

# Technische Dokumentation des Gesamtprojekts

Matthis Gördel

2. Mai 2019

## Zusammenfassung

Das MdWI-Projekt der Semester fünf und sechs wird als agiles Softwareprojekt durchgeführt. Für die Umsetzung wurde die Entscheidung getroffen, den Kurs in drei Teams aufzuteilen: *Marketing*, *Frontend* und *Backend*. Hier wird die Motivation hinter dieser Trennung, verbunden mit der technischen Architektur, beschrieben.

## 1 Ausgangssituation

Zu Beginn gab für die Umsetzung des Projekts zwei zentrale Fragen:

- Wie soll die technische Architektur aussehen?
- Wie organisieren wir uns als Kurs?

Als Rahmen war gegeben, dass das Projekt per Scrum organisiert werden soll. Dies bringt einige Implikationen für die Entwicklung mit sich, da Scrum interdisziplinäre Entwicklerteams fordert, im Gegensatz zu z. B. einer Aufteilung in Entwicklungs-, Datenbank-, und Test-Team.

Scrum fordert außerdem Teams von rund sieben Mitarbeitern. Den Kurs in drei Teams aufzuteilen, schien deshalb sinnvoll. Daraus ergab sich die Frage nach einer geeigneten technischen Architektur, bei der mehrere Entwicklerteams parallel an einem Produkt arbeiten können. Da eine Webapplikation entwickelt werden sollte, wurden zwei Optionen gesehen:

- Die Umsetzung als Monolith.
- Die Umsetzung als Single-Page-Anwendung mit getrenntem Front- und Backend.

Eine Umsetzung als *monolithische* Applikation hätte zum Beispiel so aussehen können, dass zum Beispiel mit PHP und HTML oder einem Webframework mit HTML-Template-Engine von mehreren Teams eine einzige Applikation entwickelt wird.

Die Alternative wäre, eine *Single-Page-Anwendung* mit Webservice-Backend zu entwickeln. Hier entwickelt ein Team eine Webanwendung, welche vom

	Monolith	Single-Page-Application
Vorteile	<ul style="list-style-type: none"> <li>• Alle arbeiten mit einer Sprache, einem Framework</li> <li>• Flexiblerer Einsatz der Entwickler</li> </ul>	<ul style="list-style-type: none"> <li>• Entkopplung in zwei Anwendungen: Frontend und Backend</li> <li>• Zwei Teams können parallel arbeiten</li> </ul>
Nachteile	<ul style="list-style-type: none"> <li>• Eventuell komplexer Spaghetti-Code</li> <li>• Schwierig, &gt; 10 Entwickler parallel zu beschäftigen</li> </ul>	<ul style="list-style-type: none"> <li>• Herausforderung der Absprache zw. Front- und Backend</li> <li>• Abhängigkeiten</li> </ul>

Tabelle 1: Vor- und Nachteile der techn. Ansätze

Endnutzer als ein einziges HTML-Dokument heruntergeladen wird, welches daraufhin anzuzeigende Inhalte dynamisch von einem Backend-Server abfragt, der von einem anderen Team entwickelt wird. Hier ist die Website eine Art *Fat-Client*, da die Website, die der Nutzer herunterlädt, viel mehr Logik enthält als die verschiedenen Webseiten, die eine monolithische Applikation generiert.

Es wurden die Ansätze abgewogen (vgl. Tabelle 1) und sich gegen den monolithischen Ansatz entschieden. Daraus resultierte nun endgültig die Teilung in *Frontend* und *Backend* (und *Marketing*).

## 2 Single-Page-Anwendung, Frontend, Backend

Abbildung 1 zeigt die Funktionsweise einer Single-Page-Anwendung. Im ersten hervorgehobenen Rechteck wird das Laden der Anwendung dargestellt. Ein Anwender ruft zum Beispiel die Webseite der Projekt-Anwendung auf. Sein Browser schickt dann eine GET-Request nach der `index.html`, der zentralen HTML-Seite. Diese wird vom Webserver beantwortet. Nun hat der Anwender die Anwendung in den Speicher seines Browsers geladen.

