6 Qualitätssicherungs-Manifest: Projekt zKalkulator

Architektonische Integrität, Mathematische Invarianz und Ephemere Validierung in Finanzanwendungen

Qualitätssicherungs-Manifest: Projekt zKalkulator

Architektonische Integrität, Mathematische Invarianz und Ephemere Validierung in Finanzanwendungen

Präambel: Die Notwendigkeit einer deterministischen Qualitätssicherung

In der Entwicklung moderner Finanzapplikationen wie dem Projekt 'zKalkulator' stellt die Qualitätssicherung (Quality Assurance, QA) nicht lediglich eine nachgelagerte Phase der Fehlersuche dar, sondern fungiert als das fundamentale Rückgrat der gesamten Systemarchitektur. Angesichts der technologischen Konstellation aus Supabase als Backend-as-a-Service, Deno als sicherer Laufzeitumgebung und React (spezifisch im Kontext von Lovable) für das Frontend, sehen wir uns mit einer Komplexität konfrontiert, die herkömmliche Testmethoden obsolet erscheinen lässt. Das bloße Vorhandensein von Unit-Tests reicht nicht aus, um die Korrektheit finanzieller Berechnungen unter allen denkbaren Randbedingungen zu garantieren, noch vermag es die strukturelle Integrität einer wachsenden Codebasis zu sichern.

Dieses Manifest definiert daher eine kompromisslose Strategie, die auf vier unverhandelbaren Säulen ruht: der strikten Durchsetzung architektonischer Grenzen durch statische Analyse, der mathematischen Beweisführung durch Property-Based Testing, der Validierung der Datensicherheit durch ephemere CI-Datenbanken und der inhaltsbasierten Verifikation finaler Artefakte. Wir bewegen uns weg von der stichprobenartigen Prüfung hin zu einem systemischen Ansatz, der Fehlerklassen kategorisch ausschließt. Ziel ist es, ein System zu schaffen, dessen Korrektheit nicht nur behauptet, sondern reproduzierbar demonstriert werden kann.

I. Architektonische Integrität durch Striktes Linting und Modulgrenzen

Die Langlebigkeit und Wartbarkeit einer Softwarelösung korreliert direkt mit der Disziplin, mit der ihre internen Abhängigkeiten verwaltet werden. In einem TypeScript-basierten Ökosystem, das Deno und React verbindet, tendieren unregulierte Importstrukturen dazu, sich in einen monolithischen "Spaghetti-Code" zu verwandeln, in dem zyklische Abhängigkeiten und unklare Verantwortlichkeiten jede Refaktorisierung zu einem Risiko machen. Für 'zKalkulator' implementieren wir daher ein rigides "Feature-Sliced Design" (FSD), das nicht durch Konventionen, sondern durch harte Linting-Regeln erzwungen wird.

1.1 Die Durchsetzung von Schichtgrenzen mittels ESLint

Das Fundament unserer Architekturprüfung bildet das eslint-plugin-boundaries. Dieses Werkzeug erlaubt uns, die zulässigen Kommunikationswege zwischen verschiedenen Teilen der Anwendung explizit zu definieren und Abweichungen als Build-Fehler zu klassifizieren.[1] Anders als herkömmliche Linter, die sich auf Syntax oder Stil konzentrieren, analysiert dieses Plugin die Semantik der Dateisystemstruktur und mappt diese auf architektonische Konzepte.

Wir definieren für zKalkulator vier primäre Schichten, deren Interaktion streng hierarchisch geregelt ist. Zuunterst liegt der **Core (Kern)**, der in Verzeichnissen wie src/shared residiert. Dieser Layer enthält atomare UI-Komponenten, Hilfsfunktionen und Typdefinitionen, die absolut keine Abhängigkeiten zu anderen Projektteilen aufweisen dürfen. Darauf baut die **Domain-Schicht** (src/entities) auf, welche die geschäftliche Logik abbildet – im Falle eines Finanzrechners also Zinsmodelle, Steuerklassen oder Amortisationstabellen. Die **Feature-Schicht** (src/features) orchestriert diese Entitäten zu nutzbaren Funktionen, beispielsweise einem "Hypotheken-Vergleichsrechner". An der Spitze steht die **App-Schicht** (src/app oder src/pages), die das Routing und die Komposition der Seiten übernimmt.[1]

