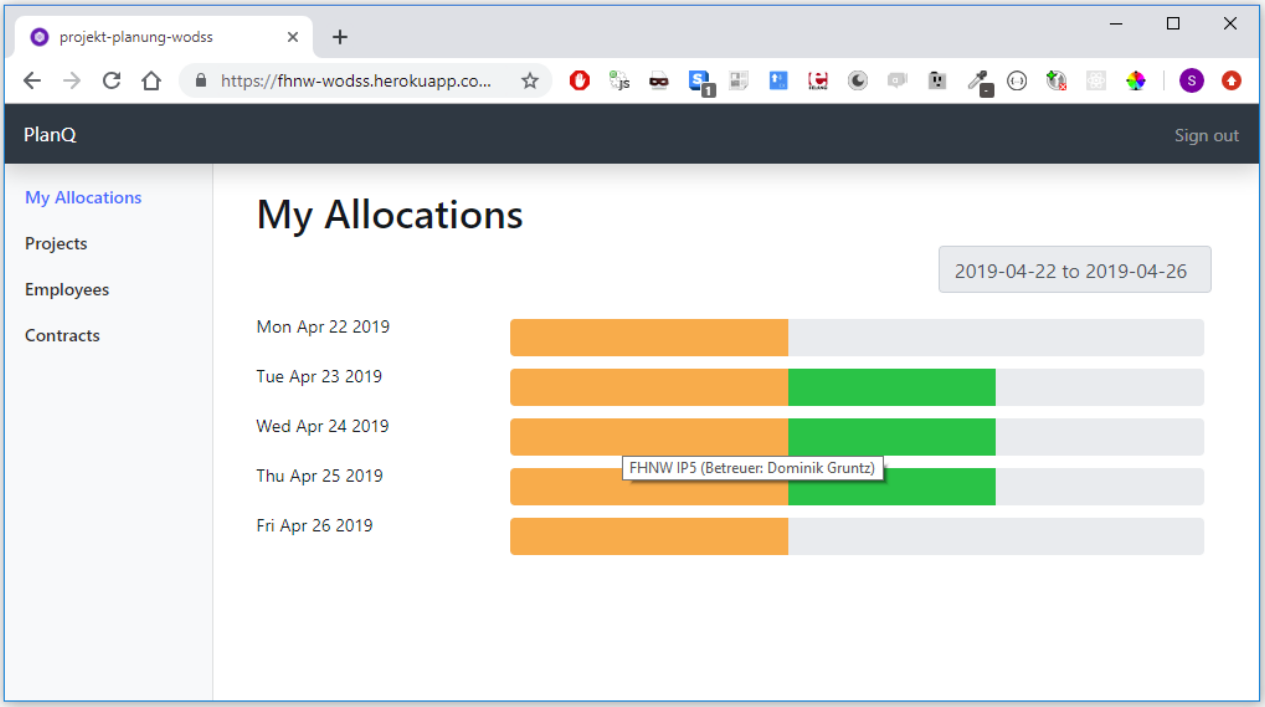
Bericht Workshop (wodss)



Projektname: PlantQ (Planungstool Workshop wodss)
Autoren: Thibault Gagnaux, Philipp Lüthi und Simon Wächter
Version: 1.0.0

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 Backend zu verwenden:

- 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 passsen. 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

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
  - 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 TestContainer, 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)

3 Lessons Learned

3.1 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])

3.2 Datenbankwrapper jOOQ

Dadurch das die API an mehreren Stellen Doppeldeutigkeiten aufwies und wir eine datenbanknahe Library wie jOOQ verwendet haben, konnten wir recht einfach und elegant auf diese Schwierigkeiten eingehen. 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)

Nachteilig 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.

3.3 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.

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.

3.4 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.

4 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 & interessante Libraries doch wirklich verwenden wollen oder ob der versprochene Mehrwert nicht doch trügerisch ist (Eigentlich genau das Gegenteil, was Herr König erreichen wollte). Genau so hätten wir in den letzten beiden Wochen in der Realisierung des Frontends die Energie verlieren und Schiffbruch erleiden können.

Für die weitere Ausführung des Workshops würden wir uns wünschen, dass die Ziele und Erwartungen im Vorfeld bekannt (dazu noch mehr in der Präsentation).

5 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 würden:

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 anderern 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 z
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?

6 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