CONFINALE
Digital Banking applied

Confinale Übungsaufgabe

1 Summary

Dieses Dokument begleitet die Übungsaufgabe, welche mir im Rahmen eines Assessment seitens Confinale Ende Juli 2019 gestellt wurde. Das Dokument beinhaltet ergänzende Informationen, welche gegebenenfalls aus dem Quellcode nicht direkt ersichtlich sind, Designüberlegungen und soll insgesamt dazu dienen, meine Herangehensweise realistisch beurteilen zu können.

Wie am Kennenlerngespräch erwähnt, verfügte ich bei der Übergabe der Aufgabe über *keinerlei* Skills im Bereich Angular und Spring Framework. Die *technische Umsetzung* ist daher nur bedingt repräsentativ und würde mit vertieften Kenntnissen der Frameworks sicherlich anders ausfallen (insbesondere Abstrakter und in mehreren Views!). Das *Vorgehen* bei einem Entwicklungsprojekt halte ich jedoch für stehts ähnlich – was auch hier zutrifft, also: Requirements Engineering, Grobkonzept, Abnahme vom Kunden, Iterative Umsetzung (beispielsweise nach SCRUM).

Aufgrund meiner hohen beruflichen und privaten (wir erwarten in Kürze Nachwuchs) Belastung, hatte ich die Aufgabe hauptsächlich nachts und über mehrere Tage verteilt zu lösen.

Hiermit bestätige ich, Florian Dürr, geb. 17.3.1981, dass ich die Aufgabe ohne fremde Hilfe, welche mir nicht auch sonst zur Verfügung stehen würde (Google, Stackoverflow, Youtube, Technet, Spring.io…), bearbeitete und die unten angegebenen Zeitdauern für die Teilaufgaben korrekt (+/- 10%) sind.

2 Aufgabenstellung

Programmiere ein WG-Einkaufsprotokoll, welches die folgenden Features haben sollte (sortiert nach absteigender Priorität der Features):

1. MOCKUP: Benutzer der Apps können Einkäufe (Name des Einkäufers, Name des Produktes, Datum, Preis) abspeichern (Unklar: soll der Benutzer beliebige andere Einkäufer «simulieren/übernehmen» können, oder ist ein «1 Benutzer» gleichbedeutend mit «1 Einkäufer», sprich: sollte nicht eigentlich ein Benutzer nur *seine eigenen* Einkäufe mutieren können? → Zielgruppe der App definieren! Ferner: Datum des Einkaufs oder Datum des Produkts?) → in einem «richtigen» Projekt würde ich diese und weitere (siehe unten) Punkte mit dem Kunden klären und vor der ersten Zeile Code das Mockup mit dem Kunden besprechen.
2. SOLVED: Im Frontend wird das Einkaufsprotokoll (also die Liste aller erfassten Einkäufe) angezeigt
3. SOLVED: Gespeicherte Einkäufe können (einzeln) gelöscht werden.
4. SOLVED: Beim Einkaufsprotokoll wird die Summe der Preise aller getätigten Einkäufe angezeigt.
5. PARTIAL: Gespeicherte Einkäufe können (einzeln) bearbeitet werden.
6. SOLVED: Beim Einkaufsprotokoll wird die Summe der Einkäufe pro Einkäufer angezeigt, am liebsten in einem Kuchendiagramm. (Hier würde ich dem Endkunden empfehlen, dies in eine eigene Dashboard-View auszugliedern – siehe Mockup – um die Liste nicht zu überfrachten)
7. TODO: Eine Funktion soll es ermöglichen, Einkaufssummen-Differenzen zwischen Einkäufern auszugleichen. Dazu sollen die entsprechenden Beträge, die ein spezifischer Einkäufer einem anderen Einkäufer abgeben muss, angezeigt werden und bei Bestätigung die entsprechenden Einträge ins Einkaufsprotokoll gesetzt werden. (Unklar: was ist mit «abgeben» gemeint? Die Businesslogik einer derartigen Funktion erschliesst sich mir nicht. Siehe Mockup – meine Interpretation ist vermutlich nicht korrekt)

