NEXTGEN
- DEVELOPMENT -

Testkonzept

NextGen Development

Version 1.1

4. Mai 2025

Teammitglieder:
Julian Lachenmaier
Ayo Adeniyi
Din Alomerovic
Cedric Balzer
Rebecca Niklaus

Verantwortlich für dieses Dokument:
Ayo Adeniyi

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Dieses Dokument beschreibt das Testkonzept für das im Rahmen des Projekts entwickelte Eventmanagement-System. Ziel des Testkonzepts ist es, eine strukturierte und nachvollziehbare Vorgehensweise für die Planung, Durchführung und Auswertung aller Testaktivitäten zu definieren.

Das Eventmanagement-System ermöglicht die Erstellung, Verwaltung und Durchführung von Veranstaltungen, einschließlich Teilnehmerregistrierung, Organisationsverwaltung und Veranstaltungsplanung. Um eine hohe Qualität, Stabilität und Benutzerfreundlichkeit der Anwendung zu gewährleisten, ist ein systematischer Testprozess unerlässlich.

Das vorliegende Testkonzept legt sowohl konzeptionelle als auch organisatorische Grundlagen für den gesamten Testablauf fest. Es definiert, was getestet wird, wie die Tests durchgeführt werden, wer welche Verantwortlichkeiten trägt und wie Testergebnisse dokumentiert und nachverfolgt werden. Dabei werden sowohl funktionale als auch nicht-funktionale Anforderungen berücksichtigt.

2 Testkonzept

2.1 Testumfang

Im Projekt werden verschiedene Arten von Tests durchgeführt, um unterschiedliche Aspekte der Softwarequalität abzudecken. Die Auswahl orientiert sich am Qualitätssicherungsplan und den Projektanforderungen.

- **Unit Tests:** Einzelne Komponenten des Backends werden vom Entwicklerteam isoliert getestet, um sicherzustellen, dass sie die erwartete Funktionalität korrekt umsetzen. Dies betrifft insbesondere Backend-Komponenten wie Services und Controller. Unit-Tests werden nach Abschluss der Entwicklung einzelner Komponenten geschrieben und regelmäßig durchgeführt.
- **Integrationstests:** Diese Tests prüfen die Kommunikation zwischen mehreren Module, z. B. ob die API korrekt mit der Datenbank kommunizier oder das Frontend erwartungsgemäß mit dem Backend interagiert.
- **Manuelle Tests:** Manuelle Tests werden durchgeführt, um das Frontend und das System als ganzes abzudecken. Frontend-Tests überprüfen dabei das Verhalten der Benutzeroberfläche auf Fehler, während Systemtests die Funktionalität des gesamten Systems abdecken.

2.2 Unit Tests

Der Entwickler, der eine Methode schreibt, ist für das Testen dieser verantwortlich. Somit wird gewährleistet, dass neue und bereits bestehende Funktionen fortlaufend getestet werden.

Für die Unit Tests steht innerhalb des Backend-Codes ein eigenes Teilprojekt zur Verfügung. Ziel ist es, für jeden Service und dessen Funktionen jegliche Fälle abzudecken, die auftreten können. Pro Service wird folglich eine **ServicenameServiceTests** Klasse erstellt. Die Benennung der Testfunktionen folgt einem klaren Muster: **ZuTestendeFunktion_Input_ActionResult**. Den Test-Methoden werden konsistente Mock-Daten zur Verfügung gestellt.

Da für die Entwicklung eines Feature ein eigener Branch erstellt wird, werden die benötigten Unit Tests ebenfalls auf diesem implementiert. Für die Vollständigkeit und Korrektheit der Tests ist das dem Feature zugeteilte Pair Programming Team zuständig. Dementsprechend wird nach dem 4-Augen-Prinzip vorgegangen. Der Testumfang ist dabei von der Wichtigkeit und dem Umfang der Funktion abhängig und muss pro Funktion individuell angepasst werden.

Da der Workflow sowohl nach Push- als auch Pull-Requests durchgeführt werden, soll die regelmäßige Korrektheit aller bestehenden Funktionen regelmäßig getestet werden. Entwickler sind dazu verpflichtet, sowohl die Funktionalität ihrer neu implementierten als auch die bereits implementierter Funktionen auf ihrem Branch zu gewährleisten. Sollte es im Rahmen einer Pull-Request dazu kommen, dass Tests nicht bestanden werden, wird diese Request automatisch abgelehnt.

