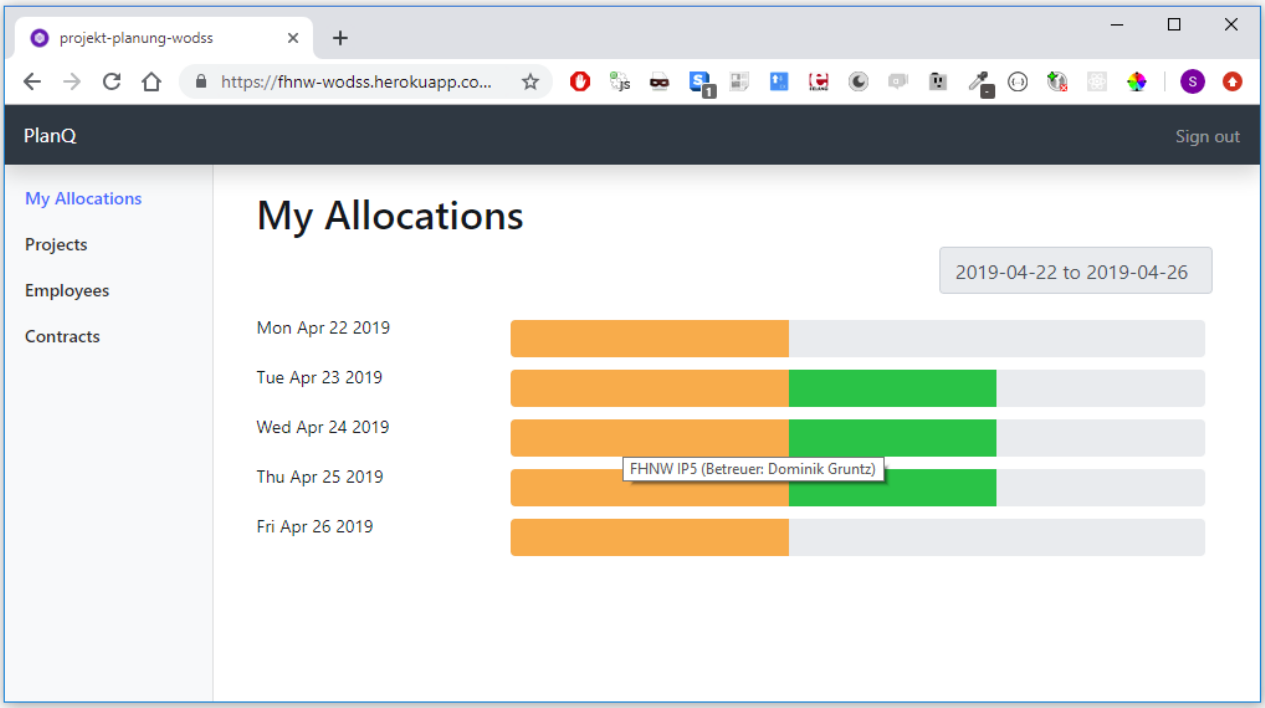
Bericht Workshop (wodss)



Projektname: PlantQ (Planungstool Workshop wodss)
Autoren: Thibault Gagnaux, Philipp Lüthi und Simon Wächter
Version: 2.0.0

Inhaltsverzeichnis

1 Vorüberlegungen

Nach der Vorstellung der Aufgabenstellung in der ersten Woche haben wir uns entschlossen, je eine uns noch unbekannte Technologie im Frontend als auch im Backend zu verwenden (Dies möchten wir gerne als **Bonus** in die Bewertung einfliessen lassen):

- Im Backend hat sich sehr schnell jOOQ als Kandidat herausgestellt. Diese Library, entwickelt durch Lukas Eder aus St. Gallen, betreibt unter anderem ein Reverse Engineering einer SQL Datenbank zur Buildzeit und ermöglicht so typensichere SQL Abfragen auf Basis von JDBC.
- Im Frontend haben wir etwas länger diskutiert: React als solches ist uns bekannt aber verhältnismässig schwergewichtig. Zudem litt es längere Zeit unter Lizenzproblemen, welche aber inzwischen behoben worden sind. Preact preist sich dagegen als leichtgewichtige Alternative zu React an und schien deshalb zu passen. Schlussendlich haben wir uns auf Preact geeinigt, da uns die Dozenten von den Vorteilen etwas Neues auszuprobieren überzeugen konnten.

Wir haben auch noch überlegt, zusätzlich im Backend Spark zu verwenden, haben diesen Gedanken aber wieder verworfen als wir erfahren haben, dass sich unser viertes Teammitglied aus dem Workshop ausgetragen hat und wir somit nur noch zu dritt sind.