Wenn nun der Anwender zum Beispiel seine Mitgliedsdaten sehen möchte, klickt er im Browser auf den entsprechenden Link. Es wird jedoch keine neue HTML-Seite aufgerufen und heruntergeladen, sondern die Webanwendung schickt stattdessen eine Anfrage an einen Webservice, das *Backend*. Dieser prüft dann zum Beispiel, ob die Anfrage berechtigt ist und greift auf die Datenbank zu um die angeforderten Daten zu lesen.

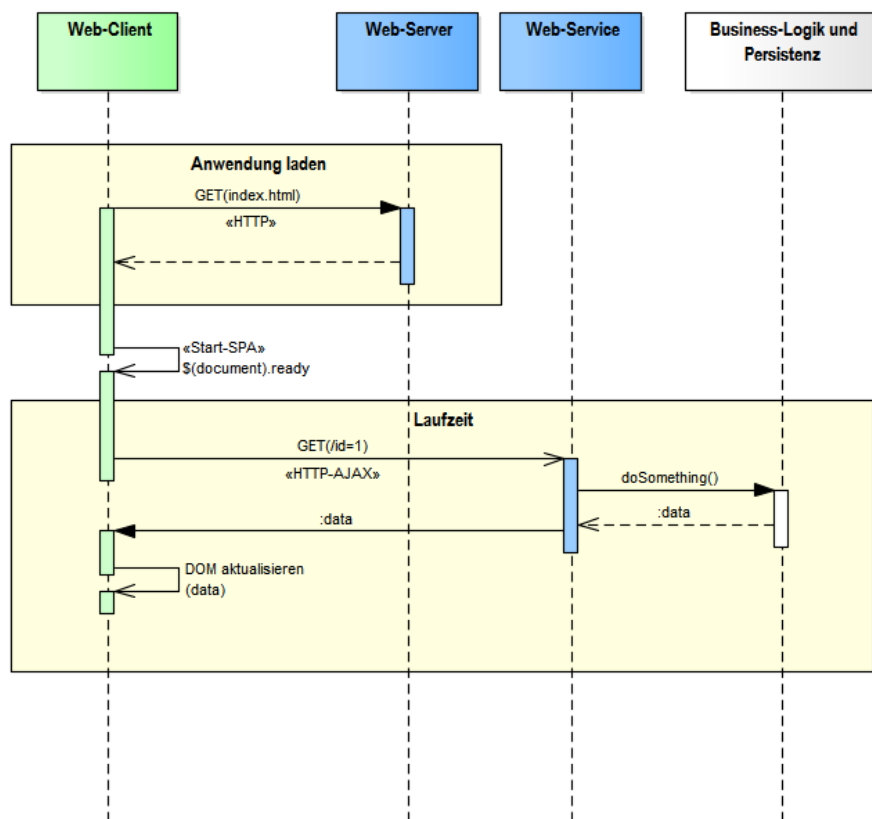


Abbildung 1: Skizze des Ablaufs einer Single-Page-Webanwendung

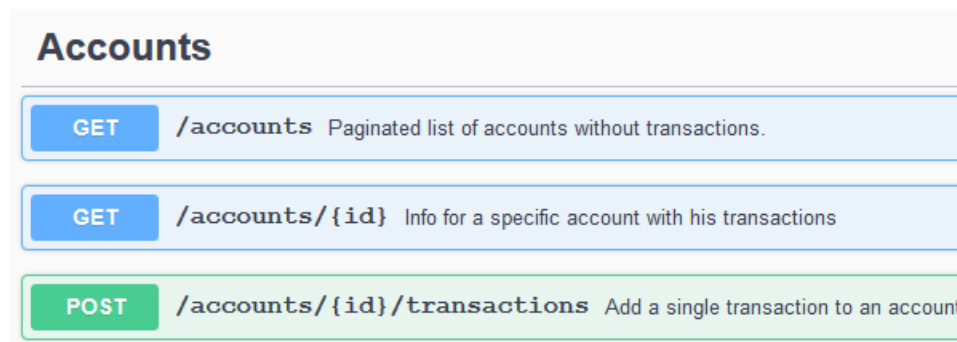
Daraufhin schickt er die Daten an die Webanwendung, die im Browser des Nutzers läuft, zurück. Die Webanwendung aktualisiert nun das *Document Object Model (DOM)*, passt also das HTML der Webseite an, um die neu empfangenen Daten anzuzeigen.

