Testkonzept

Erstellt von:
- **Matti Bos**

Verifiziert von:
- **Sebastian Brosch**

Version	Erstellung	Beschreibung
0.1	09.04.2025	Erstellung Dokument
0.2	11.04.2025	Erweiterung
0.3	13.04.2025	Erweiterung
0.4	20.04.2025	Einarbeitung Feedback
1.0	21.04.2025	Korrektur

Inhalt

Zielsetzung und Übersicht

Dieses Testkonzept bietet eine umfassende Übersicht über die geplante Teststrategie, die eingesetzten Methoden und die Zuständigkeiten innerhalb des Projekts. Es erläutert unterschiedliche Teststrategien, sowie die grundlegenden Prinzipien und bewährten Vorgehensweisen, die während der Testentwicklung und -durchführung Anwendung finden. Darüber hinaus beschreibt das Testkonzept klar, welche Verantwortlichkeiten einzelne Teammitglieder übernehmen und zu welchem Zeitpunkt Tests durchgeführt werden sollen, um Fehler frühzeitig zu identifizieren und zu beheben. Zusätzlich werden Aspekte der Testautomatisierung und die Nutzung entsprechender Werkzeuge behandelt, um die Effizienz und Zuverlässigkeit der Tests sicherzustellen.

Ziel dieses Dokuments ist es, Entwicklern, Testern und Stakeholdern eine gemeinsame und klare Grundlage für die Testdurchführung zu bieten, um eine effiziente Zusammenarbeit zu fördern und eine hohe Qualität des Softwareprodukts zu gewährleisten.

Testprinzipien

Auch beim Schreiben von Tests gibt es verschiedene Paradigmen. Im Folgenden werden einige wesentliche näher beschrieben, um unsere Entscheidung für eines zu begründen.

Test-Driven Development (TDD)

Beim TDD wird vor der Existenz einer Funktion bereits ein Test für diese implementiert. Der Test soll zu Beginn fehlschlagen. Im Anschluss wird die eigentliche Funktion implementiert, bis sie in der Lage ist den geschriebenen Test zu bestehen. Nun wird der Code refactored, um sein Design zu verbessern, ohne dass er dabei seine Fähigkeit verliert den Test zu bestehen. Dann wird für jede neue Funktionalität zyklisch fortgefahren.

Vorteile:

- Frühzeitige Fehlererkennung
- Hohe Testabdeckung
- Großes Vertrauen in Funktionalität des Codes
- Klare Anforderungen durch Tests vor der Implementierung
- Sicherstellung der Korrektheit bei Umstrukturierung des Codes

Nachteile:

- Höherer Zeitaufwand
- Erfordert viel Erfahrung für effiziente Testgestaltung

FIRST-Prinzip

F.I.R.S.T. ist ein Akronym, das wichtige Eigenschaften guter Unit Tests allgemein beschreibt. Es ist eine weniger strikte Vorgabe an die Behandlung von Tests als TDD, ohne essenzielle Punkte zu vernachlässigen.

Die Buchstaben stehen für folgende Eigenschaften:

- **Fast** (Tests laufen schnell)
- **Independent** (Tests sind unabhängig voneinander)
- **Repeatable** (Tests sind wiederholbar)
- **Self-validating** (Tests validieren sich selbst)
- **Timely** (Tests werden zeitnah erstellt)

Tests sollen also in der Lage sein, auch auf verschiedenen Maschinen schnell ausgeführt zu werden und dabei zuverlässig dieselben Ergebnisse zu liefern. Auch soll keine tiefere Analyse erforderlich sein, um das Ergebnis eines Tests zu bestimmen. Sie sollen so konzipiert sein, dass ein bestandener Test gewünschtes Verhalten, und ein fehlgeschlagener Test Probleme signalisiert. Der letzte Punkt wird von den vorherigen impliziert, Tests sollen „rechtzeitig“ geschrieben werden.

