STRATEGY GAME

DISTRIBUTED SYSTEMS PROJEKT – FRÜHLINGSSEMESTER 2021

DENIS NAULI, FLORIS STAUB, JAN HUBER

THEMEN

01 Motivation und Idee

02 Demo

03 Architektur

04 Implementation

05 Fragen

IDEE

MOTIVATION


Sid Meier's
CIVILIZATION


AGE
of
EMPIRES

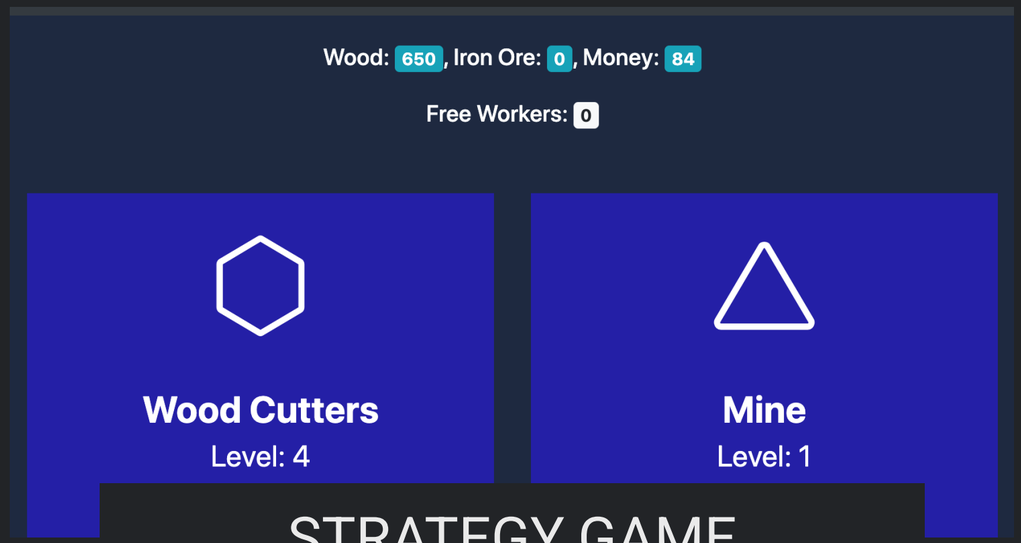

FARMVILLE


CLASH
of CLANS

FARM VILLE vs. STRATEGY GAME

FARM VILLE

- 👎 In-App-Käufe
- 👎 Basierte auf Flash († 2021)



STRATEGY GAME

- 👍 Kostenlos
- 👍 HTML-Oberfläche
- 👍 Search-Engine-Optimierter Name

DEMONSTRATION

ARCHITEKTUR

Fokus

- Neue Technologien
- Ausprobieren

TECHNOLOGIEN

Django

Django war für uns eine neue Technologie, wir haben dieses Framework gewählt, um die Sprache Python besser kennenzulernen.

