FACHHOCHSCHUL-MASTERSTUDIENGANG

RSE

DevOps Projekt

Dokumentation

Wintersemester 2025/26

Protokolltitel

Testautomatisierung mit GitHub Actions für die Verwaltung von Kontaktdaten

Verantwortlicher Autor / Matrikelnummer:

Michael Hörtenhuber / S2410828004

Verantwortlicher Autor / Matrikelnummer:

Philipp Weismann / S2410828009

LVA-Leiter:

Dr. Ernest Wallmüller

Abgabetermin:

12.12.2025

Inhaltsverzeichnis

1 Aufgabenstellung

Die Verwaltung von Kontaktdaten ist ein typisches Beispiel für eine kleine, aber realitätsnahe Anwendung, die sich hervorragend eignet, um moderne DevOps-Praktiken praktisch umzusetzen. Ziel dieses Projekts war es daher nicht nur, eine funktionale Kontaktdatenverwaltung zu entwickeln, sondern vor allem einen vollständigen, automatisierten Test- und Build-Prozess aufzubauen. Damit soll demonstriert werden, wie Qualitätssicherung, Testautomatisierung und kontinuierliche Integration in einem professionellen Entwicklungsworkflow zusammenspielen.

Im Rahmen der Aufgabenstellung war eine einfache Kontaktdatenverwaltung zu implementieren, die das Anlegen, Bearbeiten, Löschen und Suchen von Kontakten ermöglicht. Zusätzlich mussten grundlegende Validierungsregeln (z. B. E-Mail-Format, nichtleere Telefonnummer) umgesetzt werden. Auf Basis dieser Funktionalität war eine Testspezifikation zu erstellen sowie Unit- und Integrationstests zu implementieren. Schließlich sollten diese Tests über GitHub Actions automatisch ausgeführt werden, sodass bei jedem Push und Pull-Request die Korrektheit der Anwendung überprüft und entsprechende Testprotokolle bereitgestellt werden.

2 Analyse & Spezifikation

2.1 Anforderungen und Funktionsumfang der Applikation

Die Kontaktdatenverwaltung dient als digitale Lösung zur strukturierten Erfassung und Pflege von personenbezogenen Kontaktdaten. Der funktionale Mindestumfang orientiert sich an der Aufgabenstellung der Lehrveranstaltung und umfasst das Anlegen, Bearbeiten, Löschen und Suchen von Kontakten. Jeder Kontakt besteht aus den Feldern *Name*, *Firma*, *Telefonnummer*, *E-Mail-Adresse* und *Adresse*. Zusätzlich wurden einfache, aber klar definierte Validierungsregeln umgesetzt: Der Name und die Telefonnummer dürfen nicht leer sein, und die E-Mail-Adresse muss ein gültiges Grundformat aufweisen (enthält „@“, enthält einen Punkt, keine Leerzeichen). Weiterhin bietet die Anwendung eine Suche sowohl nach vollständigen als auch nach teilweisen Namensbestandteilen sowie eine Filterfunktion über mehrere Felder, wie etwa Name und Firma. Diese Anforderungen stellen sicher, dass die Anwendung für typische CRUD- und Suchszenarien geeignet ist und gleichzeitig eine verlässliche Datenqualität gewährleistet wird.

2.2 Domänenmodell

Das Domänenmodell bildet die zentrale Datenstruktur der Anwendung ab und definiert, welche Eigenschaften ein Kontakt besitzen muss. Die Klasse *Contact* repräsentiert dabei einen einzelnen Kontakteintrag und enthält die Felder *Id*, *Name*, *Company*, *Phone*, *Email* und *Address*. Die Id wird beim Erstellen eines Kontakts automatisch generiert und ermöglicht die eindeutige Identifikation im gesamten System. Das Modell ist bewusst schlank gehalten, um eine klare Trennung zwischen Datenstruktur, Geschäftslogik und Persistenzschicht zu gewährleisten. Gleichzeitig ist es so gestaltet, dass es sich leicht testen und erweitern lässt, beispielsweise um zukünftige Attribute oder Validierungsregeln. Dieses einfache, klar definierte Domänenmodell bildet die Grundlage für die Repository-Logik, die Validierung sowie die Suchfunktionen.

