachen
07.02
20.03
04.04
20.05
Project: Personal Canban
Date: Fre 28.02.14
Geplanter Meilenstein
Geplanter Sammelvorgang
Task
Split
Milestone
Summary
Project Summary
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
External Tasks
External Milestone
Deadline
Progress
Manual Progress
Puffer
Geplant
Page 1

Personal Canban



D Quellen

4. Implementierung

4.1. Vorbereitung
Jetbrains: PhpStorm – Version v. 07.05.2014 - http://www.jetbrains.com/phpstorm/

5. Testing
RESTClient – Version 2.0.3 - https://addons.mozilla.org/de/firefox/addon/restclient/