Projekt: **Event-Tool**

Testkonzept – Version 1.0

Abgabedatum: 21.04.2025

Projektgruppe: BitWorks
Ansprechpartner: Tobias Schuhmacher – *Projektleiter*

Inhaltsverzeichnis

1 Einleitung 3
2 Organisatorisches 4
2.1 Test-Driven-Development 4
2.2 Verantwortlichkeiten 4
2.3 Zentralisierung und Abgabe der Testergebnisse 4
3 Unit Tests 4
3.1 Frameworks für Unit Tests 5
3.1.1 xUnit und bUnit 5
3.1.2 MS-Test 5
3.2 Verwendung 6
3.3 Eingaben und Ausgaben 6
4 Integrationstests 6
4.1 Verwendung 7
4.2 Mock-Objekte 7
5 Systemtests 7
5.1 Durchführung 8
5.2 Testziele 8
5.3 Testumgebung 8
5.4 Dokumentation 8
6 Github 9
6.1 Allgemein 9
6.2 Umgang mit Entwicklungsfehlern 9
6.2.1 Fehlerhafte Commits auf dem Feature-Branch 9
6.2.2 Merge-Konflikte beim Pull Request 10
6.2.3 Build- oder Testfehler nach dem Push 10
6.2.4 Unnötige Branches 10
6.2.5 Entwicklung und Veröffentlichung 10

1 Einleitung

Dieses Dokument behandelt das Testkonzept der Gruppe BitWorks im Rahmen der Entwicklung des ‚Event-Tool‘-Projektes, in welchem ein Event-Management-Tool konzipiert, entwickelt und implementiert wird. Das Projekt wird in C# mit dem .NET 8 Framework sowie RazorPages als Frontend umgesetzt.

Ziel des Testkonzeptes ist es, die Funktionalität, Integrität und Qualitätssicherung der Software sicherzustellen. Da das System lokal betrieben wird, liegt ein erhöhtes Augenmerk auf stabilen Datenbankoperationen, konsistenter Oberfläche und Qualitätssicherung der kritischen Funktionen.
Die im Folgenden vorgestellten Testarten stellen Unit-, Integrations- und Systemtests ab.
Wichtige Frameworks sind

- xUnit für Unit- und Integrationstests für das Backend
- MSTest für Unit- und Integrationstests für das Backend
- bUnit für Unit- und Integrationstests für das Frontend (RazorPages)
- manuelles Testen und falls nötig Playwright für E2E Tests

um die gewünschte Abdeckung des Stacks zu erreichen.

2 Organisatorisches

2.1 Test-Driven-Development

Ein wichtiger Bestandteil des Testkonzeptes besteht in der Art und Weise, nach der Tests konzipiert, geschrieben und durchgeführt werden.
Beim *TDD (Test-Driven-Development)* werden für jeden Codeabschnitt Tests geschrieben, sodass es eine hohe Abdeckung gibt.
Auch die Entwicklung von Code folgt dem Prinzip, Tests für jede Methode, Klasse und weiteres zu schreiben, **bevor** dieser implementiert wird. In diesem Projekt werden Bestandteile des TDD genutzt, um eine optimale Abdeckung, eine Test-geführte Entwicklung und ein erfolgreiches Produkt zu schaffen.

2.2 Verantwortlichkeiten

Die Einrichtung des Testpakets und das Erstellen der einzelnen Tests ist eine Verantwortlichkeit des Testers. Dieser richtet ebenso die GitHub-Actions ein und kümmert sich somit um das automatisierte Testen von neuem Code.

Die Entwickler und restlichen Projektteilnehmer haben die Verantwortlichkeit, mit dem Tester zusammen Code/ Tests zu entwickeln, um die gewünschte Abdeckung zu erreichen. Hierbei kann, je nach Auslastung, vom Tester Arbeit an die restlichen Gruppenmitglieder deligiert werden.