Die Konfiguration des eslint-plugin-boundaries fungiert hierbei als architektonische Firewall. In der .eslintrc-Konfiguration legen wir fest, dass ein Modul vom Typ feature zwar auf domain und core zugreifen

darf, niemals aber auf ein anderes feature oder die app-Schicht.[2] Dies verhindert die horizontale Kopplung, die dazu führt, dass Änderungen in einem Feature unerwartete Seiteneffekte in einem völlig anderen Bereich auslösen. Ein direkter Import wie import { useLoanCalc } from '../../features/loan-calc' innerhalb eines Auth-Features würde sofort vom Linter blockiert werden. Dies zwingt Entwickler dazu, geteilte Logik bewusst in den shared-Layer zu abstrahieren, anstatt schnelle, aber schädliche Abkürzungen zu nehmen.[3]

1.2 Vermeidung von Zyklen und Kapselungsverletzungen

Neben der vertikalen Schichtung ist die Vermeidung zyklischer Abhängigkeiten (Circular Dependencies) kritisch, insbesondere in Deno-Umgebungen, wo Module zur Laufzeit aufgelöst werden. Ein Zyklus entsteht, wenn Modul A Modul B importiert, welches wiederum Modul A benötigt. Solche Strukturen sind oft die Ursache für schwer debugbare Laufzeitfehler und "Maximum Call Stack"-Exceptions.

Zur Detektion solcher Anomalien integrieren wir dependency-cruiser in unsere CI-Pipeline. Dieses Werkzeug generiert einen gerichteten Graphen aller Modulabhängigkeiten und validiert diesen gegen einen Satz von Regeln.[4] Ein CI-Job schlägt fehl, sobald ein Zyklus erkannt wird. Darüber hinaus prüfen wir auf "Waisenkinder" (Orphans) – Dateien, die zwar im Repository existieren, aber von keinem Einstiegspunkt aus erreichbar sind. Dies hält die Codebasis schlank und verhindert das Ansammeln von totem Code ("Dead Code").

Ein weiterer Aspekt der architektonischen Hygiene ist die Kapselung. Wir erzwingen die Nutzung von "Barrel Files" (index.ts) als einzige öffentliche Schnittstelle eines Moduls. Der direkte Zugriff auf interne Dateien eines Moduls (z.B. import... from 'feature/internal/helper') wird durch die Regel boundaries/no-private unterbunden.[5] Dies erlaubt den Entwicklern, die interne Struktur eines Moduls jederzeit zu refaktorisieren, ohne die Konsumenten dieses Moduls zu brechen, solange die öffentliche API im index.ts stabil bleibt.

In Monorepos, wie sie oft mit Tools wie Nx verwaltet werden, können diese Regeln noch granularer gestaltet werden, indem Tags in der project.json genutzt werden, um Abhängigkeiten basierend auf Projekt-Typen (z.B. type:ui vs type:data-access) zu steuern.[6] Für zKalkulator adaptieren wir diese Logik auf Ordner-Ebene, um auch ohne komplexes Monorepo-Tooling die gleiche Strenge zu erreichen.

II. Mathematische Invarianz: Property-Based Testing mit Deno

Im Kontext von Finanzanwendungen ist das klassische "Example-Based Testing" unzureichend. Tests, die prüfen, ob 100 + 200 = 300 ist, bestätigen lediglich die Korrektheit einer einzigen Instanz einer Berechnung. Sie sagen nichts darüber aus, wie sich das System bei extrem großen Zahlen, negativen Werten, Bruchzahlen oder unerwarteten Eingabekombinationen verhält. Um mathematische Korrektheit systemisch zu beweisen, setzen wir auf Property-Based Testing (PBT) unter Verwendung von fast-check innerhalb der Deno-Testumgebung.[8]

2.1 Philosophie der Invarianten

Anstatt spezifische Eingabewerte und erwartete Ausgabewerte manuell zu definieren, beschreiben wir allgemeingültige Eigenschaften (Invarianten), die das System unter allen Umständen erfüllen muss. fast-check fungiert dabei als Fuzzer, der tausende von zufälligen Eingaben generiert, um diese Eigenschaften zu widerlegen.[10] Findet das Tool einen Fehler, startet es einen "Shrinking"-Prozess: Es versucht, das einfachste Beispiel zu finden, das den Fehler noch reproduziert, um das Debugging zu erleichtern.[11]

Für den zKalkulator definieren wir folgende Kern-Invarianten, die in jedem Build geprüft werden:

Invarianz-Typ	Beschreibung	Relevanz für zKalkulator
Idempotenz	f(f(x)) = f(x)	Mehrfaches Anwenden von Bereinigungsfunktionen (z.B. Runden, Steuerberechnung) darf das Ergebnis nach dem ersten Lauf nicht verändern.
Round-Trip	decode(encode(x)) = x	Daten, die in Supabase gespeichert und wieder geladen werden, müssen bitgenau dem Ursprung entsprechen.

