NEXTGEN
- DEVELOPMENT -

Testkonzept

NextGen Development

Version 1.0

21. April 2025

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

Verantwortlich für dieses Dokument:
Ayo Adeniyi

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Testkonzepts

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 Ziel und Zweck des Testkonzepts

2.1 Ziel der Testaktivitäten

Das Hauptziel der Testaktivitäten im Rahmen dieses Projekts ist die **frühzeitige Erkennung und Behebung von Fehlern**, um die Stabilität, Funktionalität und Nutzbarkeit des Eventmanagement-Systems sicherzustellen. Durch konsequente Tests während des gesamten Entwicklungsprozesses sollen kritische Bugs, Sicherheitslücken und unerwartete Verhaltensweisen vermieden werden, bevor es zu einer Auslieferung an Endnutzer*Innen kommt.

Zudem dienen die Tests dazu, die korrekte Integration einzelner Komponenten sicherzustellen und Regressionen bei zukünftigen Änderungen zu vermeiden.

2.1.1 Welche Qualitätssicherungsziele sollen erfüllt werden?

Die Qualitätssicherung im Projekt verfolgt mehrere konkrete Ziele:

- **Funktionale Korrektheit:** Das System soll alle vorgesehenen Funktionen gemäß den Anforderungen korrekt umsetzen (z. B. Event-Erstellung, Benutzerregistrierung, Teilnehmerverwaltung).
- **Sicherheit:** Vertrauliche Nutzerdaten (z. B. Login-Informationen) müssen sicher verarbeitet und in der Datenbank gespeichert werden.
- **Benutzerfreundlichkeit:** Die Bedienung des Systems soll intuitiv, klar strukturiert und konsistent sein.
- **Stabilität und Zuverlässigkeit:** Das System soll auch bei hoher Nutzerlast oder unerwarteten Eingaben stabil bleiben.
- **Skalierbarkeit:** Die Anwendung soll auf wachsende Anforderungen (z. B. steigende Teilnehmerzahlen) vorbereitet sein.
- **Wartbarkeit:** Änderungen und Erweiterungen am Code sollen durch klare Struktur und Testabdeckung risikoarm möglich sein.

2.2 Testumfang

2.2.1 Welche Komponenten werden getestet? Welche nicht?

Im Rahmen dieses Projekts werden folgende Komponenten des Eventmanagement-Systems getestet:

- **Frontend:** Die Benutzeroberfläche (z. B. Eventanzeige, Registrierung, Login) wird auf ihre Funktionalität, Benutzerfreundlichkeit und responsives Design überprüft.
- **Backend:** Alle Server-seitigen Funktionen wie Eventerstellung, Benutzerverwaltung, Authentifizierung und Datenbankzugriffe werden intensiv getestet.
- **API:** Die Schnittstellen zwischen Frontend und Backend werden auf korrekte Datenverarbeitung, Fehlerbehandlung und Performance getestet.

Nicht Bestandteil dieses Testkonzepts sind z. B. externe Dienste Dritter, auf die lediglich zugegriffen wird (etwa E-Mail-Dienste), sofern deren Funktionsweise nicht direkt durch das System beeinflussbar ist.

2.2.2 Berücksichtigung von funktionalen und nicht-funktionalen Anforderungen

Die Tests berücksichtigen sowohl:

- **Funktionale Anforderungen** – z. B. “Ein Benutzer kann sich registrieren”, “Ein Organisator kann ein Event erstellen”, “Ein Teilnehmer kann sich für ein Event anmelden”.
- **Nicht-funktionale Anforderungen** – z. B. Reaktionszeiten, Sicherheit der Benutzerdaten, Systemverhalten bei hoher Last, Zugänglichkeit (Accessibility).