Vorteile:

- Geringerer Zeitaufwand als TDD
- Flexibel und pragmatisch, besonders bei studentischen Projekten
- Schneller Einstieg und einfache Wartbarkeit der Tests

Nachteile:

- Tests entstehen häufig erst nach der Implementierung
- Potenziell geringere Testabdeckung verglichen mit TDD

Entscheidung für unser Projekt:

TDD ist eine gute Wahl für Projekte mit langer Entwicklungs- und Lebenszeit. Im Rahmen unseres studentischen Projekts wollen wir durch die Verwendung des FIRST-Prinzips die positiven Aspekte nutzen und durch besondere Vorsicht die Risiken ausgleichen.

Insbesondere werden wir strikt darauf achten, dass kein zu testender Code ohne Tests ausgeliefert wird. Die Tests müssen mindestens parallel zur Funktionalität selbst entwickelt werden, wenn nicht sogar davor. Code Coverage selbst ist keine Metrik die als Grundlage dienen kann Aussagen über die Qualität des Codes zu treffen. Wir werden unsere Tests mit besonderem Augenmerk auf „Use-Case-Coverage“ entwickeln. Das bedeutet, die Funktionen primär zu testen die vom endgültigen User tatsächlich verwendet werden. Somit wird das Vertrauen in Funktionen gestärkt, die dem Nutzer auch zur Verfügung stehen und den Fallen des „Test-Users“ und dem Test von reinen Implementationsdetails aus dem Weg gegangen.

Testverantwortlichkeit und Zeitpunkt

Wie bereits im Dokumentationskonzept beschrieben sollten Tests allgemein von dem Autor geschrieben werden, der auch den zu testenden Code verfasst hat. Auch zeitlich soll die Entstehung der Tests nahe am eigentlichen Code liegen.

Tests können auch zuerst geschrieben werden, also bevor die Implementierung des Codes existiert. In diesem Fall ist der Autor nicht relevant. Absicht der Regelung ist, das Aufschieben von Tests zu vermeiden. Falls sie zuerst geschrieben werden kann dies nicht passieren.

Frühzeitig geschriebene Tests können frühzeitig fehlschlagen. So werden Fehler nicht weiter eingearbeitet und können sofort identifiziert und schneller beseitigt werden. Die Korrektur selbst kann ebenfalls leichter fallen, da keine erneute Einarbeitung in eine Funktion erfolgen muss.

Aus diesen Gründen werden die Tests mindestens parallel, also vor oder gleichzeitig mit dem Code entwickelt. Direktester Vorteil hiervon ist die generelle Existenz von Tests und breite Abdeckung. Zusätzlich wird der Entwickler durch die Überlegung von Testfällen angeregt, sich mit möglichen Sondereingaben und anderer Edge-Cases zu befassen. Die Korrektheit des Programms kann so bei unterschiedlichen Eingaben verifiziert werden.

Testdaten

Die Qualität der Testdaten ist ein entscheidender Faktor für die Qualität der Testergebnisse. Nur wenn mit Daten getestet wird, die der normalen Nutzung entsprechen, kann man von aussagekräftigen Tests sprechen. Ebenfalls müssen Tests wiederholbar sein, die getesteten Daten werden also im Rahmen mehrerer Tests nicht verändert. Trotzdem können auch sie mit dem restlichen Projekt weiterentwickelt werden, um neue Fälle abzudecken.

Darum werden diese Daten sorgfältig erstellt und gesammelt. Primär ist der Verantwortliche für Tests dafür zuständig, die Daten zu erstellen und zu organisieren. Alle Teammitglieder sind angehalten den Datensatz zu erweitern, falls die Testqualität von einer Ergänzung der Daten profitiert.

Qualitätskriterien

Gute Daten spiegeln die Anwendungsfälle und Benutzerinteraktionen der realen Anwendung wider. Es wird auf eine breite Abdeckung verschiedener Use-Cases geachtet. Auch diverse Rand- und Fehlerfälle werden in die Palette der geprüften Szenarien aufgenommen. Somit kann ein korrektes Verhalten der Applikation auch in diesen besonderen Situationen gesichert werden.