Kommutativität	a + b = b + a	Die Reihenfolge von Teilzahlungen sollte das Endergebnis nicht beeinflussen (sofern keine zeitliche Komponente vorliegt). [12]
Monotonie	x > y => f(x) >= f(y)	Wenn der Zinssatz steigt, darf die Gesamtsumme der Rückzahlung niemals sinken (ceteris paribus).

2.2 Umgang mit Fließkommazahlen und Währungen

Ein notorisches Problem in JavaScript (und fast allen Programmiersprachen, die IEEE 754 nutzen) ist die Ungenauigkeit von Fließkommazahlen (z.B. 0.1 + 0.2!= 0.3).[13] Für eine Finanz-App ist dies inakzeptabel. Wir nutzen daher strikt Bibliotheken wie decimal.js für alle monetären Berechnungen.[14]

Die Herausforderung im PBT besteht darin, fast-check dazu zu bringen, valide Decimal-Objekte zu generieren, da die Standard-Arbitraries nur native number-Typen erzeugen. Wir implementieren daher benutzerdefinierte Arbitraries ("Arbs"), die Decimal-Instanzen erzeugen, welche realistische Geldwerte repräsentieren.

Ein Beispiel für einen solchen Test in Deno könnte wie folgt aussehen:

TypeScript

```
import fc from "npm:fast-check";
import Decimal from "npm:decimal.js";
import { assert } from "https://deno.land/std/testing/asserts.ts";
// Custom Arbitrary für Währungsbeträge (z.B. -1Mrd bis +1Mrd)
const moneyArb = fc.bigInt({ min: -1000000000n, max: 1000000000n })
 .map(n => new Decimal(n.toString()).div(100)); // Konvertierung zu Decimal mit 2 Nachkommastellen
Deno.test("Zinsberechnung ist monoton steigend", () => {
  fc.assert(
    fc.property(moneyArb, fc.float({ min: 0, max: 1 }), (principal, rate) => {
      // Vorbedingung: Nur positive Kredite betrachten
      fc.pre(principal.gt(0));

      const resultLow = calculateInterest(principal, new Decimal(rate));
      const resultHigh = calculateInterest(principal, new Decimal(rate + 0.01));

      // Assertion: Höherer Zinssatz muss zu höheren/gleichen Zinsen führen
      assert(resultHigh.gte(resultLow), `Verletzung der Monotonie: ${rate}% ergab mehr als ${rate +
0.01}%`);
    })
  );
});
```

Dieser Test führt standardmäßig 100 Ausführungen mit verschiedenen Werten durch. In der CI-Pipeline erhöhen wir diese Anzahl via Konfiguration signifikant, um auch seltene Randfälle ("Corner Cases") abzudecken.[11]

2.3 Integration in den Entwicklungsprozess

Die Integration von fast-check in Deno ist nahtlos, da Deno native Unterstützung für TypeScript und Web-Standards bietet. Wir nutzen den Deno Test Runner (deno test), der Features wie --fail-fast bietet, um bei der ersten fehlgeschlagenen Property sofort abzubrechen und Ressourcen zu sparen.[17] Dies ist besonders in CI-Umgebungen wichtig, um schnelles Feedback zu ermöglichen. Komplexere Testszenarien können auch asynchrone Eigenschaften (fc.asyncProperty) prüfen, was essenziell ist, wenn Berechnungslogik Datenbankabrufe oder API-Calls simuliert.[10]

III. Die Ephemere Wahrheit: CI-Datenbank-Audits mit Supabase

In vielen Entwicklungsumgebungen wird die Datenbank als persistentes Artefakt behandelt oder durch In-Memory-Mocks ersetzt. Für zKalkulator, das stark auf Supabase-Features wie Row Level Security (RLS) und Datenbank-Funktionen setzt, ist dies riskant. Mocks verhalten sich nie exakt wie die echte Datenbank, und persistente Dev-Datenbanken driften oft vom Produktionsschema ab. Die Lösung ist der Einsatz von *ephemeren* (flüchtigen) Datenbank-Instanzen in der CI-Pipeline.