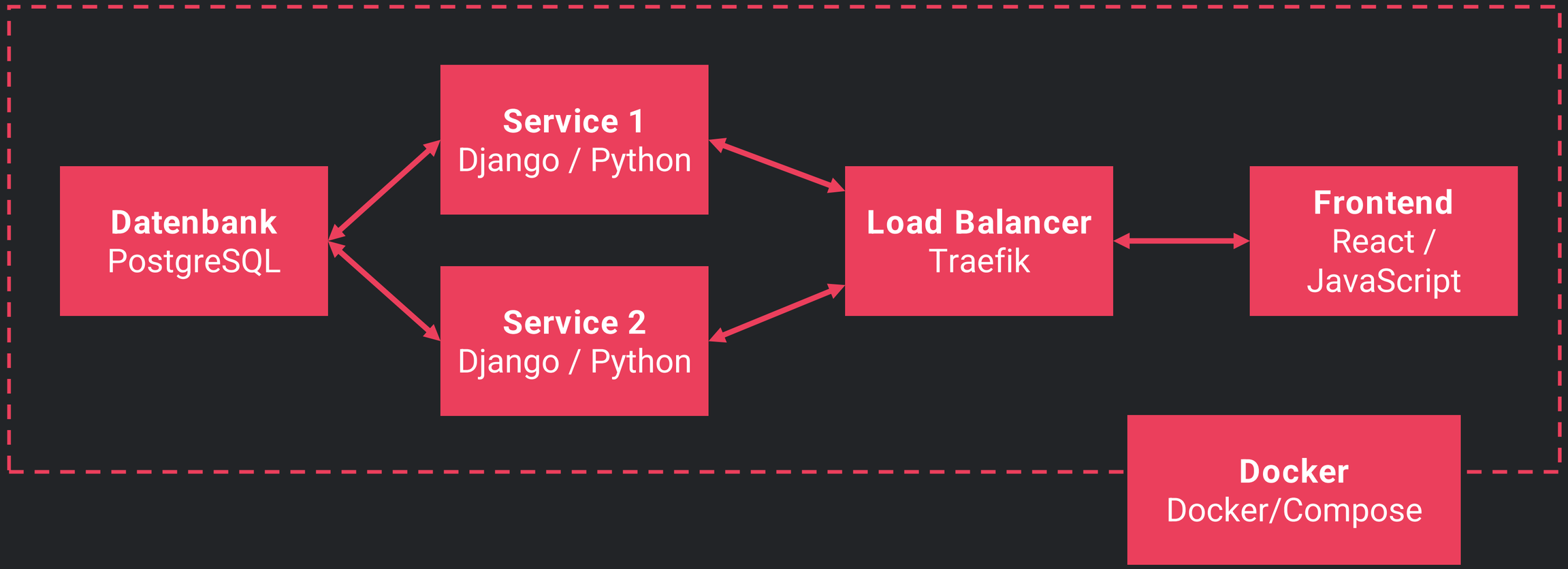
React

Da das Frontend aus mehreren oft Wiederverwendbaren Komponenten besteht, hat uns die React-Library einiges an Arbeit abgenommen.

Traefik

Wir haben Traefik aufgrund seiner einfachen Konfigurationsmöglichkeiten sowie aufgrund der Einführung im Unterricht ausgewählt

AUFBAU


Service 1
Django / Python
Datenbank
PostgreSQL
Service 2
Django / Python
Load Balancer
Traefik
Frontend
React /
JavaScript
Docker
Docker/Compose

Docker-compose

```
➜  ds1-strategygame git:(master) ✗ docker-compose up
Docker Compose is now in the Docker CLI, try `docker compose up`

Starting frontend   ... done
Recreating postgres ... done
Recreating service2 ... done

ERROR: for service1  Container "5fd47f624644" is unhealthy.
ERROR: Encountered errors while bringing up the project.
```

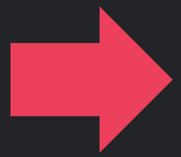
```
Success. You can now start the database server using:

    pg_ctl -D /data/postgres -l logfile start

waiting for server to start....2021-05-15 16:49:55.483 UTC [56] FATAL:  data directory "/data/postgres" has invalid permissions
2021-05-15 16:49:55.483 UTC [56] DETAIL:  Permissions should be u=rwx (0700) or u=rwx,g=rx (0750).
pg_ctl: could not start server
Examine the log output.
 stopped waiting
```

Lösung

```yaml
postgres:
  container_name: postgres
  image: postgres:13-alpine
  environment:
    - POSTGRES_DB=database
    - POSTGRES_USER=${USER}
    - POSTGRES_PASSWORD=${PASSWORD}
    - PGDATA=/data/postgres
  ports:
    - 5432:5432
  volumes:
    - ./pg_data:/data/postgres
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 5s
    timeout: 30s
    retries: 20
  networks:
    - backend
```



```yaml
postgres:
  container_name: postgres
  image: postgres:13-alpine
  environment:
    - POSTGRES_DB=database
    - POSTGRES_USER=${USER}
    - POSTGRES_PASSWORD=${PASSWORD}
    - PGDATA=/data/postgres
  ports:
    - 5432:5432
  volumes:
    - pg_data:/data/postgres
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 5s
    timeout: 30s
    retries: 20
  networks:
    - backend
```

IMPLEMENTATION

ENTSCHEIDUNGEN

Stateless Services

Die Stateless Services erlauben uns eine besonders einfache Skallierung. Es können weitere Services hinzugefügt oder entfernt werden, ohne dass dabei ein laufender Workflow unterbrochen wird.