2 Technologiestack

2.1 Übersicht

Im Vergleich zu regulären Projektaufgaben musste der Technologiestack definiert werden, bevor überhaupt die Aufgabenstellung genau ausgearbeitet und via Requirements Engineering erfasst werden konnte (Das ist doch recht unüblich). Wir sind rasch bei den uns bekannten und bewährten Tools gelandet:

Architektur:

- Presentation
  - SPA Frontentapplikation auf Basis von Preact (React Klon)
  - React Redux mit Thunk & Log Middleware für das State Management in Preact → Dieser Teil wurde später durch die Vue Applikation der Gruppe 5 ersetzt
  - Spring Web auf Servletbasis[1] mit Spring Security
  - JJWT zur Erstellung & Validierung von JWT Tokens
  - Integrierter Swagger Client zum Bedienen der API
- Business
  - Spring für die Dependency Injection
  - Sonst reguläre & handgeschriebene Businessklassen
- Persistence
  - Generisches CRUD Repository auf Basis von jOOQ + Möglichkeit auf eigene, komplexere Queries ausweichen zu können
  - Mapping der jOOQ Records zu DTO via MapStruct
  - Datenbankmigration via Flyway
  - PostgreSQL 11 als Datenbankserver (Gehostet durch Heroku)

[1] Wir wären auch an einer reaktiven Implementierung auf Basis von Spring WebFlux & Netty interessiert gewesen, haben uns aber dann dagegen entschieden, da wir eine relationale Datenbank auf Basis einer blockierenden JDBC Basis verwenden. Asynchrone & reaktive SQL Libraries wie R2DBC (https://github.com/r2dbc) stecken noch in den Kinderschuhen.

Tooling:

- Sourcecodeverwaltung: Git mit Repository auf GitHub: http://github.com/swaechter/fhnw-wodss
- Buildsystem: Gradle 5.3.1 mit Plugins für Flyway (Datenbankmigration), Node/NPM (Bauen und integrieren des Frontends)

Testing:

- JUnit 5 mit einer lokal in Docker gestarteten PostgreSQL Datenbank (Integration von Test-Container, welche vor den Unit Tests einen Docker Container startet und eine JDBC Verbindung zur Verfügung stellt) für Integration Tests.
- Gemockte Klassen zur Abdeckung der Services

Deployment:

- Java Version: Version 11, nicht Jigsaw modularisiert
- Art: Lokal oder via Docker Container als Cloud-Applikation auf Heroku (Beispiel: http://fhnw-wodss.herokuapp.com)

2.2 Beschreibung der Schnittstelle

Basierend auf den initialen Arbeiten von David und Christian hat Simon die API erweitert und möchte über diese am 18. März abstimmen lassen. Der Kundenwunsch von Herrn König zum Integrieren von Zeitpensen wurde umgesetzt. Spezifikation: https://github.com/swaechter/fhnw-wodss-spec

2.3 Beschreibung Authentifizierung

Die Schnittstelle basiert auf dem JWT Mechanismus, welcher wie folgt abläuft:

1. Client besitzt noch keine Authentifizierung.
2. Client steuert POST /api/token mit einer Emailadresse samt Passwort als Request Parameter an.
3. Der Server verifiziert diese Informationen und stellt ihm ein JWT Token in der Response aus. In diesem JWT Token ist der ganze Mitarbeiter als Claim «employee» integriert.
4. Der Client speichert dieses Token ab (Local Storage).

Der Aufruf einer geschützten Schnittstelle läuft wie folgt ab:

1. Der Client liest das Token aus dem Local Storage.
2. Der Client schickt das Token als HTTP Header «Authorization» im Format «Bearer TOKEN» in der jeweiligen Anfrage mit.
3. Der Server validiert via Filter die Signatur des Tokens und lässt dementsprechend den Zugriff zu.

3 Installation und Inbetriebnahme

3.1 Inbetriebnahme des Back- und Frontend

3.1.1 Überlegungen und Möglichkeiten

Wir haben uns entschlossen, das Projekt via Heroku als Docker Applikation zu betreiben und für andere zugänglich zu machen. Dies resultiert in 2 möglichen Verwendungszwecken:

1. Verwendung unseres Backends unter https://fhnw-wodss.herokuapp.com (**Hinweis**: Die Applikation wird im Free-Tier Modell deployt und bei Nichtverwendung schlafen gelegt. Bei einem späteren Aufruf dauert das Wecken bis zu einer Minute).
2. Eigenes Deployment eines Docker Container via Heroku (Nachfolgend genauer beschrieben).

3.1.2 Variante 1: Verwendung der durch uns gehosteten Heroku Lösung

- Link zur Webapplikation: https://fhnw-wodss.herokuapp.com
- Link zum Swagger Interface: https://fhnw-wodss.herokuapp.com/swagger-ui.html

Anmeldenamen	Passwort	Rolle
simon.waechter@students.fhnw.ch	123456aA	Administrator
philipp.luethi@students.fhnw.ch	123456aA	Administrator
thibault.gagnaux@students.fhnw.ch	123456aA	Administrator

Hinweis: Via Registration können neue Benutzer erstellt werden (Diese müssen aber danach noch durch einen Administrator freigeschalten werden).

3.1.3 Variante 2: Eigenes Heroku Deployment

Vor dem eigentlichen Beginn muss ein Heroku Account erstellt und die Heroku CLI heruntergeladen werden: https://devcenter.heroku.com/articles/heroku-cli

Nach der Installation meldet man sich lokal an:

Anmelden an der Heroku CLI
heroku login -i

Für das Bauen und Hochladen der containerisierten Applikation muss ferner Docker installiert werden.