3.1 Der Lebenszyklus einer CI-Datenbank

Für jeden Pull Request und jeden Commit auf dem Main-Branch wird eine vollständig isolierte Supabase-Instanz innerhalb des GitHub Actions Runners hochgefahren. Dies wird durch die Supabase CLI ermöglicht, die den gesamten Stack (Postgres, GoTrue, PostgREST, Realtime, Storage) in Docker-Containern lokal orchestriert.[18]

Der Ablauf gestaltet sich wie folgt:

1. **Initialisierung:** Der Runner checkt den Code aus und installiert die Supabase CLI.
2. **Start:** supabase start bootet die Container. Da dies beim ersten Mal Zeit in Anspruch nehmen kann (Image Download), nutzen wir Caching-Strategien für die Docker-Layer in GitHub Actions.[20]
3. **Migration:** Mittels supabase db reset oder supabase migration up wird das exakte Schema aus dem Repository (supabase/migrations) auf die leere Datenbank angewendet.[18] Dies validiert gleichzeitig, dass die Migrationskette intakt und fehlerfrei ist.
4. **Seeding:** Ein spezielles seed.sql Skript füllt die Datenbank mit deterministischen Testdaten (User, Profile, Basiskonfigurationen), die für die nachfolgenden Tests benötigt werden.[21]
5. **Audit & Test:** Die eigentlichen Tests laufen gegen diese Instanz.
6. **Teardown:** Nach Abschluss werden die Container gestoppt und verworfen.

Dieser Ansatz garantiert, dass jeder Testlauf auf einer "Tabula Rasa" beginnt, was "Flaky Tests" durch Datenreste aus vorherigen Läufen eliminiert.

3.2 Sicherheits-Audits und RLS-Verifikation

Ein kritischer Aspekt von Supabase-Anwendungen ist die Sicherheit auf Zeilenebene (Row Level Security). Eine falsch konfigurierte Policy kann dazu führen, dass Benutzer A die Finanzdaten von Benutzer B sehen kann. Um dies zu verhindern, führen wir in der CI-Pipeline "Negative Tests" durch. Wir nutzen das Konzept der "Canary Queries". Das sind SQL-Abfragen, die bewusst versuchen, Sicherheitsregeln zu verletzen. Diese werden über supabase db execute ausgeführt. Ein Skript prüft dabei, ob eine Abfrage Ergebnisse liefert, die sie *nicht* liefern sollte.

SQL

```
-- Beispiel eines Security-Regression-Tests
DO $$
BEGIN
  -- Versuch: Zugriff auf Daten eines anderen Users als 'anon' oder fremder User
  -- Wir simulieren hier die Perspektive eines Angreifers
  IF EXISTS (
    SELECT * FROM private_financial_records
    WHERE user_id = 'target-victim-uuid'
  ) THEN
    RAISE EXCEPTION 'SICHERHEITSLECK: Datenzugriff trotz RLS möglich!';
  END IF;
END $$;
```

Wenn diese Abfrage auch nur eine Zeile zurückgibt, bricht die CI-Pipeline sofort ab (RAISE EXCEPTION führt zu Exit-Code!= 0).[22] Zusätzlich aktivieren wir die supa_audit Extension.[23] In einem Testszenario führen wir legitime Änderungen durch und verifizieren anschließend, ob diese Änderungen korrekt im

Audit-Log (audit.record_history) verzeichnet wurden. Dies stellt die Nachvollziehbarkeit und Non-Repudiation sicher, was für Finanzanwendungen oft regulatorisch gefordert ist.[24]

IV. Inhaltsbasierte End-to-End Tests: PDF-Validierung

Der Output des zKalkulators ist oft ein PDF-Bericht, der als Vertragsgrundlage oder Archivdokument dient. Herkömmliche End-to-End (E2E) Tests mit Tools wie Playwright prüfen meist nur, ob der "Download"-Button klickbar ist und ob eine Datei im Download-Ordner erscheint. Für zKalkulator ist dies unzureichend. Wir müssen sicherstellen, dass die *in* der PDF enthaltenen Zahlen exakt mit den Berechnungsergebnissen übereinstimmen. Ein "leeres" PDF oder eines mit falschem Encoding wäre fatal.

