htlperg
innovativ | kreativ | praxisnah

EngelConnect

Automatisierte Überprüfung und UI-gestützte Auswertung von OPC-UA Schnittstellen

DIPLOMARBEIT

Höhere Abteilung für Informatik

01/07/2025 – 26/03/2026

Projektmitglieder: Thomas Kastner
Ádám Halász

Betreuerin: Maria Inreiter, MMSc

Eidestaatliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Bei der Erstellung der Arbeit haben wir generative KI-Tools wie ChatGPT zu folgendem Zweck verwendet: Rechtschreib- und Grammatikprüfung, Formulierungshilfe, Generierung von Code-Kommentaren.

Unterzeichner Thomas Kastner
Datum und Uhrzeit 26.03.2026, 11:46 (GMT+01:00)

ID Austria

Dieses Dokument ist digital signiert

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr. 910/2014 vom 23. Juli 2014 ("EIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Informationen zur Prüfung der elektronischen Signatur finden Sie unter: https://www.signaturpruefung.gv.at

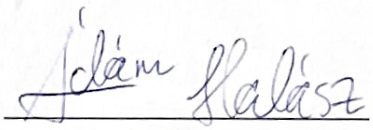
Perg, 26.03.2026

Ort, Datum

Unterschrift, Kastner Thomas

Perg, 26.03.2026

Ort, Datum



Unterschrift, Ádám Halász

Gendererklärung

Um die Lesbarkeit dieser Diplomarbeit zu erleichtern, werden geschlechtsspezifische Bezeichnungen, die sowohl Frauen als auch Männer einschließen, in der üblichen männlichen Form verwendet. Dies geschieht ausschließlich aus Gründen der sprachlichen Vereinfachung und soll in keiner Weise eine Benachteiligung oder Missachtung des Gleichheitsgrundsatzes darstellen.

Danksagung

Wir möchten uns herzlich bei allen Personen bedanken, die uns während der Erstellung dieser Diplomarbeit unterstützt haben. Besonders bedanken möchten wir uns bei unserer Betreuerin Prof. Ing. Maria Inreiter, MSc, die uns während der Ausarbeitung der Diplomarbeit wertvolle Tipps gegeben hat.

Ein besonderer Dank gilt unserem Auftraggeber, der ENGEL Austria GmbH, einem der weltweit führenden Unternehmen in der Forschung und Produktion von Spritzgießmaschinen für die Kunststoffverarbeitung sowie der dazugehörigen Automatisierungstechnik.

Unser besonderer Dank richtet sich an Manuel Buchinger, der die Betreuung innerhalb des Unternehmens übernommen und uns während des gesamten Projektverlaufs unterstützt hat.

Während der technischen Umsetzung wurde unser Projektteam zusätzlich von Jakob Haider begleitet, der sich insbesondere mit dem Frontend sowie dem Design beschäftigte und damit wesentlich zum Gelingen der Arbeit beigetragen hat.

Ebenso möchten wir uns bei Christian Schuhmayr bedanken, der uns bei der Implementierung des Backends unterstützte und dem Projektteam insbesondere bei Fragen zur Softwarearchitektur mit wertvollen fachlichen Ratschlägen zur Seite stand.

Kurzfassung

Die manuelle Überprüfung von OPC-UA-Schnittstellen bei Spritzgussmaschinen ist zeit- und kostenintensiv, da bei jeder Beanstandung eine Verbindung zur Maschine hergestellt und zahlreiche Attribute einzeln geprüft werden müssen. Diese Diplomarbeit beschreibt eine Softwarelösung für die ENGEL Austria GmbH, die diesen Prozess automatisiert und sowohl vor der Auslieferung als auch beim Kunden eine effiziente und standardisierte Analyse ermöglicht.

Die entwickelte Anwendung führt automatisierte Tests für verschiedene OPC-UA- und Euromap-Schnittstellen durch. Beim Start erkennt das System, welchem Standard ein Gerät am ehesten entspricht, und schlägt passende Testpakete vor. Zusätzlich werden sogenannte „Common Tests“ durchgeführt, die grundlegende Eigenschaften unabhängig vom konkreten Standard prüfen.

Die Teststruktur ist baumartig aufgebaut und ermöglicht eine flexible Auswahl. Benutzer können vorgeschlagene Tests direkt ausführen („Run Selected“), individuell konfigurieren oder im „Run All“-Modus alle Tests starten, um unbekannte Schnittstellen zu identifizieren. Die Tests werden sequenziell ausgeführt, wobei Ergebnisse unmittelbar angezeigt werden. Neben „bestanden“ und „fehlgeschlagen“ wird eine zusätzliche Bewertungsstufe verwendet, da ENGEL teilweise strengere Anforderungen definiert als die Standards vorgeben.

Die Benutzeroberfläche wurde auf Basis von Figma-Mockups entworfen und in Angular umgesetzt. Zentrale Funktionen sind eine Baumstruktur zur Testauswahl, eine Breadcrumb-Navigation zur Orientierung, eine Detailansicht für Testergebnisse sowie eine Statusanzeige der Geräteverbindung. Testergebnisse können als JSON exportiert werden.

Im Zuge der Implementierung wurden zusätzliche Funktionen ergänzt, darunter das Abbrechen laufender Tests, die Anzeige von Untertests sowie eine regelmäßige Überprüfung der Verbindung. Diese Erweiterungen wurden durch die Verwendung bestehender UI-Komponenten ermöglicht, wodurch Entwicklungszeit eingespart und die Anwendung weiter verbessert werden konnte.

Die entwickelte Lösung reduziert den manuellen Prüfaufwand erheblich, verkürzt Reaktionszeiten bei Kundenanfragen und liefert strukturierte sowie nachvollziehbare Testergebnisse. Gleichzeitig wurde eine erweiterbare Grundlage geschaffen, sodass zukünftige Anpassungen und zusätzliche Testpakete einfach integriert werden können.

Abstract

The manual verification of OPC UA interfaces in injection molding machines is time-consuming and costly, as each issue requires establishing a connection to the machine and checking numerous attributes individually. This diploma thesis presents a software solution for ENGEL Austria GmbH that automates this process and enables efficient and standardized analysis both before delivery and directly at the customer site.

The developed application performs automated tests for various OPC UA and Euromap interfaces. At startup, the system identifies the most likely standard implemented by the connected device and suggests appropriate test packages. In addition, so-called "common tests" are executed to verify fundamental properties independent of a specific standard.

The test structure is organized in a hierarchical tree, allowing flexible test selection. Users can execute suggested tests directly ("Run Selected"), configure tests manually, or run all available tests ("Run All") to identify unknown interfaces. Tests are executed sequentially, while results are displayed immediately. In addition to "passed" and "failed", an additional evaluation level is introduced, as ENGEL defines stricter requirements than those specified in the standards.

The user interface was designed using Figma mockups and implemented in Angular. Key features include a tree-based test selection, breadcrumb navigation, detailed result views, and a connection status indicator. Test results can be exported as JSON files.

During implementation, additional features such as test cancellation, subtest visualization, and periodic connection monitoring were introduced. These enhancements were enabled by the use of existing UI components, resulting in reduced development time and improved usability.

The developed solution significantly reduces manual effort, shortens response times, and provides structured and reproducible test results. Furthermore, it establishes a scalable foundation for future extensions and additional test packages.

Inhaltsverzeichnis

1 Einführung

Nach der Einführung in die Thematik und die grundlegende Motivation des Projekts werden im Folgenden die Ausgangslage, die Zielsetzung sowie der inhaltliche und organisatorische Rahmen der Arbeit näher beschrieben. Dabei wird zunächst die bestehende Problemstellung im Unternehmenskontext erläutert. Anschließend wird die Zielsetzung der entwickelten Lösung definiert, bevor der Projektinhalt sowie das Projektumfeld als Grundlage für die weiteren Kapitel dargestellt werden.

1.1 Ausgangslage

Das Unternehmen Engel Austria GmbH stellt Spritzgussmaschinen mit verschiedenen Schnittstellen zur Verfügung. Diese Schnittstellen basieren auf dem Standard Open Platform Communications Unified Architecture (OPC-UA) und werden sowohl weltweit von Kunden als auch intern in der Engel Maschine genutzt. In der Praxis zeigte sich, dass die bereitgestellten Schnittstellen nicht immer den jeweiligen Standards entsprechen, da jedes externe Gerät sich dabei selbständig und fehlerfrei an den Standard von OPC-UA halten muss, um eine einwandfreie Funktion zu gewährleisten.

Die Kunden der Engel Austria GmbH berichten häufig, dass die von Engel-Maschinen bereitgestellten Schnittstellen nicht dem versprochenen OPC-UA Standard entsprechen. Dies führte dazu, dass bei jeder Beschwerde ein Mitarbeiter sich zu der Schnittstelle der Maschine verbindet und jedes einzelne Attribut bzw. jede Funktion manuell überprüft und schließlich die Diagnose schriftlich dokumentiert. Diese manuelle Bearbeitung der Beschwerden ist zeit- und kostenintensiv. Zudem verlängert sich dadurch die Antwortzeit gegenüber dem Kunden erheblich.

Aus diesem Grund entstand die Idee, den gesamten Testprozess durch eine Softwareanwendung zu automatisieren. Ziel ist es, bereits vor der Auslieferung der Maschine alle ENGEL-internen OPC-UA-Schnittstellen mithilfe dieser Testsoftware zu überprüfen. Zusätzlich kann es auch nach der Inbetriebnahme beim Kunden zu Problemen zwischen dem Kundennetzwerk und der Maschine kommen. In solchen Fällen soll die Anwendung ebenfalls Abhilfe schaffen, indem der

Kundenservice mithilfe der Software umgehend eine Rückmeldung erhält, welche Funktionen oder Attribute nicht OPC-UA-konform sind.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die Überprüfung der Schnittstellen mithilfe einer Software zu automatisieren. Die Anwendung soll dabei nicht nur Tests für eine einzelne OPC-UA-Schnittstelle durchführen, sondern sämtliche implementierten OPC-UA-Schnittstellen unterstützen. Der Benutzer kann dabei auswählen, welche Tests tatsächlich ausgeführt werden sollen. Weiters sollen sogenannte "Common Tests"durchgeführt werden, die bei allen Standards gleich sind und alle Grundfunktionalitäten, wie z.B. Connection-Tests, überprüfen sollen.

Zusätzlich soll die Anwendung beim Start erkennen, welchem OPC-UA bzw. Euromap-Standard das Gerät am ehesten entspricht. Dies erleichtert die Arbeit von Mitarbeitern, die mit den jeweiligen Standards nicht vertraut sind.

Die Tests werden in Echtzeit ausgeführt. Das bedeutet, dass die Tests sequenziell nacheinander laufen und nach Abschluss eines Tests dessen Ergebnis sofort angezeigt wird, während im Hintergrund trotzdem die übrigen Tests weiterlaufen.

Das Testergebnis soll die Fehlerursache sowie den erwarteten und den tatsächlich gemessenen Wert eines Attributs enthalten. Zusätzlich soll es neben den Ergebnissen „bestanden“ und „nicht bestanden“ eine dritte Bewertungsstufe geben, da ENGEL teilweise strengere Anforderungen definiert als in den entsprechenden Standards vorgesehen sind.

Abschließend ist zu erwähnen, dass die Aufgabe der Projektgruppe darin bestand, eine leicht erweiterbare Grundlage für ENGEL zu schaffen. In weiterer Folge soll die Applikation von den jeweiligen Betreuern der Schnittstellen selbst erweitert werden, da diese über das größte Fachwissen hinsichtlich des jeweiligen Standards verfügen. Das konkrete Ziel der Arbeit lag daher in der Umsetzung der automatischen Erkennung der Standards, das Erstellen der Benutzeroberfläche sowie der sequenziellen Ausführung der Tests in Echtzeit.

1.3 Projektinhalt

Dieses Kapitel beschreibt den inhaltlichen und technischen Aufbau des Projekts. Zunächst wird ein kompakter Überblick über das Gesamtsystem gegeben. Darauf aufbauend werden die wesentlichen architektonischen Komponenten und deren Zusammenspiel näher erläutert.

1.3.1 Kommunikation zwischen Computer und Maschine

Frontend zum Backend

Grundsätzlich war es ein Ziel, möglichst einfache und klar definierte Schnittstellen zwischen den drei voneinander unabhängigen Teilprojekten zu schaffen. Die Kommunikation zwischen Frontend und Backend erfolgt über im Voraus definierte ContextBridge-Schnittstellen, die als Kommunikationskanäle („Tunnel“) dienen. Einige dieser Schnittstellen liefern Daten auf Anfrage, während andere dynamisch Werte an das Frontend übermitteln.

Testumgebung zum Node Client

Innerhalb des Backends greift die Testumgebung auf einzelne benötigte Funktionen des Node-OPC-UA-Clients zurück. Aus diesem Grund ist es wichtig, dass der Client korrekt innerhalb der Testumgebung initialisiert wird. Der Client selbst verwendet das node-opcua-Framework, um die Kommunikation mit dem simulierten Server umzusetzen.

OPC-UA Client zur Server

Dieses Framework stellt eine Reihe von Funktionen zur Verfügung, die für den Zugriff auf den Server verwendet werden können. Auf Basis dieser Funktionen wurden möglichst allgemeine Methoden implementiert, sodass die Testumgebung selbst möglichst einfach und übersichtlich gehalten werden kann.

1.3.2 Benutzeroberfläche

Im Rahmen des Projekts ist die Entwicklung einer Benutzeroberfläche vorgesehen, die den Anwender strukturiert durch den gesamten Testprozess führt. Ziel ist es, eine übersichtliche und klar verständliche Interaktion zu ermöglichen, die sich auf die wesentlichen Schritte der Testdurchführung konzentriert.

Die Benutzeroberfläche soll dabei aus einer reduzierten Anzahl zentraler Ansichten bestehen, die den vollständigen Ablauf abbilden. Insgesamt sind drei Hauptseiten vorgesehen:

Auswahl des Geräts

Zu Beginn soll der Benutzer eine Ansicht erhalten, in der das zu prüfende Gerät ausgewählt werden kann. Diese Auswahl bildet die Grundlage für alle weiteren Schritte, da die durchzuführenden Tests vom jeweiligen Gerät abhängig sind. Die Oberfläche soll hierbei eine einfache und nachvollziehbare Auswahl ermöglichen.

Start und Durchführung des Tests

Nach der Geräteauswahl ist eine Seite vorgesehen, auf der der Testprozess gestartet werden kann. Diese Ansicht dient der Initiierung der Tests und soll dem Benutzer eine klare Möglichkeit bieten, den Vorgang zu beginnen. Während der Durchführung ist vorgesehen, den aktuellen Status des Tests darzustellen, sodass der Fortschritt jederzeit ersichtlich ist.