Nach den Vorbereitungen und einem Git Clone (http://github.com/swaechter/fhnw-wodss) wechselt man in das Projektverzeichnis und erstellt eine neue Heroku Applikation. Der Name „fhnw-wodss“ muss dabei durch einen anderen Namen ersetzt werden, da dieser schon belegt ist (z.B. fhnw-wodss-john):

Erstellen einer neuen Heroku Applikation
heroku create fhnw-wodss

Der erstellten Applikation soll jetzt auch eine PostgreSQL Instanz angehängt werden:

Anhängen einer PostgreSQL Instanz

```
heroku addons:create heroku-postgresql:hobby-dev
```

Nach dem Erstellen der PostgreSQL Instanz müssen wir deren Credentials anzeigen und aufsplitten:

Anzeigen der PostgreSQL Credentials

```
heroku config
```

Der Aufbau der URL ist wie folgt:

postgres://**DATENBANKBENUTZER**:**DATENBANKPASSWORT@DATENBANKHOST:5432/DATENBANKNAME**

Da unser Build und die Applikation diese Credentials benötigen, kopieren wir das Template «config/application.properties.template» nach «config/application.properties» und tragen diese dort ein. Unser Buildsystem als auch die Aplikation selber verwenden untereinander unterschiedliche & inkompatible URL Formate, weshalb die URL zwei Mal zu konfigurieren ist (aber in einem unterschiedlichen Format):

- postgresql statt jdbc:postgresql
- Integration von Benutzername und Passwort in den Link

Eine fiktive Beispielkonfiguration sieht wie folgt aus:

Fiktive Konfiguration config/application.properties

```
# Runtime settings for our Spring application (Required for runtime and development)
PORT=8000
DATABASE_URL=postgres://uvyrwlvnktglsa:44ce2bd0901d80f3be93968c0552e59ea29ddc3a284cbd7edaa20ed224ad1eb4@ec2-54-221-113-7.compute-2.amazonaws.com:5432/d6hg874i8q0qau

# Build time settings for the JOOQ SQL table generation + Flyway data migration (Required for development only)
JDBC_DATABASE_URL=jdbc:postgresql://ec2-54-221-113-7.compute-2.amazonaws.com:5432/d6hg874i8q0qau
JDBC_DATABASE_USER=uvyrwlvnktglsa
JDBC_DATABASE_PASSWORD=44ce2bd0901d80f3be93968c0552e59ea29ddc3a284cbd7edaa20ed224ad1eb4
```

Nach dem Setzen der Credentials kann die Applikation in Docker gebaut und hochgeladen werden (Dieser Prozess dauert beim ersten Mal circa 5 Minuten und danach circa 1 Minute):

Bauen und Hochladen der Applikation

```
heroku container:login
heroku container:push web
heroku container:release web
```

Nach dem Hochladen muss auf eine Applikationsinstanz hochskaliert werden. Das Starten der ersten Instanz kann dabei gleich beobachtet werden:

Setzen der Skalierung
heroku ps:scale web=1 heroku logs --tail

Die Applikation kann nach dem Start auch direkt im Browser geöffnet werden:

Öffnen der Applikation in einem Browser
heroku open

3.2 Inbetriebnahme des Frontends

Diese Kapitel setzt sich mit der Inbetriebnahme des Frontends auseinander und ist für die integrierende Gruppe wichtig.

3.2.1 Klonen des Projektes

Das Frontend befindet sich im Verzeichnis **src/main/javascript** unseres Projektes, welche zuerst einmal geklont werden muss:

Klonen des Projektes
git clone https://github.com/swaechter/fhnw-wodss.git cd src/main/javascript

3.2.2 Anpassen der Backend URL

Nach dem Klonen muss die URL zum Backend in der Datei **src/main/javascript/src/services/config.js** angepasst werden (**Wichtig**: Die URL muss mit einem Slash enden).

3.2.3 Bauen des Frontends

Das Frontend kann wie folgt gebaut werden:

Bauen des Projektes
npm install npm run build

Der Output des Builds landet dabei im Verzeichnis **src/main/resources/public** und würde bei einem regulären Build gleich in die Applikation integriert werden (Spring Boot Applikation mit SPA Applikation und REST Backend). Das Verzeichnis kann in **src/main/javascript/package.json** im Skriptbefehl **build** angepasst werden

Möchte man das Frontend im Development Modus mit Live Reload starten, kann ein **npm run watch** ausgeführt werden.