Polling statt Websocket

Das Spiel ist von der Zeit abhängig: Alle X Minuten gewinnt der Benutzer / die Benutzerin neue Ressourcen, die im Frontend angezeigt werden müssen.

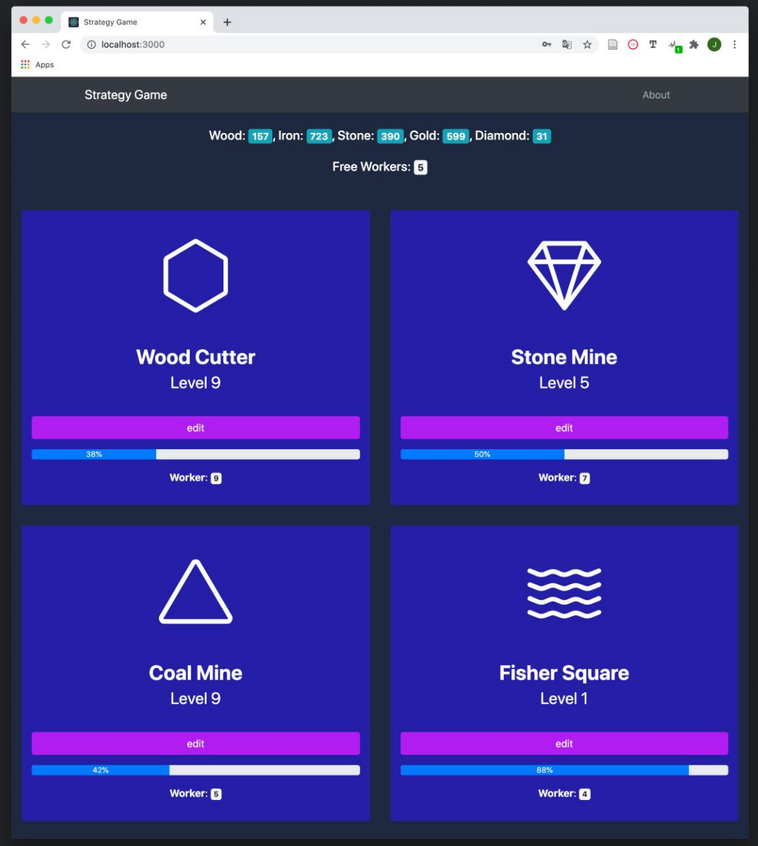
Um die Komplexität tief zu halten hohlt sich das Frontend jede Minute die aktualisierten Werte vom Backend. Alternativ hätte man eine Event-Basierte Architektur mit Websockets erstellen können.

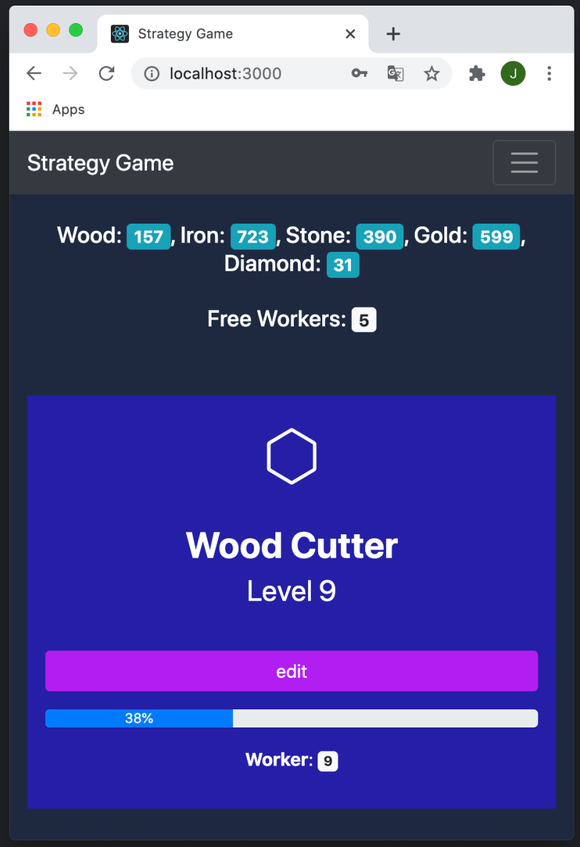
Client-Side Rendering

Das Frontend wird auf der Client-Seite gerendert und kommuniziert über eine HTTP-API mit dem Backend. Somit waren wir gezwungen, saubere Schnittstellen zu definieren und konnten die Arbeit einfacher aufteilen. Zudem könnte man so einfacher weitere Plattformen bedienen (zum Beispiel mit einer nativen Smartphone-App)