2.3 Schnittstellen & Architekturentscheidungen

Die Architektur der Anwendung folgt einem modularen Aufbau, der eine klare Trennung von Verantwortlichkeiten sicherstellt und die Testbarkeit erhöht. Zentrales Element ist das Repository-Pattern, das über das Interface IContactRepository definiert ist. Es kapselt sämtliche Datenzugriffe und bietet Methoden zum Erstellen, Aktualisieren, Löschen und Suchen von Kontakten. Die konkrete Implementierung erfolgt im JsonContactRepository, das alle Kontakte in einer JSON-Datei speichert. Durch diese Abstraktion bleibt der restliche Anwendungscode unabhängig von der Persistenztechnologie und kann im Testkontext durch ein InMemoryContactRepository ersetzt werden.

Für die Eingabevalidierung wurde eine eigene Komponente (ContactValidator) eingeführt, die zentrale Validierungsregeln bündelt und einheitlich aufruft. Dadurch kann sowohl die UI als auch jede zukünftige API oder Backend-Komponente denselben Validierungsmechanismus nutzen.

Die Benutzeroberfläche wurde mit WinForms umgesetzt, da sie unter Windows einfach einzurichten ist und eine klare Trennung zwischen UI und Logik erlaubt. Die UI interagiert ausschließlich über das Repository und den Validator mit dem Backend, sodass keine Logik in der Oberfläche selbst enthalten ist. Dieser modulare Aufbau unterstützt DevOps-Praktiken wie automatisierte Tests, refactor-sicheren Code und eine saubere Continuous-Integration-Pipeline.

3 Softwareentwicklung

3.1 Architekturüberblick und Tech-Stack

Die Applikation basiert auf dem .NET-8 Framework und folgt einer modularen Schichtenarchitektur, die eine klare Trennung zwischen Datenmodell, Persistenz, Validierung und Benutzeroberfläche sicherstellt. Als UI-Technologie kommt WinForms zum Einsatz, da sie sich unter Windows einfach integrieren lässt und nur minimale Abhängigkeiten benötigt. Die Datenspeicherung erfolgt über ein JSON File, wodurch sich Daten ohne Datenbankserver lokal speichern und problemlos versionieren lassen. Die Tests wurden mit xUnit implementiert und über GitHub Actions automatisiert ausgeführt, was eine kontinuierliche Qualitätssicherung ermöglicht. Durch das Repository-Pattern ist der Anwendungskern unabhängig von der realen Datenquelle und damit hervorragend testbar – ein zentraler DevOps-Grundsatz.

3.2 Klassen

Contact.cs:

Das Domänenmodell definiert die Struktur eines Kontakts und enthält ausschließlich Daten, keine Logik. Es bildet die Grundlage für alle CRUD-Operationen und ermöglicht eine klare Trennung zwischen Modell und Verarbeitung.

ContactValidator.cs:

Diese Klasse enthält die zentralisierte Validierungslogik für Name, Telefonnummer und E-Mail-Adresse. Dadurch bleibt die Geschäftslogik konsistent und die UI sowie mögliche zukünftige APIs können dieselben Regeln verwenden.

IContactRepository.cs:

Das Interface legt alle notwendigen CRUD- und Suchmethoden fest und schafft eine einheitliche Zugriffsschicht auf Kontaktdaten. Durch diese Abstraktion wird die Anwendung entkoppelt und kann verschiedene Speicherimplementierungen verwenden, ohne dass sich der Rest des Systems ändert.

JsonContactRepository.cs:

Diese Klasse implementiert das Repository-Pattern mithilfe einer JSON-Datei als persistentem Speicher. Sie eignet sich ideal für eine leichtgewichtige Anwendung und trennt Dateizugriff vollständig von UI- und Validierungslogik.