Darstellung der Testergebnisse

Nach Abschluss des Testvorgangs soll eine Ergebnisansicht angezeigt werden. In dieser Ansicht ist vorgesehen, dem Benutzer eine Rückmeldung über den Ausgang der Tests zu geben. Dabei soll ersichtlich sein, ob die Tests erfolgreich oder fehlgeschlagen sind. Zusätzlich sollen weiterführende Informationen zu einzelnen Testergebnissen bereitgestellt werden, um Transparenz über mögliche Fehlerquellen oder besondere Hinweise zu schaffen.

Die geplante Struktur der Benutzeroberfläche orientiert sich somit an einem klaren, linearen Ablauf: Auswahl des Geräts, Start der Tests sowie anschließende Auswertung der Ergebnisse. Ziel ist eine intuitive Bedienbarkeit und eine nachvollziehbare Darstellung des gesamten Testprozesses.

1.3.3 Auswahl und Definition der Testfälle

Im Rahmen des Projekts sollen ausreichend viele Standardtestfälle implementiert werden, um die zentralen Kernfunktionalitäten des Systems überprüfen zu können. Ziel dieser Testfälle ist es, sicherzustellen, dass die grundlegenden Funktionen der Anwendung erwartungsgemäß arbeiten und typische Anwendungsfälle korrekt verarbeitet werden.

Die Auswahl der Testfälle orientiert sich daher primär an den wichtigsten Funktionalitäten des Systems. Es wird nicht angestrebt, sämtliche theoretisch möglichen Szenarien abzudecken, sondern vielmehr eine ausreichende Anzahl repräsentativer Standardtests bereitzustellen, mit denen die Kernfunktionen zuverlässig überprüft werden können.

1.3.4 Weiterführende Entwicklung

Das entwickelte System stellt in erster Linie eine funktionsfähige Grundlage dar, welche die Kernfunktionen sowie die notwendige Infrastruktur für die Durchführung von Tests bereitstellt. Dazu zählen insbesondere die Kommunikationsschnittstellen, die Testumgebung sowie die Mechanismen zur dynamischen Ausführung und Auswertung von Testfällen.

Die zukünftige Aufgabe besteht darin, weitere Tests zu implementieren, die zusätzliche Aspekte des OPC UA-Standards überprüfen. Dadurch kann das System schrittweise erweitert werden, um eine größere Anzahl an möglichen Testszenarien abzudecken.

1.4 Projektumfeld

Im folgenden Abschnitt wird das Projektumfeld näher beschrieben. Dabei werden insbesondere das Projektteam, die Betreuung sowie der Auftraggeber vorgestellt.

1.4.1 Projektteam

Das Projektteam setzt sich aus Adam Halasz und Thomas Kastner zusammen, beide Schüler der Höheren Technischen Lehranstalt Perg. Thomas Kastner hat in diversen Projekten in der HTL schon viele Erfahrungen mit Angular sowie Figma gemacht und war deshalb für das Gestalten des Designs und Entwicklung des Frontend zuständig. Adam Halasz erwarb ebenfalls durch schulische Projekte fundierte Kenntnisse im Bereich der Backendmodellierung und war in diesem Projekt für die Implementierung und Ausführung der automatisierten Tests zuständig.

1.4.2 Betreuung

Die Diplomarbeit wurde im technischen und theoretischen Teil durch Maria Inreiter, MMSc., mit ihrem Wissen über Webentwicklung tatkräftig unterstützt.

1.4.3 Auftraggeber

Auftraggeber dieser Arbeit ist die ENGEL Austria GmbH, eines der weltweit führenden Unternehmen in der Forschung und Produktion von Spritzgießmaschinen für die Kunststoffverarbeitung sowie der dazugehörigen Automatisierungstechnik. Die Betreuung innerhalb des Unternehmens erfolgte durch Manuel Buchinger. Während der technischen Umsetzung wurde das Projektteam zusätzlich von Jakob Haider unterstützt, der sich hauptsächlich mit dem Frontend sowie dem Design beschäftigte. Christian Schuhmayr leistete Unterstützung bei der Implementierung des Backends und stand dem Projektteam insbesondere bei Fragen rund um die Softwarearchitektur beratend zur Seite.

Abbildung 1: Logo ENGEL Austria GmbH

2 Theoretische und fachpraktische Grundlagen und Methoden

Die in der vorliegenden Diplomarbeit verwendeten Technologien, Werkzeuge und Standards werden in diesem Kapitel im Detail beschrieben. Die Auswahl orientierte sich sowohl an unternehmensinternen Vorgaben der ENGEL Austria GmbH als auch an fachlichen, technischen und organisatorischen Anforderungen des Projekts.

2.1 Technologien

Im Folgenden werden die im Projekt eingesetzten Technologien und Werkzeuge im Detail vorgestellt. Den Anfang bildet die Programmiersprache TypeScript, die als zentrale Grundlage für die Implementierung dient. Weiter geht es mit dem Frontend Framework Angular, Npx, Node.js sowie Git und Azure DevOps.

2.1.1 TypeScript

Für die Implementierung der Anwendung wurde die Programmiersprache TypeScript verwendet. TypeScript stellt eine typsichere Erweiterung von JavaScript dar und ermöglicht durch statische Typisierung eine frühzeitige Fehlererkennung bereits zur Entwicklungszeit [1].

Ein wesentlicher Beweggrund für die Wahl dieser Technologie lag in der unternehmensinternen Präferenz von ENGEL sowie in der strategischen Entscheidung, sowohl Frontend als auch Backend auf Basis derselben Programmiersprache umzusetzen. Dadurch konnte eine einheitliche Codebasis geschaffen, der Wartungsaufwand reduziert und die Zusammenarbeit im Projektteam effizient gestaltet werden.

Darüber hinaus verbessert TypeScript durch klar definierte Schnittstellen, Klassenstrukturen und Typdefinitionen die Lesbarkeit sowie die langfristige Wartbarkeit des Quellcodes [2].

Abbildung 2: TypeScript Logo
[3]

2.1.2 Angular

Als Frontend-Framework wurde Angular eingesetzt. Angular ist ein komponentenbasiertes Framework zur Entwicklung moderner Single-Page-Applikationen und gilt als Industriestandard in zahlreichen Unternehmensanwendungen [4]. Angular bietet eine klar strukturierte Softwarearchitektur, integriertes Routing, Dependency Injection sowie umfangreiche Testmöglichkeiten, was die Entwicklung einer skalierbaren und wartbaren Benutzeroberfläche unterstützt [5].

Innerhalb der ENGEL Austria GmbH wird Angular konzernweit als Standardtechnologie für Webanwendungen eingesetzt, weshalb dessen Verwendung auch im Rahmen dieser Diplomarbeit naheliegend war. Zusätzlich verfügte das Projektteam bereits über praktische Erfahrung mit Angular aus der schulischen Ausbildung, wodurch ein effizienter Projekteinstieg gewährleistet werden konnte.

Abbildung 3: Angular Logo
[6]

2.1.3 Npx

Npx ist das primäre Kommandozeilenwerkzeug, das im Rahmen des Projekts verwendet wird. Es wird in Kombination mit npm bereitgestellt und ermöglicht die Ausführung von Node-Packages, ohne dass diese global installiert werden müssen.

Im Rahmen des Projekts wurden zudem mehrere Npx-Skripts zum Starten und Builden des Monorepos erstellt, da das gesamte Monorepo-Setup ebenfalls mithilfe von Npx erstellt wurde [7].

2.1.4 Node.js

Node.js ist eine JavaScript Laufzeitumgebung. Im Rahmen des Projekts wurde Node.js jedoch in Kombination mit dem TypeScript-Compiler verwendet, sodass die Implementierung in TypecSript erfolgte.

Die Verwendung von TypeScript bietet den Vorteil statischer Datentypen, wodurch Fehler bereits während der Entwicklung beziehungsweise zur Compile-Zeit erkannt werden können. Dies verbessert die Wartbarkeit des Codes und erhöht die Typensicherheit innerhalb der Anwendung.

Darüber hinaus wurde Node.js für den Build-Prozess sowie für die Kompilierung der Backend-Komponenten verwendet. Die einheitliche Verwendung von TypeScript in Kombination mit Node.js ermöglichte eine konsistente Technologiearchitektur über alle Anwendungsschichten hinweg [8].

2.1.5 Git und Azure DevOps

Zur Versionsverwaltung wurde Git eingesetzt. Git ermöglicht eine verteilte Versionskontrolle, wodurch Änderungen am Projekt nachvollziehbar dokumentiert und effizient zwischen Teammitgliedern ausgetauscht werden können

Als zentrale Plattform für Repository-Verwaltung und Projektkoordination diente Azure DevOps. Da Azure DevOps unternehmensintern bei ENGEL verwendet wird, war dessen Nutzung verpflichtend. Insbesondere die Möglichkeit, Pull Requests strukturiert zu prüfen und freizugeben, unterstützte die Qualitätssicherung sowie die Zusammenarbeit mit den projektbetreuenden Personen [9, 10, 11].

(a) Git Logo
[12]

(b) Azure DevOps Logo
[13]

Abbildung 4: Verwendete Tools

2.1.6 Electron und ContextBridge

Ziel des Projekt war eine Desktopanwendung. Genau dafür ist Electron gut geeignet. Außerdem wird in Electron mit der Webtechnologie Angular entwickelt, was auch für das Projektteam schon mehr als gut bekannt ist. Electron nutzt zwei zentrale Technologien: [14]

Chromium

Ist für die grafische Anzeige zuständig. Dieser wird auch als der **Renderer-Prozess** bezeichnet [15].

Node.js

Diese Komponente dient dem Zugriff auf das Betriebssystem selbst und wird als **Main-Prozess** bezeichnet [8].

Die Kommunikation zwischen den beiden Prozessen wird von der ContextBridge gesichert, da sie voneinander getrennt sind. Die ContextBridge kann man sich wie einen Tunnel vorstellen, in dem zuvor definiert wurde, welche Funktionen und Daten übertragen werden [16].

2.1.7 Jasmine

Jasmine war ursprünglich die vom Projektteam gewählte Testumgebung. Dabei handelt es sich um ein etabliertes Testing-Framework für JavaScript und TypeScript, das häufig für Unit-Tests eingesetzt wird.

Während der Implementierungsphase zeigte sich jedoch, dass die vorhandenen Funktionalitäten von Jasmine für die Anforderungen des Projekts nur eingeschränkt geeignet sind. Insbesondere stellte die fehlende Unterstützung für dynamische Reports eine wesentliche Einschränkung dar. Auch die standardmäßig verfügbaren Reporting-Funktionen erwiesen sich als nicht optimal für den vorgesehenen Anwendungsfall [17].

2.1.8 Vitest

Vitest wurde im weiteren Verlauf des Projekts als Ersatz für Jasmine ausgewählt. Dabei handelt es sich um ein vergleichsweise neues Testing-Framework, das in den letzten Jahren zunehmend an Popularität gewonnen hat.

Ein entscheidender Vorteil von Vitest ist die vergleichsweise einfache Konfiguration sowie der geringe Integrationsaufwand in bestehende Projekte. Darüber hinaus bietet Vitest die Möglichkeit, Tests dynamisch auszuführen, was für die Anforderungen des Projekts besonders relevant ist. Dennoch weist auch dieses Framework gewisse Einschränkungen auf, die bei der Implementierung berücksichtigt werden müssen [18].

2.2 Entwicklungssysteme

Im folgenden Abschnitt werden die im Projekt eingesetzten Entwicklungssysteme näher beschrieben. Dabei wird zunächst die verwendete Entwicklungsumgebung vorgestellt, bevor anschließend auf weitere unterstützende Werkzeuge eingegangen wird.

2.2.1 VS Code

Als Entwicklungsumgebung wurde Visual Studio Code verwendet. Visual Studio Code ist ein plattformübergreifender, erweiterbarer Code-Editor mit umfangreicher Unterstützung für TypeScript, Angular und Node.js [19].

Die Entscheidung fiel zugunsten von VS Code, da dieses Werkzeug unternehmensintern bei ENGEL etabliert ist. Alternativen wie WebStorm wurden zwar in Betracht gezogen, jedoch aufgrund der betrieblichen Standardisierung sowie der vorhandenen Extensions und Konfigurationsmöglichkeiten nicht weiterverfolgt.

Abbildung 5: VS Code Logo
[20]

2.2.2 Figma

Für die Konzeption und Gestaltung der Benutzeroberfläche wurde Figma eingesetzt. Figma ist ein webbasiertes Design- und Prototyping-Werkzeug, das insbesondere durch seine kollaborativen Funktionen überzeugt [21].

Ein wesentlicher Vorteil war die Möglichkeit, interaktive und klickbare Prototypen („Clickable Dummys“) zu erstellen. Dadurch konnten Benutzerflüsse frühzeitig visualisiert und mit den Projektbetreuern abgestimmt werden. Zudem verfügte das Projektteam bereits über praktische Erfahrung mit Figma aus der schulischen Ausbildung, wodurch ein effizienter Einsatz gewährleistet war.

Abbildung 6: Figma Logo
[22]

2.3 Bibliotheken und Plug-Ins

2.3.1 PrimeNG

Für die Umsetzung der Benutzeroberfläche wurde die Komponentenbibliothek PrimeNG verwendet. PrimeNG stellt eine umfangreiche Sammlung vorgefertigter UI-Komponenten für Angular bereit [23].

Der Einsatz dieser Bibliothek ermöglichte eine effiziente Implementierung komplexer Oberflächenelemente, beispielsweise eines selektiven Tree-Components, dessen Eigenentwicklung mit erheblichem Zeitaufwand verbunden gewesen wäre. Darüber hinaus wird PrimeNG ebenfalls im Unternehmen eingesetzt, wodurch eine technologische Konsistenz gewährleistet werden konnte.

Abbildung 7: PrimeNG Logo
[24]

2.3.2 Node OPC-UA

Da im Rahmen des Projekts ein Node.js-Backend entwickelt wird und dieses mit einem OPC-UA-Server kommunizieren muss, wird ein Open-Source-OPC-UA-Framework verwendet, das

für Node.js verfügbar ist. Dieses Framework ermöglicht das einfache Durchsuchen komplexer OPC-UA-Serverstrukturen, das Auslesen beliebiger Werte, das Schreiben von Daten auf den Server sowie das Aufrufen von Funktionen, die vom Server bereitgestellt werden. Um diese Technologie effektiv nutzen zu können, stellte die Firma Engel ein Beispielprojekt für den Node OPC-UA-Client zur Verfügung.

2.4 Verwendete Standards/Protokolle

Aufbauend auf den eingesetzten Entwicklungssystemen werden im Folgenden die verwendeten Standards und Protokolle beschrieben. Dabei wird zunächst das im Projekt verwendete Repository-Konzept anhand des Nx-Monorepos erläutert, bevor anschließend auf weitere relevante Industriestandards eingegangen wird.

