Dokumentation Gruppe 35

https://github.com/hhu-propra2-ws22/projekt_gruppe_35

Einführung und Ziele:
Ziel dieser Anwendung ist es, unter einer Gruppe von Personen Ausgaben und Schulden minimal auszugleichen.

Lösungsstrategie:
Da wir auf eine Onion-Architektur eingeschränkt sind, ist unsere Lösungsstrategie eine Dekomposition der Architekturkomponenten in Domain, Service, Web und Persistenz. Zur Kollaboration haben wir CodeWithMe von Jetbrains benutzt, was zu inkonsistenten gitHub-Namen im Code führt. Zudem mussten wir deswegen anfangs auch Commits auf mehrere Commits aufteilen, die wir dann nummeriert haben (man muss diese also als einen zusammenhängenden Commit sehen).

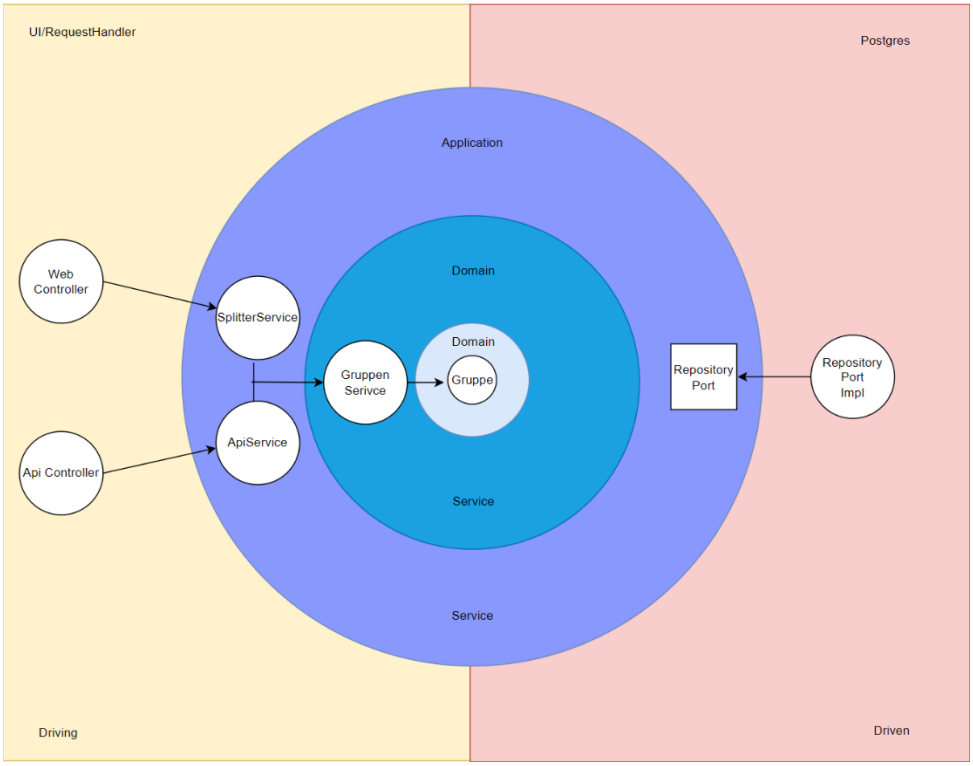
Komponente:
Domain:
Als einziges Aggregat haben wir uns für ein Gruppen-Objekt entschieden.
Dieses besitzt die zwei Value Objekte Transaktion und Ausgabe. ‚Ausgabe' sind die vom User eingetragenen Ausgaben, und Transaktion sind der Anwendung ausgerechneten minimalen Ausgleichungen.

Domain Service:
Rechnet minimale Transaktionen aus, erzeugt neue Gruppen, fügt Mitglieder hinzu und rechnet Schulden aus.

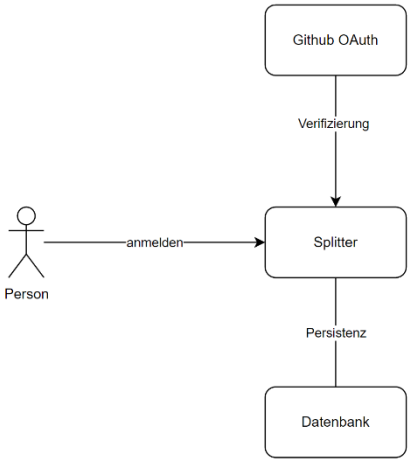
Application Service:
Ruft den Domain Service auf, kommuniziert mit der Datenbank und wird vom Controller aufgerufen.

Controller:
Sorgt für das UI und RequestHandling der Webanwendung, und nimmt Eingabe entgegen.