InMemoryContactRepository.cs:

Diese Implementierung dient speziell für Tests und speichert alle Daten in einer temporären Speicherstruktur. Dadurch werden Integrationstests deterministisch, schnell und unabhängig vom Dateisystem oder externen Ressourcen.

Form1 (WinForms UI):

Die UI-Klasse stellt die Benutzeroberfläche dar und delegiert sämtliche Geschäftslogik an Repository und Validator. Durch diese Trennung bleibt die Oberfläche frei von Logik und die Kernfunktionen lassen sich unabhängig automatisiert testen.

In der folgenden Abbildung sieht man die Projektstruktur in die einzelnen Ordner und Dateien:

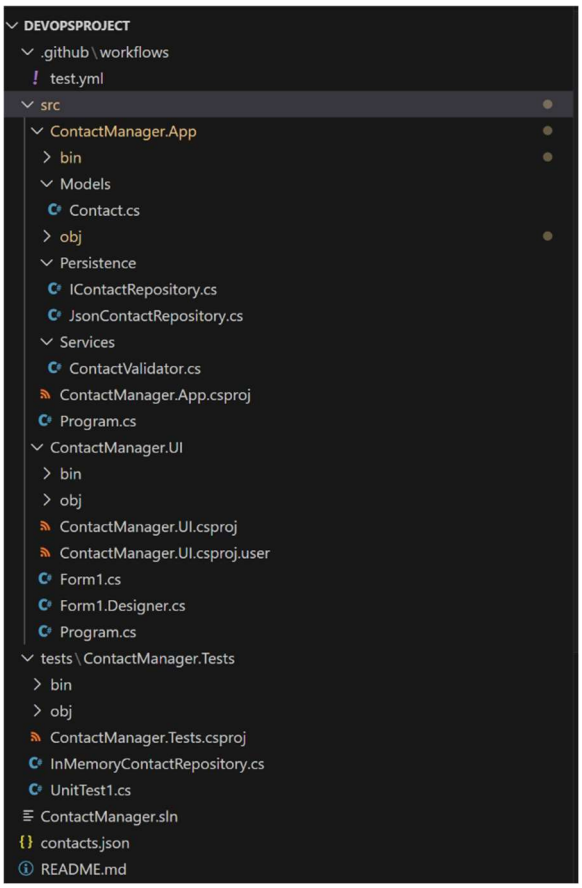

Figure 1 Projektstruktur

3.3 Benutzeroberfläche

Die Benutzeroberfläche wurde mit WinForms umgesetzt und dient als einfache, intuitive Steuerzentrale für alle CRUD-Operationen. Auf der linken Seite zeigt eine Liste alle vorhandenen Kontakte an, während rechts Eingabefelder und Buttons zum Hinzufügen, Bearbeiten, Löschen und Suchen bereitstehen. Die UI enthält keine Geschäftslogik, sondern nutzt ausschließlich das Repository (IContactRepository) zum Lesen und Schreiben der Daten sowie den ContactValidator zur Überprüfung der Eingaben. Dadurch bleibt die Logik vollständig vom Frontend getrennt, was nicht nur die

Wartbarkeit erhöht, sondern auch automatisierte Tests ermöglicht. Jede Benutzeraktion – wie das Klicken auf „Add“ oder „Update“ – löst ein Event aus, das die Validierung durchführt und anschließend die entsprechende Repository-Methode aufruft. In der folgenden Abbildung sieht man die Benutzeroberfläche.