2.3 Testarten

2.3.1 Definition

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 (Modultests):** Einzelne Methoden oder Klassen werden isoliert getestet, um sicherzustellen, dass sie die erwartete Funktionalität korrekt umsetzen. Dies betrifft insbesondere Backend-Komponenten wie Services und Controller. Beispiel für Unit Tests für das Eventmanagement-System

```
1   public class EventControllerTests
2   {
3       public async Task
            CreateEvent_ShouldReturnCreatedEvent_
            WhenEventIsValid()
4
5       {
6           // Arrange
7           var eventDto = new EventCreateDto { Title = "Test
                ␣Event", Location = "Stuttgart" };
8           var createdEvent = new EventBasicDetailedDto { Id
                 = 1, Title = "Test␣Event", Location = "
                Stuttgart" };
9
10          _mockEventRepository.Setup(repo => repo.AddAsync(
                It.IsAny<EventCreateDto>()))
11              .ReturnsAsync(createdEvent);
12
13          // Act
14          var result = await _controller.CreateEvent(
                eventDto);
15
16          // Assert
17          var actionResult = Assert.IsType<ActionResult<
                EventBasicDetailedDto>>(result);
18          var createdAtActionResult = Assert.IsType<
                CreatedAtActionResult>(actionResult.Result);
19          Assert.Equal(201, createdAtActionResult.
                StatusCode);
20          Assert.Equal("Test␣Event", createdAtActionResult.
                Value.Title);
```

```
21              }
22          }
```

Listing 2.1: Unit Test für CreateEvent Methode in EventController

- **Integrationstests:** Diese Tests prüfen die Kommunikation zwischen mehreren Module, z. B. ob die API korrekt mit der Datenbank kommuniziert, das Frontend erwartungsgemäß mit dem Backend interagiert (siehe Listing 2.2).

```
1             [Fact]
2             public async Task
                  CreateEvent_ShouldPersistEventInDatabase()
3             {
4                 // Arrange
5                 var eventDto = new EventCreateDto { Title = "Test␣
                      Event", Location = "Stuttgart" };
6                 var options = new DbContextOptionsBuilder<
                      ApplicationDbContext>()
7                     .UseInMemoryDatabase(databaseName: "
                          TestDatabase")
8                     .Options;
9
10                using (var context = new ApplicationDbContext(
                      options))
11                {
12                    var eventRepository = new EventRepository(
                          context);
13                    var controller = new EventController(
                          eventRepository, new Mock<ILogger<
                          EventController>>().Object);
14
15                    // Act
16                    var result = await controller.CreateEvent(
                          eventDto);
17
18                    // Assert
19                    var createdEvent = await context.Events.
                          FirstOrDefaultAsync();
20                    Assert.NotNull(createdEvent);
21                    Assert.Equal("Test␣Event", createdEvent.Title)
                          ;
22                }
23            }
```

Listing 2.2: Integration Test für die Kommunikation zwischen EventController und EventRepository

- **Systemtests:** Hierbei wird das gesamte Eventmanagement-System als Ganzes getestet, um zu überprüfen, ob es die Anforderungen vollständig und korrekt erfüllt. Diese Tests beinhalten typische Nutzerflüsse wie Registrierung, Eventerstellung oder Eventteilnahme. In Listing2.3 ist ein Beispiel zu finden, das ermöglicht Anwender*innen/Organizer*innen ein Event zu erstellen.

```
[Fact]
public async Task UserCanCreateEvent()
{
    // Arrange
    var user = new User { UserName = "testuser", Password = "
        Password123", UserRole="Admin" or "␣Organizer" };
    var eventDto = new EventCreateDto { Title = "Test␣Event",
        Location = "Stuttgart" };
    var mockUserService = new Mock<IUserService>();
    mockUserService.Setup(s => s.GetUserByUsername(It.IsAny<
        string>())).ReturnsAsync(user);

    var eventService = new EventService(mockUserService.Object,
        new EventRepository(_dbContext));
    var controller = new EventController(eventService);

    // Act
    var result = await controller.CreateEvent(eventDto);

    // Assert
    var createdEvent = await _dbContext.Events.
        FirstOrDefaultAsync(e => e.Title == "Test␣Event");
    Assert.NotNull(createdEvent);
    Assert.Equal("Test␣Event", createdEvent.Title);
}
```