2.4.1 Nx-Monorepo

Im Gegensatz zu einem modularen Repository werden bei einem Monorepo alle Bestandteile einer Anwendung in einem einzigen Repository verwaltet und gespeichert. Dies vereinfacht das Dependency-Management, da Versionierungskonflikte reduziert werden. Ein weiterer Vorteil besteht darin, dass das gesamte Projekt als gemeinsamer Release betrachtet werden kann. Das bedeutet, dass die einzelnen Bestandteile stets miteinander kompatibel und auf dem aktuellen Stand sind.

Trotz dieser Vorteile bringt ein Monorepo auch gewisse Herausforderungen mit sich. Eine davon ist, dass das Repository rasch sehr groß und komplex werden kann, wodurch das Projektteam häufig Anpassungen in den Konfigurationsdateien vornehmen muss.

Im Rahmen dieses Projekts wurde dafür ein Monorepo unter Verwendung des Build-Systems Nx umgesetzt. Dieses Werkzeug unterstützt die Organisation mehrerer Anwendungen und Bibliotheken innerhalb eines gemeinsamen Repositories. [25]

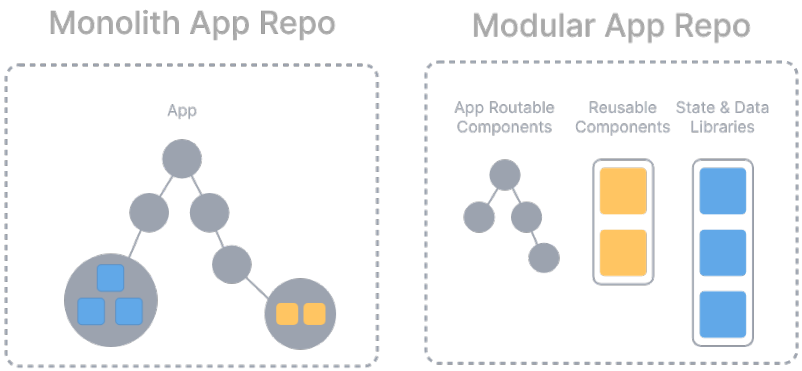


Abbildung 8: Monorepo im Vergleich zu einem modularen Repo
[26]

2.4.2 EUROMAP

Der EUROMAP-Standard stellt eine Sammlung internationaler Schnittstellenstandards für die Kunststoff- und Gummiverarbeitung dar, insbesondere im Bereich von Spritzgießmaschinen. EUROMAP definiert, welche Daten zwischen Maschinen, Peripheriegeräten und Leitsystemen ausgetauscht werden [27]. Da ENGEL diesen Standard in seinen Maschinen implementiert, war dessen Berücksichtigung im Rahmen des Projekts zwingend erforderlich.

Abbildung 9: Euromap Logo
[28]

2.4.3 OPC-UA

OPC Unified Architecture (OPC UA) ist ein plattformunabhängiger Industriestandard für den sicheren und herstellerneutralen Datenaustausch zwischen Maschinen, Sensoren und IT-Systemen [29].

Während EUROMAP definiert, **welche** Informationen übertragen werden, beschreibt OPC UA, **wie** diese Informationen strukturiert und übertragen werden. OPC UA basiert intern auf einer hierarchischen Baumstruktur, in der Datenpunkte als Knoten organisiert sind. Der Einsatz dieses Standards ermöglichte eine sichere, skalierbare und zukunftsorientierte Integration in industrielle Industrie-4.0-Umgebungen.

Abbildung 10: OPC-UA Logo
[30]

2.5 Sonstige verwendete Software

Im folgenden Abschnitt werden ergänzende Werkzeuge vorgestellt, die im Projekt unterstützend eingesetzt wurden. Den Beginn bildet das Diagrammtool draw.io, das insbesondere für die Visualisierung von Strukturen und Abläufen verwendet wurde.

2.5.1 Draw.io

Zur Erstellung des Projektstrukturplans sowie technischer Diagramme wurde draw.io verwendet. draw.io ist ein webbasiertes Werkzeug zur Erstellung von Diagrammen und Visualisierungen [31].

Dieses Werkzeug unterstützte die visuelle Planung der Projektphasen und half dabei, die Softwarearchitektur schrittweise zu strukturieren und Missverständnisse innerhalb des Teams zu vermeiden.

Abbildung 11: Draw.io Logo
[32]

2.5.2 Paint

Zu Beginn des Projekts wurde Microsoft Paint für einfache, spontane Skizzen verwendet. Obwohl es sich um ein sehr einfaches Zeichenprogramm handelt, erwies es sich als zweckmäßig für rasche architektonische und strukturelle Ideenskizzen, insbesondere im Rahmen informeller Abstimmungen während Daily-Meetings.

Abbildung 12: Paint Logo
[33]

2.5.3 Euromap Simulator

Für Testzwecke wurde ein unternehmensseitig bereitgestellter EUROMAP-Simulator verwendet. Dieser ermöglichte die Durchführung realistischer Testszenarien, ohne auf eine physische Maschine zugreifen zu müssen. Dadurch konnte die Implementierung der Kommunikationsschnittstelle praxisnah validiert werden.

2.5.4 Clockify

Zur Zeiterfassung wurde Clockify eingesetzt. Clockify ist ein webbasiertes Tool zur strukturierten Dokumentation von Arbeitszeiten. Im Rahmen der Diplomarbeit diente es der transparenten Erfassung des Projektaufwands sowie der Nachvollziehbarkeit einzelner Entwicklungsphasen [34].

Abbildung 13: Clockify Logo
[35]

3 Konzept

Im folgenden Abschnitt wird der strukturelle Aufbau des Systems sowie die Organisation des Quellcodes näher erläutert. Dabei wird insbesondere auf die gewählte Monorepo-Architektur und deren konkrete Umsetzung im Projekt eingegangen. Ziel ist es, die zugrunde liegende Struktur nachvollziehbar darzustellen und die daraus resultierenden Vorteile für Entwicklung, Wartbarkeit und Zusammenarbeit im Projektteam aufzuzeigen.

Anschließend wird die Konzeption und Umsetzung der Benutzeroberfläche beschrieben. Dabei liegt der Fokus auf den einzelnen Ansichten, die dem Benutzer zur Interaktion mit dem System zur Verfügung stehen, sowie auf deren funktionaler und gestalterischer Ausprägung. Die Beschreibung umfasst sowohl den grundlegenden Ablauf der Benutzerführung als auch die visuelle Gestaltung und Struktur der einzelnen Screens.

3.1 Struktureller Aufbau des Systems

Im folgenden Kapitel werden die Anforderungen an das Gesamtsystem beschrieben und erläutert. Darüber hinaus wird der Ansatz des Projektteams zur Erreichung dieser Ziele dargestellt.

3.1.1 Ziele für das Gesamtsystem

Das grundlegende Ziel für das Backend ist es, die dynamische und Echtzeit-Ausführung von Testfällen zu ermöglichen. Das bedeutet, dass die Tests unabhängig voneinander ablaufen, egal ob der Benutzer alle oder nur bestimmte Tests auswählt. Das Ergebnis eines Tests wartet nicht darauf, dass alle anderen Tests abgeschlossen sind; stattdessen werden die Ergebnisse sofort an das Frontend zurückgeliefert.

Das Frontend zeigt nur die Testfälle an, die bereits von den Entwicklern implementiert wurden. Das bedeutet, dass es eine Schnittstelle gibt, die die vorhandenen Tests zurückliefert. Der Grund dafür ist, dass eine sehr große Anzahl von OPC-UA-Tests existieren und das Produkt wahrscheinlich bereits im Betrieb genutzt wird, bevor alle Tests implementiert sind. Darüber hinaus sollte die Schnittstelle zwischen Frontend und Backend möglichst einfach und zentral gestaltet sein, sodass Frontend-Entwickler die interne Backend-Logik nicht im Detail kennen müssen. Dies

reduziert die Komplexität auf Frontend-Seite, erleichtert die Wartung und verhindert unnötige Abhängigkeiten zwischen den Komponenten.

3.1.2 Systemaufbau und Design Patterns

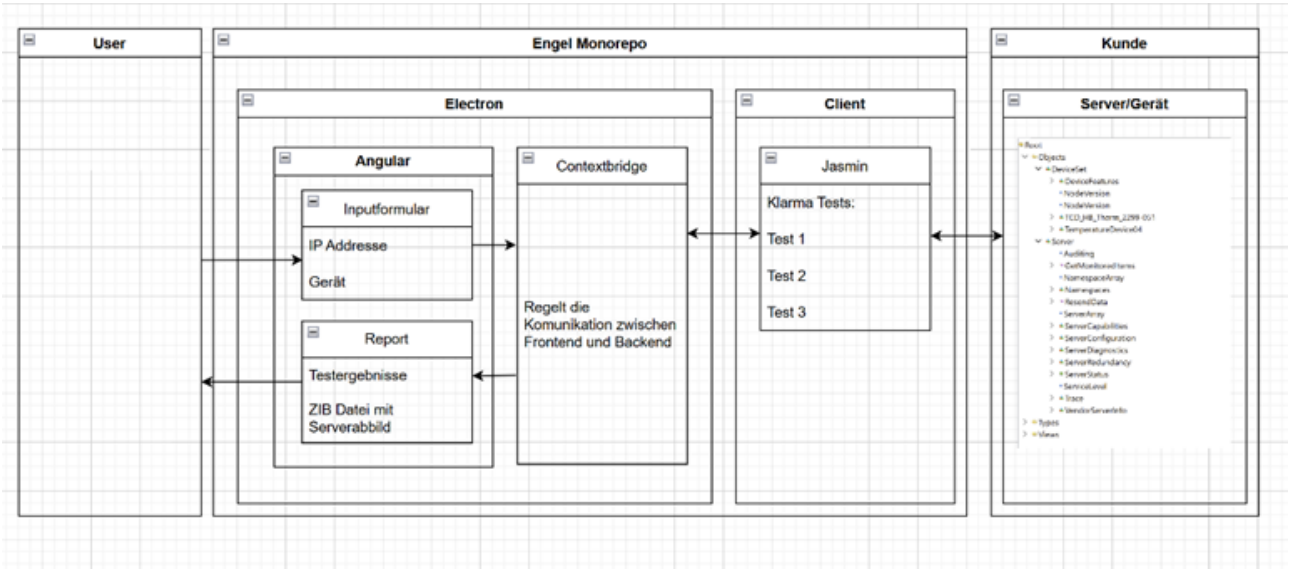


Abbildung 14: Systemaufbau

Das Projekt besteht aus einer Electron-Weboberfläche, wodurch der Benutzer den gedachten OPC-UA-Standard von den Servern zurückgegeben bekommt. Weiters hat der Benutzer die Möglichkeit, die Testfälle, die er abdecken möchte, aus einer Liste auszuwählen, in der die programmierten Tests dynamisch geladen werden. Als Letztes wird der Benutzer an eine Report-Seite weitergeleitet, wo er die Ergebnisse der einzelnen Tests anschauen und überprüfen kann. Da in Angular das MVVM Design Pattern der Standard ist, hat das Projektteam das Frontend auch damit implementiert [36].

Wie man in der Abbildung (14) sieht, beruht die Kommunikation auf den Ideen des Facade Design Pattern. Dieses Pattern ermöglicht es dem Frontend-Entwickler, dass er sich von der komplexen Backend-Logik distanziert und nur über eine möglichst einfache Schnittstelle die gewünschten Funktionen aufzurufen. In der Realität ist diese zentrale Schnittstelle die ContextBridge von Electron, die die Funktionen aus dem komplexen Backend zentral zur Verfügung stellt [36].

Eine der beiden Hauptkomponenten im Backend ist die Testumgebung. Diese nutzt eine Kombination aus zwei Design Patterns, nämlich dem Command- und dem Observer Pattern. Die Ausführung der Tests funktioniert mit dem Command Pattern, da die ContextBridge einen sogenannten "TestRunneräufruft, der dann in weiterer Folge die richtigen Tests startet. Die Rückgabe der Testergebnisse erfolgt über das Observer Pattern, da die ContextBridge stets

darauf achtet, ob ein Test fertig ist. Falls ja, muss sie das Ergebnis sofort zurückliefern. Somit laufen die Tests dynamisch und in Echtzeit [36].

Die Testumgebung greift auf den letzten Hauptbestandteil zu, nämlich auf den Node-OPC-UA-Client. Dieser ist ein Projekt, das einen Client für den OPC-UA-Server simuliert und somit die hierarchische Baumstruktur des Servers durchsucht. Dieser sendet in regelmäßigen Zeitabständen den Status der Verbindung zwischen ihm und dem Server zurück. Weiters bietet er eine Schnittstelle an, welche den erkannten OPC-UA-Standard zurückgibt.

3.1.3 Schnittstellen

Die nachfolgenden Schnittstellen wurden bereits im Rahmen des Projektstarts geplant.

vorhandene Tests erhalten

Diese Schnittstelle ist essenziell, da das Frontend andernfalls eine hardcodierte Liste von Tests anzeigen müsste, was in weiterer Folge zu Fehlern führen könnte. Die Aufgabe dieser Schnittstelle besteht darin, die programmierten Testdateien zu durchsuchen, die darin implementierten Testfunktionen auszulesen und diese in einem vordefinierten Format über die ContextBridge zurückzugeben.

Tests starten

Laut Plan existiert eine Schnittstelle, über die das Frontend OPC-UA Standat Liste übermitteln kann, für welche Tests Ergebnisse benötigt werden. Dabei ist zu beachten, dass diese Schnittstelle keine Resultate liefert, sondern ausschließlich den Testlauf initiiert. Dieses Verhalten ist bewusst so konzipiert, da die Ergebnisse dynamisch und in Echtzeit zurückgeliefert werden müssen.

Ergebnisse erhalten

Bei dieser Schnittstelle handelt es sich um eine Live-Schnittstelle. Das bedeutet, dass das Frontend auf eingehende Ergebnisse wartet. Da es sich hierbei um die komplexeste der definierten Schnittstellen handelt, wurden zwei Umsetzungsvarianten vorgesehen (Plan A und Plan B).

Der primäre Ansatz (Plan A) sieht vor, dass das Backend selbstständig erkennt, wann der Testdurchlauf abgeschlossen ist, und die Schnittstelle eigenständig schließt. Dies würde die Implementierung im Frontend insofern vereinfachen, als dort nicht zusätzlich geprüft werden muss, wann das Ergebnis des letzten Tests eingelangt ist.

Plan B hingegen sieht vor, dass das Frontend selbstständig erkennt, wann der Testlauf beendet ist, und die Schnittstelle entsprechend eigenständig öffnet und schließt.