Nehmen wir die Klasse UserServiceTests als Beispiel. Um den `UserService` zu testen, wurde eine Testklasse `UserServiceTests` erstellt. Die Tests verwenden das xUnit-Framework sowie Mocking-Techniken mit Moq, um die Interaktionen mit externen Abhängigkeiten zu simulieren und somit isolierte Unit-Tests durchzuführen.

Zuerst werden alle erforderlichen Mocks eingerichtet und der `UserService` initialisiert. Dabei wird die `UserManager`-Klasse gemockt, um die Benutzerverwaltung zu simulieren.

```
1  private readonly Mock<UserManager<User>> _userManagerMock;
2  private readonly Mock<IUserRepository> _userRepoMock;
3  private readonly UserService _userService;
4
5  public UserServiceTests(ITestOutputHelper output)
6  {
7      var store = new Mock<IUserStore<User>>();
8      _userManagerMock = new Mock<UserManager<User>>(
9          store.Object, null, null, null, null, null, null, null, null)
               ;
10
11     _userRepoMock = new Mock<IUserRepository>();
12     _userService = new UserService(
13         _userManagerMock.Object,
14         Mock.Of<SignInManager<User>>(),
15         _userRepoMock.Object,
16         Mock.Of<ILogger<UserService>>());
17 }
```

In diesem Code-Fragment werden Mocks für den `UserManager` und das `IUserRepository` erstellt. Der `UserService` wird dann mit diesen gemockten Abhängigkeiten initialisiert.

Dies stellt sicher, dass keine echten Datenbankoperationen oder externe API-Aufrufe während des Tests durchgeführt werden.

Im nächsten Schritt definiert man einen Unit-Test für die Methode `CreateUserAsync`, die einen neuen Benutzer erstellt. Der Test überprüft, ob bei gültigen Eingabedaten ein Benutzer korrekt zurückgegeben wird.

```
[Fact]
public async Task CreateUserAsync_ValidData_ReturnsUser()
{
    var userDto = new UserCreateDto
    {
        Email = "test@example.com",
        Password = "ValidP@ss1",
        FirstName = "Test",
        LastName = "User"
    };

    _userManagerMock.Setup(x => x.FindByEmailAsync(userDto.Email))
        .ReturnsAsync((User)null);

    _userManagerMock.Setup(x => x.CreateAsync(It.IsAny<User>(),
        userDto.Password))
        .ReturnsAsync(IdentityResult.Success);

    _userRepoMock.Setup(x => x.CreateUserAsync(It.IsAny<UserCreateDto
        >()))
        .ReturnsAsync(new UserResponseDto { Email = userDto.Email });

    var result = await _userService.CreateUserAsync(userDto);

    result.Should().NotBeNull();
    result.Email.Should().Be(userDto.Email);
}
```

2.3 Integrationstests

Integrationstests werden in einem separaten xUnit-Testprojekt geschrieben, um sie klar von den Unit Tests zu trennen. Ziel ist es, die Zusammenarbeit mehrerer Komponenten unter realitätsnahen Bedingungen zu testen. Dabei kommt eine dedizierte Testdatenbank zum Einsatz, um realistische Szenarien zu simulieren, ohne produktive Daten zu beeinflussen.

Die Integrationstests werden automatisiert im Rahmen der CI/CD-Pipeline ausgeführt. Sie bestehen aus mehreren, logisch zusammenhängenden Testmethoden, die in bestimmten Reihenfolgen oder Abhängigkeiten zueinander stehen können. Die Organisation und Steuerung der Testausführung erfolgt über Mechanismen des xUnit-Frameworks, wie z. B. Fixtures für gemeinsame Setups, Teardowns für Ressourcenfreigabe und benutzerdefinierte Kategorien zur Gruppierung.