2.3 Zentralisierung und Abgabe der Testergebnisse

Die Tests sollen auch im Rahmen unseres Projektes gesichert bzw. zentralisiert gesammelt werden. Hierfür werden durch xUnit/bUnit proprietäre Mittel genutzt, um Tests an wichtigen Eckpunkten des Projektverlaufes zu visualisieren und darzustellen.

Diese Visualisierungen bzw. Testprotokolle werden regelmäßig mit den restlichen Projektartefakten auf der Firmenwebsite gepflegt.
Über die GitHub-Actions Oberfläche lassen sich ebenfalls die jeweils durchgeführten Tests betrachten.

3 Unit Tests

Eine der bekanntesten Testarten ist der Unit-Test, auch als Modul bzw. Komponententest bezeichnet. Unit-Tests prüfen einzelne Methoden oder Funktionen auf ihr korrektes Verhalten. Dabei wird eine

Methode mit bereits definierten Eingabewerten aufgerufen und die Ausgabe wird mit dem erwarteten Ergebnis verglichen.
Da der Quellcode bekannt ist und gezielt getestet wird, handelt es sich bei Unit-Tests um sogenannte **White Box Tests**.
Diese Tests lassen sich sehr gut automatisieren und in den Entwicklungsprozess integrieren. Sie ermöglichen schnelles Feedback bei Änderungen und helfen dabei, Regressionen frühzeitig zu erkennen.

3.1 Frameworks für Unit Tests

3.1.1 xUnit und bUnit

Für das „Event-Tool“-Projekt kommen **xUnit** und **bUnit** als Unit-Test-Framework zum Einsatz. Bei xUnit handelt sich um ein modernes, weit verbreitetes .NET-Testframework mit guter Integration in Visual Studio und GitHub.
Wichtiger Bestandteil des Testens sind die *Assert*-Methoden. Diese Methoden zielen darauf ab, etwas sicherzustellen, indem sie prüfen, ob ein erwartetes Ergebnis tatsächlich gegeben ist.
Typische Assert-Methoden des xUnit Frameworks:

Methode	Beschreibung
Assert.Equal(expected, actual)	Prüft, ob zwei Werte gleich sind.
Assert.True(condition)	Erwartet, dass die Bedingung true ist.
Assert.False(condition)	Erwartet, dass die Bedingung false ist.
Assert.Throws<TException>(() => ...)	Erwartet, dass beim Ausführen der Methode eine bestimmte Exception auftritt.

Weitere wichtige Attribute sind ‚Fact’, welche eine normale Testmethode, also ein einzelner, eigenständiger Test ist und ‚Theory’ sowie ‚InlineData()‘, welche sich besonders gut für das Testen von z.B. Grenzwerten eignet.
Testklassen werden automatisch erkannt, wenn sie öffentlich sind und Methoden das [Fact]-Attribut besitzen. Setup- und Cleanup-Logik kann über Konstruktor umgesetzt werden.
XUnit wird im Rahmen dieses Projektes zum Einsatz im in der Backend-Logik kommen. Somit werden die wichtigen Klassen, Methoden und Strukturen, welche für den logischen Aufbau im Hintergrund verantwortlich sind, mit xUnit getestet.

bUnit ist ein auf Blazor spezialisiertes Test-Framework und wird gerade im Frontend im Bezug auf die Razor-Pages Verwendung finden. Hierbei können gerade Ansichten, Buttons und alles was mit RazorPages zu tun hat gut getestet werden.

3.1.2 MS-Test

MSTest ist ein von Microsoft entwickeltes .NET-Testframework, welches ebenfalls direkt in Visual Studio verfügbar ist. In MSTest werden Testklassen angelegt und darin Testmethoden definiert. Auch hier gibt es wieder Assert-Methoden wie ‘Assert.AreEqual()’, ‘Assert.IsTrue’ und weitere Methoden.