Listing 2.3: Beispiel für einen Systemtest: Eventerstellung durch einen Nutzer

- **Akzeptanztests (User Acceptance Tests):** Diese Tests werden unter realitätsnahen Bedingungen durchgeführt, um sicherzustellen, dass das System die Erwartungen der Endnutzer*innen erfüllt. Idealerweise wirken hier auch Stakeholder*innen oder Endnutzer*innen mit. In Listing2.4 ist ein Beispiel zu finden, das erlaubt Endnutzer*innen sich zu registrieren und dann sich einloggen.

```
1
2       [Fact]
3   public async Task UserShouldBeAbleToRegisterAndLogIn()
4   {
5       // Arrange
6       var userDto = new UserDto { UserName = "newuser", Password =
            "␣Password123" };
7       var userService = new UserService(_dbContext);
8       var controller = new UserController(userService);
9
10      // Act
11      var registrationResult = await controller.Register(userDto);
12      var loginResult = await controller.Login(new LoginDto {
            UserName = "newuser", Password = "Password123" });
13
14      // Assert
15      Assert.IsType<OkObjectResult>(registrationResult);
16      Assert.IsType<OkObjectResult>(loginResult);
17  }
```

Listing 2.4: Beispiel für einen Akzeptanztest: Endnutzer Registration und Einloggen

- **Regressionstests:** Bei Änderungen im Code wird überprüft, ob bereits getestete Funktionen weiterhin korrekt arbeiten. Diese Tests helfen, unbeabsichtigte Seiteneffekte frühzeitig zu erkennen. In Listing2.5 ist ein Beispiel zu finden, die bestehende oder ausgewählte Events aktualisieren, wenn Änderungen stattgefunden haben.

```
1
2       [Fact]
3   public async Task GetEvent_ShouldReturnEventAfterUpdate()
4   {
5       // Arrange
6       var eventId = 1;
7       var eventDto = new EventCreateDto { Title = "Updated␣Event",
            Location = "Stuttgart" };
8
9       // Initial Event Creation
10      var createdEvent = new Event { Id = eventId, Title = "Old␣
            Event", Location = "Stuttgart" };
11      _dbContext.Events.Add(createdEvent);
12      await _dbContext.SaveChangesAsync();
13
14      // Update Event
15      var controller = new EventController(new EventRepository(
            _dbContext));
16      await controller.UpdateEvent(eventId, eventDto);
17
18      // Act
19      var result = await controller.GetEvent(eventId);
20
21      // Assert
22      var updatedEvent = Assert.IsType<OkObjectResult>(result.
            Result).Value as Event;
23      Assert.NotNull(updatedEvent);
24      Assert.Equal("Updated␣Event", updatedEvent.Title);
25  }
```

Listing 2.5: Beispiel für einen Regressionstests: Event Aktualisierung nach einem/mehreren Update(s)

2.3.2 Einsatzzeitpunkte

- **Unit-Tests:** Werden parallel zur Entwicklung einzelner Komponenten geschrieben. Ziel ist es, die Funktionalität kleiner Einheiten (z. B. Methoden oder Services) isoliert zu testen.
- **Integrationstests:** Werden nach Fertigstellung mehrerer Module durchgeführt, um sicherzustellen, dass diese korrekt miteinander interagieren (z. B. Repository mit Datenbank).
- **Ende-zu-Ende-Tests (E2E):** Überprüfen das Verhalten des Systems aus Sicht des Benutzers – von der Benutzeroberfläche bis zur Datenbank.
- **Manuelle Tests:** Kommen in frühen Phasen oder bei nicht automatisierbaren Fällen (z. B. UI-Design, Usability) zum Einsatz.

2.4 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.

2.4.1 Bereitstellung der Testdaten

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.
- **Systemtests:** Hier kann auf externe JSON/XML-Dateien mit vorgefertigten Testdaten zugegriffen werden. Alternativ werden automatisierte Skripte verwendet, die die Datenbank mit realistischer Testbefüllung versorgen.
- **Produktionsnahe Tests:** Anonymisierte oder synthetisch erzeugte Produktionsdaten dienen der realitätsnahen Testdurchführung.

Für die Konsistenz und Wiederverwendbarkeit der Testdaten werden Versionskontrolle (z. B. über Git) und einheitliche Namenskonventionen verwendet.

2.4.2 Beispiele für Testdaten

- Benutzer mit unterschiedlichen Rollen (Admin, Organizer, Teilnehmer)
- Events mit verschiedenen Statuswerten (geplant, aktiv, abgeschlossen)
- Organisationen mit mehreren zugeordneten Mitgliedern
- Registrierungen mit gültigen und ungültigen Eingaben

2.4.3 Datenschutz

Es werden ausschließlich synthetische Testdaten verwendet. Personenbezogene oder sensible Daten realer Nutzer finden in der Testumgebung keine Anwendung.