3.2.4 Aufbau

Die Applikation im Verzeichnis src/main/javascript ist wie folgt aufgebaut:

Verzeichnis	Beschreibung
src/actions src/reducers	Enthält alle Redux Actions & Reducers. Ursprünglich wurde das Adminfrontend separat entwickelt, weshalb es noch gewisse Admin Action & Reducers übrig sind
src/components	Enthält wiederverwendbare Komponenten wie Fehlerdialoge, Navigation und Locks (Locks enforcen eine Aktion, z.B. dass jemand angemeldet sein muss oder andernfalls eine Fehlermeldung/Redirect erhält)
src/pages	Enthält alle einzelnen gerouteten Seiten
src/services	Enthält alle Services mit ihren jeweiligen Funktionen, z.B. zum Anmelden
src/utils	Enthält reine Utilityfunktionen, z.B. für Umrechnungen von Daten

3.2.5 Hinweise

Folgende Punkte sind noch erwähnenswert

- Aufgrund der Verwendung von async & await muss unser Projekt die experimentelle Preact CLI in Version 3 verwenden. Die stabile Version 2 ist inkompatibel.
- Das Token muss den Claim „employee" besitzen, ansonsten kann die Authentifizierung nicht erfolgen → Das Token wird im Local Storage gespeichert.
- Wir haben in unserem Backend CORS Preflights ignoriert, sprich alles akzeptiert → Das Frontend wurde nicht mit striktem CORS getestet.
- Benutzer können sich an unserem Backend registrieren und dann freischalten lassen → POST /api/employee sehen wir als ungeschützten Endpoint an.

4 Lessons Learned und Designentscheidungen

4.1 Gestaltung API & Dokumentation

Wir als Gruppe haben früh versucht, mit den anderen Gruppen eine API zu spezifizieren und haben deshalb vorgeschlagen, dass jede Gruppe eine verantwortliche Person stellt und diese sich in den ersten Wochen treffen. Leider war das Echo sehr gering und ein paar wenige Personen haben zusammen die API via Swagger ausgearbeitet.

Die Gruppe konnte ihre Entscheide aber in einem eigens für das Modul erstellten Slack Channel posten und so wertvolles Feedback erhalten. Es hat sich rasch herausgestellt, dass sowohl die einzelnen Gruppen als sehr wahrscheinlich auch die Dozierenden selbst eine unterschiedliche Vorstellung vom Projekt besitzen. Wir konnten nach mehreren Diskussionsrunden und unter Zuhilfenahme von Herrn Gruntz für eine Diskussionsrunde eine Version spezifizieren und diese dann so einreichen (Quelle: https://github.com/swaechter/fhnw-wodss-spec).

Nach der Eingabe haben wir rasch festgestellt, dass eine reine Dokumentation via Swagger ungenügend ist. Eine Kontextdokumentation, welche die Umgebung und alle Überlegungen dokumentiert, wäre sehr wertvoll gewesen. Da allfällige Fragen aber recht direkt via Slack besprochen werden konnten, haben wir keine weitere Kontextdokumentation geschrieben (Für uns Studierende klappt dieser Ansatz wahrscheinlich, nicht aber für die Dozierenden, welche keinen Zugang zu unserem Slack Channel haben).

Nach dem Schreiben des Backend und Frontend sind wir der Meinung, dass unsere API zu doppeldeutig ist, sprich wir versuchen, mit wenigen Endpoints zu viele Use Cases abzudecken. Ein Aufbohren der /project API wäre eine sehr gute Idee gewesen (Beispiel: /api/project/{id}/[employees | allocations | contracts]).

4.2 Datenbankwrapper jOOQ

Obwohl die API an mehreren Stellen Doppeldeutigkeiten aufwies, konnten wir mit der datenbanknahen Library jOOQ diese elegant behandeln. Eine Auswahl von Problemen:

- Möchte man die aktuellen Projekte eines Developers erfahren, so muss man von der Project Tabelle über Allocations auf Contracts samt ID des Entwicklers zugreifen. Wir möchten nicht wissen, wie sich sowas mit JPA ohne (n + 1)(n + 1) Problem umsetzen liesse, doch war dies in SQL mit jOOQ und einem doppelten JOIN recht einfach lösbar: https://github.com/swaechter/fhnw-wodss/blob/master/src/main/java/ch/fhnw/wodss/webapplication/components/project/ProjectRepository.java#L72
- Ein Projekt, Contract oder Allocation erstreckt sich über eine Zeitspanne, nach welcher gefiltert werden kann. Beim Filtern entstehen fünf mögliche Szenarien:
    - Der jeweilige Datensatz startet vor dem Startfilter und endet in der Periode
    - Der jeweilige Datensatz startet in der Periode und endet nach dem Endfilter
    - Der jeweilige Datensatz startet vor dem Startfilter und endet auch erst nach dem Endfilter
    - Der jeweilige Datensatz startet nach dem Startfilter und endet auch vor dem Endfilter
    - Der Datensatz streift die Periode gar nicht
- Durch das Verkreuzen der Vergleichslogik und dem künstlichen Setzen der beiden Filterdaten bei Abwesenheit (1.1.1900 und 1.1.2100) konnte das Filtering auf die Datenbank ausgelagert werden: (https://github.com/swaechter/fhnw-wodss/blob/master/src/main/java/ch/fhnw/wodss/webapplication/components/project/ProjectRepository.java#L77).

Wir haben ferner die gängigen CRUD Operationen in eine generische Repository-Klasse ausgelagert. Diesem Repository können SQL Filterausdrücke übergeben werden, was die Klasse sehr elegant erscheinen lässt: (https://github.com/swaechter/fhnw-wodss/blob/master/src/main/java/ch/fhnw/wodss/webapplication/utils/GenericCrudRepository.java).