4.1 Der "In-Memory" Parsing Ansatz

Da das Speichern und Lesen von Dateien in CI-Containern oft zu Problemen mit Dateirechten und Pfaden führt, nutzen wir einen Ansatz, der den Download-Stream direkt im Arbeitsspeicher abfängt. Playwright bietet hierfür das download-Event an, dessen Stream wir in einen Node.js Buffer konvertieren.[26]

Diesen Buffer übergeben wir an eine Parsing-Bibliothek wie pdf-parse. Diese Bibliothek extrahiert den reinen Textinhalt aus der binären PDF-Struktur (welche auf PostScript basiert).

Der Testablauf in Playwright (TypeScript) sieht wie folgt aus:

1. **Aktion:** Der Test-User füllt das Formular aus und klickt "Export PDF".
2. **Interzeption:** page.waitForEvent('download') fängt den Download ab.
3. **Extraktion:** Der Stream wird mittels createReadStream() gelesen und in einen Buffer konkateniert.
4. **Parsing:** pdf-parse wandelt den Buffer in einen String um.[27]
5. **Assertion:** Wir nutzen Regex-Matcher, um kritische Inhalte zu prüfen.

4.2 Validierung jenseits von Text

Die reine Textprüfung reicht oft nicht aus, wenn es um Layout-Compliance geht (z.B. "Steht der Disclaimer unten auf der Seite?"). Hierfür können wir die Metadaten des PDFs (Seitenzahl, Autor, Erstellungsdatum) prüfen, die ebenfalls von pdf-parse geliefert werden.[26]

Ein entscheidender Vorteil dieses Ansatzes ist die Unabhängigkeit vom Rendering. Während Screenshot-Tests ("Visual Regression Testing") oft "flaky" sind, weil Fonts je nach OS (Linux im CI vs. MacOS lokal) leicht unterschiedlich rendern, ist der extrahierte Text deterministisch. Wir prüfen beispielsweise:

- **Berechnungsergebnisse:** Taucht "Gesamtsumme: 10.500,00 €" im Text auf? [28]
- **Rechtliche Pflichtangaben:** Ist der Text "Alle Angaben ohne Gewähr" enthalten?
- **Metadaten:** Stimmt der Titel des Dokuments?

Für den Fall, dass das PDF als Base64-String von der API zurückkommt (anstatt als direkter Download), passen wir den Test an, indem wir den API-Response abfangen und den Body direkt parsen.[29] Dieser "White-Box"-Ansatz in den E2E-Tests schließt die letzte Lücke zwischen der Benutzeroberfläche und dem generierten Geschäftsdokument.

V. Synthese und strategische Empfehlung

Das hier dargelegte Qualitätssicherungs-Manifest transformiert die QA von einer reaktiven Notwendigkeit zu einem proaktiven Konstruktionsprinzip. Durch die Kombination aus **FSD-Architektur-Linting** schaffen wir eine Codebasis, die gegen Entropie resistent ist. **Property-Based Testing** liefert uns die mathematische Gewissheit, die im Finanzsektor unabdingbar ist, weit über das hinaus, was menschliche Tester abdecken könnten. Die **ephemeren CI-Datenbanken** eliminieren die Unsicherheit geteilter

Zustände und ermöglichen echte Sicherheits-Audits bei jedem Commit. Schließlich garantiert die **inhaltsbasierte PDF-Analyse**, dass das, was der Nutzer in Händen hält, auch der Wahrheit entspricht. Für das Projekt 'zKalkulator' ist die Implementierung dieser Maßnahmen nicht optional. Die initiale Investition in diese Pipeline wird sich exponentiell durch reduzierte Wartungskosten, eliminierte Regressionen und – am wichtigsten – durch das vertrauen der Endanwender in die Präzision der Berechnungen amortisieren. Wir empfehlen, mit der Einrichtung der eslint-plugin-boundaries zu beginnen, da dies die Struktur für alle weiteren Maßnahmen vorgibt, gefolgt von der Stabilisierung der CI-Pipeline durch die Supabase CLI.