Die Hauptziele der Integrationstests sind:

- **Einhaltung der Architekturvorgaben und Entwurfsmuster:**
  - Wird das geplante Design korrekt umgesetzt?
  - Werden objektorientierte Konzepte wie Vererbung oder Aggregation sachgerecht angewendet?
  - Werden die Best Practices korrekt befogt?
- **Prüfung von Performance und Stabilität häufig genutzter Anwendungsfälle:**
  - Werden zentrale Geschäftsprozesse effizient abgearbeitet?
  - Wie stabil funktionieren diese Prozesse unter typischer Last?
- **Validierung der Datenbankintegration:**
  - Werden Daten korrekt geschrieben, gelesen, verändert und gelöscht?
  - Werden Fehler- oder Sonderfälle beim Datenbankzugriff sauber behandelt?
  - Erfolgt die Kommunikation mit der Datenbank gemäß den fachlichen Anforderungen?

Die Integrationstests werden überwiegend von Entwicklern erstellt, die für das jeweilige Feature verantwortlich sind, da sie über die notwendige Kontextkenntnis verfügen. Aufgrund ihres Umfangs und ihrer Komplexität wird empfohlen, sie zusätzlich durch ein Testreview auf fachliche und technische Korrektheit prüfen zu lassen. Wenn möglich, soll das von den jeweiligen Pair-Programming Gruppen übernommen werden, die für die Funktion zuständig sind.

Jede getestete Komponente erhält eine eigene Testklasse mit der Benennung **KomponentennameIntegrationTests**. Methoden folgen dem Schema **Funktion_Szenario_Erwartung**, z. B. `CreateUser_ValidInput_ReturnsCreatedUser()`.

Zur Initialisierung gemeinsamer Ressourcen wie z. B. der Testdatenbank wird die `IClassFixture<T>` Schnittstelle verwendet. Testdaten werden in einem zentralen `TestDbContext` bereitgestellt, der bei jedem Testlauf zurückgesetzt wird. Für Datenbankzugriffe wird eine isolierte Testinstanz (PostgreSQL in Docker) verwendet, um eine realitätsnahe Umgebung zu schaffen.

Ein vereinfachtes Beispiel für den Aufbau eines Integrationstests mit Setup:

```
public class UserServiceIntegrationTests : IClassFixture<
    TestDatabaseFixture>
{
    private readonly TestDatabaseFixture _fixture;
    private readonly UserService _userService;

    public UserServiceIntegrationTests(TestDatabaseFixture fixture)
    {
        _fixture = fixture;
        _userService = new UserService(_fixture.DbContext, ...);
    }

    [Fact]
    public async Task CreateUser_ValidInput_ReturnsUser()
    {
        // Arrange
        var userDto = new UserCreateDto { Email = "new@example.com",
            ... };

        // Act
        var result = await _userService.CreateUserAsync(userDto);

        // Assert
        result.Should().NotBeNull();
        result.Email.Should().Be("new@example.com");
    }
}
```

Die Klasse `TestDatabaseFixture` stellt sicher, dass der Datenbankkontext initialisiert und vor bzw. nach jedem Test bereinigt wird:

```
public class TestDatabaseFixture : IDisposable
{
    public AppDbContext DbContext { get; }

    public TestDatabaseFixture()
    {
```

```
7           var options = new DbContextOptionsBuilder<AppDbContext>()
8               .UseInMemoryDatabase("TestDb")
9               .Options;
10
11          DbContext = new AppDbContext(options);
12          DbContext.Database.EnsureCreated();
13      }
14
15      public void Dispose()
16      {
17          DbContext.Database.EnsureDeleted();
18          DbContext.Dispose();
19      }
20  }
```

Die Tests sind dadurch voneinander unabhängig, reproduzierbar und für die automatische Ausführung geeignet.

2.4 Manuelle Tests

Das Ziel manueller Tests ist es, das System aus der Perspektive eines Benutzers zu überprüfen. Besonders getestet werden Komponenten im Frontend, wobei der Schwerpunkt auf der Nachbildung typischer Benutzeraktionen liegt. Ziel ist es, die Bedienbarkeit und Benutzerfreundlichkeit sowie die korrekte Umsetzung der spezifizierten Anforderungen zu evaluieren. Jede Anforderung, die von außen überprüfbar ist, soll einem eigenen Test unterzogen werden.