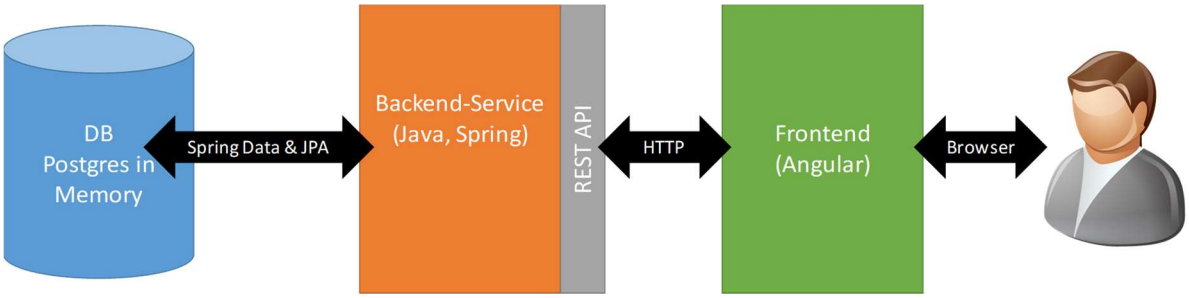
(Optional: Um es dir etwas einfacher zu machen kannst du auf das User-Handling verzichten: D.h. Benutzer müssen sich nicht einloggen und alle Benutzer der App dürfen alle Einträge sehen, löschen und bearbeiten.)

2.1 Erwartetes Resultat

1. Einen Link zu einem Git Repository (bei einem beliebigen Hoster) mit *kompilierbarem* Code (→ GitHub)
2. Ein Dokument, in welchem du deine Arbeit kurz beschreibst: Wo gab es Probleme? Womit würdest du weitermachen? Was gefällt dir nicht an deinem Code? (→ dieses Dokument)
3. Testcode (→ UnitTests)

