

Schlachtplan – Verkehrs-Spiel: Test & Design Guide

Ein rundenbasiertes Lernspiel über innerstädtischen Verkehr, Zeitbedarf und CO₂-Emissionen. Jede Runde wählen Spieler ihr Verkehrsmittel. Diese Wahl beeinflusst Verkehr, Emissionen und das Spielgeschehen. Dieser Schlachtplan zeigt Sebastian, wie Tests und Game-Design koordiniert werden.

1. Architektur (Kurzfassung)

Frontend: React + Vite (Spiel, Chat, Animationen)

Backend: Django + DRF + Channels + Redis (API, Logik, WebSockets)

Proxy: nginx (Routing, statische Dateien)

Auth: Django-Session (Login / Register / Spielzugang)

2. Rollen

Lukas: Architektur, Backend, API, Frontend-Integration

Sebastian: Tests (Python) & Game-Design (Spielfluss, Balancing, Feedback)

3. Teststrategie (alles in Python)

Verwendete Tools: Django TestCase, pytest + DRF APIClient, Channels TestClient, Selenium.

Ziele: Modelle, API, WebSockets und Nutzerfluss absichern.

4. Beispieltests

Modelle – Datei: games/tests/test_models.py

```
from django.test import TestCase
from games.models import Game

class GameModelTests(TestCase):
    def test_defaults(self):
        g = Game.objects.create(code="ABC123", host_id=1)
        self.assertEqual(g.status, "lobby")
        self.assertEqual(g.round, 1)
```

API – Datei: games/tests/test_api.py

```
from rest_framework.test import APIClient, APITestCase
from django.contrib.auth.models import User

class GameAPITests(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user("sebastian", password="1234")
        self.client = APIClient()
        self.client.login(username="sebastian", password="1234")

    def test_create_game(self):
        res = self.client.post("/api/games/", {"name": "Testspiel"})
```

```
self.assertEqual(res.status_code, 201)
```

WebSockets – Datei: games/tests/test_ws.py

```
from channels.testing import WebsocketCommunicator
from django.test import TransactionTestCase
from trafficgame.asgi import application

class WSTests(TransactionTestCase):
    async def test_chat(self):
        comm = WebsocketCommunicator(application, "/ws/game/TEST123/")
        connected, _ = await comm.connect()
        assert connected
        await comm.send_json_to({"type": "chat.send", "text": "Hallo"})
        msg = await comm.receive_json_from()
        assert msg["type"] == "chat.new"
        await comm.disconnect()
```

Integrationstest – Datei: tests/test_flow.py

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.by import By

class GameFlowTests(StaticLiveServerTestCase):
    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_login_and_lobby(self):
        self.browser.get(self.live_server_url + "/login/")
        self.browser.find_element(By.NAME, "username").send_keys("testuser")
        self.browser.find_element(By.NAME, "password").send_keys("password")
        self.browser.find_element(By.CSS_SELECTOR, "button[type=submit]").click()
        assert "Lobby" in self.browser.page_source
```

5. Testablauf für Sebastian

1. Umgebung aufsetzen (Python, Redis, WebDriver).
2. Django starten: python manage.py runserver
3. Redis starten: redis-server &
4. Tests ausführen:

```
python manage.py test
pytest games/tests/
pytest e2e/
```

6. Wochenplan

- Woche 1: Modelle & API-Tests
- Woche 2: Lobby- und WebSocket-Tests
- Woche 3: Selenium – Login → Lobby → Runde starten
- Woche 4: Game-Design-Feedback (Balancing, Spaßfaktor)

7. Game-Design-Aufgaben

- Spielfluss beobachten und bewerten (Tempo, Verständlichkeit, Spannung).

- Rückmeldung zu Balancing und Spielmechaniken.
- Unfaire Verkehrsmittel melden ('Auto zu stark' = Bug).
- Feedback zum Spielgefühl und UI geben.

8. Kommandos (Spickzettel)

```
python manage.py test      # Django-Tests
pytest games/tests/        # API-Tests
pytest e2e/                 # Browsertests
python manage.py runserver  # Backend starten
npm run dev                 # Frontend starten
```

9. Langfristige Ziele

- CI-Pipeline mit automatischen Tests.
- Coverage über 80 %.
- Später: Playwright statt Selenium.
- Jedes Feature braucht Tests & Feedback.

10. TL;DR – Unser Plan

- Django + React bleiben gesetzt.
- Tests komplett in Python.
- Sebastian kümmert sich um Testabdeckung + Game-Design.
- Ziel: Stabile, faire und nachvollziehbare Spielerfahrung.

Motto: Wenn Sebastian es nicht kaputtkriegt, darf's online.