Das Projektteam priorisiert Plan A, da das Frontend möglichst einfach und ohne zusätzliche Logik zur Zustandsüberwachung des Testlaufs gehalten werden soll.

Status des Servers erhalten

Darüber hinaus soll eine Schnittstelle bereitgestellt werden, die den Verbindungsstatus des Servers an das Frontend übermittelt. Dies ist die einzige Schnittstelle, die direkt auf die node-opcua-Bibliothek zugreift, da die Testumgebung selbst keine Informationen über den aktuellen Status der Serververbindung besitzt.

3.1.4 Monorepo und Ordnerstruktur

Diese drei Bestandteile sind unabhängige Projekte, die jeweils für sich ausführbar sind. Deswegen ist die App in ein Monorepo eingebettet. Ein Monorepo, auch bekannt als "monolithic repository", ist ein Versionskontroll-Repository, das den Source-Code mehrerer Projekte, Bibliotheken oder Microservices an einem zentralen Ort speichert.

Diese Vorgehensweise fördert die Wiederverwendbarkeit von Code-Teilen und erleichtert die Arbeit des Projekt-Teams beim Abhängigkeitsmanagement.

Das Projektteam verwendet das Nx-Setup für die Struktur des Monorepos. Das bedeutet, dass dieses in drei wichtige Ordner unterteilt ist. Als Erstes der Apps-Ordner, der alle Teil-Projekte initialisiert und die Anwendungen startet. Weiters existiert ein Libs-Ordner, in dem sich grundsätzlich die Implementierungen der Teilprojekte befinden. Diese Teile sind möglichst unabhängig voneinander gehalten. Der letzte Ordner wird von Node vorgegeben, nämlich der Dist-Ordner. Hier befinden sich die in JavaScript kompilierten Dateien. In diesem Ordner arbeiten die einzelnen Bestandteile des Monorepos zusammen. Aus diesem Grund ist die Struktur der kompilierten Dateien essenziell im Dist-Ordner.

Aus diesem Grund hat das Projektteam gleich am Anfang überlegt welche Startskripte benötigt werden. Zusätzlich zu den Standard-Build- und Start-Skripten gibt es ein gemeinsames Skript für den Build- und Startprozess des Node-Projekts und der Testumgebung. Es gibt auch ein Build Skript, das den gesamten Monorepo baut, sowie eines, das alles in einem Schritt baut und startet.

3.2 Benutzeroberfläche

Im Folgenden wird dargestellt, welche Ansichten für die Benutzerinteraktion erforderlich sind und wie diese den Ablauf der Anwendung strukturieren.

3.2.1 Welche Seiten werden benötigt?

Damit der Enduser mit dem System interagieren kann, stellen drei Teile die Basis der grafischen Benutzeroberflächen dar:

Beim Start der Anwendung bekommt der User eine Login-Ansicht zu Gesicht. In dieser gibt er bekannt, mit welcher Maschine er sich verbinden möchte. Das kann er mithilfe der Eingaben von IP-Adresse und Port. Optional kann er auch noch einen Usernamen und ein Passwort eingeben, für Maschinen, die nur für bestimmte Personen zur Verfügung stehen und dadurch abgesichert sind. Während dann der User eine Ladeanimation sieht, wird im Hintergrund versucht, sich auf die Maschine zu verbinden. Die Rückmeldung, ob es funktioniert hat wird dann dem User angezeigt mit dem Bonusfakt, um welche Maschine es sich handelt z.B. ein Temperiergerät.

Wenn das Gerät richtig erkannt wird und ein fixer Euromap Standard für dieses Gerät festgelegt ist, wird dieser dem User als Empfehlung vorgeschlagen z.B Temperiergerät hat den Standard Euromap 82.1. Der User kann nun entscheiden ob ihm der Vorschlag recht ist oder er lieber manuell entscheiden möchte welcher Euromap Standard zur Überprüfung der Schnittstelle genommen werden soll. Diese Auswahl soll so dynamisch gestaltet sein. Damit ist gemeint, dass der User mehrere Standards auf einmal prüfen kann, wenn er sich nicht sicher ist, welcher der Richtige ist. Weiters kann er sich auch dafür entscheiden nur spezifische Tests auszuführen und nicht einen ganzen Euromap Standard.

Unabhängig davon, ob der Benutzer den empfohlenen Einstellungen folgt oder selbst konfiguriert, welche Tests ausgeführt werden sollen, gelangt er abschließend immer zum Ergebnisbildschirm. Dieser bietet einen Überblick darüber, welche Tests erfolgreich abgeschlossen wurden und welche fehlgeschlagen sind. Zusätzlich wird angezeigt, welche Tests noch nicht ausgeführt wurden bzw. welcher Test aktuell durchgeführt wird, um dem Benutzer eine laufende Rückmeldung über den aktuellen Status zu geben. Darüber hinaus sollen die einzelnen Tests auch im Detail einsehbar sein, sodass spezifische Informationen darüber abgerufen werden können, welche Fehler konkret aufgetreten sind.

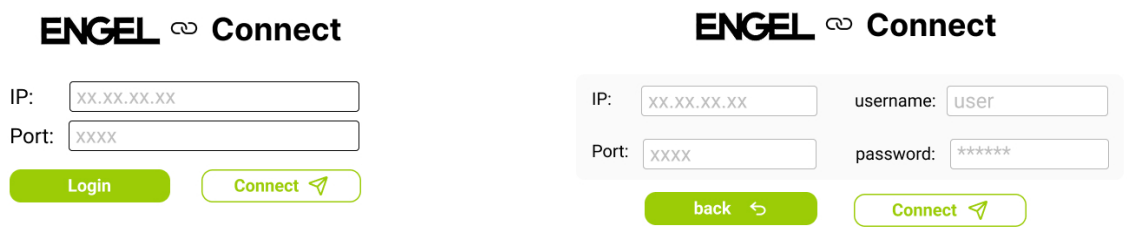
3.2.2 Skizzierung mit Figma

Das Mockup wurde bereits in den ersten Tagen des Praktikums konzipiert, um einen klaren Anhaltspunkt für die weitere Entwicklung zu schaffen. Dadurch konnte sich das Projektteam frühzeitig mit den Zielen der ENGEL-Mitarbeiter vertraut machen. Zudem sollten durch die frühzeitige Ausarbeitung mögliche Unstimmigkeiten während der Frontend-Entwicklung vermieden werden, sodass ein möglichst reibungsloser Arbeitsablauf gewährleistet werden konnte.

3.2.3 Startscreens

Wie in Abbildung 15 dargestellt, gibt es — wie bereits zuvor erläutert — zwei Möglichkeiten, eine Verbindung zur Maschine herzustellen. Zum einen erfolgt die standardmäßige Verbindung, wie in Abbildung 15a dargestellt, und zum anderen eine Verbindung mit zusätzlicher Authentifizierung mittels Benutzername und Passwort, wie in Abbildung 15b gezeigt.

Zwischen den beiden Ansichten kann mithilfe der grünen Buttons navigiert werden. Nachdem alle erforderlichen Eingaben vorgenommen wurden, gelangt man durch Betätigen des „Connect“-Buttons zum Ladebildschirm 16a.



(a) Startscreen ohne Login (Standard)

(b) Startscreen mit Login falls benötigt

Abbildung 15: Vergleich der beiden Verbindungsmöglichkeiten

3.2.4 Connectionscreens

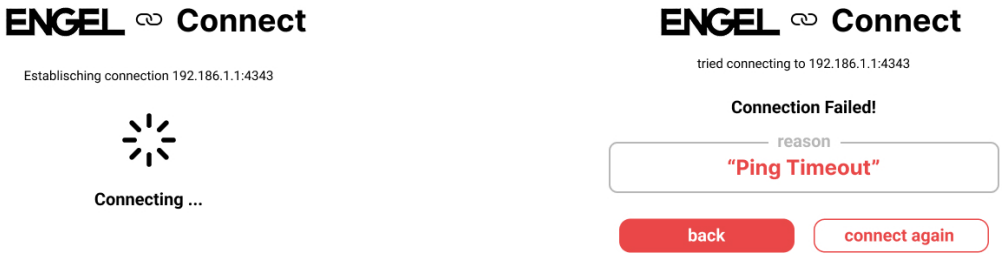
Während des Ladebildschirms (16a) wird versucht, eine Verbindung zur Maschine herzustellen. Sollte dies fehlschlagen, gelangt der Benutzer zur Seite „Verbindung fehlgeschlagen“ (16b), auf der eine entsprechende Fehlermeldung angezeigt wird, um eine klare Rückmeldung über die Ursache des Fehlers zu geben.

Der Benutzer kann anschließend entscheiden, ob er zum Startbildschirm zurückkehren möchte, beispielsweise wenn er festgestellt hat, dass ein falscher Port eingegeben wurde oder ein Tippfehler

beim Passwort vorliegt. Alternativ kann er, falls er sich sicher ist, dass die Eingaben korrekt sind — etwa wenn die Maschine zuvor noch nicht eingeschaltet war — nach dem Einschalten erneut mit denselben Daten eine Verbindung herstellen.

Wenn die Verbindung erfolgreich hergestellt wurde, gelangt der Benutzer zur Seite „Connection Successful“ (17). Dort wird — sofern möglich — angezeigt, mit welchem Gerät eine Verbindung hergestellt wurde, beispielsweise mit einem Temperiergerät.

Der Benutzer hat nun zwei Möglichkeiten: Einerseits kann, sofern anhand des Maschinentyps erkannt wird, welche Schnittstelle die Maschine verwendet, der von der Anwendung vorgeschlagene Test unmittelbar ausgeführt werden. Andererseits besteht die Möglichkeit, die Tests manuell zu konfigurieren und auszuwählen, welche Tests, wie viele Tests oder auch nur Teilbereiche eines Tests ausgeführt werden sollen.



(a) Loading Screen (b) Verbindung fehlgeschlagen

Abbildung 16: Mögliche Zwischenscreens vor einem erfolgreichen Verbindungsaufbau

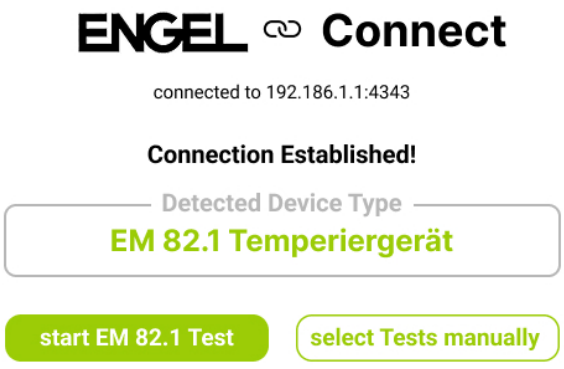


Abbildung 17: Verbindung erfolgreich

3.2.5 Testauswahlscreen

Bei der manuellen Auswahl der Tests, wie in Abbildung 18 dargestellt, erhält der Benutzer eine Liste aller verfügbaren Schnittstellentests. Diese können ausgewählt werden, wenn beispielsweise das gesamte Testpaket, etwa für Euromap 82.1, ausgeführt werden soll. Alternativ besteht die Möglichkeit, einen Test aufzuklappen und nur einzelne darunterliegende Testpakete auszuwählen.

Die Struktur ist — wie zuvor erläutert — baumartig bzw. rekursiv aufgebaut. Dabei gilt die Regel, dass bei der Auswahl eines Testpakets automatisch alle darunterliegenden Testpakete ebenfalls als ausgewählt gelten.

Ist der Benutzer mit seiner Auswahl zufrieden, wie beispielsweise in der Abbildung dargestellt, kann er rechts oben über den Button „Run Selected“ die ausgewählten Tests starten. Falls der Benutzer unsicher ist, welcher Test der richtige ist — etwa wenn kein Gerät erkannt wurde und somit unklar ist, welche Schnittstelle implementiert wurde — besteht alternativ die Möglichkeit, alle Tests über den Button „Run All“ links neben dem zuvor genannten Button auszuführen. Anhand der späteren Auswertung der Testergebnisse kann anschließend erkannt werden, welche Schnittstelle am ehesten zutrifft.

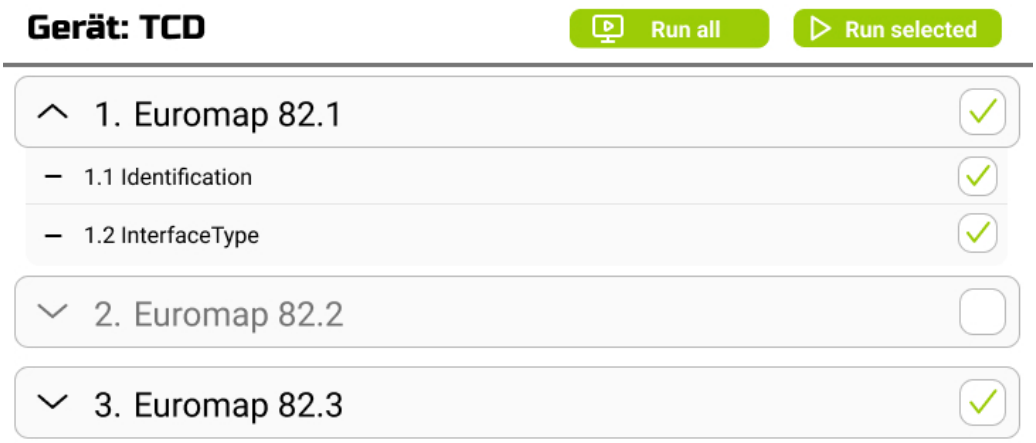


Abbildung 18: Manuelle Testauswahl

3.2.6 Ansichten aller laufenden und abgeschlossenen Tests

Beim Ausführen der Tests erscheint eine Ansicht, wie in Abbildung 19 dargestellt. Im oberen Bereich wird angezeigt, wie viele Tests bereits ausgeführt wurden, beispielsweise drei von fünf, wie in der Abbildung 19 zu sehen. Zusätzlich wird dargestellt, wie viele Tests bereits fehlgeschlagen sind, beispielsweise einer.

Darunter werden die vier möglichen Statuszustände eines Tests visualisiert. Einerseits gibt es die grün umrandeten Tests, die als „Successful“ gekennzeichnet sind. Dabei handelt es sich um Tests, die erfolgreich abgeschlossen wurden. Das Gegenstück dazu sind die rot umrandeten „Failed“-Tests, welche fehlgeschlagene Tests darstellen.

Weiters gibt es einen normal dargestellten Test mit schwarzer Umrandung, wie im Beispiel Euromap 82.3 (Abbildung 19). Dieser stellt den aktuell laufenden Test dar. Dabei wird zusätzlich angezeigt, wie weit der Test fortgeschritten ist, beispielsweise drei von sieben Untertests abgeschlossen.