Tests von MSTest laufen direkt im Test-Explorer und benötigen keine Extra-Pakete, was die Anwendung vereinfacht.
In diesem Projekt soll MSTest eine unterstützende Rolle einnehmen für Anwendungsfälle, bei denen xUnit nicht gut passt.

3.2 Verwendung

Im Projekt soll vor dem Implementieren einer Methode Unit-Tests für diese geschrieben werden. Nicht jede einfache Methode benötigt einen Test (z. B. einfache Getter/Settermethoden), jedoch sollten alle **wichtigen Methoden** abgedeckt werden – insbesondere solche mit Berechnungen, Validierungen oder Zustandsänderungen.
Ziel ist es, mindestens **75 % *Code Coverage*** zu erreichen.
Best Practice:

- Pro Testmethode möglichst nur einen logischen Testfall prüfen.
- Aussagekräftige Namen für Testmethoden wählen (MethodeXY_GibtErwartetesErgebnis_FallsBedingungZ), um durch die Benamung der Methode bereits zu wissen, was diese macht.
- Keine unnötige Abhängigkeit von der Datenbank – stattdessen werden Mock-Objekte (Simulierte Platzhalter für externe Strukturen) verwendet.

3.3 Eingaben und Ausgaben

Beim Erstellen der Tests sollten realistische und auch grenzwertige Eingaben getestet werden. Besonders bei Datenbankoperationen ist das wichtig.
Zu beachten:

- **Grenzfälle** prüfen (z. B. leere Eingabe, maximale Länge oder unerwartete Zeichen)
- **Zahlenbereiche** abdecken (z. B. negative, null und positive Werte bei Berechnungen und Eingaben)
- **Exakte Vergleiche** durchführen, wenn möglich (mit z.B. „Assert.Equal“, ebenfalls bei Bereechnugen und Eingaben)
- **Fehlerverhalten** testen, z. B. bei ungültiger Eingabe oder Ausnahmebedingungen

4 Integrationstests

Eine weitere Art der Tests sind Integrationstests. Sie prüfen, ob mehrere Komponenten der Software – wie Datenbankzugriffe, Services, Razor Pages oder API-Endpunkte – korrekt miteinander funktionieren. Dabei werden ganze Funktionseinheiten als ein Objekt getestet, also z. B. ein Controller mit Datenbankanbindung.
Integrationstests sind aufwendiger als Unit-Tests, da sie Abhängigkeiten wie Datenbanken einbinden. Sie sind essenziell, um sicherzustellen, dass z. B. Benutzer korrekt gespeichert, Events erstellt oder Prozessschritte verarbeitet werden. Da dabei keine internen Implementierungsdetails getestet

werden, handelt es sich um sogenannte **Black Box Tests**, der Tester hat somit keinerlei Wissen über die interne Logik der Strukturen.

4.1 Verwendung

Für jede zentrale Funktionseinheit (z. B. Eventverwaltung, Benutzerregistrierung oder Rechtevergabe) sollen Integrationstests erstellt werden. Diese werden mit dem Framework **xUnit** umgesetzt.
Die Tests führen echte Aufrufe auf HTTP-Ebene durch z. B. ein POST-Request auf /Event/Create und prüfen sowohl die Rückgaben als auch die Änderungen in der Datenbank.
Beispielhafte Tests:

- POST /Register legt Benutzer in DB an
- GET /Event/Details gibt vollständige Eventdaten zurück
- Event mit unvollständigen Angaben wird abgelehnt

Integrationstests werden regelmäßig lokal ausgeführt – vor allem nach Änderungen an Repositories oder Views. Ein fehlschlagender Integrationstest muss analysiert und wieder angepasst werden.