Erst mit solchen Repositories, welche die Datenbankfunktionalität abkapseln, lässt sich eine saubere drei Schichtenarchitektur realisieren. Mit JPA wäre dies so ohne Weiteres nicht möglich, da sich die JPA Entitäten in die Businesslogik hinaufschleichen. Kapselt man diese mit DAOs ab, würde man wieder die Mächtigkeit von JPA künstlich vernichten.

Etwas mühselig an jOOQ hat sich die Integration in die Buildchain erwiesen, da zum Builden des Projektes immer ein PostgreSQL Server vorhanden sein muss. Da wir aber Heroku mit einer angebotenen PostgreSQL Datenbank verwenden, konnten wir dieses Problem elegant umgehen. Um die Datenbank auf dem neusten Stand zu halten, wurde ferner Flyway verwendet, welches die aktuellen Datenbankschemen ausführt und integriert.

4.3 Frontend-Library Preact

Die Preact Library haben wir verwendet, da sie sich mit nur 3 KB statt 45 KB als leichtgewichtiges React «Drop In Replacement» angepriesen hat und wir daran Gefallen gefunden haben. Für allfällige Kompatibilitätsprobleme konnte dabei auf eine 2 KB grosse Library «preact-compat» zurückgegriffen werden, welche weitere, aber schwergewichtigere Kompatibilitätslayers für React anbietet (Namentlich «react-dom», die DOM Manipulationen ermöglicht und worauf viele/alle React Libraries aufbauen).

Wir mussten aber schnell feststellen, dass durch das Fehlen von «react-dom» faktisch keine React Library wie beispielsweise reactstrap (Bootstrap Library, wo jQuery durch React ersetzt worden ist + alle gängigen Bootstrap Komponenten als React Komponenten realisiert worden sind) verwendet werden konnten. Wir haben mit dem Gedanken gespielt, die Library «preact-compat» zu integrieren, sind aber an der momentan noch instabilen Preact CLI in Version 3 gescheitert (Wir konnten aufgrund mehrerer fehlenden Features die stabile Version 2 nicht verwenden und mussten auf die instabile Version 3 wechseln).

Für uns hiess das also, dass wir alle Bootstrap Elemente via Vanilla HTML ausschreiben und stylen mussten (Also keine reactstrap Komponenten verwenden konnten), was sich als sehr mühsam und nicht intuitiv herausgestellt hat. So konnten wir beispielsweise nicht die aus webfr bekannten & sehr eleganten modalen Dialoge aus reactstrap zurückgreifen und mussten diese in eigene Seiten auslagern. Dies hatte zur Folge, dass wir in den letzten beiden Wochen vor Abgabe in einen grossen Zeitstress geraten sind, obwohl wir stetig & aktiv am Projekt gearbeitet haben (In einem kommerziellen Projekt hätte man an dieser Stelle wohl Preact wieder mit React ersetzt).

Durch das Integrieren von Bootstrap samt deren Abhängigkeiten wie jQuery und Popper.js als auch Redux sind wir dann bei einer Bundlegrösse von über 600 KB gelandet (Ohne GZIP). Das von uns gesetzte Ziel, mit Preact eine leichtgewichtige Applikation (Von Redux jetzt einmal abgesehen), konnten wir so nicht erfüllen (Dafür hatten wir ohne weitere Konfiguration PWA Support).

Man darf Preact aber nicht in die böse Ecke stellen und der Library ihre Fähigkeiten absprechen. Ist man sich den Eigenheiten von Preact bewusst und setzt konsequent auf leichtgewichtige Libraries (z.B. redux-zero mit 2 KB anstelle von Redux) und schreibt sein CSS selber (Kein Bootstrap o.ä.) so ist es problemlos möglich, eine Basisapplikation von 5 KB zu schreiben (Mit GZIP).

Nur konnte die Library diese Vorteile in unserem Fall nicht ausspielen, da das Tooling der Preact CLI noch nicht gut genug war.

4.4 Andere Designentscheidungen

Neben den beiden oben genannten Technologien gab es auch diverse weitere Designentscheidungen, auf welche wir rasch eingehen möchten:

- Eine Applikation ohne Registrationsmöglichkeit ist sinnlos, darum haben wir eine Registration gebaut. Der Benutzer muss danach durch einen Administrator freigeschalten werden. Allenfalls müsste der Administrator bei der Benutzererstellung das Passwort des Benutzers kennen und ihm dieses später mitteilen → Gefährlich
- Das Planen und Realisieren einer drei Schichtenarchitektur im Backend hat für viel Klarheit und Ordnung gesorgt.
- Die Möglichkeit, dass ein Service (Business) mehrere Repositories (Persistence) verwenden kann, hat dem ganzen wieder viel Flexibilität verliehen. (Dazu gibt es eine Frage an die Dozierenden am Ende dieses Berichtes.)
- Das Abstrahieren von jOOQ macht es möglich, die Persistenceschicht auszutauschen. (z.B. mit dem MongoDB Java Driver: https://mongodb.github.io/mongo-java-driver/).
- Das Verwenden von Conditional Rendering in Preact (https://reactjs.org/docs/conditional-rendering.html) hat sehr viel vereinfacht: So werden Seiten und Komponenten nur dann gezeichnet, wenn der Benutzer beispielsweise über die benötigte Rolle verfügt (Beispielsweise gewisse Links in der Navigation) oder aber er eingeloggt ist (Sonst wird die Loginpage gezeichnet). Diese Komponenten (Auth Lock, Role Lock, Error) konnten auf den Seiten sehr oft wiederverwendet werden,