Persistenz:
Wird vom Service durch den Port aufgerufen, transferiert Objekte in DTO's und speichert diese in der Datenbank ab.

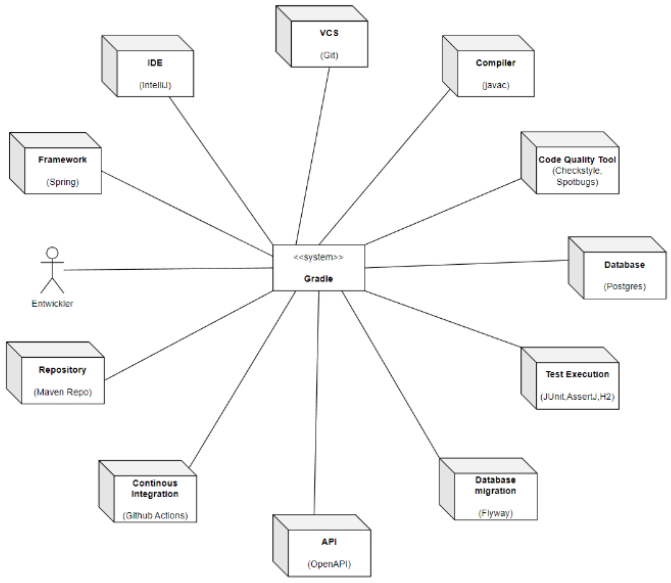


Interaktion mit der Anwendung soll wie folgt aussehen:
Eine Person soll sich mit Github auf der Seite einloggen,
diese wird von der Anwendung verifiziert
und die Person darf die Anwendung nutzen.
Dessen Eingaben und intern ausgeführte Rechnung
werden in der Datenbank persistiert.



Runtime:
Es werden alle Fehler auf der Seite abgefangen und entsprechende Error Nachrichten ausgegeben.
Die Anwendung verwendet zwei eigene Exceptions.

Deployment:
Die Anwendung basiert auf folgenden Komponenten:



Konzepte:
Es wird die Onion-Architecture verwendet.
Wir verwenden das DDD-Pattern in der Domain.
Genauso verwenden wir des Ports und Adapters Pattern im Application Service mit der Persistenz.
Die Datenbank basiert auf Postgres und verwendet das DAO-Pattern.
Die Architektur wird anhand ArchRules festgelegt und aufrechterhalten.
Das DAO-Pattern wird auch in der API für Json verwendet.

<u>Architektur Entscheidung:</u>
Domain:
Wir haben uns dazu entschieden die Person nicht als einzelnes Objekt anzulegen, da diese bei uns nur einen Github Namen speichert und sonst keine weitere Logik beinhaltet. Des Weiteren ist es leichter eine Person als String zu persistieren und zu referenzieren. Wir sind uns bewusst, dass dies Primitive Obsession ist.
'Transaktion' und 'Ausgabe' sind Value-Objekte, da diese immutable sind, und nur ersetzt werden.
Service:
Wir haben uns dazu entschieden Domain- und Application-Service zu verwenden, statt einen einzelnen Service, um Wartbarkeit und Leserlichkeit aufrecht zu erhalten.
So gelingt es uns z.B. einen API-Service anzulegen, ohne die Logik der Domain zu verändern.
Controller:
Wir haben zwei verschiedene Controller für die Webanwendung und API angelegt, und dabei die API mit @RestController annotiert, da diese nur Json-Formate ausgeben soll.
Persistenz:
Wir verwenden jeweilige Debitoren- und Mitglieder-DTO's, um diese abgegrenzt voneinander in der Datenbank zu persistieren.

<u>Risiken und Technische Schulden</u>:
AusgabeDto: @Id-Annotation aufgrund einer 1:n Beziehung in der Datenbank, da Postgres von einer Entität ausgeht. Wir sind uns bewusst, dass AusgabeDto keine @Id-Annotation haben darf, da es kein eigenes Aggregat repräsentiert.

Service/Controller Konstruktor: @SuppressFBWarnings(value=„EI_EXPOSE_REP2“) Annotation an den Konstruktoren, da Spotbugs die statischen Services als Objekte behandelt, und dadurch denkt, dass interne Repräsentation preisgegeben wird.

Domain: Person wird weder als Aggregat noch als Value Objekt behandelt, da sie nur einen String speichert und sonst keine Logik besitzt. Wir sind uns über Primitive Obsession bewusst.

Webcontroller Methode addAusgabe: Wir lesen die Parameter selbst aus der ServletRequest aus, ohne diese in der Form zu validieren, da die Validierung die entsprechenden Objekte selbst betrifft und somit nur im Service möglich ist.