Auch müssen die Daten frei von Duplikaten sein. Diese verlangsamen den Ablauf unnötig, ohne neue Informationen über den Zustand des Systems zu liefern. Der Begriff Duplikate ist dabei nicht beschränkt auf das mehrfache Vorkommen desselben Datensatzes. Vielmehr bezieht er sich auf sehr ähnliche Testfälle die nur unwesentliche Unterschiede besitzen und Teil derselben Äquivalenzklassen sind.

Quelle

Die Testdaten stammen aus zwei Hauptquellen: bestehende Daten aus unserem Prototyp und eigens für das Projekt erstellte synthetische Daten. Die synthetischen Daten werden sorgfältig

erstellt, um spezifische Szenarien zu simulieren, die in der realen Anwendung auftreten könnten. Dabei achten wir besonders darauf, dass diese Daten eine repräsentative Abdeckung verschiedener Use-Cases bieten. Dies umfasst sowohl reguläre Nutzungsszenarien als auch Rand- und Fehlerfälle, die für die Sicherstellung der Robustheit und Zuverlässigkeit der Anwendung entscheidend sind.

Testobjekte

Ziel ist es, eine möglichst gute und breite Testabdeckung des Codes zu erreichen. Dazu werden sowohl Unit-, als auch Komponenten- und Integrationstests durchgeführt werden. Die steigende Komplexität wird hierbei berücksichtigt. Es soll also viele Unit Tests geben, einige Komponententests und ein paar Integrations- und Systemtests.

Das Testkonzept orientiert sich dabei an den technischen Rahmenbedingungen unseres Projekts. Die genaue Nennung der verwendeten Services und Versionen gehen aus dem Designentwurf hervor. Für das Front-end werden wir das React verwenden. Die API wird mit C# und dem ASP.NET Framework erstellt werden. In beiden Fällen gibt es verbreitete Test-Suiten, die Tests und deren Einbindung in automatisierte Prozesse ermöglichen. So kann per Knopfdruck die Gesamtheit aller Unit Tests ausgeführt werden. Wir haben uns aufgrund ihrer Verbreitung, leichter Bedienbarkeit und guter Dokumentation für Jest und xUnit entschieden.

Getestet werden sowohl die technische Funktionalität der einzelnen Komponenten als auch ihr Zusammenspiel. Beispielsweise soll überprüft werden, ob die Datenbank in der Lage ist neue Einträge zu speichern und wieder auszugeben. Auch die Interne Funktion der API und des restlichen Back-Ends soll kontrolliert werden.

Systemtests werden manuell ablaufen und sich an den, im Pflichtenheft dokumentierten, Produktfunktionen ausrichten. Sie werden, wie später beschrieben, schriftlich festgehalten, um die zeitliche Entwicklung nachvollziehbar zu machen. Die Ergebnisse der automatisierten Tests können von der Pipeline ebenfalls automatisch gespeichert werden.

Test-Ausnahmen

Eine vollständige Testabdeckung aller Komponenten eines Gesamten Projekts ist nahezu unmöglich und wenig sinnvoll. Insbesondere bei Projekten mit zeitlichem Limit, wie diesem. Wir werden daher einige Teile des Systems nicht mit in die Tests einbeziehen:

- **Unit Tests im Front-End**
  Es wird nur der von der API übertragene Status des Systems dargestellt. Die Komponenten selbst enthalten keine Logik („Dumb-Components“-Prinzip). Weil keine explizite Forderung für Unit-Tests im Front-End besteht können wir auch aus zeitlichen Gründen auf diese verzichten. Die Funktionen werden im Rahmen der Systemtests überprüft.
- **Front- & Back-End-Integration**
  Es werden keine expliziten Integrationstests zwischen Front- und Back-End durchgeführt. Die korrekte Anzeige der Komponenten wird im Rahmen der Systemtests ausreichend verifiziert werden.