FRONTEND

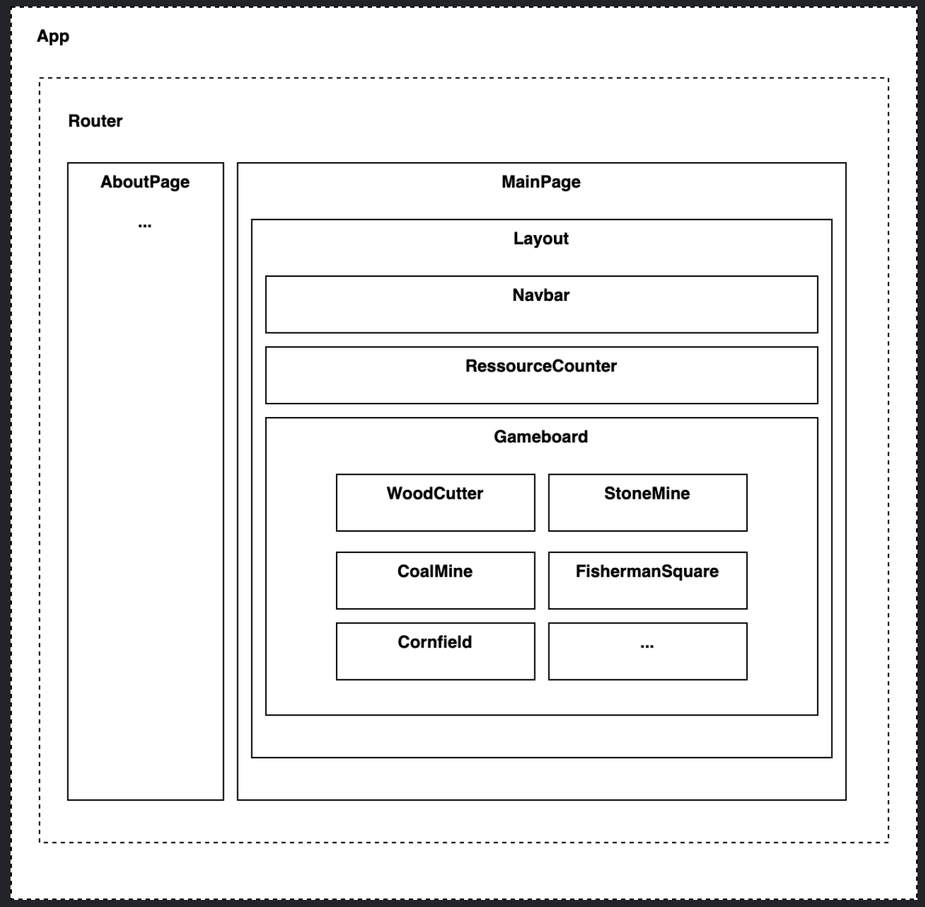
Responsives Layout mit CSS Grid





FRONTEND

Einsatz von React



FRONTEND

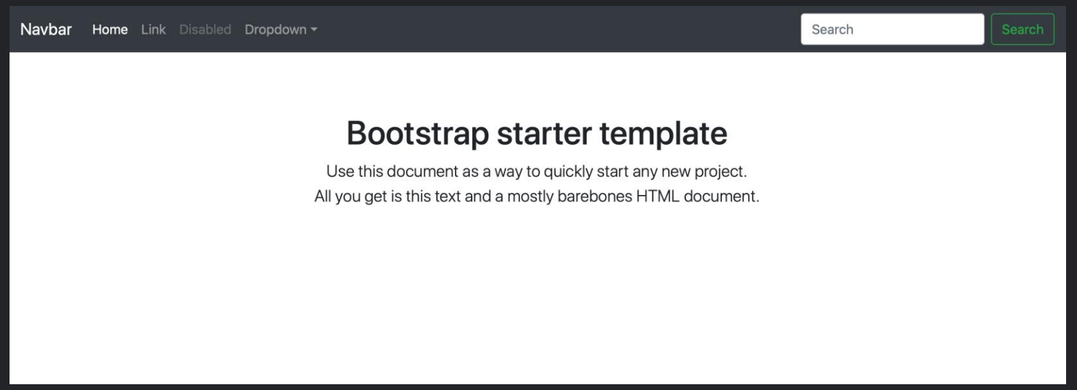
Einsatz von React

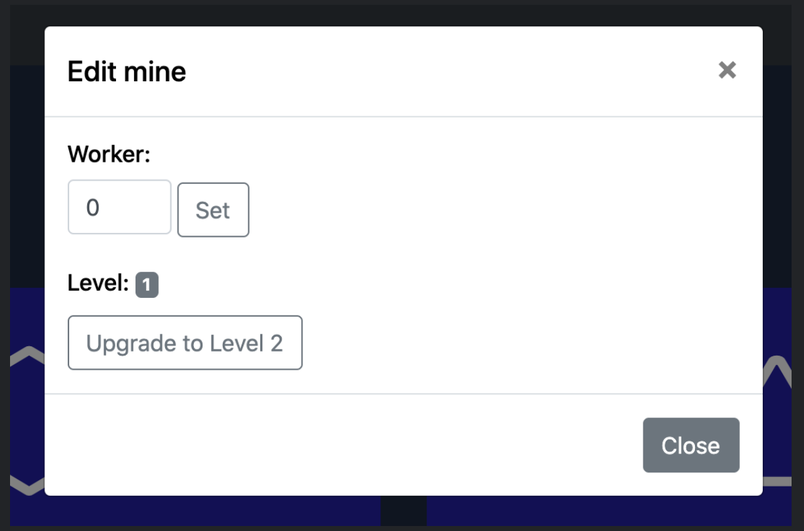
```
<GameBoard>
   <WoodCutters
      level={this.props.woodCutter.level}
      worker={this.props.woodCutter.worker}
      updateHandler={this.props.updateBuildingHandler}>
   </WoodCutters>

   <Mine
      level={this.props.mine.level}
      worker={this.props.mine.worker}
      updateHandler={this.props.updateBuildingHandler}>
   </Mine>
</GameBoard>
```

FRONTEND

Einsatz von Bootstrap





FRONTEND

Testing

+

React Testing Library

Jest

FRONTEND

Testing

```
it('renders title', () => {
    render(<App/>);
    const textElement = screen.getByText(/Welcome to Strategy Game/i);
    expect(textElement).toBeInTheDocument();
});
```