Abschließend gibt es noch die ausgegrauten Tests. Diese stellen Tests dar, die noch nicht ausgeführt wurden.

Dieses Farbschema bleibt in der gesamten Applikation sowie auf allen Detailseiten konsistent, um für den Benutzer eine klare und verständliche Orientierung zu gewährleisten.

Sobald alle Tests abgeschlossen sind, kann der Benutzer über den Button rechts oben eine ZIP-Datei mit den Testergebnissen herunterladen. In der dargestellten Situation ist dies noch nicht möglich, da die Tests noch laufen, weshalb der Button ausgegraut bzw. deaktiviert dargestellt ist.

Zusätzlich kann der Benutzer einzelne Tests auswählen, um die jeweiligen Untertestpakete einzusehen. Dies ist unabhängig vom aktuellen Status eines Tests möglich und wird in Abbildung 19 näher erläutert.

3.2.7 Detailansicht für gelaufene Tests

Wählt der Benutzer einen der Tests aus, gelangt er zu einer Detailansicht. Befindet sich der Test noch im laufenden Zustand, wird eine Ansicht angezeigt, die vom Projektteam konzipiert wurde, wie in Abbildung 20a dargestellt.

Eine entsprechende Ansicht existiert ebenfalls für erfolgreich abgeschlossene Tests (siehe Abbildung 20b) sowie für fehlgeschlagene Tests (siehe Abbildung 21).

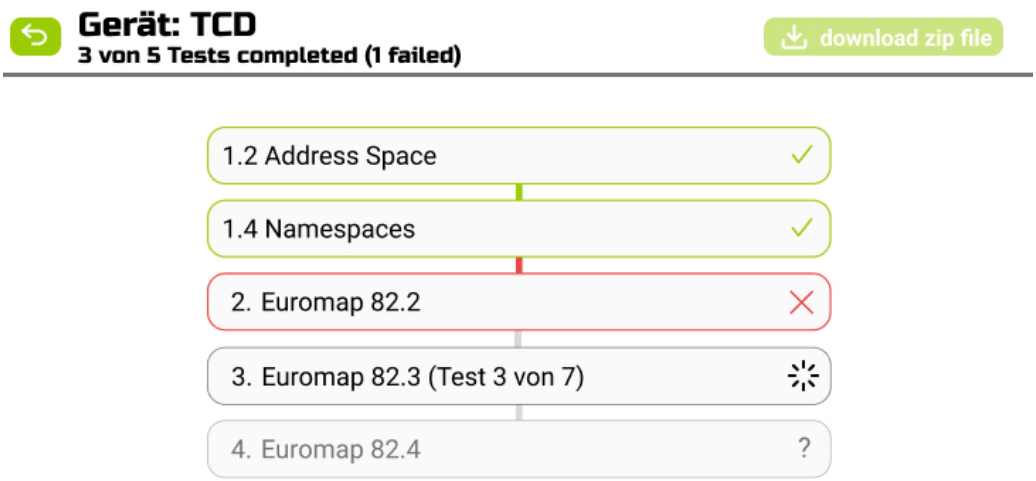
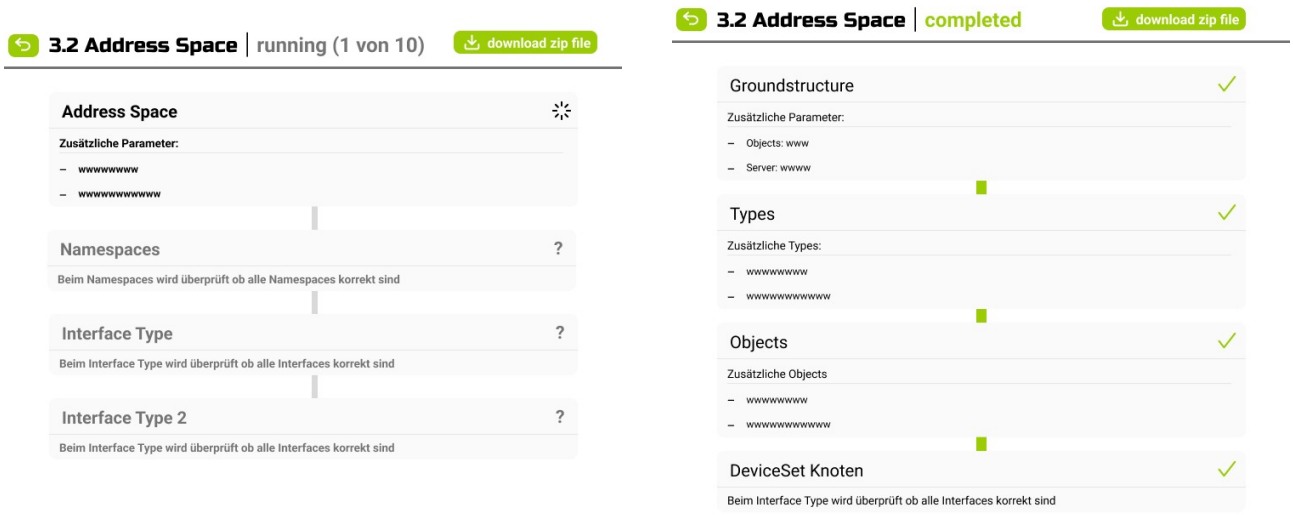


Abbildung 19: Ansicht aller Testfälle

Wie bereits zuvor erwähnt, werden auch hier dieselben Symbole und Farben verwendet, um den jeweiligen Status eindeutig darzustellen, wie in allen drei Abbildungen im oberen Bereich ersichtlich ist. Dabei ist zu beachten, dass bereits ein einzelner fehlgeschlagener Test dazu führt, dass das gesamte Testpaket als fehlgeschlagen gilt, wie in Abbildung 21 dargestellt.

Darüber hinaus kann der Benutzer in dieser Ansicht detailliert nachvollziehen, welche Fehler aufgetreten sind, wie beispielsweise beim fehlgeschlagenen Test in Abbildung 21, bei dem die Modellbezeichnung nicht gefunden wurde.

Auch bei erfolgreichen Tests wird eine Rückmeldung angezeigt. Dies ist relevant, da es vorkommen kann, dass zwar alle erforderlichen Parameter vorhanden sind und der Test somit erfolgreich abgeschlossen wird, jedoch zusätzliche Parameter erkannt wurden. Diese führen zwar nicht zum Fehlschlagen des Tests, können für den Benutzer jedoch dennoch hilfreiche Informationen darstellen.



(a) Detailview Test in Progress (b) Detailview Successful Test

Abbildung 20: Screens bei einem normalen Testablauf, bei dem nichts schiefgelaufen ist

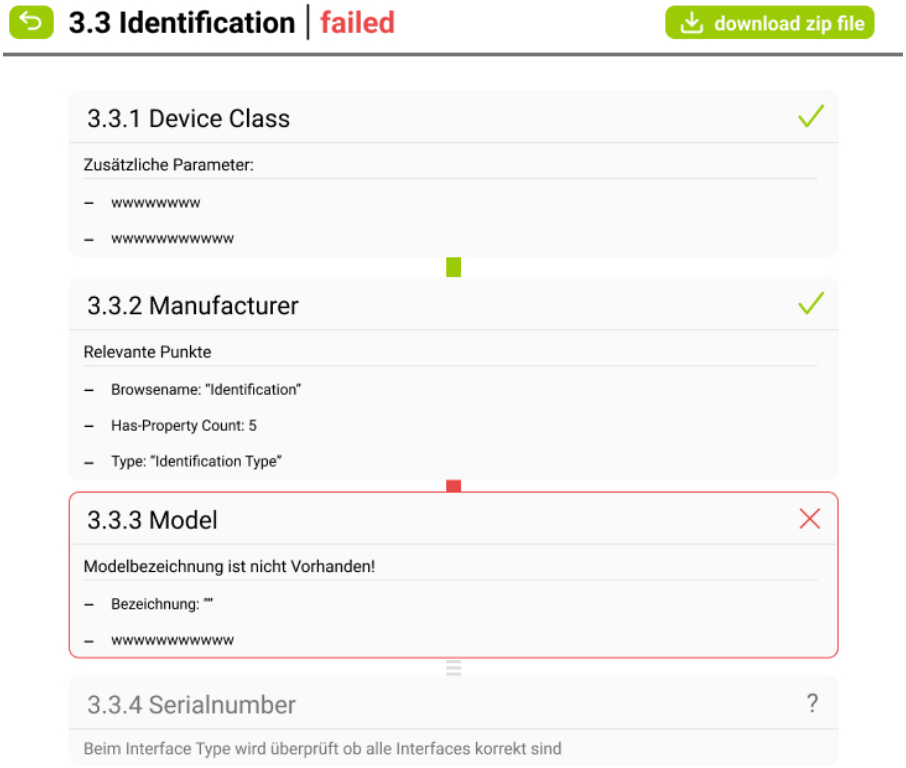


Abbildung 21: Detailview Failed Test

4 Implementierung

Im folgenden Kapitel wird erläutert, wie das Projekt umgesetzt wurde. Dies umfasst insbesondere die Implementierung der ContextBridge, des Node OPC-UA Clients, der Testumgebung sowie des zugehörigen Angular-Frontends.

4.1 ContextBridge

Die Gestaltung der ContextBridge ist in zwei Kanäle unterteilt. Der erste Kanal, der als 'testHandler' bezeichnet wird, stellt die Verbindung zur Testumgebung her und stellt Funktionen bereit, mit denen die Ausführung der Tests durch verschiedene Instrumente gesteuert werden kann.

Der zweite Kanal ist als 'deviceHandler' benannt und stellt die Verbindung zum Node OPC-UA Client her. Über diesen Kanal können Funktionen aufgerufen werden, mit denen Informationen über den aktuellen Verbindungsstatus zum Server abgerufen werden können.

Diese Umsetzung stellt eine leichte Abweichung vom ursprünglich geplanten Konzept dar. Während zu Beginn vorgesehen war, drei getrennte ContextBridge-Schnittstellen zu verwenden, zeigte sich im Verlauf des Projekts, dass keine separate ContextBridge erforderlich ist, solange auf dieselbe Anwendung zugegriffen wird. Dadurch konnten die Schnittstellen zwischen den Anwendungen übersichtlicher gestaltet und die Gesamtstruktur der Kommunikation vereinfacht werden.

4.1.1 Allgemeiner Aufbau

Um die Funktionsweise der einzelnen Schnittstellen zu verstehen, ist es wichtig, den allgemeinen Aufbau eines ContextBridge-Kanals nachzuvollziehen zu können.

Request/Response Schnittstelle

Request/Response-Schnittstellen funktionieren analog zu einfachen API-Calls: Ein Request wird vom Main Prozess an den Renderer Prozess gesendet, woraufhin eine entsprechende Response zurückgeliefert wird.

Renderer Prozess

Im Main-Prozess muss die ContextBridge über die Funktion *.exposeInMainWorld()* definiert werden. Der erste Parameter dient dabei zur Benennung der Schnittstelle. In diesem einfachen Beispiel wird die Schnittstelle als *'Interface'* benannt, wodurch sie vom Frontend über diesen Namen aufgerufen werden kann.

Innerhalb dieser Schnittstellen können über den *ipcRenderer* mehrere Funktionen als Callbacks hinterlegt werden, die vom Frontend verwendet bzw. aufgerufen werden können. Für Request/Response-Funktionen wird die Methode *.invoke()* verwendet, um einen spezifischen Kanal zu definieren. Auch hier wird der erste Parameter als für die Benennung reserviert, sodass er im Renderer-Prozess eindeutig identifiziert werden kann. Dieser ist im Beispiel als *MessageChannel* bezeichnet. Anschließend können beliebige Parameter an die Funktion übergeben werden.

Listing 1: Beispiel Renderer Prozess bei einer Request/Response Schnittstelle

```
1      contextBridge.exposeInMainWorld('Interface', {
2          action: (input: any) => {
3              ipcRenderer.invoke('MessageChannel', input);
4          },
5          //Alle anderen Channels definieren
6      });
```

Main-Prozess Bei dieser Art von Schnittstelle muss die Methode *.handle()* des *ipcMain*-Objekts verwendet werden. Der erste Parameter legt fest, auf welchen Kanal vom Renderer-Prozess reagiert werden soll. Anschließend wird als Callback definiert, welche Funktionalität die Schnittstelle ausführen soll.

Neben den Übergabeparametern gibt es zudem einen *event*-Parameter, der grundsätzlich eine Instanz des Objekts *Electron.IpcMainEvent* darstellt. Dieser Parameter spielt insbesondere bei anderen Arten von Kanälen eine Rolle, ist jedoch auch hier obligatorisch anzugeben, um die Schnittstelle korrekt zu definieren.

Listing 2: Beispiel Main Prozess bei ein Request/Response Schnittstelle

```
1       ipcMain.handle('MessageChannel', async (event, input)=> {
2           return await doStuff(input);
3       });
```

Eventbasiertes Schnittstelle

Diese Schnittstellen sind Fire-and-Forget-basiert. Das bedeutet, dass der Renderer-Prozess die Nachricht an den Main-Prozess senden kann, ohne auf eine sofortige Antwort zu warten. Der Main-Prozess verarbeitet die Nachricht asynchron, und der Renderer kann währenddessen weiterarbeiten.

Renderer Prozess

Im Renderer-Prozess besteht der Hauptunterschied darin, dass statt der Methode *.invoke()* die Methode *.send()* verwendet wird. Diese Methode liefert selbst keinen Rückgabewert zurück, weshalb ein separater Kanal benötigt wird, um Antworten zu empfangen.

Dieser Antwortkanal wird mit einem Parameter definiert, der eine Callback-Funktion darstellt. Über diese Callback-Funktion kann das Frontend auf die zurückgegebenen Ergebnisse reagieren und diese bei Bedarf verarbeiten.

Die Funktion nutzt die Methode *.on()* und wartet auf eine Nachricht vom Kanal, dessen Name als erster Parameter angegeben wurde. Im vorliegenden Beispiel wartet der Renderer-Prozess auf Nachrichten des Kanals *'ReturnChannel'*.

Listing 3: Beispiel Renderer Prozess bei einer eventbasierten Schnittstelle

```
1  contextBridge.exposeInMainWorld('Interface', {
2    action: (input: any) => {
3      ipcRenderer.send('MessageChannel', input);
4    },
5    setListener: (callback: (event , result ) => void) => {
6      ipcRenderer.on(ReturnChannel, (event, data) => {
7        callback(event, data);
8      });
9    }
10   //Alle anderen Channels definieren
11 });
```

Main Prozess

Wie bereits erwähnt, besteht der Unterschied darin, dass hier keine klassische Rückgabe (return) möglich ist. Um Daten an den Sender zurückzugeben, muss ein eigener Kanal definiert werden,

der die Methode *.reply()* des *Event*-Objekts verwendet. Diese Methode verhält sich ähnlich wie *.send()*, die jedoch nicht im Kontext des *event* Objekts verfügbar ist.