Der Ablauf erfolgt wiefolgt: Jede Woche wird ein Tester ernannt, der für die Durchführung der manuellen Tests zuständig ist. Ein Test wird so lange wiederholt, bis alle Bedingungen erfüllt sind. Die Durchführung orientiert sich an einer standardisierten Vorlage, die eine strukturierte und nachvollziehbare Dokumentation sicherstellt. Diese umfasst folgende Abschnitte:

- **Allgemeine Informationen:**
  - Eindeutige Testprotokollnummer und Versionsangabe
  - Name der testenden Person
  - Datum und Uhrzeit der Durchführung
  - Testergebnis (erfolgreich/nicht erfolgreich)

- **Testumgebung:**
  - Ausführungsumgebung (z. B. lokaler Host, Testserver)
  - Verwendeter Webbrowser
  - Geprüfter Branch im Versionsverwaltungssystem
- **Testziel:**
  - Beschreibung des zu testenden Anwendungsfalls oder der Funktion
  - Erwartetes Ergebnis bzw. Verhalten des Systems
- **Testergebnisse:**
  - Welche Erwartungen wurden erfüllt?
  - Welche Abweichungen oder Fehler traten auf?
  - Detaillierte Reproduktionsschritte bei Fehlern
  - Identifizierte Verbesserungspotenziale

Die Testprotokolle sind so zu gestalten, dass sie sowohl für einzelne GUI-Komponenten als auch für komplexe Geschäftsprozesse oder Performance-Anforderungen verwendet werden können. Wichtig ist, dass alle festgestellten Probleme nachvollziehbar dokumentiert und für das Entwicklungsteam verständlich beschrieben werden, sodass eine zügige Behebung möglich ist. Werden schwerwiegende Fehler entdeckt, informiert der Tester das verantwortliche Pair-Programming Team.

2.5 Testdaten

Für die effektive Durchführung von Tests im Eventmanagement-System werden verschiedene Arten von Testdaten benötigt. Diese orientieren sich sowohl an typischen Anwendungsszenarien als auch an Grenz- und Fehlersituationen:

- **Gültige Daten:** Beispielhafte Events mit sinnvollen Titeln, Datumsangaben, Veranstaltungsorten und zugehörigen Nutzern etc.
- **Ungültige Daten:** Fehlende Pflichtfelder, Datumsangaben in der Vergangenheit oder unzulässige Rollen bei Nutzern*innen.
- **Grenzwerte:** Maximale Feldlängen bei Titel und Beschreibung, Events mit sehr vielen Teilnehmern, leere Felder.

- **Sicherheitsrelevante Daten:** Versuchter Zugriff auf geschützte Ressourcen durch unautorisierte Rollen oder nicht authentifizierte Benutzer*innen.

Um die konsistente Bereitstellung von beispielhaften Testdaten sicherzustellen, werden für jegliche Tests eine beschränkte Menge an Testdaten zur Verfügung gestellt. Die Bereitstellung und Verwaltung der Testdaten erfolgt auf unterschiedliche Weise, abhängig vom Testtyp:

- **Unit-Tests:** Die Daten werden direkt im Code durch Mock-Objekte oder In-Memory-Datenbanken (z. B. `UseInMemoryDatabase`) definiert.
- **Integrationstests:** Vor dem Testlauf werden Testdaten in die Testdatenbank eingespielt. Dies geschieht automatisiert über Seeder-Methoden oder Setup-Blöcke im Testcode.
- **Manuelle Tests:** Hier kann auf externe JSON/XML-Dateien mit vorgefertigten Testdaten zugegriffen werden. Alternativ werden automatisierte Skripte verwendet, die die Datenbank mit realistischer Testbefüllung versorgen.

Für die Konsistenz und Wiederverwendbarkeit der Testdaten werden Versionskontrolle mithilfe von Github und einheitliche Namenskonventionen verwendet.