4.2 Mock-Objekte

Für bestimmte Tests, bei denen externe Abhängigkeiten noch nicht verfügbar sind oder nicht benötigt werden, können sogenannte ***Mock-Objekte*** verwendet werden. Diese simulieren beispielsweise den Versand von E-Mails, um das Systemverhalten zu testen, ohne wirklich E-Mails zu verschicken und vereinfachen somit die Tests.
Auch Services wie Benutzerverwaltung oder Rollenprüfung können über Mock-Objekte simuliert werden. Dabei gilt jedoch:
So viele echte Komponenten wie möglich einbinden – Mocking nur zur Isolierung oder wenn notwendig.

5 Systemtests

Systemtests, auch als End-to-End-Tests bekannt, überprüfen die gesamte Anwendung als ein Gesamtsystem. Dabei wird das Verhalten der Software aus Sicht eines Benutzers, von der Anmeldung über das Anlegen von Events bis zur Verarbeitung von Teilnehmern oder dem Abrufen von Dateien, simuliert.
Systemtests prüfen somit, ob alle Bestandteile des Systems wie UI, Backend, Datenbank und/oder externe Schnittstellen im Zusammenspiel korrekt funktionieren. Ziel ist es, reale Nutzungsszenarien vollständig durchzugehen und potenzielle Fehler zu erkennen, die in isolierten Tests nicht sichtbar wären.

5.1 Durchführung

Da in diesem Projekt kein spezielles Test-Framework geplant ist, erfolgt die Durchführung der Systemtests **manuell,** sollte hierbei der Aufwand zu hoch werden, kann auch **Playwright** eingesetzt werden. Die Tests werden bei manueller Prüfung in Testfällen dokumentiert und von Projektmitgliedern durch Klick-Tests in der laufenden Anwendung durchgeführt.
Testpersonen folgen vordefinierten Testskripten bzw. Anleitungen und dokumentieren ihre Beobachtungen und Testergebnisse. Änderungen am Verhalten oder Design der Benutzeroberfläche müssen dabei ebenfalls berücksichtigt werden.

5.2 Testziele

Die wichtigsten Ziele der Systemtests sind:

- **Registrierung** neuer Benutzer mit Bestätigungs-E-Mail
- **Anmeldung** und **Wechsel** zwischen Organisationen
- **Erstellung von Events** inkl. Prozessschritten und Datei-Funktionen
- **Einladung und Verwaltung** von Teilnehmern
- **Eventeinsicht und -teilnahme** von Mitgliedern
- **Automatisiertes Auslösen von Prozessschritten**
- **Archivierung** und **Zugriff** auf vergangene Events

Alle diese Anwendungsfälle sollten vollständig geprüft werden, mit jeweils mit typischen Eingaben, aber auch mit fehlerhaften Daten wie fehlende Felder, doppelte Namen, große Dateien, da diese Punkte auch die Kernlogik des gesamten Produktes bilden.

5.3 Testumgebung

Die Tests werden in einer lokalen Umgebung durchgeführt:

- Zugriff erfolgt über localhost
- Es wird eine vorkonfigurierte PostgreSQL-Testdatenbank verwendet
- Bei Bedarf werden Test-Accounts mit unterschiedlichen Rollen (Admin, Owner, Organisator, Mitglied) angelegt
- Tests werden im *späteren* Verlauf des Projektes über GitHub-Actions durchgeführt, wobei bei jedem Commit einmal alle Tests durchgeführt werden

5.4 Dokumentation

Die Testergebnisse werden in einem **Testprotokoll** dokumentiert. Dieses enthält Informationen wie Testfall-ID und die zugehörige Beschreibung, Eingaben und das zu erwartende Verhalten, das

tatsächliche Verhalten, den Status ob Erfolg oder Fehlschlag und Hinweise oder Screenshots bei entsprechendem Misserfolg.

6 Github

6.1 Allgemein

Für die Entwicklung des Event-Tools soll **GitHub** als zentrale Plattform für die Quellcodeverwaltung und Zusammenarbeit verwendet werden.
Die Versionsverwaltung wird mithilfe von Branches organisiert, um eine parallele Entwicklung zu ermöglichen und stabile Releases erstellen zu können.