In diesem Fall ist das *event* Objekt eine Instanz von *Electron.IpcRendererEvent*. Es enthält Metadaten über die vom Main-Prozess empfangene Nachricht und ermöglicht dem Renderer-Prozess, auf die Nachricht zu reagieren und die entsprechenden Daten über den definierten Kanal zurückzusenden.

Listing 4: Beispiel Main Prozess bei einer eventbasierten Schnittstelle

```
1 run: (param: string[]) => {
2     ipcRenderer.send('<start-channel>', param);
3 },
```

4.1.2 ContextBridge: testHandler

Die nachfolgenden Funktionen sind Teil der ContextBridge, die für die Verwaltung der Testläufe zuständig ist. Sie übernehmen zentrale Aufgaben wie das Starten und Stoppen von Testdurchläufen, das Zurückliefern von Testergebnissen in Echtzeit, das Übermitteln von Statusinformationen an das Frontend sowie die Organisation der Testhierarchie. Durch diese Funktionen wird sichergestellt, dass die Kommunikation zwischen Frontend, Backend und Testumgebung effizient, zuverlässig und konsistent abläuft, wodurch die Testläufe dynamisch und nachvollziehbar gesteuert werden können.

Funktion: runTests

Renderer Prozess

Diese eventbasierte Schnittstelle erwartet ein Array von Strings, das die Testfälle enthält, die ausgeführt werden sollen. Dabei ist es entscheidend, dass die Testnamen exakt mit den vorhandenen Tests übereinstimmen, da sie sonst später nicht korrekt zugeordnet werden können.

Die korrekte Formatierung der Testnamen wird indirekt durch die ContextBridge-Funktion *listAll()* sichergestellt, welche alle vorhandenen Tests richtig zurückliefert und somit eine fehlerfreie Zuordnung ermöglicht. 4.1.2

Listing 5: runTests-Backend Funktion

```
1 runTests: (testNames: string[]) => {
2     ipcRenderer.send('run-multiple-tests', testNames);
3 }
```

Main Prozess

Im Main-Prozess startet dieser Kanal alle Tests, die vom Frontend übergeben wurden. Dies erfolgt durch den Aufruf der Funktion *runVitest()*. Damit die Tests für den korrekten OPC-UA-Server ausgeführt werden, benötigt diese Funktion zusätzlich die IP-Adresse und den Port des Servers. Diese Variablen sind global im entsprechenden File definiert, da zuvor der Kanal *getDevice()* aufgerufen werden muss, um die Variablen zu initialisieren.

Für die Response gibt es zwei mögliche Fälle:

- Falls keine Fehler auftreten, wird nach Abschluss eines Tests die Methode *.reply()* aufgerufen, ähnlich wie im Code dargestellt. Der Unterschied besteht darin, dass hier das Testergebnis zurückgegeben wird, anstatt eines Fehlers. Dieser Aufruf muss tief verschachtelt erfolgen, da nur über einen bestimmten Vitest-Reporter auf den Abschluss eines Tests reagiert werden kann.
- Treten Fehler auf, werden diese über die Methode *.reply()* an den Renderer-Prozess zurückgeschickt.

In beiden Fällen ist der Kanalname gleich. Dieser wird als zufällig generierte UUID vergeben, um einzelne Testdurchläufe eindeutig voneinander zu unterscheiden. Die genaue Logik hierzu wird in der Funktion *stop()* detailliert erläutert. (Siehe 4.1.2)

Listing 6: runTests Main Prozess

```
ipcMain.on('run-multiple-tests', async (event:
    Electron.IpcMainEvent, testNames: string[]) => {
  try {

    await runVitest(event, testNames, ip_local, port_local);

  } catch (error) {
    event.reply(RESULT_LISTENER_ID, { testNames, error: error });
  }
});
```

Funktion - Renderer Prozess: setTestResultListener

Hier werden die zuvor zurückgelieferten Werte empfangen. Es ist wichtig zu beachten, dass auf die Rückgabe jeweils nur eines einzelnen Testfalls reagiert wird. Dadurch ist es möglich, dass die Tests dynamisch, also in Echtzeit, ausgeführt werden.

Innerhalb dieser Funktion werden zur Sicherheit alle Kanäle mit demselben Namen gelöscht, um Konflikte zwischen Kanälen zu vermeiden, falls das Frontend den vorherigen Testlauf nicht selbst beendet hat.

Für die Verarbeitung der Ergebnisse muss das Frontend einen Callback übergeben, über den die Testergebnisse weiterverarbeitet oder manipuliert werden können.

Listing 7: setTestResultListener im Renderer Prozess

```
1 setTestResultListener: (callback: (event:
     Electron.IpcRendererEvent, result: Result) => void) => {
2     ipcRenderer.removeAllListeners(RESULT_LISTENER_ID);
3     ipcRenderer.on(RESULT_LISTENER_ID, (event, data) => {
4       callback(event, data);
5     });
6 }
```

Funktion - Renderer Prozess: setOnTestReadyListener

Darüber hinaus muss das Frontend wissen, welcher Test als nächstes ausgeführt wird. Da diese Information nicht im Ergebnis eines Tests enthalten ist, musste eine eigene Logik entwickelt werden.

Im zuvor beschriebenen Vitest-Reporter kann auf den Start eines Tests reagiert werden, bevor dieser ausgeführt wird. In diesem Fall wird die Methode *.reply()* aufgerufen und das Ereignis über einen Kanal mit einer zufällig generierten UUID zurück an das Frontend übermittelt.

Die UUID wird beim Stoppen des Testlaufs jeweils neu gesetzt. Darüber hinaus berücksichtigt die ContextBridge auch den Fall, dass das Stoppen vom Frontend übersehen wurde, um eine konsistente und fehlerfreie Testausführung sicherzustellen.

Listing 8: setTestResultListener Main Prozess

```
1 setOnTestReadyListener: (callback: (event:
     Electron.IpcRendererEvent, nextTestPath: string) => void) => {
2     ipcRenderer.removeAllListeners(NEXT_LISTENER_ID);
3     ipcRenderer.on(NEXT_LISTENER_ID, (event, data) => {
4       callback(event, data);
5     });
```

```
6 }
```

Funktion - Renderer Prozess: stop

Bei dieser Funktion ist der Name etwas irreführend, da der Testlauf selbst nicht tatsächlich gestoppt wird. Dies liegt daran, dass ein direktes Beenden eines laufenden Testdurchlaufs in Vitest nicht vorgesehen ist.

Aus diesem Grund wird das Problem durch eine alternative Vorgehensweise umgangen. Anstatt den laufenden Testlauf zu beenden, generiert die Funktion neue Kanalnamen, damit keine Daten aus einem vorherigen Testlauf übernommen werden. Dies erfolgt mithilfe der Funktion *crypto.randomUUID()*.

Diese sogenannte UUID (Universally Unique Identifier) besteht aus 36 hexadezimalen Zeichen und wird intern über den Zufallszahlengenerator der Web Crypto API erzeugt.

Die Verwendung einer neuen UUID ist notwendig, da es vorkommen kann, dass ein Benutzer das Ende eines Testlaufs nicht abwartet und unmittelbar einen neuen Testlauf startet. Durch die Generierung neuer Kanalnamen wird sichergestellt, dass sich mehrere Testläufe nicht gegenseitig beeinflussen und eindeutig voneinander unterschieden werden können.

Listing 9: Stop Funktion im Renderer Prozess

```
1 stop: () => {
2     setResultListenerId(crypto.randomUUID());
3     setNextListenerId(crypto.randomUUID())
4 }
```

Funktion: listAll

Renderer Prozess

Ziel dieser Schnittstelle ist es, auf Anfrage alle verfügbaren Tests an das Frontend zurückzuliefern. Das Format der Rückgabe wurde vom Projektteam gemeinsam festgelegt.

Dabei handelt es sich um eine Baumstruktur, die aus sogenannten *TreeNodes* besteht. Diese können wiederum weitere *TreeNodes* als *children* enthalten. Auf diese Weise entsteht eine hierarchische Struktur von Tests, die sowohl vom Frontend als auch vom Backend für die Implementierung genutzt werden kann.

Listing 10: ListAll Funktion im Renderer Prozess

```
listAll: (): Promise<TreeNode<TreeNodeData>[]> => {
    return ipcRenderer.invoke('list-all-tests');
}
```

Main Prozess

Listing 11: ListAll Funktion im Main Prozess

```
ipcMain.handle('list-all-tests', async () => {
    return await getTestCases();
})
```

Betrachtet man den Main-Prozess, ist auch diese Funktion in der ContextBridge vergleichsweise einfach aufgebaut. Sie ruft die Funktion *getTestCases()* der *Testumgebung* auf. Dabei handelt es sich um eine asynchrone Funktion, die aus den vorhandenen Tests die zuvor beschriebene Baumstruktur erstellt und im entsprechenden Format zurückliefert.

Durch diese Funktion wird indirekt sichergestellt, dass das Frontend die Testnamen korrekt formatiert. Aus diesem Grund kann der Kanal *runTests()* davon ausgehen, dass die Funktion vom Frontend mit gültigen und korrekt formatierten Testnamen aufgerufen wird.

4.1.3 deviceHandler

Die nachfolgende ContextBridge ist für die Kommunikation mit dem OPC-UA-Client zuständig. Sie stellt die Schnittstelle bereit, über die das Backend Informationen vom Server abrufen, Verbindungsstatus überwachen und Funktionsaufrufe durchführen kann. Dadurch wird eine strukturierte und zuverlässige Interaktion zwischen der Testumgebung und dem OPC-UA-Client gewährleistet.

Funktion: getDevice

Renderer Prozess

Diese Schnittstelle muss als erste aufgerufen werden. Der Kanal dient dazu, die Verbindung zum OPC-UA-Server herzustellen beziehungsweise zu überprüfen.

Eine weitere Funktion besteht darin, den vermuteten OPC-UA-Standard des Servers an das Frontend zu übermitteln.

Da manche Geräte eine Authentifizierung erfordern, bietet diese Schnittstelle zusätzlich optionale Parameter für *username* und *password* an.

Listing 12: getDevice Renderer Prozess

```
getDevice: (ip: string, port: number, username?: string,
    password?: string): Promise<string> => {
     return ipcRenderer.invoke('get-device-info', ip, port,
        username, password);
}
```

Main Prozess

Beim Aufruf dieser Funktion wird zunächst der OPC-UA-Server angepingt, um sicherzustellen, dass die Verbindung grundsätzlich funktioniert. Falls dieser Schritt fehlschlägt, wird eine entsprechende Fehlermeldung zurückgegeben.

Bei einer erfolgreichen Verbindung werden die Variablen *ip_local* und *port_local* gesetzt, sodass diese Informationen nicht bei jedem weiteren ContextBridge-Aufruf erneut vom Frontend übergeben werden müssen.

Anschließend wird die Funktion *findProtocolName()* des OPC-UA-Clients aufgerufen, um den verwendeten OPC-UA-Standard des Geräts zu ermitteln. Wichtig ist dabei, dass der Rückgabewert der Funktion in folgendem Format vorliegt: *protocolName/deviceName*.

Der erste Teil der Rückgabe ist jedoch noch nicht in einer benutzerfreundlichen Form formatiert und muss daher mithilfe der Funktion *formatTestName()* angepasst werden. Der endgültige Rückgabewert entspricht somit dem Format *<OPC-UA-Standard>/<vollständiger Gerätename>*.

Listing 13: getDevice Main Prozess

```
ipcMain.handle('get-device-info', async (event:
    Electron.IpcMainEvent, ip: string, port: number, username?:
    string, password?: string) => {
     try {
       await opc_ua_client.withTimeout(
          opc_ua_client.initClient(ip, port), 5000);
     } catch (e) {
       return "error - connection couldn't been established";
     }

     ip_local = ip;
     port_local = port;

     const device: string = await opc_ua_client.findProtocolName();

     await opc_ua_client.disconnect();

     const parts = device.split("/");
     const devicename = parts[1];
```

```
17      let testName = formatTestName(parts[0]);
18      const reconstructed = testName + '/' + devicename;
19      return reconstructed;
20  })
```

Funktion: startPing

Renderer Prozess

Sobald ein Gerät verbunden ist, wird diese Schnittstelle aufgerufen, die einen eigenen Prozess startet. Dabei handelt es sich um eine eventbasierte Schnittstelle, weshalb sie keinen direkten Rückgabewert erwartet.

Listing 14: startPing Renderer Prozess

```
1  startPing: () => {
2      ipcRenderer.send('start-opc-ping');
3  }
```

Main Prozess

Wenn dieser Kanal aufgerufen wird, wird zunächst der Renderer-Prozess zur Liste der abonnierten Prozesse hinzugefügt. Anschließend wird überprüft, ob bereits ein *PingWorker* existiert. Falls dies nicht der Fall ist, wird ein neuer Worker erstellt.

Der *PingWorker* ist im Wesentlichen ein Thread, der in regelmäßigen Zeitabständen den OPC-UA-Server anpingt. In diesem Fall beträgt das Intervall fünf Sekunden. Diese Vorgehensweise ist notwendig, da nicht alle OPC-UA-Server mehrere parallele Verbindungen gleichzeitig unterstützen.

Der *PingWorker* übermittelt in den festgelegten Intervallen, ob der Server online oder offline ist. Diese Informationen werden sofort über den Kanal *'opc-ping-status'* an den Renderer zurückgeliefert.

Sobald der Thread beendet wird, wird dies über den *'exit'*-Kanal signalisiert, woraufhin der Worker zurückgesetzt wird. Im Falle von Fehlern erfolgt eine Konsolenausgabe zu Debugging-Zwecken.

Listing 15: startPing Main Prozess

```
ipcMain.on('start-opc-ping', (event) => {
    pingListeners.add(event.sender);

    if (pingWorker) return;

    pingWorker = new Worker(
      path.join(__dirname, '../../libs/libs-opc-ua-client
         /src/libs/ping-worker.js'),
      {
        workerData: { ip_local, port_local, interval: 5000 },
      }
    );

    pingWorker.on('message', (msg) => {
      const status = msg.reachable ? 'online' : 'offline';
      for (const listener of pingListeners) {
        if (!listener.isDestroyed()) {
          listener.send('opc-ping-status', status);
        }
      }
    });

    pingWorker.on('error', (err) => {
      console.error('Ping worker error:', err);
    });

    pingWorker.on('exit', () => {
      pingWorker = null;
      pingListeners.clear();
    });
});
```

Funktion: stopPing

Renderer Prozess

Diese Schnittstelle wird vom Frontend aufgerufen, wenn kein Statusupdate zur Verbindung benötigt wird. Dabei handelt es sich um eine eventbasierte Schnittstelle, die somit keine Antwort vom Main-Prozess erwartet.