2.5 Testumgebung

2.5.1 Systemumgebung für die Testdurchführung

Für eine zuverlässige und konsistente Durchführung der Tests wird eine dedizierte Testumgebung verwendet, die weitestgehend der Produktionsumgebung entspricht. Dies stellt sicher, dass erkannte Fehler realitätsnah sind und spätere Überraschungen bei der Auslieferung vermieden werden.

2.5.2 Technische Komponenten der Testumgebung

- **Backend:** ASP.NET Core Web API, Version `8.0`
- **Frontend:** React (JavaScript/TypeScript)
- **Datenbank:** PostgreSQL, optional auch InMemory für Unit-Tests
- **Entwicklungsumgebung:** Visual Studio Code, .NET SDK, Node.js
- **Testframeworks:** xUnit für Unit- und Integrationstests, Postman/Newman für API-Tests, Playwright für UI-Tests
- **Betriebssystem:** Windows 11 / Linux /MacOS (je nach Entwickler-Setup)
- **Versionierung:** Git mit GitHub als Remote Repository.
- **CI/CD-Plattform:** GitHub Actions (für automatische Testausführung bei Commits)

2.5.3 Testwerkzeuge

- **xUnit:** Für Unit- und Integrationstests im Backend.
- **Postman:** Für manuelles Testen der REST-API.
- **Swagger:** Zur API-Dokumentation und zum Testen einzelner Endpunkte.
- **Docker:** Zur Isolierung von Services und einfachen Reproduktion der Umgebung.
- **EF Core InMemory:** Für schnelle und unabhängige Tests der Datenzugriffsschicht.
- **Playwright / Cypress:** Für End-to-End-Tests im Frontend – etwa zur Simulation von Benutzerinteraktionen in der Oberfläche.

- **Jest / React Testing Library:** Für Komponententests im Frontend (v. a. bei React-Anwendungen).

2.5.4 Spezielle Konfigurationen

Um die Testausführung effizient und zuverlässig zu gestalten, wurden folgende Konfigurationen vorgenommen:

- **Separate Konfigurationsdateien** (z. B. `appsettings.Test.json`) für Umgebungsvariablen wie Test-Datenbankverbindung.
- **Mocking von externen Diensten** (z. B. Authentifizierungsdienste, Zahlungsanbieter) mit Hilfe von Bibliotheken wie Moq.
- **Containerisierung:** Für Integrationstests wird die Anwendung optional in Docker-Containern getestet (z. B. PostgreSQL via Docker Compose).
- **Automatischer Datenbank-Reset** vor jedem Integrationstest, um konsistente Startbedingungen sicherzustellen.

2.5.5 Zugriff und Deployment

Die Testumgebung kann lokal über einen Startbefehl in Visual Studio Code oder über ein Docker-Compose-Skript gestartet werden. Für Teammitglieder ist der Zugriff über einen gemeinsamen GitHub-Workflow möglich, der das Testsystem bei jedem Push automatisch aufsetzt und validiert.

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		
Systemtests	C	R		
CI/CD einrichten			R	
Akzeptanztests		C		R
Fehlerdokumentation	R	R		
Teststrategie definieren	C	R	C	A

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

4 Testdokumentation

4.1 Zweck der Testdokumentation

Die Testdokumentation dient als nachvollziehbare Aufzeichnung aller durchgeführten Testaktivitäten und ihrer Ergebnisse. Sie ermöglicht die Überprüfung der Qualität des Eventmanagement-Systems, schafft Transparenz im Entwicklungsprozess und unterstützt die Kommunikation zwischen Entwicklerteam, Projektleitung und Stakeholdern.

4.2 Inhalte der Testdokumentation

Die Dokumentation umfasst folgende Elemente:

- **Teststrategie:** Beschreibung der eingesetzten Testarten und Testmethoden (Unit-, Integrations-, Systemtests etc.).
- **Testfälle:** Konkrete Testbeschreibungen inklusive Eingaben, erwarteten Ausgaben, Vorbedingungen und Nachbedingungen.
- **Testergebnisse:** Auflistung der durchgeführten Tests mit Angaben zu Erfolg/Misserfolg, ggf. inkl. Fehlermeldungen oder Screenshots.
- **Testabdeckung:** Angabe, welche Anwendungsbereiche, Komponenten und Use Cases abgedeckt wurden.
- **Fehlermanagement:** Dokumentation entdeckter Fehler, ihrer Klassifikation (z. B. kritisch/nicht-kritisch), ihres Status (offen/behoben) und der zugehörigen Änderungen im Code.