3 Architektur

3.1 High-Level



3.2 View(s)

Abbildung 1: Home-View

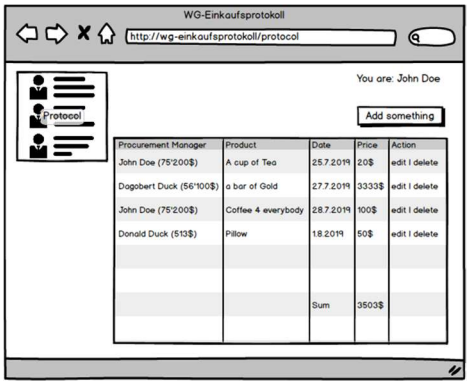


Abbildung 2: Protocol-View

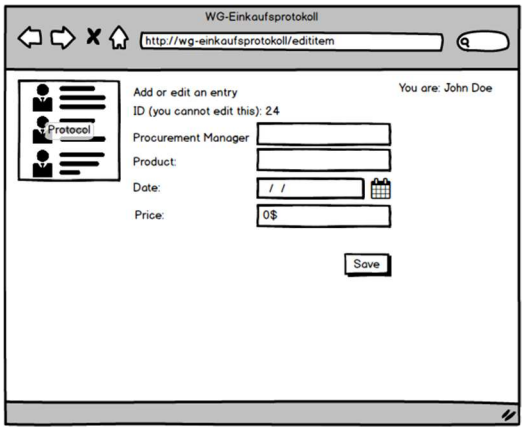


Abbildung 3: Edit an Item from Protocol-View

Abbildung 4: Dashboard-View

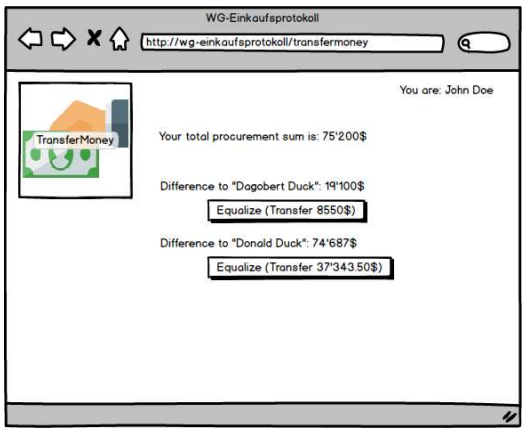
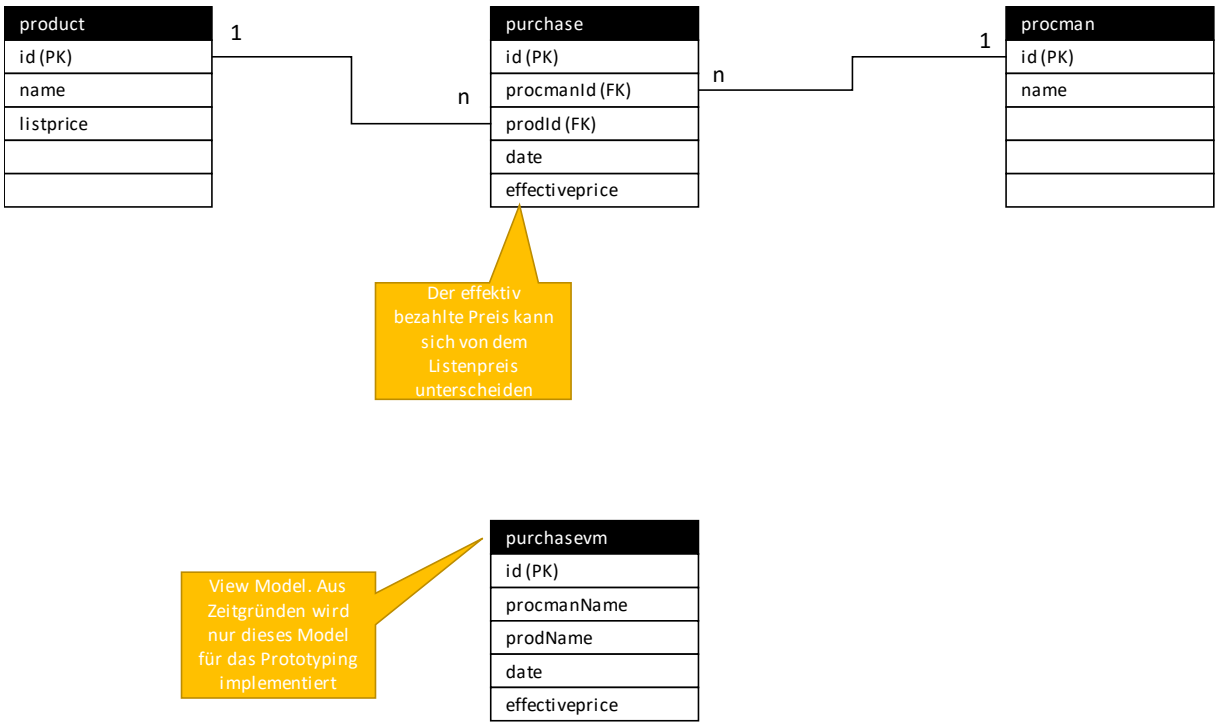