Es ist im Verlauf des Projekts geplant, **automatisierte Test und Build-Prozesse** über **GitHub Actions** einzurichten. Dadurch sollen bei jedem Commit:

- Builds automatisch ausgeführt
- alle Tests (xUnit, bUnit) validiert
- Fehler mit einem roten Kreuz markiert werden

Solange diese Automatisierung noch nicht umgesetzt ist, wird die Prüfung **manuell** erfolgen.

6.2 Umgang mit Entwicklungsfehlern

Auch wenn man möglichst fehlerfrei entwickeln möchte, lassen sich Probleme beim Programmieren und Zusammenführen von Code nie ganz vermeiden. In Kombination mit GitHub und Visual Studio gibt es jedoch einige Mechanismen, mit denen Fehler erkannt, rückgängig gemacht oder sauber aufgelöst werden können. Dieses Kapitel beschreibt den geplanten Umgang mit typischen Problemen.

6.2.1 Fehlerhafte Commits auf dem Feature-Branch

Falls auf einem Feature-Branch fehlerhafte Commits gemacht werden, sollen diese mithilfe von *Reverts* oder *git reset* rückgängig gemacht werden. In Visual Studio kann dies direkt über die Git-UI erfolgen. Falls die Änderungen bereits gepusht wurden, wird ein Pull Request mit entsprechenden Korrekturen erstellt.
Wenn ein Feature-Branch instabil ist, soll dieser **nicht** in den main-Branch geführt werden. Die betroffenen Teammitglieder stimmen sich ab und beheben die Probleme lokal.

6.2.2 Merge-Konflikte beim Pull Request

Kommt es beim Merge eines Pull Requests zu Konflikten, sollen diese in Visual Studio oder über die GitHub-Weboberfläche manuell aufgelöst werden. Dabei ist darauf zu achten, dass keine funktionale Logik überschrieben oder versehentlich entfernt wird.
Falls der Merge zu unübersichtlich wird, kann der Feature-Branch nochmals mit dem aktuellen main-Stand synchronisiert werden, um Konflikte im Vorfeld zu bereinigen.

6.2.3 Build- oder Testfehler nach dem Push

Sollten nach einem Push Fehler beim Build oder bei Tests auftreten, sollen diese lokal mit Visual Studio und dem integrierten Test-Explorer überprüft werden. Alle Tests im Projekt müssen **erfolgreich** durchlaufen, bevor der Code über einen Pull Request zusammengeführt wird.
Die Build-Fehler werden über die Fehlerliste in Visual Studio angezeigt und müssen behoben werden, bevor erneut gepusht wird.

6.2.4 Unnötige Branches

Nicht mehr benötigte Branches wie abgebrochene Features sollten in GitHub geschlossen **und** gelöscht werden, um die Übersichtlichkeit zu wahren. Dies geschieht nach dem Merge oder wenn das Feature verworfen wurde.
Wenn Branches veraltet sind, kann mit ‚git pull origin main‘ der Branch aktualisiert und neu gestartet werden. Falls der Branch irreparabel ist, wird er gelöscht und ein neuer Branch basierend auf main angelegt.

6.2.5 Entwicklung und Veröffentlichung

Die Entwicklung des Projektes erfolgt im Github-Repository Main-Branch. Für jede neue wöchentliche Abgabe wird ein eigener Feature-Branch erstellt. In diesem Feature-Branch wird die Implementierung und Entwicklung des wöchentlichen Softwareinkrements abgearbeitet. Sobald der 1-Wochen-Sprint abgeschlossen ist, wird der Feature-Branch im Main-Branch zusammengeführt und von diesem Main-Branch wird ein Release-Branch erstellt, welcher lauffähig ist und dem Kunden zu Demonstrationszwecken zur Verfügung gestellt wird.
Tests werden auf seperaten Test-Branches entwickelt, die aus den jeweiligen Feature-Branches entstehen.