Es ist also ein bisschen so, als ob der Nutzer beim Öffnen der Seite ein Programm herunterlädt und mit diesem interagiert. Das Programm wiederum greift auf einen Server zu, schickt und holt Daten. Schließt der Anwender das Browserfenster löscht/deinstalliert er das Programm sozusagen.

Die Kommunikation zwischen der Anwendung im Browser und dem Webservice erfolgt über eine Webschnittstelle. Die Anwendung ruft also in gewisser Weise selbst Webseiten auf, wobei diese aber nicht für Menschen gedacht sind, sondern einfach nur Daten in einer relativ rohen Form enthalten.

### 3 Die API-Spec

Im Projekt erfolgt die Definition und Dokumentation dieser Web-Schnittstelle über die sogenannte [API-Spec](#). Hier werden die Routen, die die Frontend-Anwendung aufrufen kann, definiert. Genauso werden die eventuell notwendigen Parameter der Anfragen und die möglichen Ergebnisse beschrieben.



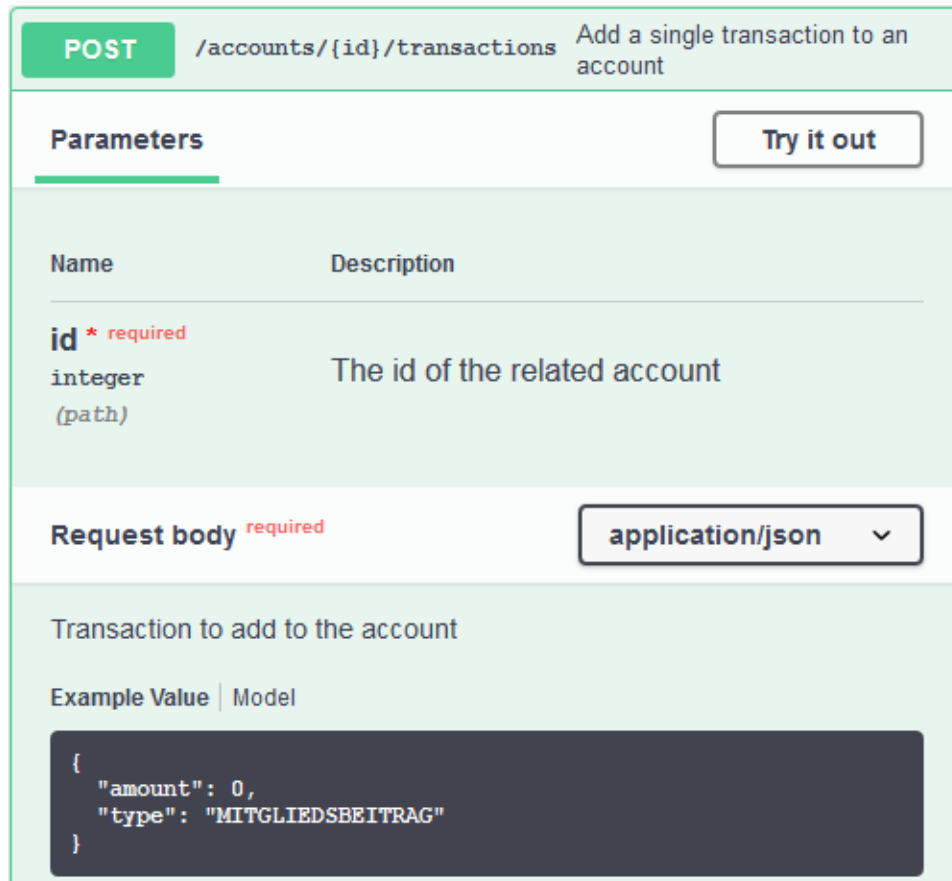
Accounts	
GET	/accounts Paginated list of accounts without transactions.
GET	/accounts/{id} Info for a specific account with his transactions
POST	/accounts/{id}/transactions Add a single transaction to an account

Abbildung 2: Übersicht der möglichen Routen zu Accounts

Abbildung 2 zeigt einen Auszug aus der API-Spec. Hier wurden unter der Überschrift *Accounts* drei verschiedene Routen definiert, auf die jeweils mit verschiedenen HTTP-Methoden zugegriffen wird. Rechts steht eine Beschreibung, was mit dem Aufruf geschieht. Dies ist jedoch nur die Kompakt-Ansicht, die sich expandieren lässt.

