

ABSCHLUSSPRÄSENTATION EPRO WS21/22

Konzeption und Entwicklung einer API für das OKR-Framework

Unser Team:

- Mario Golawsky
- Thomas Hormes
- Franck Ngami

Inhaltsverzeichnis



- I. Tooling
- II. Analyse und Design
- III. Implementation
 - a. Datenbank
 - b. Security
 - c. Modell, Controller, Services
 - d. HATEOAS
 - e. Unit & Integration Tests
- IV. Fazit
- V. Live Demo

Tooling

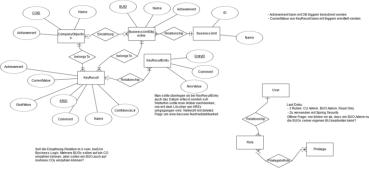


- Postgres als DB Dockerized
- Intellij Ultimate als IDE
- Hoppscotch als HTTP Client
- Git (Github) als VCS
- Testcontainers für Integration Tests
- Maven als Buildtool

Analyse und Design



- Zunächst Datenmodell erstellt
- Anhand des Datenmodell weiteres Vorgehen geplant
- Ziel: möglichst klare Trennung zwischen Business Logik, Präsentationslogik und Datenbankzugriff



Implementation – Datenbank



- Verwendung von PostgreSQL
 - Meiste Erfahrung mit diesem DBMS im Team vorhanden
- Entwicklung DB-Modell nach erarbeitetem ER-Diagramm
- Möglichst viel Datenverarbeitung in der Datenbank
 - Achievement von Objectives und KeyResults wird mit generated Columns berechnet
 - Trigger zur Historisierung

Implementation - Security



- Erster Versuch mit Basic Auth dann Wechsel zu Authentifikation mit JWT
- Sehr einfache Authentifikation mittels JWT über mehrere Requests nun möglich
- Umsetzung von Authentifikation mittels JWT gestaltet sich schwerer
 - Dokumentation schwer verständlich/teilweise "dünn"
- Umsetzung Autorisierung gestaltete sich leichter
 - @PreAuthorize mit SpEL und Custom Expressions m\u00e4chtig und leicht zu verwenden

Implementation – Modell, Controller, Services



- Ziel: Trennung zwischen Business Logik, Präsentationslogik und Datenbankzugriff
- Datenbankzugriff über Repositories, welche @Entity Klassen zurückgeben
- Präsentationslogik wird über @RestController abgebildet
- @Services implementieren die Business Logik

Implementation – Modell, Controller, Services



- Um zu vermeiden zu viele Daten preiszugeben, werden Data Transfer Objects (DTO) benutzt
- Mapping zwischen Entity und DTO mithilfe von Mapstruct
- Bei kleinen Entities existiert nur ein DTO
- Bei größeren Entities auch DTOs, die nur für POST oder PUT sind
 - Einfachere Validation
 - Einfache Möglichkeit sicherzustellen, dass bestimmte Werte nach Post nicht mehr verändert werden können

Implementation – HATEOAS



- HATEOAS wird verwendet, um dem Nutzer der API mögliche weitere Schritte zu präsentieren
- Hierzu Verwendung von Springs RepresentationModelAssembler
- Assembler machen mithilfe von Mapstruct Mappern aus Entity ein DTO, das mit Link Relationen angereichert wurde

Implementation – Unit & Integration Tests



- Unit Tests auf Controller-Ebene, bei denen Services gemocked werden
- Unit Tests auf Service-Ebene, bei denen Repositories gemocked werden

More Info

- Keine Repository-Tests nötig, da keine Custom Queries
- Integration Tests, mit Testcontainers
 - Benötigt Installation von Docker
 - Docker-Container wird mit Datenbank für Tests gestartet, um möglichst realitätsnahe Integration Tests zu haben
- Finale Testabdeckung: 100% Class, 85% Method, 83% Line

Fazit – Vorgehen & Teamwork



- Anfangs wöchentliche Treffen Gut
 - Hat sich mit der Zeit verloren Schlecht
- Pair Programming hat viel geholfen
 - Wichtig für Verständnis -> ermöglicht schnellere Umsetzung & besseren Code
 - Mehr Pair Programming!
- Aufgaben Aufteilung zunächst gut
 - Hat sich mit der Zeit verloren, sodass immer mehr aufgabenübergreifend gearbeitet wurde -> Übersicht geht mit der Zeit verloren
- Vorgenommen nach Git Flow zu arbeiten
 - Hat nicht gut funktioniert

Uns hat eine feste Vorgehensweise gefehlt Verbindlichkeit hat gefehlt

Fazit – Architektur



- Von Anfang an klare Richtung
- Regelmäßiger Umbruch
 - Grund: nicht komplettes Durchdringen der Aufgabenstellung
- Klare Trennung, zwischen Model, View und Controller



Vielen Dank für eure Aufmerksamkeit

Hoffentlich war es für euch mehr als nur