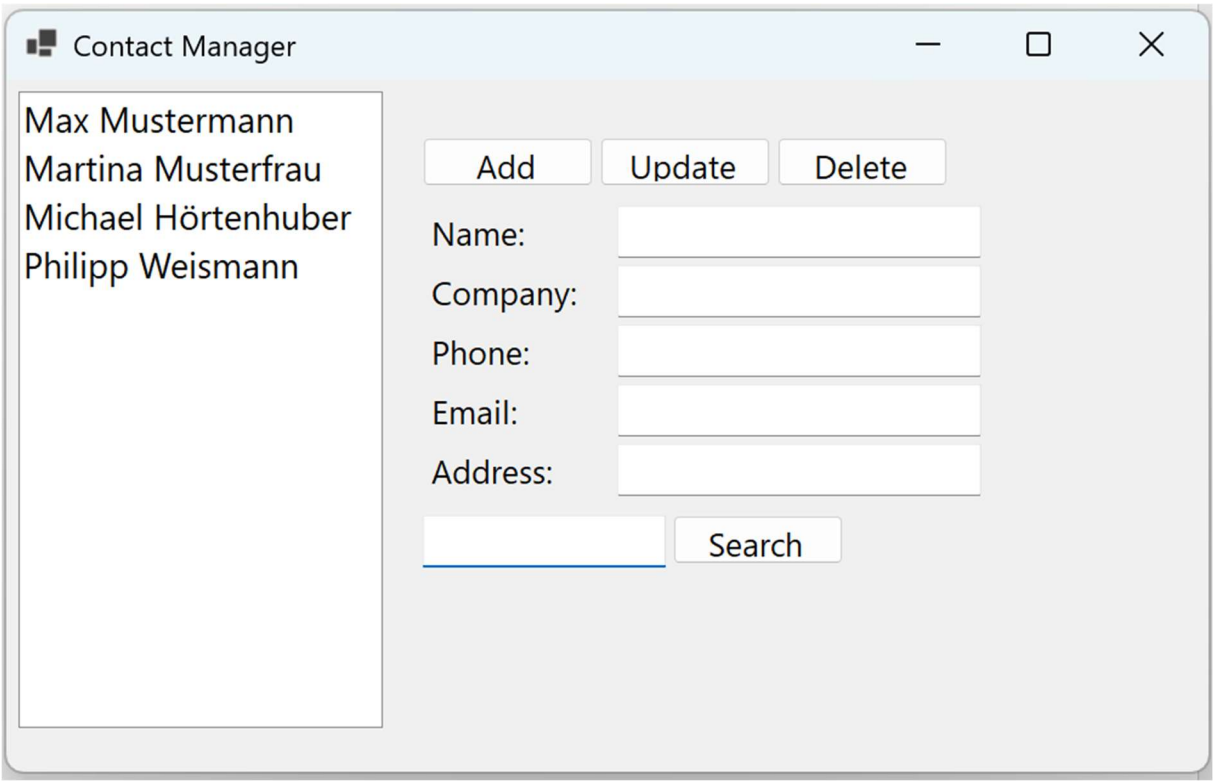

Figure 2 Benutzeroberfläche

3.4 Projekt ausführen

Für die Ausführung des Projekts werden .NET 8 SDK, Git und optional Visual Studio Code mit der C#-Extension benötigt. Nach dem Klonen des Repositories müssen lediglich die NuGet-Abhängigkeiten mit dotnet restore geladen werden. Gestartet wird die Anwendung über das WinForms-Projekt mit folgendem Befehl: dotnet run --project src/ContactManager.UI/ContactManager.UI.csproj .

Die Anwendung erzeugt beim ersten Start automatisch eine contacts.json im Projektverzeichnis oder lädt alle bestehenden Kontakte. Tests können jederzeit mit dotnet test ausgeführt werden, wobei GitHub Actions diese ebenfalls bei jedem Push automatisch ausführt.