Abbildung 5: Transfer Money-View

3.3 Datenmodell



3.4 REST API

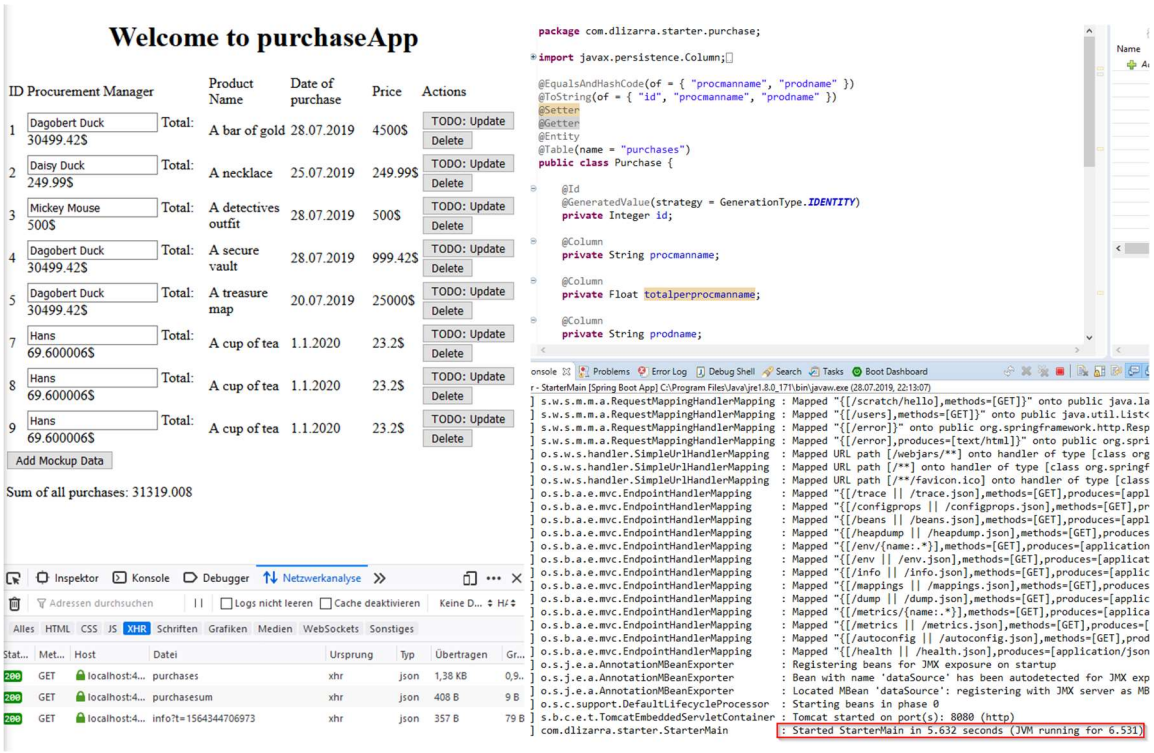
API	HTTP Methode	Pfad	Beschreibung
GET purchases	GET	/api/purchases	Eine Liste aller Einkäufe als JSON zurückgeben
POST purchase	POST	/api/purchases	Einen neues Einkaufsobjekt anlegen
GET purchase	GET	/api/purchases/{id}	Ein bestimmtes Einkaufsobjekt als JSON zurückgeben
PUT purchase	PUT	/api/purchases/{id}	Ein bestimmtes Einkaufsobjekt aktualisieren
GET purchase	GET	/api/deletepurchase/{id}	Ein bestimmtes Einkaufsobjekt löschen

4 Codeanalyse

- Offenbar wird Jackson verwendet um JSON für die REST API zu generieren → kurz einlesen insb. wegen der Syntax. Es werden offensichtlich POJOs automatisch per Annotation gemarschalt… ob zu Runtime oder Compiletime ist mir nicht klar: OK
- @RequestMapping stellt eine Route für einen Http-Call zur Verfügung : OK
- Wo ist die Datenbank?! Ich finde lediglich ein SQL-File, welches offenbar @Boot in eine Datenbank geladen wird. → postgres In Memory?