2.6 Testwerkzeuge

Um die Erstellung und Durchführung der Tests zu optimieren und eine hohe Qualität der getesteten Komponenten sicherzustellen, werden folgende Testwerkzeuge verwendet:

- **xUnit:** xUnit wird für die Erstellung von Unit- und Integrationstests im Backend verwendet. Wie im Abschnitt 2.2 bereits ersichtlich wurde, werden Mocks eingesetzt, um Abhängigkeiten zu simulieren und so eine isolierte Testumgebung zu schaffen.
- **Mocking mit Moq:** Moq wird genutzt, um abhängige Komponenten wie Logger oder Services zu isolieren. So können gezielt einzelne Klassen oder Methoden getestet werden, ohne Seiteneffekte durch abhängige Systeme zu riskieren.
- **Entity Framework Core InMemory Provider:** Dieser wird für Integrationstests verwendet, um eine schnelle und unabhängige Prüfung von Repository-Logik zu ermöglichen, ohne auf die die Produktionsdatenbank angewiesen zu sein. Dies erlaubt realistische Testszenarien bei minimalem Setup-Aufwand.

- **Swagger:** Swagger dient zum manuellen Testen einzelner Endpunkte direkt aus der Dokumentation heraus. Tester können so direkt in der Dokumentation überprüfen, wie die Endpunkte zu bedienen sind und welche Antworten sie erwarten können. Dies ermöglicht es, eine schnelle Validierung einzelner Entpunkte zu erreichen und sicherzustellen, dass die API-Endpunkte funktionstüchtig sind und die richtigen Daten liefern.

- **CI/CD-Plattform:** Die CI/CD-Plattform GitHub Actions wird genutzt, um die automatisierte Ausführung der Tests zu gewährleisten. Jedes Mal, wenn ein Push-, Pull- oder Merge-Request in das Repository erfolgt, wird ein automatisierter Build- und Testprozess angestoßen. Dies stellt sicher, dass alle neuen Änderungen nicht die bestehenden Funktionen beeinträchtigen.

  Der Testprozess wird in einem GitHub Action Workflow definiert, der die Ausführung der Unit- und Integrationstests auf einer sauberen Umgebung garantiert. Sollte ein Test fehlschlagen, wird der Merge-Request blockiert, bis der Fehler behoben ist. Dies stellt sicher, dass nur fehlerfreie, getestete Codeänderungen in die Hauptentwicklungslinie integriert werden.

  Sobald die Prozesschritte ausgeführt werden, werden während jedem Prozesschritt Logs ausgegeben. Somit kann nachvollzogen werden, wann und warum ein Prozesschritt wie beispielsweise ein Test nicht erfolgreich durchgeführt werden konnte und es kann dementsprechend reagiert werden.

  Hierfür wird folgende .yml-Datei verwendet:

```
1         name: .NET Tests
2
3         on:
4         push:
5             branches: [ main ]
6         pull_request:
7             branches: [ main ]
8
9         jobs:
10        build-and-test:
11            runs-on: ubuntu-latest
12
13            steps:
14            - name: check repo
15            uses: actions/checkout@v3
16
17            - name: install .NET
18            uses: actions/setup-dotnet@v3
```

```
19              with:
20                  dotnet-version: '8.0.x'
21
22              - name: restore dependencies
23              run: dotnet restore
24
25              - name: build
26              run: dotnet build --no-restore --configuration
                    Release
27
28              - name: run tests
29              run: dotnet test --no-build --configuration Release
                    --logger:trx
```

Um die Testausführung effizient und zuverlässig zu gestalten, werden innerhalb des ASP.NET Cores folgende Konfigurationen vorgenommen:

- **Separate Konfigurationsdateien** (z. B. `appsettings.Test.json`) für Umgebungsvariablen wie Test-Datenbankverbindung.
- **Containerisierung:** Für Integrationstests wird die Anwendung optional in Docker-Containern getestet (z. B. PostgreSQL via Docker Compose).

3 Rollen und Verantwortlichkeiten im Testprozess

3.1 Beteiligte Rollen

Im Rahmen des Eventmanagement-Projekts sind mehrere Rollen am Testprozess beteiligt. Jede dieser Rollen übernimmt spezifische Aufgaben im Qualitätsmanagement.