- **Stress-Tests**
  Die Performance der Applikation ist wichtig, jedoch nicht eines der Hauptaugenmerke. Für unseren Auftrag ist eine Verhaltensanalyse an und über der Leistungsgrenze des Systems nicht notwendig.
  Zusätzlich ist es uns auch technisch nicht möglich Aussagekräftige Ergebnisse zu liefern, da hierzu Tests auf der endgültigen Hardware des Kunden nötig wären.
- **Externe-Services**
  Datenbank und Mailserver werden als funktionsfähig vorausgesetzt. Ihre tatsächliche Implementierung ist für uns nicht relevant. Im Rahmen der Integrations- und Systemtests wird ihre Funktion im Gesamtsystem bestätigt.

Aufbau eines Tests

Um die Lesbarkeit von Tests zu erhöhen, sollen sie und ihre Beschreibung einheitlich sein. Da sie primär nach fehlgeschlagenen Durchläufen manuell überprüft werden, muss leicht erkennbar sein, was genau die getestete Funktion als Ergebnis liefern soll. Die Beschreibung eines Tests kann durch seine Benennung erfolgen, wenn der Name aussagekräftig genug ist. Andernfalls muss ein erklärender Kommentar angefügt werden.

Die Namen sollten nach folgendem Schema gebildet werden:

```
<Funktion><Ergebnis><Bedingung>
```

Beispielsweise könnte ein Test zur Beschreibung einer Kreis-Klasse wie folgt benannt sein:

```
CircleCircumferenceIsGreaterThanRadius

// Circle should throw error if provided a negative radius
CircleErrorOnNegativeRadius
```

Die Tests sollen alle auf Englisch verfasst werden.

Auch die Manuellen Systemtests werden einheitlich aufgebaut. Für jeden Testfall werden folgende Eigenschaften schriftlich festgehalten:

- **Beschreibung**
  Eine kurze Zusammenfassung des Tests. Die Beschreibung darf dabei beschreiben welche technischen Aspekte geprüft werden.
- **Use-Case**
  Die abgedeckten Use-Cases werden aufgelistet. So ist es leicht möglich, die Einsatzfähigkeit aller Use-Cases des Systems zu prüfen.
- **Vorbedingungen**
  Einige Testfälle setzen einen bestimmten Zustand der Anwendung voraus. Es muss nachvollziehbar beschrieben werden, wie dieser vom Start aus erreicht werden kann. So ist es im Sinne des FIRST Prinzips möglich Tests zu wiederholen.
- **Schritte**
  Vom Startzustand des Tests aus wird detailliert angegeben, wie der zu prüfende Endzustand erreicht wird. Detailliert bezieht sich dabei auf die Beschreibung aller Zwischenschritte und nicht auf besonders feine Erklärungen oder Begründungen der Notwendigkeit dieser Schritte.

- **Erwartetes Ergebnis**
  Der erwartete Zustand der einen erfolgreichen Test bedeutet. Falls nötig auch eine Beschreibung der Merkmale die den Eintritt des Erfolgs erkennbar machen.
- **Tatsächliches Ergebnis**
  Zustand des Systems, nachdem die Schritte ausgeführt wurden. Nur bei Abweichungen näher ausführen
- **Fehlerursachen (Optional)**
  Wenn die Ursache eines Fehlers erkennbar ist, wird diese hier angegeben.

Unit Tests

Da Unit Tests von jedem Entwickler geschrieben werden sollen, sind Richtlinien hilfreich die den Prozess genauer beschreiben. Durch ihre Einhaltung entsteht ein qualitativ hochwertiges Repertoire einheitlicher Tests.