4 DevOps-Herangehensweise

Unsere DevOps-Herangehensweise orientierte sich an den Prinzipien der Testpyramide: Eine breite Basis aus Unit-Tests stellt sicher, dass Validierungslogik und kleine Funktionseinheiten zuverlässig arbeiten, während darüber hinaus Integrationstests das Zusammenspiel von Repository, Datenmodell und Suchfunktionen abdecken. Ein Feature gilt als „Done“, wenn alle definierten Unit- und Integrationstests fehlerfrei durchlaufen, die Anwendung baubar ist und die GitHub-Actions-Pipeline erfolgreich abgeschlossen wurde. Die Projektstruktur folgt strikten DevOps-Prinzipien mit einer klaren Trennung zwischen Produktivcode im Ordner src/, Testcode im Ordner tests/ und Automatisierungslogik im Verzeichnis .github/workflows/. Dadurch bleibt der Code wartbar, nachvollziehbar und sowohl manuell als auch automatisiert testbar. Für die Zusammenarbeit wurde ein einfacher Branching-Workflow verwendet: main enthält stets stabilen Code, während neue Funktionen in Feature-Branches entwickelt und über Pull Requests integriert werden. Die GitHub-Actions-Pipeline automatisiert anschließend alle zentralen Schritte – sie baut die Lösung, führt sämtliche Tests aus und erzeugt Testartefakte wie MTP-Protokollen. Damit wird bei jeder Änderung automatisch überprüft, ob die Software korrekt funktioniert, was eine hohe Codequalität und einen zuverlässigen Entwicklungsprozess sicherstellt.

5 Testspezifikation

Ziel der Testspezifikation ist es, systematisch festzuhalten, wie die funktionalen und nicht-funktionalen Anforderungen der Kontaktdatenverwaltung überprüft werden. Die Tests sollen sicherstellen, dass sowohl einzelne Komponenten (Validator, Repository) als auch deren Zusammenspiel zuverlässig funktionieren und typische sowie kritische Nutzungsszenarien korrekt abgedeckt werden.

5.1 Abgrenzung Unit- vs. Integrationstests

Unit-Tests prüfen ausschließlich die kleinste funktionale Einheit – in diesem Projekt hauptsächlich die Validierung von Kontaktdaten. Sie testen isoliert ohne externe Abhängigkeiten und stellen sicher, dass Fehler früh erkannt werden.

Integrationstests hingegen überprüfen das Zusammenspiel mehrerer Komponenten, beispielsweise das Hinzufügen, Finden oder Ändern von Kontakten über das Repository. Durch das InMemoryContactRepository werden diese Tests realistisch, aber dennoch deterministisch und unabhängig vom Dateisystem ausgeführt.

5.2 Vorgehensweise bei der Testfallauswahl

Die Auswahl der Testfälle orientierte sich an der fachlichen Spezifikation und der vom Repository-Pattern getrennten Logik. Zunächst wurden alle Validierungsregeln ausformuliert und in einzelne Unit-Tests überführt. Für die Integrationstests wurden häufig vorkommende Szenarien (z. B. Hinzufügen, Aktualisieren, Filtern) sowie typische Benutzerabläufe berücksichtigt. Zusätzlich wurden für jede wichtige Repository-Funktion mindestens ein Erfolgsfall und ein potenzieller Problemfall getestet.

5.3 Analyse von Grenzfällen

Bei der Validierung wurden Grenzfälle berücksichtigt, die häufig zu Fehlern führen: leere Felder, fehlende Sonderzeichen in E-Mail-Adressen, Leerzeichen im E-Mail-Feld oder fehlende Telefonnummern. Bei der Repository-Schicht wurden Fälle wie doppelt angelegte Kontakte (gleicher Inhalt, aber unterschiedliche IDs), nicht vorhandene Datensätze oder Filtern mit Teilstrings geprüft. Diese Edge Cases erhöhen die Robustheit der Anwendung und stellen sicher, dass auch untypische, aber realistische Eingaben korrekt behandelt werden.

5.4 Begründung der Testabdeckung

Die gewählte Testabdeckung orientiert sich klar an der Testpyramide: Die meisten Tests sind Unit-Tests, da sie schnell, stabil und isoliert laufen. Durch die Integrationstests wird das Zusammenspiel des InMemoryContactRepository mit dem Datenmodell geprüft, wodurch zentrale Geschäftslogik realitätsnah validiert wird. Die Kombination beider Testarten deckt alle Kernfunktionalitäten ab und erfüllt vollständig die LV-Anforderungen an Unit- und Integrationstests.