Beispiel anhand des AuthLock:

```
@connect(reducers, actions)
export default class AuthLock extends Component {
    render({children}) {
        switch(this.props.auth.loginState){
            case loginState.LOGGED_IN:
                return ({children})
            default:
                return (<LoginOrRegiseterPage/>)
        }
    }
}
```

Samt der Anwendung (Ist der Benutzer nicht angemeldet, wird eben die Login/Registerseite gerendert):

```
render() {
    return (
        <AuthLock>
            <Router history={createHashHistory()}>
                <MyAllocationsPage path='/my-allocation'/>
                <ProjectPage path='/project'/>
```

4.5 Teamarbeit

Wir als Gruppe haben zwar stetig am Projekt gearbeitet, waren aber durch das Fehlen eines 4. Teammitglied im stetigen Zeitdruck. Wir haben rasch erkannt, dass ein funktionierendes Backend ab Woche 3 diverse Vorteile bietet und eine Implementierung des Frontend ansonsten nur sehr mühsam möglich wäre. Wir waren aber von der in den Anforderungen nicht ersichtlichen Komplexität beeindruckt, doch konnten diese durch jOOQ recht gut abfedern (Wenn ich ein SQL Query dafür schreiben kann, so funktioniert es auch in jOOQ). Im Backend sind diverse interessante, aber auch sehr zeitintensive Diskussionen entstanden. Generell kam es immer zu einem Konflikt zwischen «Wir sind im Workshop und würden gerne mal noch X ausprobieren» versus «Wir sind generell schon recht unter Zeitdruck, lass lieber vorwärtsmachen».

Als wir dann nach Abschluss des Backends alle Kräfte auf das Frontend verlagert haben und feststellen mussten, dass wir reactstrap nicht verwenden konnten, sind wir in eine grössere Zeitknappheit geraten und mussten diverse Überstunden und Nachtstunden schieben. Wir konnten somit dem Frontend nicht auch nochmals die Liebe & Energie geben, welche auch schon unser Backend erhalten hat, was sehr schade ist. Von dem her sind wir der Meinung, dass die erste Phase des Workshops verlängert oder umgebaut werden sollte.

5 Integrationsaufwand

5.1 Ablauf

Wir als Gruppe 6 haben das Frontend der Gruppe 5 übernommen und integriert. Die Integration lief wie folgt ab:

1. Klonen des Repositories: https://github.com/837/WODSS2019_Einsatzplanung
2. Kopieren ihres Frontends Frontend/group5-client in unser Projekt unter src/main/javascript2
3. Installieren aller Pakete via npm i
4. Starten des Frontends via npm run serve

5.2 Problem 1: Falsche Portangabe

Nach dem Starten stellte sich rasch heraus, dass ihr Backend auf einem anderen Port lief als unseres. Eine rasche Änderung des Ports in der entsprechenden Environment Datei behob das Problem. Keine grosse Sache, aber diese Änderung wurde nachher in die Installationsanleitung der Gruppe 5 übernommen.

5.3 Problem 2: Anderer Aufbau des Employee Claim

Nach dem erfolgreichen Ansprechen des Backends und der Möglichkeit, sich anmelden zu können, traten im Frontend der Gruppe 5 mehrere Anomalien auf:

- In der Navigation war der Vor- und Nachname undefined.
- Diverse Buttons für den Administrator zum Erstellen von Projekten/Mitarbeiten/Verträgen waren nicht sichtbar.

Auf dem Backend der Gruppe 5 traten diese Probleme nicht auf, sodass schnell klar wurde, dass ein Problem mit dem Token, respektive dem nicht genau spezifizierten Aufbau mit dem Claim «Employee» bestand. Das Base64 dekodierte Token des Backends der Gruppe 5 sieht wie folgt aus:


```
{
    alg: "HS256",
    typ: "JWT"
}.
{
    rawPassword: "1234",
    lastName: "admin",
    firstName: "admin",
    active: true,
    emailAddress: "admin@admin.ch",
    id: "ALrz3FOsesFL8bqDFHsG",
    role: "ADMINISTRATOR",
    iat: 1559317051,
    exp: 1559403451
}.
[signature]
```


Während das dekodierte Token unseres Backends wie folgt aussah:


```
{
    alg: "HS512"
}.
{
    iss: "FHNW wodss",
    sub: "simon.waechter@students.fhnw.ch",
    employee: {
        id: "6bae2bc7-9d1e-457d-ac92-dd78e467b7d1",
        firstName: "Simon",
        lastName: "Wächter",
        emailAddress: "simon.waechter@students.fhnw.ch",
        role: "ADMINISTRATOR",
        active: true
    },
    iat: 1559317301,
    exp: 1559319101
}.
[signature]
```


Anhand des unterschiedlichen Aufbau war klar, dass das Konzept von Claims in JWT unterschiedlich verstanden worden war (Sie wurden ja nicht spezifiziert, sondern in Slack einfach mal diskutiert): Die Gruppe 5 verwendete mehrere Claims für die einzelnen Attribute (Zudem fehlt das Subject und das Passwort sollte nicht mitserialisiert werden), wo hingegen wir das Benutzerobjekt als einen einzigen Claim definieren (So wie es ein Grossteil aller Gruppen auch verstanden hat).