4.3 Werkzeuge zur Dokumentation

Zur effizienten Testdokumentation werden folgende Tools eingesetzt:

- **Azure DevOps / GitHub Issues:** Zur Erfassung und Verwaltung von Bugs, Testfällen und Testplänen.

- **xUnit Output + Logs:** Automatisierte Protokollierung der Testergebnisse aus dem Test-Framework.
- **Markdown-Dateien:** Für manuelle Testdokumentation in strukturierter Form direkt im Repository.
- **Swagger UI:** Zur Dokumentation und interaktiven Überprüfung der API-Endpunkte.

4.4 Beispielhafter Testfall

Test-ID	TC-001
Beschreibung	Überprüfung der erfolgreichen Erstellung eines Events durch einen Organizer.
Vorbedingung	Benutzer ist als Organizer angemeldet.
Eingabe	`POST /api/events` mit gültigem EventCreateDto
Erwartetes Ergebnis	HTTP 201 Created, Event wird in der Datenbank gespeichert.
Tatsächliches Ergebnis	HTTP 201 Created, Event gespeichert.
Status	Erfolgreich

Tabelle 4.1: Beispiel für dokumentierten Testfall

5 Testabläufe

5.1 Überblick

Die Testabläufe beschreiben die konkrete Durchführung der Tests im Rahmen des Entwicklungsprozesses. Sie beinhalten organisatorische und technische Aspekte, wie Tests geplant, vorbereitet, durchgeführt und ausgewertet werden. Ziel ist es, die Effizienz und Wiederholbarkeit der Tests sicherzustellen und eine nachvollziehbare Qualitätssicherung zu ermöglichen.

5.2 Testvorbereitung

Bevor Tests durchgeführt werden konnten, wurden folgende vorbereitende Maßnahmen getroffen:

- **Testumgebung:** Eine separate Testdatenbank wurde mit `InMemoryDatabase` konfiguriert, um reproduzierbare Testergebnisse zu garantieren.
- **Testframeworks:** Für die Backend-Tests wurde `xUnit` in Kombination mit `Moq` zur Simulation von Abhängigkeiten eingesetzt.
- **Seed-Daten:** Initiale Testdaten wurden programmiert eingebunden, um typische Anwendungsszenarien realistisch abzubilden.
- **Testfälle:** Es wurden für jede zentrale Funktionalität (z. B. Event-Erstellung, Benutzerregistrierung, Rollenprüfung) Testspezifikationen definiert.

5.3 Testdurchführung

Die Tests wurden automatisiert und manuell durchgeführt:

- **Automatisierte Tests:** Unit- und Integrationstests wurden automatisiert mit `xUnit` in der CI/CD-Pipeline über GitHub Actions ausgeführt. Die Ergebnisse wurden dabei direkt in das Repository eingebunden.

- **Manuelle Tests:** Für spezifische Use Cases, vor allem im Frontend, wurden manuelle Klicktests durchgeführt, insbesondere zur Überprüfung von Eingabevalidierungen und UI-Reaktionen.
- **Testdatenpflege:** Vor jedem Testlauf wurde die Testdatenbank zurückgesetzt, um konsistente Anfangszustände sicherzustellen.

5.4 Testfrequenz

Die Tests wurden in verschiedenen Phasen ausgeführt:

- Nach Abschluss eines Features (z. B. „Create Event“) wurde lokal ein vollständiger Testlauf durchgeführt.
- Vor jedem Merge in den Haupt-Branch wurden alle automatisierten Tests durch CI/CD überprüft.
- Vor Release-Meilensteinen wurden zusätzlich vollständige Regressionstests durchgeführt.