5.5 Erklärung der einzelnen Tests

Unit-Tests:

- ValidateEmail_WithValidEmail_ReturnsEmpty
  → Prüft, dass eine gültige E-Mail kein Validierungsfehler ist.
- ValidateEmail_WithInvalidEmail_ReturnsError
  → Prüft, dass eine ungültige E-Mail (ohne „@“ und „.“) korrekt abgelehnt wird.
- ValidateEmail_WithEmailWithSpace_ReturnsError
  → Prüft, dass E-Mails mit Leerzeichen ungültig sind.
- ValidatePhone_WithValidPhone_ReturnsEmpty
  → Prüft, dass eine nicht-leere Telefonnummer akzeptiert wird.
- ValidatePhone_WithEmptyPhone_ReturnsError
  → Prüft, dass eine leere Telefonnummer abgelehnt wird.

- Validate_WithValidData_ReturnsEmpty

  → Positiver Testfall: vollständige und gültige Kontaktdaten führen zu keinem Fehler.

- Validate_WithMissingName_ReturnsError

  → Prüft, dass ein leerer Name korrekt als ungültig erkannt wird.

Integration-Tests:

- AddContact_WithValidData_AddsContactToRepository

  → Prüft, dass ein neuer Kontakt korrekt im Repository gespeichert wird.

- AddContact_WithDuplicate_AddsMultipleContacts

  → Prüft, dass zwei identische Kontakte separat gespeichert werden und unterschiedliche IDs haben.

- AddContact_GetContacts_PersistsInMemory

  → Prüft, dass mehrere hinzugefügte Kontakte vollständig abrufbar sind.

- UpdateContact_ModifiesExistingContact

  → Prüft, dass Änderungen an einem bestehenden Kontakt korrekt übernommen werden.

- FindByFilter_SearchByName

  → Prüft die Filterfunktion für Namenssuche inkl. Teilstrings.

- FindByFilter_SearchByCompany

  → Prüft die Filterfunktion für Firmennamen.

- SearchAndUpdate_Integration

  → Testet einen realistischen Ablauf: Kontakt finden → aktualisieren → erneut abrufen.

6 Continuous Integration mit GitHub Actions

Für die kontinuierliche Integration wurde eine automatisierte GitHub-Actions-Pipeline eingerichtet, die bei jedem Push und Pull-Request auf den main-Branch ausgeführt wird. Ziel dieser Pipeline ist es, sicherzustellen, dass die Anwendung stets baubar bleibt und alle Unit- und Integrationstests erfolgreich durchlaufen. GitHub Actions übernimmt dabei die Rolle des zentralen Qualitätsgatekeepers: Änderungen werden erst gemergt, wenn Build und Tests erfolgreich sind.

Die Workflow-Datei test.yml definiert alle Schritte der Pipeline. Zunächst wird das Repository ausgecheckt und eine .NET-8-Umgebung eingerichtet. Anschließend installiert die Pipeline über dotnet restore alle benötigten Abhängigkeiten und baut das Projekt im Release-Modus. Danach werden alle Tests mithilfe von dotnet test ausgeführt. Der Testlauf ist dabei so konfiguriert, dass die Ergebnisse im TRX-Format generiert werden – dem Standardformat der Microsoft Testing Platform (MTP). Damit ist die

Pipeline bereits MTP-kompatibel, da MTP TRX-Dateien als Eingangsdaten nutzt. Eine vollständige Integration in das MTP-Webportal wäre optional möglich.

Nach erfolgreicher Testausführung speichert die Pipeline die generierten TRX-Dateien als Artefakte. Diese Artefakte dienen als dokumentierter Nachweis der Testergebnisse und können direkt aus dem GitHub Actions Lauf heruntergeladen werden. Sie zeigen detailliert an, welche Tests erfolgreich waren und wo Fehler aufgetreten sind. Damit erfüllt die Pipeline vollständig die Anforderung, Testprotokolle bereitzustellen, und ermöglicht eine vollständige Nachvollziehbarkeit der Testergebnisse. In der folgenden Abbildung sieht man die ausgelöste GitHub Action bei einem Pull.