- **Projektleitung:(Julian Lachenmaier)**
  - Genehmigt Testpläne und stellt Ressourcen zur Verfügung.
  - Überwacht die Einhaltung von Qualitätssicherungsmaßnahmen.
- **Entwicklerteam: (Din Alomerovic, Rebecca Niklaus)**
  - Entwickelt und führt Unittests sowie Integrationstests durch.
  - Beseitigt entdeckte Fehler zeitnah.
  - Dokumentiert die Testfälle im Quellcode.
- **Tester/QA-Engineer: (Ayo Adeniyi)**
  - Plant und koordiniert systematische Tests (z. B. System-, Akzeptanz- und Regressionstests).
  - Führt manuelle und automatisierte Tests durch.
  - Erstellt Fehlerberichte und überwacht deren Behebung.
- **DevOps Engineer: (Cedric Balzer)**
  - Implementiert CI/CD-Pipelines für automatisierte Testdurchläufe.
  - Überwacht die Testabdeckung und analysiert Testergebnisse in der Pipeline.
- **Product Owner / Fachverantwortliche: (Ayo Adeniyi, Cedric Balzer)**
  - Prüfen im Rahmen von Akzeptanztests, ob die Implementierung den fachlichen Anforderungen entspricht.

– Definieren kritische Anwendungsfälle und Testziele aus Nutzersicht.

3.2 Zusammenarbeit und Kommunikation

Eine enge Zusammenarbeit zwischen den Rollen ist essenziell. Wichtige Aspekte:

- Regelmäßige Meetings zur Koordination von Testaufgaben und zur Besprechung von Fehlern.
- Gemeinsame Nutzung eines Bug-Trackers (z. B. GitHub Issues).
- Dokumentation der Testergebnisse und Testabdeckungen in einem zentralen Repository.

3.3 Verantwortlichkeitsmatrix (RACI)

Zur Übersicht, wer für welche Aufgaben verantwortlich ist, dient folgende RACI-Matrix:

Testaktivität	**Entwickler**	**Tester**	**DevOps**	**PO**
Unittests schreiben	R	C		
Integrationstests	R	C		
Manuelle Tests	C	R		
CI/CD einrichten			R	
Fehlerdokumentation	R	R		
Teststrategie definieren	C	R	C	A

Tabelle 3.1: RACI-Matrix: R = Responsible, A = Accountable, C = Consulted

4 Abnahmekriterien

4.1 Wann ist ein Test bestanden?

Ein Test gilt als bestanden, wenn das tatsächliche Testergebnis dem erwarteten Ergebnis entspricht. Dabei werden folgende Bedingungen zugrunde gelegt:

- Alle spezifizierten Vorbedingungen wurden erfüllt (z. B. Authentifizierung, Datenzustand).
- Die getestete Funktion liefert die korrekte Rückgabe oder führt die erwartete Aktion aus.
- Es treten keine unvorhergesehenen Fehler, Ausnahmen oder Nebeneffekte auf.
- Bei automatisierten Tests: Alle Assertions sind erfüllt.
- Bei manuellen Tests: Der Tester bestätigt das erwartete Verhalten eindeutig anhand von Testbeschreibung und Beobachtung.

Tests, die fehlschlagen, werden protokolliert und klassifiziert (kritisch, mittel, gering), um entsprechende Maßnahmen einzuleiten (z. B. Bugfixing, Nachtests).

4.2 Release

Das System gilt als „bereit für Release“, wenn die folgenden Kriterien erfüllt sind:

- **Funktionale Abdeckung:** Alle Kernfunktionalitäten (z. B. Eventverwaltung, Benutzerregistrierung, Anmeldung) wurden erfolgreich getestet.
- **Fehlerfreiheit:** Es existieren keine kritischen oder blocker-Fehler mehr im System.
- **Testabdeckung:** Eine ausreichende Abdeckung durch Unit-, Integrations- und Systemtests ist gegeben (mindestens 80 % der Kernkomponenten).
- **Sicherheitsanforderungen:** Authentifizierung, Autorisierung und Datenschutzmechanismen wurden geprüft.

- **Benutzerakzeptanz:** Falls vorgesehen, wurde ein UAT (User Acceptance Test) durchgeführt und positiv bewertet.
- **Deployment-Fähigkeit:** Das System kann erfolgreich in einer Staging- oder Produktionsumgebung bereitgestellt werden.
- **Dokumentation:** Technische und benutzerorientierte Dokumentation ist vollständig und aktuell.

Erst wenn alle genannten Bedingungen erfüllt sind und von der Projektleitung sowie den Stakeholdern freigegeben wurden, erfolgt die formale Abnahme und das Go-Live der Version des Eventmanagement-Systems.