Ein Ausschnitt der expandierten Darstellung ist in Abbildung 3 zu sehen. Hier wird beschrieben, welche Parameter für die Anfrage notwendig

sind. Das ist zum einen ein Parameter in der URL des Aufrufs, um anderen muss beim POST eine Art Anhang mitgeschickt werden, nämlich das Objekt im *Request Body*, welches den Betrag und den Transaktionstyp enthält.



The image shows a REST client interface for a POST endpoint. At the top, it says 'POST /accounts/{id}/transactions' with a description 'Add a single transaction to an account'. Below this is a 'Parameters' section with a 'Try it out' button. It lists a path parameter 'id' which is required, of type 'integer', and describes it as 'The id of the related account'. The 'Request body' section is also required and has a dropdown set to 'application/json'. It shows an example transaction object: { 'amount': 0, 'type': 'MITGLIEDSBEITRAG' }.

Name	Description
<b>id</b> * required integer (path)	The id of the related account

**Request body** required application/json

Transaction to add to the account

Example Value	Model
<pre>{   "amount": 0,   "type": "MITGLIEDSBEITRAG" }</pre>	

Abbildung 3: Notwendige Parameter für das Hinzufügen einer Transaktion zu einem Mitgliedskonto

Anhand dieser Definition wissen die Frontend-Entwickler, welche Anfragen sie stellen müssen und wie die Daten aufgebaut sind, die sie erhalten. Für die Backend-Entwickler gilt das gleiche, auch hier ist nun klar, welches Verhalten implementiert werden soll.

Dabei ist nicht nur die geschriebene API-Spec ein wichtiges Dokument, auch der Prozess des Schreibens ist von hoher Bedeutung. Hierbei werden grundlegende Design-Entscheidungen getroffen und die Entwickler müssen sich bereits überlegen, welche logische Darstellung der Daten möglichst sinnvoll aber auch leicht zu implementieren ist.

Im Idealfall wird bei der Entwicklung eines neuen Features zuerst die API-Spec von Front- und Backend-Entwickler gemeinsam definiert, danach

jeweils implementiert und am Ende gemeinsam getestet.

## 4 Die Plattform

Während die API-Spec die Schnittstelle zwischen Frontend und Backend dokumentiert, wird der Quellcode auf der Plattform *GitHub* verwaltet. Frontend und Backend haben hier jeweils ein Repository des Versionsverwaltungssystems *Git*. Hier geschieht die Entwicklung weitestgehend unabhängig, ausgenommen einiger Fälle in dem ein Mitglied des einen Teams *Issues* im Repository des anderen Teams eröffnet, um z. B. Bugs zu dokumentieren.

Um diese parallele Arbeit nun zusammenzuführen, also zu *integrieren* und um eine Instanz des aktuellen Entwicklungsstands der Applikation für den Kunden oder die *POs* zu haben, wird ein *Integrationsserver* betrieben. Auf diesem Server laufen das Frontend und das Backend.

Dabei wird die Integrations-Software *Jenkins* benutzt. Jenkins prüft jede Minute, ob es Änderungen auf dem Haupt- und Testzweig der Git-Repositories von Frontend und Backend gibt. Wird eine Änderung gefunden, lädt Jenkins den neuesten Quellcode, *baut* eine lauffähige Anwendung (*kompilieren, durchführen von Unit-Tests, verpacken zu bereitstellbaren Dateien*) und liefert diese dann an den jeweiligen Server. Das Backend läuft auf einem *Tomcat*-Applikationsserver, das Frontend wird von einem *nginx*-Webserver. Es laufen zwei Instanzen der App, *Test* und *Master*.

Für das fünfte Semester wurde als Server ein *Raspberry Pi*-Bastelcomputer mit Linux-Betriebssystem genutzt. Der *Pi* hat einen relativ schwachen Prozessor und kleinen Arbeitsspeicher, was dazu führte, dass es nicht möglich war, das Frontend auf dem *Pi* zu bauen, da dieser dabei abstürzte. Es wurde stattdessen das Frontend auf einem anderen Computer gebaut und dann *händisch* auf den *Pi* kopiert.

Zu Beginn des sechsten Semesters wurde der von Prof. Dr. Engel bereitgestellte Server in Betrieb genommen, der deutlich leistungsfähiger ist. Dadurch war es nun auch möglich, das Frontend komplett automatisiert bereitzustellen.