Durch das Anpassen des Logincode (Commit https://github.com/swaechter/fhnw-wodss/commit/281e38349388c4da37cc0dd895e63e0c75ba9406) war aber auch dieses Problem in wenigen Minuten gelöst & alles hat zufriedenstellend funktioniert.

5.4 Fazit: Warum verlief die Integration so gut?

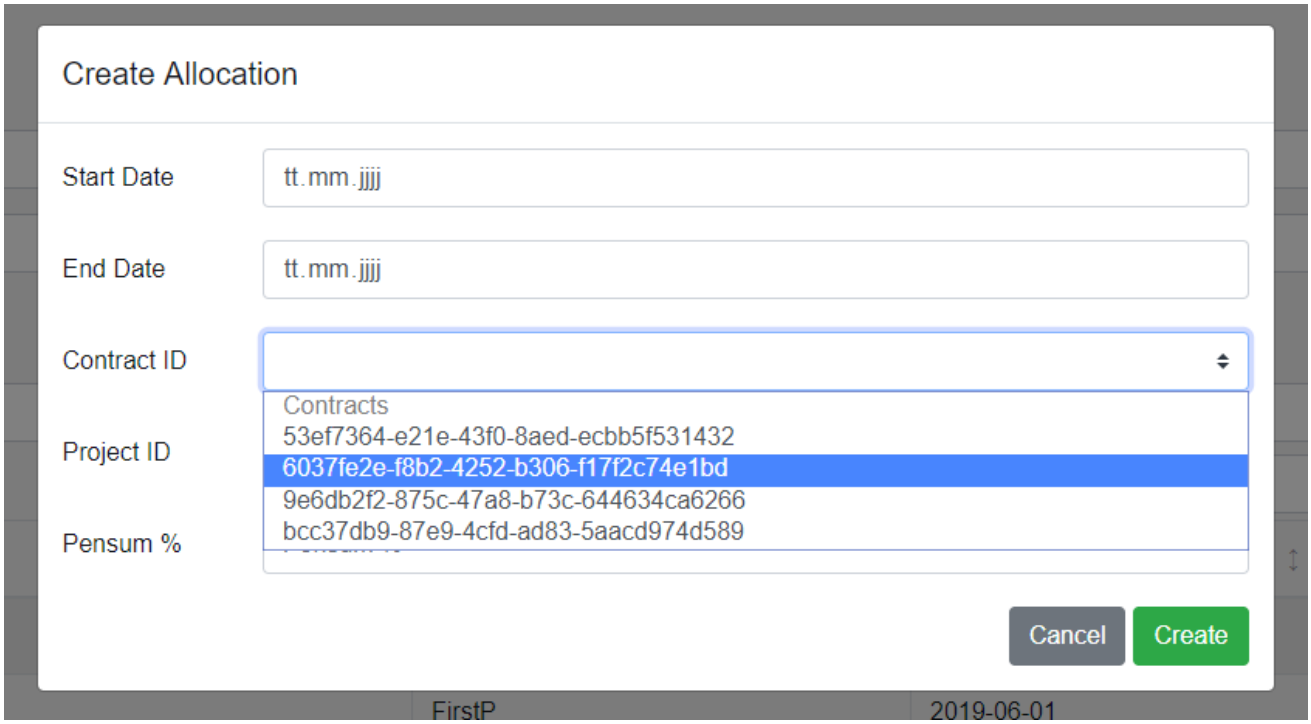
Im Grossen und Ganzen kann man also sagen, dass die Integration faktisch problemlos über die Bühne ging. Wir als Gruppe haben zusammengezählt weniger als eine einzige Stunde für die Integration aufwenden müssen, wovon wir von mehreren Gruppen beneidet worden sind.

Der Grund, warum die Integration so gut lief, lässt sich anhand einer Tatsache festmachen: Unser Backend generiert mithilfe von «Springfox» (https://github.com/springfox/springfox) und Annotationen in unserem Java-Code eine Swagger-API, welche direkt als Spezifikation verwendet werden kann und auch wurde. Die Swaggerausgabe unseres Backendes ist also die Spezifikation auf https://github.com/swaechter/fhnw-wodss-spec (Wenn wir als Gruppe schon Zeit in die API investieren und pushen müssen, so wollen wir auch etwas davon haben).

Die Gruppe 5 hat auf Basis dieses Swaggercodes ihr Front- und Backend generieren lassen, sodass viele händische Fehler/Vertipper oder schlicht Missverständnisse (URI oder in Body) schon im Vorhinein verhindert werden konnte (Sozusagen Backend = Spezifikation).