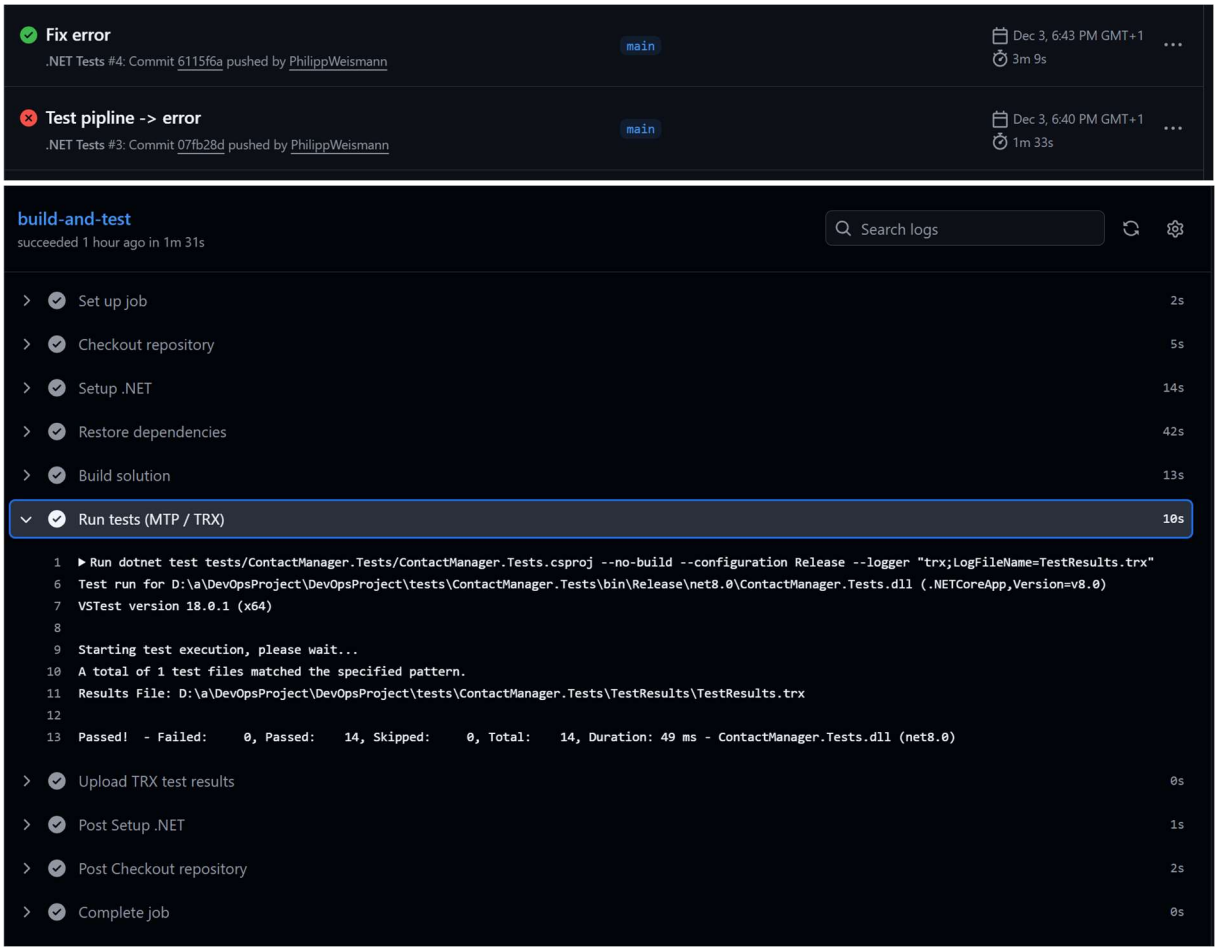

Figure 3 GitHub Actions