Listing 16: stopPing Renderer Prozess

```
stopPing: () => {
    ipcRenderer.send('stop-opc-ping');
}
```

Main Prozess

Im Main-Prozess wird zunächst der aktuelle Renderer-Prozess aus der Liste der Listener (*pingListeners*) entfernt. Anschließend wird der *PingWorker* beendet, falls kein weiterer Renderer-Prozess mehr abonniert ist und der Worker aktuell läuft.

Auf diese Weise wird sichergestellt, dass mehrere Renderer-Prozesse gleichzeitig den Worker abonnieren können, ohne dass Konflikte oder redundante Worker-Instanzen entstehen.

Listing 17: stopPing Main Prozess

```
ipcMain.on('stop-opc-ping', (event) => {
    pingListeners.delete(event.sender);

    // If no one is listening anymore, kill the worker
    if (pingListeners.size === 0 && pingWorker) {
      pingWorker.terminate();
      pingWorker = null;
    }
});
```

Funktion - Renderer Prozess: onStatusChange

Über diesen Kanal erhält das Frontend die Rückmeldungen der zuvor beschriebenen Funktion *startPing()*. Der Rückgabewert ist so konfiguriert, dass er ausschließlich die Zustände *'online'* oder *'offline'* annehmen kann.

Listing 18: onStatusChange Renderer Prozess

```
onStatusChange: (callback: (status: 'online' | 'offline') => void)
   => {
    ipcRenderer.on('opc-ping-status', (_, status: 'online' |
       'offline') => {
      callback(status);
    });
}
```

4.2 Umsetzung in Angular

Die Anwendung folgt einer komponentenbasierten Architektur unter Verwendung des Angular-Frameworks. Die Benutzeroberfläche ist in mehrere unabhängige Komponenten unterteilt, während die Geschäftslogik in zentralen Services gekapselt ist.

4.2.1 Komponentenübersicht

Im Folgenden wird zunächst eine Übersicht über die zentralen Komponenten der Anwendung gegeben. Diese dient dazu, die Struktur der Benutzeroberfläche sowie die Aufteilung der einzelnen Funktionseinheiten nachvollziehbar darzustellen. Anschließend werden die wichtigsten Komponenten im Detail beschrieben, beginnend mit der Startscreen-Komponente als Einstiegspunkt der Anwendung.

Startscreen-Komponente

Die Startscreen-Komponente stellt den Einstiegspunkt der Anwendung dar. Sie wird beim Start der Applikation als erste Ansicht geladen und dient zur Eingabe der grundlegenden Verbindungsparameter, bestehend aus IP-Adresse und Port.

Die Komponente bietet zwei zentrale Aktionen:

- Navigation zur Login-Komponente
- Direkter Verbindungsaufbau über den Loading-Screen

Zur Persistierung der eingegebenen Daten wird der BaseData-Service verwendet. Dieser ermöglicht es, die eingegebenen Werte zwischen verschiedenen Komponenten hinweg beizubehalten. Zusätzlich wird beim Initialisieren der Komponente bereits der Backend-Testbaum über den ListHistory-Service geladen, um spätere Wartezeiten zu minimieren.

Login-Komponente

Die Login-Komponente erweitert die Funktionalität des Startscreens um Benutzeranmeldedaten. Neben IP-Adresse und Port werden hier zusätzlich Benutzername und Passwort erfasst. Für die Passworteingabe wird eine Komponente aus der Bibliothek PrimeNG (`p-password`) verwendet, welche eine maskierte Darstellung sowie eine Umschaltfunktion zur Anzeige des Klartexts bietet. Die eingegebenen Daten werden über den BaseData-Service gespeichert. Die eigentliche Verbindung wird jedoch erst durch explizites Auslösen der Connect-Funktion aufgebaut.

Loading-Screen-Komponente

Die Loading-Screen-Komponente visualisiert den Verbindungsstatus zur Zielmaschine und fungiert als Zustandsanzeige mit drei möglichen Ausprägungen:

- `Verbindungsaufbau (Establishing)`: Anzeige einer Ladeanimation mittels PrimeNG ProgressSpinner.
- `Fehlgeschlagen (Failed)`: Darstellung einer Fehlermeldung inklusive Ursache sowie Optionen zum erneuten Verbindungsversuch oder zur Rücknavigation.
- `Erfolgreich (Established)`: Anzeige des erkannten Gerätetyps sowie Möglichkeit zur manuellen bzw. vorgeschlagenen Testauswahl.

Die Zustände werden reaktiv über Angular Signals aus dem BaseData-Service bezogen. Zusätzlich wird hier erstmals der ListHistory-Service verwendet, um empfohlene Tests anhand des vom Backend gelieferten Testnamens zu identifizieren.

Root-Screen-Komponente

Die Root-Screen-Komponente ermöglicht die manuelle Auswahl von Tests durch den Benutzer. Hierfür wird die Tree-Komponente von PrimeNG (`p-tree`) im Checkbox-Modus verwendet. Der dargestellte Baum wird vom ListHistory-Service bereitgestellt. Da die PrimeNG-Komponente redundante Auswahlstrukturen liefert, übernimmt der Service die Bereinigung sowie die interne Repräsentation der ausgewählten Tests. Die Benutzerinteraktion wird durch die Device-Toolbar ergänzt, welche die Steuerung der Testausführung ermöglicht.

Device-Toolbar-Komponente

Die Device-Toolbar stellt eine zentrale Steuerungseinheit dar und wird in mehreren Ansichten wiederverwendet. Ihre Funktionalität umfasst:

- Kontextabhängige Navigation (inkl. Back-Logik und Disconnect)
- Anzeige des aktuellen Gerätestatus
- Darstellung des Fortschritts von Testausführungen
- Steuerung von Testläufen (Run All, Run Selected, Cancel)
- Export von Testergebnissen

Die Komponente greift sowohl auf den BaseData- als auch auf den ListHistory-Service zu und reagiert dynamisch auf den aktuellen Anwendungszustand. In der Abbildung werden die

einzelnen Varianten (Root-Screen(Testauswahl), Testoverview und die Detailseite) von oben nach unten dargestellt.

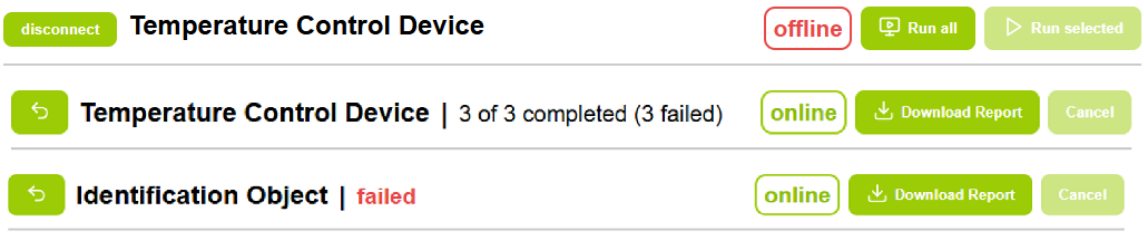


Abbildung 22: Varianten der Device-Toolbar

Tests-Overview-Komponente

Die Tests-Overview-Komponente fungiert als zentrale Ausführungsinstanz für Testläufe. Sie enthält selbst keine komplexe UI, sondern integriert folgende Komponenten:

- Device Toolbar
- Breadcrumb-Navigation
- TestBoxList

Die Ausführungslogik basiert auf dem übergebenen RunType, welcher über den Routing-State definiert wird. Abhängig davon werden unterschiedliche Testmengen ausgeführt:

- Alle Tests
- Ausgewählte Tests
- Empfohlene Tests
- Erneute Anzeige ohne Ausführung

TestBox und TestBoxList

Die TestBoxList-Komponente stellt eine Liste von Tests dar und basiert auf der Wiederverwendung der TestBox-Komponente.

Die TestBox selbst ist eine reine Präsentationskomponente, welche den Status eines einzelnen Tests visuell darstellt. Sie unterscheidet verschiedene Zustände (z. B. Running, Failed, Successful) durch entsprechende Icons und Farben.

Die Navigation innerhalb des Testbaums sowie zur Detailansicht erfolgt über den ListHistory-Service.

Breadcrumb-Komponente

Die Breadcrumb-Komponente visualisiert die aktuelle Navigationsposition innerhalb des Testbaums. Die Datenbasis wird vom ListHistory-Service bereitgestellt. Die Navigation erfolgt über einen History-Stack, welcher es ermöglicht, gezielt zu vorherigen Ebenen zurückzuspringen.

4.2.2 OpcPing-Service

Der OpcPing-Service stellt eine Schnittstelle zur nativen Umgebung dar und ermöglicht das periodische Überprüfen (alle 5 Sekunden) des Verbindungsstatus. Dabei wird ein Observable bereitgestellt, welches Statusänderungen („online“ / „offline“) an die Angular-Anwendung weiterleitet.

4.2.3 BaseData-Service

Der BaseData-Service fungiert als zentrale Datenhaltung für alle grundlegenden Verbindungsinformationen. Dazu zählen die IP-Adresse und der Port, die Zugangsdaten in Form von Benutzername und Passwort, der aktuelle Geräte- und Verbindungsstatus sowie der erkannte Gerätetyp. Darüber hinaus wird auch ein empfohlener Testname verwaltet.

Die Methode `connect()` übernimmt dabei mehrere zentrale Aufgaben innerhalb des Systems. Dazu gehören die Validierung der eingegebenen Daten, die Kommunikation mit dem Backend sowie die Verarbeitung der vom Backend gelieferten Rückgabedaten. Zusätzlich wird durch diese Methode ein Ping-Mechanismus initialisiert, der der kontinuierlichen Überwachung des Verbindungsstatus dient.

4.2.4 ListHistory-Service

Der ListHistory-Service stellt die zentrale Geschäftslogik der Anwendung dar. Er übernimmt die Verwaltung der Testdaten, die Navigation innerhalb des Testbaums sowie die vollständige Steuerung der Testausführung inklusive Ergebnisverarbeitung. Damit fungiert dieser Service gleichzeitig als State-Manager und als Ausführungsinstanz der Tests.

Gespeicherte Datenstrukturen

Der Service verwaltet mehrere interne Datenstrukturen, welche unterschiedliche Aspekte der Anwendung abbilden.

selectedTree: Speichert den vom Benutzer ausgewählten Teilbaum. Dieser Baum enthält ausschließlich jene Tests, die der Benutzer ausgewählt hat und dient als Grundlage für die Testausführung.

savedBackendTree: Speichert den vollständigen Testbaum, der vom Backend geladen wird. Dieser Baum stellt den Ursprungszustand dar und wird für Auswahl, Navigation sowie zur Suche von Tests verwendet.

savedCommonTests: Speichert spezielle Tests, die nicht vom Benutzer auswählbar sind, jedoch bei jeder Testausführung automatisch ausgeführt werden müssen. Diese werden vor jedem Testlauf dem ausgewählten Baum hinzugefügt.

currentTree: Enthält den aktuellen Zustand des Testbaums inklusive aller Testergebnisse. Dieser Baum wird während der Testausführung kontinuierlich aktualisiert.

shownSubtree: Speichert den aktuell im User Interface dargestellten Teilbaum. Dieser basiert auf dem currentTree und berücksichtigt zusätzlich die aktuelle Navigation (History).

historyStack: Speichert den Navigationspfad des Benutzers innerhalb des Testbaums in Form einer Liste von Labels. Dieser wird zur Back-Navigation verwendet.

breadcrumbs: Speichert die Breadcrumb-Daten als vollständige Objekte inklusive Referenzen auf die jeweiligen Knoten. Dadurch ist eine präzise Navigation möglich.

displayedTest: Speichert den aktuell angezeigten Test in der Detailansicht.

UI-Zustände: Zusätzlich verwaltet der Service UI-relevante Zustände wie:

- Aktivierung des *Run Selected/All*-Buttons
- Status, ob aktuell Tests ausgeführt werden (für den Cancle-Button)

setSelectedTree-Methode

Diese Methode verarbeitet die Auswahl aus der PrimeNG-Tree-Komponente und transformiert diese in eine konsistente interne Baumstruktur. Dies ist erforderlich, da PrimeNG eine Liste

aller ausgewählten Knoten zurückgibt. Dabei treten jedoch Redundanzen auf, da untergeordnete Knoten (Child-Nodes) zusätzlich bereits als „children“ in ihren jeweiligen Elternknoten enthalten sind. Dadurch würden sie bei einer einfachen Aufzählung mehrfach berücksichtigt werden, weshalb eine Bereinigung der Struktur notwendig ist.

Listing 19: Verarbeitung und Bereinigung der ausgewählten Tree-Struktur

```
1
2  setSelectedTree(selectedTree: TreeNode<TreeNodeData>[]) {
3    if (selectedTree.length === 0) {
4      this.selectedTree = [];
5      this.enableSelectedRunSubject.next(false);
6      return;
7    }
8
9    this.enableSelectedRunSubject.next(true);
10
11   // clean the selection, because user made a new one
12   this.resetSelectionInTree(this.selectedTree);
13
14   // mark final tree from the selected tree
15   this.selectedTree = this.markTreeFromSelectedTree(
16     selectedTree,
17     this.savedBackendTree.value
18   );
19
20   // cut out only the marked ones
21   this.selectedTree = this.filterSelectedTree(this.selectedTree);
22 }
```

Zunächst wird überprüft, ob überhaupt eine Auswahl existiert. Ist dies nicht der Fall, wird der Run-Button deaktiviert. Anschließend wird die vorherige Auswahl zurückgesetzt, indem alle `selected`-Flags gelöscht werden.

Danach erfolgt die Markierung der ausgewählten Knoten im ursprünglichen Backend-Baum. Abschließend wird der Baum gefiltert, sodass nur die tatsächlich ausgewählten Elemente enthalten sind.

filterSelectedTree-Methode

Diese Methode erzeugt einen minimalen Baum, der ausschließlich ausgewählte Knoten enthält.

Listing 20: Rekursive Filterung der ausgewählten Baumstruktur

```
1 private filterSelectedTree(
2   nodes: TreeNode<TreeNodeData>[]
3 ): TreeNode<TreeNodeData>[] {
```

```
4     const result: TreeNode<TreeNodeData>[] = [];
5
6     for (const node of nodes) {
7       const filteredChildren = node.children
8         ? this.filterSelectedTree(node.children)
9         : [];
10
11      const isSelected = node.data?.selected === true;
12
13      // keep node if:
14      // - it is selected itself
15      // - or at least one child remains
16      if (isSelected || filteredChildren.length > 0) {
17        result.push({
18          ...node,
19          children: filteredChildren.length > 0
20            ? filteredChildren
21            : undefined,
22        });
23      }
24    }
25
26    return result;
27  }
```

Die Methode arbeitet rekursiv. Für jeden Knoten werden zunächst die Kinder gefiltert. Ein Knoten wird genau dann beibehalten, wenn er selbst ausgewählt ist oder mindestens ein Kind ausgewählt wurde.

runTests-Methode

Die Methode `runTests` stellt die zentrale Ausführungslogik dar und wird abhängig vom gewählten RunType aufgerufen.