5.5 Empfehlung: Was könnte man noch besser machen?

Es ist jedoch schade, dass die Gruppe 5 wie wir selbst, nicht alle UI Funktionalitäten [sauber] umsetzen konnte und an mehreren Stellen auf IDs für die Ein- und Ausgabe zurückgegriffen werden muss (Anstatt die auswählbaren Elemente zu laden und anzuzeigen):



6 Verbesserungsmöglichkeiten für uns

Im Verlauf des Projektes und auch während den ersten Präsentationen haben wir festgestellt, dass wir folgende Sachen hätten besser machen können:

- Wir haben vergessen, den Dozenten einen Zugang zu unserer Instanz zu geben (Dies wurde nachträglich noch nachgeholt).
- Wie von der Gruppe 1 erwähnt haben wir durch Nichtwissen eines Teammitgliedes die package-lock.json auf die .gitignore gesetzt und diese nicht commitet. Bei der späteren Übergabe an die Gruppe 1 funktionierte dann das Frontend auf einem System, auf anderen dann aber wieder nicht (Preact ist leider nicht sehr stabil).
- Basierend auf dem oben genannten Fakt hätten auch alle Versionen in der package.json ohne ~ (Use most recent patch version) und ^ (Use most recent minor version) ausgeschrieben werden müssen, da sich im JavaScript Ökosystem leider die wenigsten Library-Autoren an semantische Versionierung halten. Das schränkt uns zwar aus Patchsicht ein, garantiert aber die langfristige Integrität von Abhängigkeiten.
- Generell mehr Zeit für das Frontend einplanen, um besser mit den Nachteilen von Preact umgehen zu können.

7 Fazit

Wie eingangs erwähnt war der Ablauf des Workshops für uns eher ungewohnt: Sich zuerst auf eine Technologie festlegen und erst dann die Anforderungen genauer ausarbeiten? Tönt riskant und ist es auch, doch hat es bei uns gerade nochmals funktioniert. In Zukunft würden wir aber hinterfragen, ob wir trendige und interessante Libraries doch wirklich verwenden wollen oder ob der versprochene Mehrwert nicht doch trügerisch ist. Genau so hätten wir in den letzten beiden Wochen in der Realisierung des Frontends die Energie verlieren und Schiffbruch erleiden können.

8 Fragen an die Dozierenden

Da die Dozierenden während des Unterrichts nicht anwesend waren und wir diverse interessante Diskussionen hatten, sind bei uns mehrere Fragen aufgetaucht, für welche wir durch die Dozierenden gerne noch Feedback einholen wollen:

1. Validierung versus. Businesslogik: Was gilt noch als Validierung und was schon als Businesslogik? Kann ich auf Service Ebene davon ausgehen, dass die Argumente (z.B. ProjectDto, EmployeeDto) valid sind oder müssen hier immer noch Validierungschecks (Null Check, Range Check etc.) gemacht werden? Wie entschärft man dieses generellen Cross-Cutting Concern?
2. Beziehung Service zu Repository: Macht es Sinn, dass ein Service strikt nur sein eigenes Repository aufrufen darf? (Für benötigte Daten von anderen Repositories würde das bedeuten, dass der Service einen anderen Service aufruft (ProjectService → EmployeeService → EmployeeRepository). Hat es mehr Vorteile, wenn ein Service mehrere Repositories aufrufen darf? (ProjectService → EmployeeRepo)? Wie geht man mit zyklischen Abhängigkeiten zwischen Services um?
3. Gibt es in unserem Falle elegantere Designpattern zu unserem „Early Return“ in den Service Methoden? Wir führen eine Aktion erst aus, wenn davor keine Exception einen Abbruch bewirkt hat (Herr Wächter mag sich erinnern, dass Herr Gruntz dieses Pattern in depa bemängelt hat.)
4. Was sind gute Ansätze, eine Datenbank nach einem Testcase wieder zurückzusetzen, sodass der nächste Testcase wieder mit einer frischen, respektive nicht beeinflussten, Datenbank starten kann? Wir wollten das Spring Transaction Management in unsere Unit Tests und jOOQ Konfiguration einbauen, um nach dem Abschliessen ein Rollback ausführen zu lassen, konnten dies aber aus Zeitgründen nicht mehr implementieren. Als Workaround haben wir eine PostgreSQL Prozedur geschrieben, welche die Daten löscht und die Grunddaten wieder erstellt (Wir rufen diese Prozedur dann nach jedem Unit Test auf). Startpunkt für Spring & jOOQ: https://www.petrikainulainen.net/programming/jooq/using-jooq-with-spring-configuration/
5. Wie kann in Redux einfach gesagt werden, wann die Daten einer Komponente geladen sind und sie sich final rendern kann? Man kann zwar auf einen Loading-State zurückgreifen, doch skaliert dieser bei mehreren API Calls nicht (Mehrere High-Low Loading Bewegungen hintereinander)
6. Wie kann die Explosion an Anzahl von Reducern und Actions in Redux verhindert werden?

9 Quellen

- Sourcecode: http://github.com/swaechter/fhnw-wodss
- Spezifikation: https://github.com/swaechter/fhnw-wodss-spec
- Testinstanz: http://fhnw-wodss.herokuapp.com
- Beschreibung Gratisinstanz Heroku: https://www.heroku.com/free