5 Selbstbeurteilung der Lösung

5.1 Screenshots



5.2 Aufgetretene Probleme

- Im Vorfeld (Out-Of-Scope): Setup der Entwicklungsumgebung bereitete Schwierigkeiten aufgrund der nicht-defaultmässig verarbeiteten Annotationen von Lombok. → SOLVED
- Konstant Schwierigkeiten und Zeitverlust bei diversen Stellen. Das «Was» war klar, aber das «Wie» und «Wo» nicht…

5.3 Nächste Schritte/Userstory bei entsprechendem Budget

- Komplettes Refactoring gemäss untenstehender Code-Kritik
- Normalisierung und Abstrahierung der Modelle

5.4 Selbstkritik am Code

- Bug: nach Löschung eines Purchase-Objektes muss die View manuell neu geladen werden um die Summenfunktionen neu zu triggern
- Die Mutationen am Purchase-Objekt sollten in einem eigenen Service implementiert werden
- Alle Daten in der View sollten pro View aus *einem* ViewModel kommen und nicht mit dem Model vermischt sein
- Die Update-Funktion übermittelt die Werte als Parameter, was hässlich und unsicher ist. Das gesamte Purchase-Objekt sollte als JSON übermittelt werden
- Das Datum sollte ein Datumsformat und kein String sein (inkl. Validierung)
- Die Summenfunktionen sollten in SQL und nicht in Code implementiert sein (insb. die Summenfunktion pro Einkäufer ist sehr schlecht implementiert → das geht in JPA garantiert besser)
- Backend und Frontend sollten separate Projekte sein
- Testcode! (In Anbetracht der vielen Neuigkeiten habe ich auf TDD verzichtet)
    - Die View ist hässlich 😉 → Twitter Bootstrap würde sich gut machen
    - Outputstring-Formatierung ergänzen

6 Arbeitsjournal

Datum	Von - Bis	Task	Dauer
21.7.2019	17:00 bis 23:00 (mit Pause)	Setup Entwicklungsumgebung in vier Betriebssystemen (W10, Ubuntu, Kubuntu, Mint). Da Out-Of-Scope der Zeitdauer, investierte ich hier aus persönlichem Interesse relativ viel Zeit.	5:50 (Out-Of-Scope)
23.7.2019	19:00 bis 22:00	Nachbesserung 4 x Entwicklungsumgebungen. Auf allen Linux-basierten Systemen Original-JDK installiert (ziemlich aufwendig). Aus persönlichem Interesse viel Zeit investiert. Email-Korrespondenz mit Kevin W.	3:00 (Out-Of-Scope)
25.7.2019	22:45 bis 00:00	Erstellung dieses Dokuments. Lesen und Verstehen der (eigentlichen) Aufgabe. Wissenslücken identifizieren. Mockup von Architektur, View und Model erstellen (Quick & Dirty)	1:15 (In-Scope)
26.7.2019	18:00 bis 18:15	Datenmodell definiert	00:15 (In-Scope)
26.7.2019	20:00 bis 21:45	Code-Analyse mit Fokus auf Architektur und Annotationen für Angular. Das Java-Backend ist relativ klar (ausser wie das SQL-File geladen wird), das Angular-Frontend schwierig → ich spiele eine Hello-World App durch, ggf. auch eine CRUD-App. Aufgrund der Zeitbeschränkung keine weitere Analyse.	01:45 (Unklar ob in Scope oder nicht)
28.7.2019	10:00 bis 14:00	Basisinformationen über zu verwendende Technologie beschafft: Spring, Angular, JPA, Lombok. Einige Hello-World-Programme erstellt. → es wird insgesamt empfohlen, Backend und Frontend in separate Projekte zu trennen.	04:00 (Unklar ob in Scope oder nicht)
28.7.2019	15:30 bis 17:45	Model für purchase im Frontend und Backend (teilweise) umgesetzt und Einzellöschfunktion implementiert -> funktioniert.	02:15 (In-Scope)
28.7.2019	20:15 bis 22:00	Summenfunktionen (quick & dirty!) implementiert. Dokumentation bereinigt.	01:45 (In-Scope)
		Total (In-Scope)	**5:30**