1. **Strukturierung:**
   Unit Tests sollen minimal sein. Nur einzelne oder wenige mögliche Ergebnisse einer Funktion werden getestet. Für breitere Abdeckung mehrerer Fälle werden Parameter mit Testdaten befüllt, um wiederholbare Resultate zu gewährleisten.
   Der Name des Tests ist ausführlich genug, um seinen Zweck zu erkennen.
2. **Präzise Assertions:**
   Der Erfolg oder Misserfolg eines Tests muss eindeutig sein. Dazu sollen wenige Assertions verwendet werden die das Gesamtergebnis des Tests prüfen. Falls sie nur in komplexeren Kombinationen auswertbar sind, muss der Test in mehrere kleinere Tests aufgeteilt werden.
3. **Minimales Set-Up:**
   Nur wirklich notwendige Daten werden für jeden Test bereitgestellt. Durchlaufen der Tests soll möglichst schnell sein, um eine regelmäßige Ausführung der Tests zu motivieren. Ein Aufbau komplexer Umgebungen trägt nichts zur Kontrolle bei und bremst das System unnötig aus.

Beispiel C#

API

```
// ProductController.cs
using Microsoft.AspNetCore.Mvc;
namespace MyApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductController : ControllerBase
    {
        [HttpGet]
        public ActionResult<string> Get()
        {
            var product = "Product1";
            return Ok(product);
        }
    }
}
```

Unit Test der API

```
// ProductControllerTests.cs
using Xunit;
using MyApi.Controllers;
using Microsoft.AspNetCore.Mvc;
namespace MyApi.Tests
{
    public class ProductControllerTests
    {
        [Fact]
        public void Get_ReturnsProduct()
        {
            // Arrange
            var controller = new ProductController();
            // Act
            var result = controller.Get();
            // Assert
            Assert.NotNull(result);
            Assert.IsType<OkObjectResult>(result.Result);
            var okResult = result.Result as OkObjectResult;
            Assert.Equal("Product1", okResult.Value);
        }
    }
}
```

Beispiel Typescript

Einfache Komponente

```
import React from ‚react’;


type GreetingProps = {
	name: string;
};

const Greeting: React.FC<GreetingProps> = ({name}) = > {
	return <h1>Hello, {name}!</h1>;
};

export default Greeting;
```

Unit-Test der Komponente

```
import React from 'react';
import { render } from '@testing-library/react';
import Greeting from './Greeting';


test('Greeting component renders correct message', () => {
	const { getByText } = render(<Greeting name="World" />);
	const greetingElement = getByText(/Hello, World!/i);
	expect(greetingElement).toBeInTheDocument();
});
```

Umgang mit Fehlern

Falls eigene Tests nach einer Änderung fehlschlagen, sind die Probleme zu beheben. Wenn Programmteile betroffen sind, die von anderen Teammitgliedern geschrieben wurden, soll sich mit diesen kurzgeschlossen werden. Der Autor eines Stück Codes lässt sich durch den Einsatz von GIT leicht feststellen.

Bei Fehlern, die während Systemtests auftreten ist ein Vermerk im jeweiligen Protokoll zu führen. Im Anschluss wird die Ursache analysiert und das Problem idealerweise behoben. Bei besonders hartnäckigen Problemen kann das Projekt eventuell durch GIT auf einen älteren Stand zurückgesetzt werden, der den Fehler noch nicht enthält.

Systemtests

Vor jedem Release wird das System auf dem Entwicklungsserver getestet. Systemtests können zudem jederzeit lokal durchgeführt werden. Eine Checkliste, die auch verschiedene Benutzerrollen berücksichtigt, dient zur systematischen Prüfung aller implementierten Funktionen.

Dabei werden die Qualität und Nutzbarkeit des vollständigen Produkts geprüft. Die im Pflichtenheft vereinbarten Funktionen sollen getestet werden. Dabei soll unbedingt auf Randfälle eingegangen werden, die im Nutzererlebnis enthalten sind. Abhängig vom zeitlichen Budget kann die Abdeckung etwas eingeschränkt werden. Definitiv getestet werden jedoch mindestens die folgenden Funktionen:

- Registrieren
  - Erstellen eines Accounts
  - An-/Abmelden
    - Umgang mit falschen Zugangsdaten und nicht vorhandenen Accounts
  - Account löschen
- Benutzer
  - Übersicht geplanter Events („Meine Events“)
    - Von Event abmelden
  - Verfügbare Events
    - Zu Event anmelden
  - Bearbeitung des eigenen Profils
  - Liste der vergangenen und teilgenommenen Events (Archiv)
- Moderator
  - Alle Berechtigungen des Benutzers
  - Bearbeitung nur zugewiesener Events (inkl. Benutzer und Prozesse)
    - Einladen von Besuchern
      - Einladung versenden
    - Entfernen von Besuchern
    - Erstellen von Prozessen
    - Ändern bestehender Prozesse
    - Löschen von Prozessen
  - Liste der vergangenen und zugewiesenen Events (Archiv)
- Organisator
  - Alle Berechtigungen des Moderators
  - Startseite
    - Übersicht aller Events
    - Events die der Organisator selbst besucht („Meine Events“)
  - Alle Events
    - Übersicht über alle Events
    - Möglichkeit zum Anzeigen von Details
    - Möglichkeit zur Anmeldung
  - Alle Prozesse
    - Globale Prozesse Verwalten
      - Erstellen
      - Bearbeiten
      - Löschen
  - Events Verwalten
    - Teilnehmer auflisten
    - Prozesse auflisten

        - Event erstellen
            - Neues Event anlegen
            - Event bearbeiten
            - Event Löschen
    - Mitglieder
        - Alle Mitglieder der Organisation anzeigen
            - Moderatoren ernennen und entfernen
    - Archiv
        - Liste vergangener Events
- Organisationsadministrator
    - Alle Berechtigungen des Organisators
    - Einsicht und Bearbeitung aller Events
    - Einsicht und Bearbeitung aller Prozesse
    - Erstellen und Löschen von Benutzern
        - Ernennung und Entfernen von Organisatoren
- Plattformadministrator
    - Organisation erstellen
    - Organisation löschen

Zu Systemtests wird ein Bericht angefertigt. Dieser basiert auf einer eigenen Vorlage und wird während der Durchführung der Tests erstellt. Der Inhalt dieser Berichte ist im Abschnitt „Aufbau eines Tests“ näher definiert.

Testautomatisierung

Über GitHub Actions kann die Ausführung der Tests bei jeder Änderung automatisiert ablaufen. In der sogenannten Pipeline wird dazu ein eigener Abschnitt angelegt. Die Pipeline ist mit dem Repository verbunden und hat so stets Zugriff auf die aktuelle Version des Quellcodes. Bei Veröffentlichung von Änderungen, also bei Push oder Pull-Requests wird sie gestartet. Die hinterlegten Schritte zum Builden und Testen werden direkt ausgeführt. Damit ist der Code in jedem Entwicklungsstadium nachweislich in der Lage, die aktuell definierten Tests zu bestehen.

Auch vor der Veröffentlichung auf GitHub sollte jeder Entwickler die Testläufe prüfen. In der IDE gibt es hierzu gute Einbindung der Test-Frameworks, sodass der Aufwand hierzu minimal ist. Durch einen Button können alle definierten Tests gestartet werden, nach einigen Sekunden liefern sie ihre Resultate. Der Entwickler kann so jederzeit die Funktionsfähigkeit seiner Implementierung verifizieren.

Nicht alle Tests werden in die Pipeline aufgenommen. Beispielsweise die Automatisierung von Systemtests ist zu komplex. Sie werden ausschließlich manuell durchgeführt. Auch ein Teil der Integrationstests wird nicht automatisiert werden, falls der notwendige Aufwand den erbrachten Nutzen sonst übersteigt.

Testergebnisse

Durch den automatisierten Ablauf der Unit-Tests innerhalb der Pipeline können die Ergebnisse im Verlauf der Entwicklung nachvollzogen werden. Weitere Dokumentation von Unit-Tests ist nicht notwendig.

Die Systemtests werden wie beschrieben anhand eines Protokolls geführt. Nach Abschluss wird dieses innerhalb der Dateiablage in Microsoft Teams dauerhaft gespeichert. So ist es für jedes Teammitglied möglich, die Resultate nachzuvollziehen.