Einzelreflexion Eric Wolf

Ausgangssituation und Einordnung in den SE-Prozess

Im Projekt StudIQ habe ich als Full-Stack-Entwickler sowohl im Frontend, Backend als auch in der Infrastruktur gearbeitet. Unsere Arbeitsweise folgte einem **iterativen Ansatz** (Sprints), wobei wir **GitHub Projects** für das Backlog-Management nutzten. Meine Hauptaufgaben lagen in der Analyse & Entwurf (Datenmodell, Systemarchitektur), der Implementierung (Projekt-Setup, Scraper-Modul, Frontend-Teile) sowie technischen Entscheidungen und Deployment-Vorbereitung.

Ausgewählte Aufgaben

1. Technisches Projekt-Setup und API-Architektur

Zu Projektbeginn habe ich das initiale Setup für Django (Backend), Vue.js (Frontend) und die API-Verbindung übernommen (Issue #30). Dies ermöglichte dem Team, sofort mit echten API-Endpunkten zu arbeiten statt auf Mock-Daten angewiesen zu sein.

Der Lösungsweg orientierte sich an der **Schichtenarchitektur**: Django-Models und Serializer nach **Domain-driven Design** (Commit `1a9dca3`), REST-API mit ViewSets und Frontend-Service-Modulen (services/), sowie Umgebungsvariablen für sauberes Konfigurationsmanagement.

Ein wichtiger Meilenstein war die vollständige Definition der Client-Server-Schnittstelle (Commit `e3b8bb3`): Ich implementierte ein strukturiertes API-Service-Modul im Frontend mit zentraler Endpunkt-Verwaltung (`endpoints.js`), einem wiederverwendbaren API-Client mit CSRF-Handling (`client.js`) und Error-Management (`errors.js`). Parallel dazu passte ich das Backend-Datenmodell an die UML-Spezifikationen an und erweiterte die REST-API. Die Dokumentation im Server-README listete alle verfügbaren Endpunkte, HTTP-Methoden und Berechtigungen auf.

Learning: Ein minimaler "Walking Skeleton" ist extrem hilfreich für agiles Arbeiten. Die zentrale Definition und Dokumentation der API-Endpunkte in `endpoints.js` schafft Klarheit darüber, welche Schnittstellen verfügbar sind und wie diese zu nutzen sind.

2. Automatisierte Datenbeschaffung (Modulux-Scraper)

Um die App mit echten Inhalten zu füllen, entwickelte ich ein Scraper-Modul (modulux_scraper.py), das Studiengänge und Module der HTW Dresden aus der Modulux-Website extrahiert. Da keine offizielle API existierte und die Seite serverseitig rendert, musste ich das HTML parsen. Ein Migrations-Skript (populate_db.py) importiert die Daten strukturiert in die Datenbank und stellt Beziehungen zwischen Studiengängen und Modulen her.

Learning: Ich unterschätzte den Aufwand erheblich. Die HTML-Struktur-Analyse und

Datenbereinigung waren arbeitsintensiver als erwartet, für zukünftige Web-Scraping-Projekte muss ich realistische Zeitpuffer einplanen.

3. Frontend-Entwicklung und Responsive Design

Ich implementierte wesentliche UI-Komponenten wie die Navigation und die Startseite (Commit `c5d55a7`). Grundlage hierfür waren die **User Stories** und **Personas** aus der Anforderungsphase, sowie die Wireframes und Design-Konzepte aus dem Team. Ich setzte diese in Vue.js um, mit Fokus auf Responsivität durch Flexbox/Grid sowie die Umsetzung von Gamification-Elementen wie dem Pinguin-Maskottchen (Penguin.vue) und den Flammen (Streak-Indikator), um die Nutzermotivation zu erhöhen.

Challenge: Das Team arbeitete ohne einheitliches Design-System. Verschiedene Entwickler implementierten Features mit ganz unterschiedlichen Stilen, von minimalistische bis zu knalligen Gradients. Am Ende erforderte das ein großes UI-Refactoring, um alle Komponenten konsistent zu gestalten.

Learning: Ein konsistentes Design-System sollte früh definiert werden. Ohne gemeinsame Regeln (Farben, Abstände, Komponenten) führt zu viele parallele Entwicklung schnell zu Inkonsistenzen. Diese Standardisierung hätte uns am Ende viel nachträgliche Arbeit erspart.

Gesamtreflexion und Fazit

Erfolge:

- Der Umstieg von SQLite auf MySQL (PR #244) und das Deployment liefen reibungslos.
- Durch das schnelle Setup konnte das Team früh echte Features bauen.
- **Code Reviews** via Pull Requests etablierten sich als wichtiges Werkzeug zur Qualitätssicherung und Wissensverteilung.

Herausforderungen:

- Der häufige Kontextwechsel zwischen Backend und Frontend kostete viel Energie, fokussierte Blöcke hätten mir gutgetan.
- Wir hätten uns früher auf einen Dokumentationsstandard (z.B. Doxygen) einigen sollen. Da wir das erst spät definierten, mussten wir bestehende Funktionen und APIs nachträglich dokumentieren, ein zeitaufwändiger Prozess, der hätte vermieden werden können, wenn die Dokumentation von Anfang an parallel zur Implementierung entstanden wäre.
- Tests wurden erst spät in den Entwicklungsprozess integriert. Hätten wir von Anfang an **Test-Driven Development** (TDD) praktiziert und Tests in die GitHub Actions CI/CD-Pipeline eingebunden, wären viele Bugs früher aufgefallen.

Wichtigstes Learning: Ein gutes Fundament (Walking Skeleton, Tests, Dokumentierungen), klares Design-System und solide Test-Abdeckung zahlen sich am Ende massiv aus. Was anfangs wie "Umweg" wirkt, spart echte Zeit und Kopfschmerzen. Für zukünftige Projekte: Standards früh definieren, gemeinsam einhalten und so groß angelegte Refactorings von vornherein vermeiden :)