FRONTEND

Testing

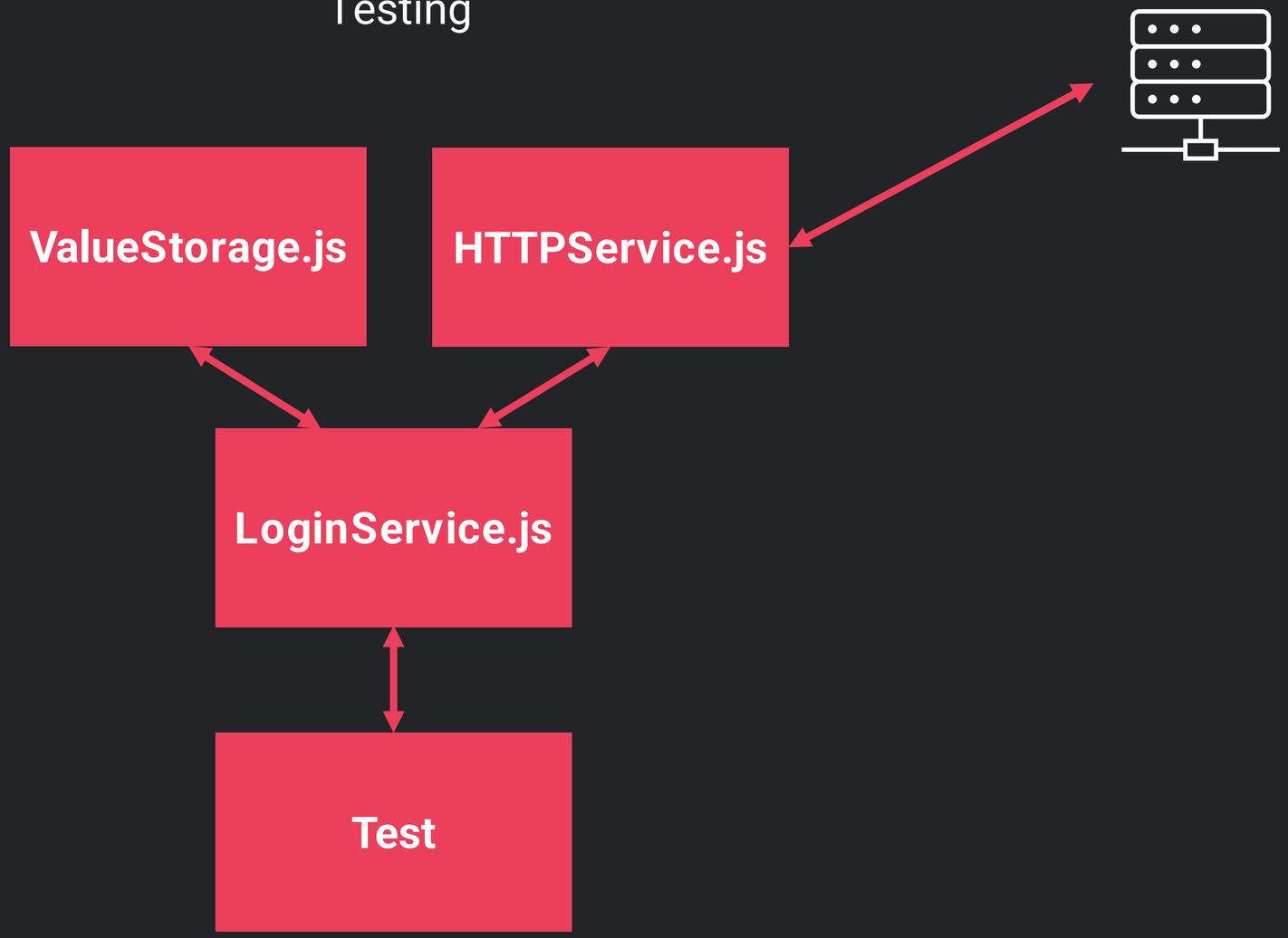
+

Sinon

Jest

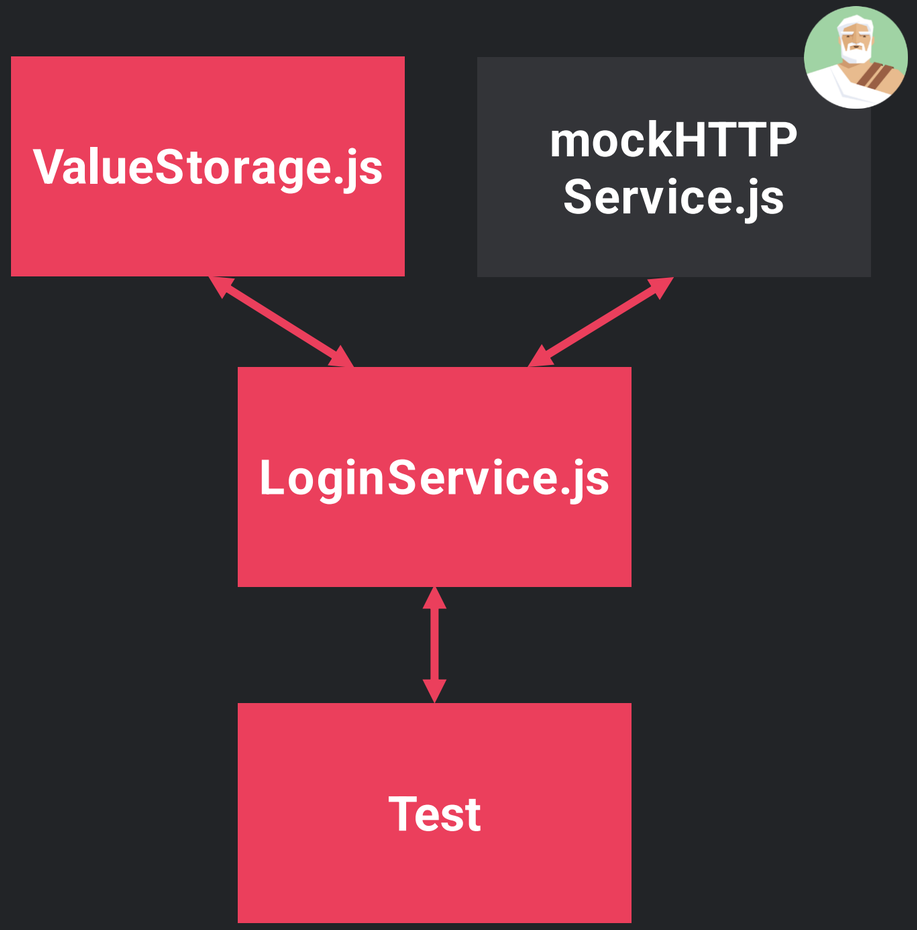
FRONTEND

Testing



FRONTEND

Testing



FRONTEND

Testing

```
mockHttpService.expects("ajax")
            .withArgs("GET", "/api/woodcutters/")
            .once();

//Rest of the test

mockHttpService.verify();
```

FRONTEND

Lessons Learned

Entwicklungs-Umgebung

In der frühen Entwicklung wurde das Frontend jeweils über Docker-Compose gestartet um die Änderungen zu testen. Das dauerte immer sehr lange. Später haben wir die Abhängigkeiten zwischen Frontend und Backend gekappselt um das Frontend unabhängig vom Backend starten und testen zu können.

Redux

Redux wird oft gemeinsam mit React verwendet, wir haben jedoch darauf verzichtet. Dadurch mussten wir im Frontend viel State-Management-Code schreiben, den uns Redux abgenommen hätte.

Standarttools

Der Einsatz von Standardtools wie Bootstrap, CSS Normalize, Templates hat Zeit in der Entwicklung gespart.

BACKEND

urls

```
urlpatterns = [
    path('mine', mineRequests),
]
```

views

```
@api_view(['GET', 'PUT'])
def mineRequests(request):
          if request.method == 'GET':
          serializer = MineSerializer(Mine.objects.get(user=request.user))
          return Response(serializer.data)
```

serializers

```
class MineSerializer(serializers.ModelSerializer):
    class Meta:
        model = Mine
        fields = ('amountCoal', 'amountIronOre', 'amountDedicatedWorkers', 'buildinglevel’)
```

models

```
class Mine(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    amountCoal = models.IntegerField(default=0)
    amountIronOre = models.IntegerField(default=0)
    amountDedicatedWorkers = models.IntegerField(default=0)
    buildinglevel = models.IntegerField(default=1)
```

JWT / LOGIN

Datenbank

Service

Traefik

Client

2: Benutzer wird gespeichert

Username	Passwort-Hash
demo_player	.7cNk4aqnjGCOU0PDD3Sbej2y...
example_user	2ETBEnM6wweu2JxBAtKaO.3hj...
max_mustermann	F1XnqiuCX5.w2zuELeB8Z.5Df...