5.5 Beispielhafter Ablauf: Event-Erstellung

1. Entwickler implementiert neue Event-Erstellungslogik im Backend.
2. Er schreibt dazu einen Unit-Test mit `xUnit` (siehe Listing 2.2).
3. Test wird lokal ausgeführt und überprüft.
4. Änderungen werden gepusht – GitHub Actions prüft automatisiert den Teststatus.
5. Bei Erfolg erfolgt Merge in den Haupt-Branch; bei Fehlern erfolgt Analyse und Korrektur.

5.6 Ergebnisanalyse

Am Ende jedes Meeting wurde eine kurze Testauswertung erstellt:

- Anzahl durchgeführter Tests
- Anzahl fehlgeschlagener Tests
- Identifizierte Fehler und Bugfixes
- Offene Tickets für zukünftige Tests

5.7 Zusammenfassung

Durch strukturierte Testabläufe und den Einsatz moderner Testwerkzeuge konnte sichergestellt werden, dass Fehler frühzeitig entdeckt und behoben wurden. Die Kombination aus automatisierten Tests im Backend und manuellen Tests im Frontend trug wesentlich zur Qualität des Gesamtsystems bei.

6 Abnahmekriterien

6.1 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).

6.2 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 des Eventmanagement-Systems.

7 Fazit und Zusammenfassung

Im Rahmen dieses Projekts wurde ein umfassendes Testkonzept für das Eventmanagement-System entwickelt und umgesetzt. Ziel war es, die Qualität der Software durch strukturierte, automatisierte und manuelle Tests sicherzustellen sowie eine nachhaltige Teststrategie für zukünftige Erweiterungen zu etablieren.

7.1 Entscheidungen über Teststrategie und Werkzeuge

Basierend auf den Anforderungen und den eingesetzten Technologien im Projekt wurde folgende Teststrategie gewählt:

- **xUnit** wurde als Framework für Unit- und Integrationstests im Backend ausgewählt, da es gut mit dem .NET-Ökosystem integriert ist, eine einfache Teststruktur bietet und von der Community breit unterstützt wird.
- **Entity Framework Core InMemory Provider** kam in Integrationstests zum Einsatz, um eine schnelle und unabhängige Prüfung von Repository-Logik zu ermöglichen, ohne auf eine echte Datenbank angewiesen zu sein. Dies erlaubt realistische Testszenarien bei minimalem Setup-Aufwand.
- **Mocking (z. B. mit Moq)** wurde genutzt, um abhängige Komponenten wie Logger oder Services zu isolieren. So konnten gezielt einzelne Klassen oder Methoden getestet werden, ohne Seiteneffekte durch abhängige Systeme zu riskieren.
- **Swagger UI** wurde verwendet, um die REST-Schnittstellen dokumentiert und interaktiv testbar zu machen. Es ermöglichte dem Team wie auch externen Testern, API-Endpunkte manuell zu überprüfen.
- **Postman** wurde ergänzend zur Verifikation komplexerer HTTP-Kommunikation und Authentifizierungsmechanismen eingesetzt, z. B. für Tests mit Bearer-Token.
- **Docker** wurde in der späteren Testphase für das Deployment in einer isolierten Umgebung genutzt. So konnte sichergestellt werden, dass die Services auch außerhalb der Entwicklungsumgebung stabil laufen.

- **Manuelle Tests im Frontend** (z. B. über Browser Developer Tools und Benutzerinteraktionen) wurden durchgeführt, jedoch nicht automatisiert. Aufgrund der Projektpriorität lag der Fokus auf der Sicherung der Backend-Funktionalität.

7.2 Ausblick

Für zukünftige Versionen des Systems wird empfohlen:

- Die Testabdeckung weiter auszubauen, insbesondere im Bereich Frontend-Tests (z. B. mit Playwright oder Cypress).
- E2E-Tests (End-to-End) einzuführen, um Benutzerworkflows systemübergreifend zu testen.
- Continuous Integration/Delivery (CI/CD) Pipelines mit automatischem Testlauf und Reporting zu etablieren.

7.3 Schlusswort

Die gewählten Technologien und Methoden haben sich im Projektverlauf als sinnvoll und effizient erwiesen. Durch die Teststrategie konnte eine hohe Codequalität sichergestellt und das Risiko kritischer Fehler deutlich reduziert werden. Die Testdokumentation bietet zudem eine solide Grundlage für Wartung, Weiterentwicklung und spätere Releases des Systems.