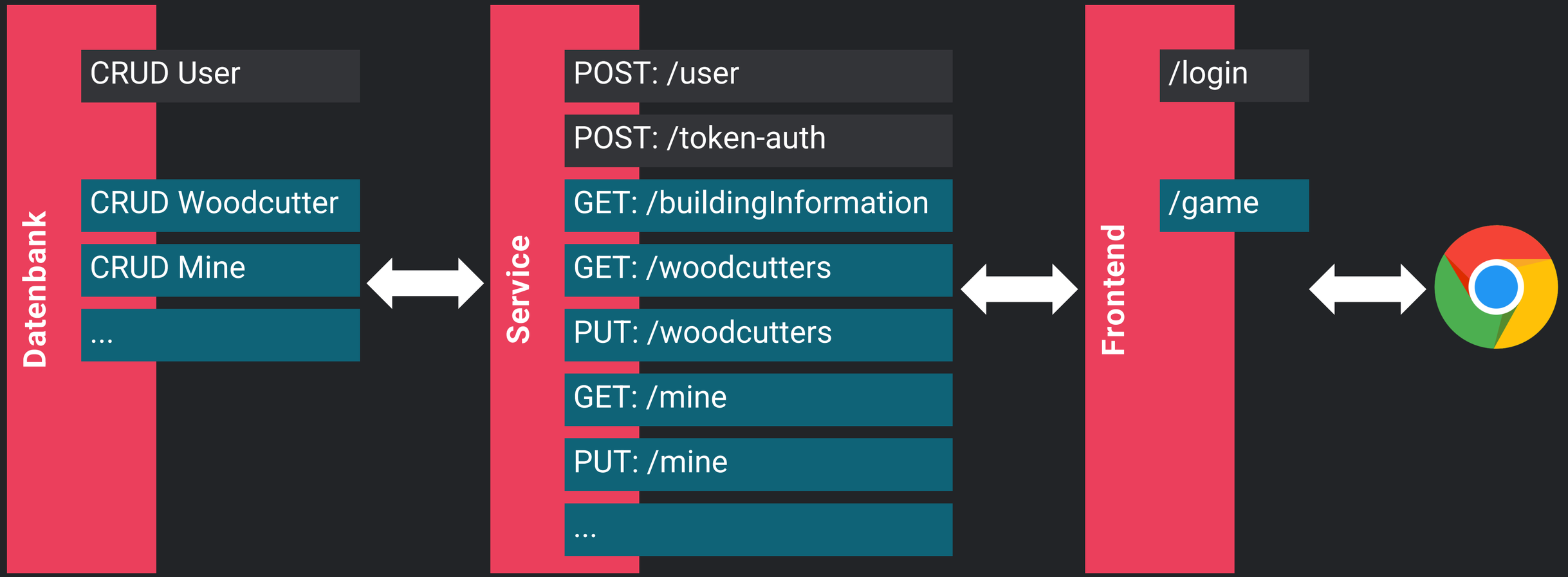
4: Token wird generiert und zurückgegeben

1: Registrierungs-Request mit Benutzername und Passwort

3: Login-Request mit Benutzername und Passwort

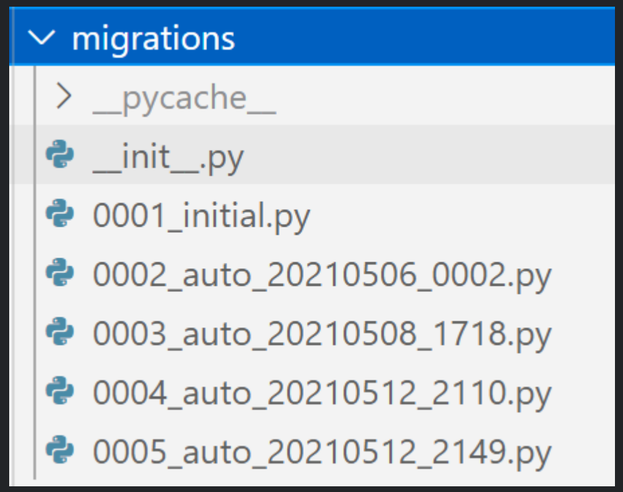
5: Token wird bei allen weiteren Requests mitgesendet

SCHNITTSTELLEN



DATENBANK

```
python manage.py makemigrations
```




```
python manage.py migrate
```

```
                List of relations
Schema |             Name              | Type  |  Owner
-------+-------------------------------+-------+----------
public | api_armycenter                | table | postgres
public | api_mine                      | table | postgres
public | api_player                    | table | postgres
public | api_townhall                  | table | postgres
public | api_woodcutters               | table | postgres
```

DATENBANK

user_id	amountCoal	amountIronOre	amountDedicatedWorkers	buildinglevel
1	62	24	2	5
2	3	3	2	3

```
class Mine(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, primary_key=True)
    amountCoal = models.IntegerField(default=0)
    amountIronOre = models.IntegerField(default=0)
    amountDedicatedWorkers = models.IntegerField(default=0)
    buildinglevel = models.IntegerField(default=1)
```

DANKE

WEITERE FRAGEN?