Zunächst werden die Common Tests zum ausgewählten Baum hinzugefügt:

Listing 21: Ausführungslogik (aufgeteilt in mehrere Codeteile zur besseren Lesbarkeit).

```
1 let tree = commonNode ? [commonNode, ...selectedTests] :
    selectedTests;
```

Anschließend werden alle ausführbaren Tests (Leaf Nodes) extrahiert:

```
1 const allLeafNames = this.extractLeafTestNames(tree);
```

Danach wird die Benutzeroberfläche vorbereitet, indem alle Knoten auf den Status *Pending* gesetzt werden und der aktuelle Baum gespeichert wird:

```
1 tree = this.setAllNodeStatus(tree, Status.Pending);
2 this.currentTree$.next(tree);
3 this.updateUiTreeWithCurrent();
```

Zusätzlich wird der nächste auszuführende Test auf den Status *Running* gesetzt, um eine visuelle Rückmeldung zu ermöglichen.

Die Verarbeitung der Testergebnisse erfolgt eventbasiert über einen Listener:

```
(window as any).testHandler.setTestResultListener(...)
```

Für jedes Testergebnis wird ein neuer Knoten erzeugt:

```
const updatedLeafNode = {
  label: result.test,
  data: {
    description: result.fullName,
    status: result.status,
    response: result.message,
    failureMessage: result.failureMessage
  },
  children: [],
};
```

Dieser wird anschließend in den bestehenden Baum eingefügt:

```
this.recursiveUpdateTree(tree, testPathParts, updatedLeafNode);
```

Danach wird der Status der übergeordneten Knoten neu berechnet:

```
this.updateTreeWithAggregatedStatus(tree);
```

Zum Abschluss wird der aktualisierte Baum erneut gespeichert und im UI dargestellt.

recursiveUpdateTree-Methode

Die Methode dient der rekursiven Aktualisierung eines spezifischen Knotens innerhalb der Baumstruktur anhand eines übergebenen Pfads. Dabei wird der Baum schrittweise entlang der einzelnen Pfadsegmente durchsucht, bis der entsprechende Blattknoten erreicht ist. Anschließend werden dessen Daten gezielt aktualisiert, ohne die bestehende Struktur zu verändern. Durch dieses Vorgehen wird sichergestellt, dass eingehende Testergebnisse präzise in die bestehende Baumstruktur integriert werden können (siehe Listing 22).

Listing 22: Rekursive Aktualisierung eines Knotens anhand eines gegebenen Pfads

```
  ...
  const currentLabel = parts[0];

  const node = nodes.find(n => n.label === currentLabel);

  if (!node) {
    console.warn('Node not found:', currentLabel);
```

```
8       return false;
9     }
10
11    if (parts.length === 1) {
12      // leaf node reached -> update data
13      if (!updatedLeafNode.data) {
14        console.warn('updatedLeafNode.data is undefined:',
            updatedLeafNode);
15        return false;
16      }
17
18      const updatedData = updatedLeafNode.data;
19      ...
```

updateTreeWithAggregatedStatus-Methode

Die Methode dient der rekursiven Aktualisierung des Status von Elternknoten auf Basis der Statuswerte ihrer untergeordneten Knoten. Zunächst werden alle Kindknoten aktualisiert, bevor anschließend ein aggregierter Status für den jeweiligen Elternknoten bestimmt wird. Dadurch wird sichergestellt, dass der Zustand der gesamten Baumstruktur konsistent und nachvollziehbar dargestellt wird.

updateUiTreeWithCurrent-Methode

Die Methode dient der Aktualisierung der in der Benutzeroberfläche dargestellten Baumstruktur basierend auf dem aktuellen Zustand des Baumes sowie dem Navigationsverlauf des Benutzers. Hierzu wird entlang des gespeicherten Pfades iteriert, um den aktuell relevanten Teilbaum zu bestimmen. Anschließend wird entweder die entsprechende Unterstruktur oder, falls kein spezifischer Knoten ausgewählt ist, der gesamte Baum zur Anzeige gebracht. Dadurch wird eine dynamische und kontextabhängige Navigation innerhalb der Baumstruktur ermöglicht (siehe Listing 23).

Listing 23: Aktualisierung der angezeigten Baumstruktur basierend auf Navigationsverlauf

```
1  // uses the current tree and navigation history to update the UI
       tree
2  public updateUiTreeWithCurrent() {
3    let currentNodes = this.currentTree$.value;
4    let currentNode: TreeNode<TreeNodeData> | undefined;
5
6    for (const segment of this.historyStack.value) {
7      currentNode = currentNodes.find(n => n.label === segment);
8
9      if (!currentNode) return null; // invalid path
10     if (!currentNode.children) break;
11
12     currentNodes = currentNode.children;
```

```
13    }
14
15    if (currentNode) {
16      this.shownSubtree.next(currentNode.children!);
17    } else {
18      this.shownSubtree.next(this.currentTree$.value);
19    }
20
21    return;
22  }
```

Fazit

Der ListHistory-Service bildet das zentrale Element der Anwendung. Er kombiniert rekursive Datenverarbeitung, Zustandsmanagement und eventbasierte Kommunikation mit dem Backend. Dadurch ermöglicht er eine konsistente und nachvollziehbare Darstellung sowie Steuerung der Testausführung.

4.2.5 Verwendete Design Patterns im Frontend

Im Rahmen der Implementierung wurden mehrere bewährte Design Patterns eingesetzt, um eine klare Struktur, Wartbarkeit und Erweiterbarkeit der Anwendung zu gewährleisten.

Service Pattern

Das Service Pattern stellt die Grundlage der Softwarearchitektur dar. In Angular werden Services verwendet, um Geschäftslogik von der Darstellung zu trennen.

In der vorliegenden Anwendung übernehmen insbesondere die Services **BaseData**, **ListHistory** und **OpcPing** zentrale Aufgaben. Sie kapseln Logik, verwalten Zustände und stellen diese den Komponenten zur Verfügung [36].

Vorteile

- Klare Trennung zwischen Logik und Benutzeroberfläche
- Wiederverwendbarkeit von Funktionen
- Zentrale Datenhaltung

Observer Pattern

Das Observer Pattern wird durch die Verwendung von **BehaviorSubject** und **Observable** realisiert. Komponenten können sich auf Datenströme subscriben und reagieren automatisch auf Änderungen [36].

Listing 24: Observer Pattern Beispiel

```
1 private currentTree$ = new BehaviorSubject([]);
```

Vorteile

- Reaktive Datenverarbeitung
- Entkopplung zwischen Datenquelle und Konsumenten
- Automatische Aktualisierung der Benutzeroberfläche

State Management Pattern

Der ListHistory-Service fungiert als zentraler State Manager der Anwendung. Er verwaltet sämtliche relevanten Zustände, darunter ([36]):

- den aktuellen Testbaum
- die Navigation (History und Breadcrumbs)
- den Status der Testausführung

Vorteile

- Konsistenter Zustand über alle Komponenten hinweg
- Zentrale Steuerung komplexer Abläufe
- Vereinfachte Synchronisation zwischen UI und Daten

Facade Pattern

Der ListHistory-Service übernimmt zusätzlich die Rolle einer Fassade. Er stellt eine vereinfachte Schnittstelle für komplexe interne Logik bereit, sodass Komponenten nicht direkt mit der Baumverarbeitung oder Testausführung interagieren müssen [36].

Vorteile

- Reduzierung der Komplexität in den Komponenten

- Klare und einfache Schnittstellen
- Bessere Wartbarkeit

Separation of Concerns

Die gesamte Softwarearchitektur folgt dem Prinzip der Trennung von Verantwortlichkeiten. Komponenten sind ausschließlich für die Darstellung zuständig, während Services die Geschäftslogik übernehmen.

Vorteile

- Verbesserte Struktur und Übersichtlichkeit
- Einfachere Erweiterbarkeit
- Erhöhte Testbarkeit der einzelnen Bestandteile

Zusammenfassung

Durch den gezielten Einsatz dieser Design Patterns entsteht eine modulare und skalierbare Softwarearchitektur. Insbesondere die Kombination aus Service Pattern, Observer Pattern und zentralem State Management ermöglicht eine saubere Trennung von Logik und Darstellung sowie eine effiziente Verarbeitung komplexer Datenstrukturen.

4.2.6 Vergleich: Baumstruktur und Rekursion vs. listenbasierter Ansatz

Im Rahmen der Implementierung stellte sich die grundlegende Frage, wie die Testdaten im Frontend strukturiert und verarbeitet werden sollen. Zwei mögliche Ansätze wurden dabei betrachtet: die Verwendung einer rekursiven Baumstruktur sowie ein listenbasierter Ansatz.

Baumstruktur und Rekursion

Die implementierte Lösung basiert auf einer Baumstruktur, welche die hierarchische Organisation der Testpakete direkt abbildet. Jedes Testpaket kann wiederum weitere Testpakete enthalten, wodurch sich eine natürliche rekursive Struktur ergibt.

Diese Struktur wird im Frontend durch verschachtelte **TreeNode**-Objekte dargestellt, wobei jedes Element sowohl Daten als auch eine Liste von Kindknoten enthält.

Vorteile

- Direkte Abbildung der realen Struktur der Testdaten
- Intuitive Navigation durch hierarchische Beziehungen
- Möglichkeit zur rekursiven Verarbeitung (z. B. Traversierung und Updates)
- Einfache Aggregation von Zuständen, beispielsweise zur Berechnung eines Elternstatus basierend auf den Statuswerten der Kinder

Nachteile

- Hoher Implementierungsaufwand durch rekursive Algorithmen
- Erhöhte Komplexität bei der Fehlersuche und beim Debugging
- Notwendigkeit mehrerer Hilfsfunktionen wie Traversierung, Suche oder Update-Mechanismen

Listenbasierter Ansatz

Eine alternative Herangehensweise besteht in der Verwendung einer flachen Liste (Array), in der alle Tests unabhängig voneinander gespeichert werden. Die hierarchische Struktur würde in diesem Fall erst zur Laufzeit für die Darstellung erzeugt werden.

Ein solcher Ansatz könnte beispielsweise über eine Struktur realisiert werden, bei der jeder Eintrag eine Referenz auf seinen Elternknoten enthält.

Vorteile

- Einfachere Datenmanipulation durch lineare Struktur
- Keine rekursiven Algorithmen erforderlich
- Geringere Komplexität bei grundlegenden Operationen wie Filtern oder Suchen

Nachteile

- Zusätzliche Transformationslogik notwendig, um die Daten als Baum darzustellen
- Komplexere Berechnung von Eltern-Kind-Beziehungen
- Schwierigeres Mapping zwischen UI-Darstellung und Datenstruktur
- Erschwerte Aggregation von Statuswerten entlang der Hierarchie

Bewertung und Entscheidung

Aufgrund der stark hierarchischen Natur der Testdaten wurde die Baumstruktur als geeigneter Ansatz gewählt. Obwohl die Verwendung von Rekursion die Implementierung komplexer macht, ermöglicht sie eine direkte und verständliche Abbildung der Problemstellung.

Insbesondere die Möglichkeit, Statusinformationen entlang der Baumstruktur zu propagieren sowie gezielt Teilbäume zu selektieren, stellt einen entscheidenden Vorteil gegenüber einem listenbasierten Ansatz dar.

Zusammenfassend lässt sich festhalten, dass die Baumstruktur trotz höherer Implementierungskomplexität die fachlich sinnvollere Lösung darstellt.

4.2.7 Fazit und Gesamtbewertung

Im Rahmen der Implementierung wurde eine modulare und klar strukturierte Frontend-Architektur auf Basis von Angular realisiert. Durch die konsequente Trennung von Präsentationslogik (Komponenten) und Geschäftslogik (Services) konnte eine hohe Wartbarkeit und Erweiterbarkeit der Anwendung erreicht werden.

Ein zentrales Element stellt der ListHistory-Service dar, welcher als State-Manager und Ausführungsinstanz fungiert. Durch die Bündelung der komplexen Logik zur Baumverarbeitung, Navigation und Testausführung in einem Service wird die Komplexität aus den UI-Komponenten herausgelöst und an einer zentralen Stelle kontrolliert.

Die Entscheidung für eine rekursive Baumstruktur erwies sich als fachlich sinnvoll, da sie die hierarchische Natur der Testdaten direkt abbildet. Trotz des erhöhten Implementierungsaufwands durch rekursive Algorithmen ermöglicht dieser Ansatz eine intuitive Navigation sowie eine konsistente Aggregation von Statusinformationen.

Zusätzlich wurde durch den Einsatz von Angular Signals und dem Observer Pattern eine reaktive Datenverarbeitung umgesetzt. Dadurch reagieren alle Komponenten automatisch auf Änderungen im Zustand, was insbesondere bei der dynamischen Testausführung von großer Bedeutung ist.

Eine besondere Herausforderung stellte die asynchrone Kommunikation mit dem Backend dar. Da dieses keine garantierte Reihenfolge der Testergebnisse liefert, wurde die Steuerung der Testausführung bewusst in das Frontend verlagert. Dies erhöht zwar die Komplexität der Implementierung, ermöglicht jedoch eine präzise und nachvollziehbare Darstellung des aktuellen Testfortschritts.

Durch die Kombination aus bewährten Design Patterns, einer klaren Softwarearchitektur und einer durchdachten Datenstruktur konnte eine robuste und flexible Anwendung entwickelt werden. Die Implementierung bildet somit eine solide Grundlage für zukünftige Erweiterungen sowie die Integration zusätzlicher Funktionalitäten.

4.3 Node OPC-UA Backend

Im folgenden Kapitel wird die Funktionsweise sowie die Implementierung der Node OPC-UA Clients beschrieben.

4.3.1 Zielsetzung und Planung

Ziel dieses Systems ist einen OPC-UA Client zu simulieren, was dann von der Testumgebung verwendet werden kann. Grundsätzlich hat diese allgemeine Methoden mit denen man in den ÖPC-UA-Baum"navigieren kann. Sie liest Werte aus und liefert sie an die Testumgebung.

4.3.2 Funktionen

initClient

Dabei wird grundsätzlich eine Session über die IP-Adresse und den Port des Servers aufgebaut. Ist die Verbindung erfolgreich, wird eine neue Session zurückgegeben, über die anschließend mit dem Server interagiert werden kann.

Listing 25: Funktion zur Initialisierung der Node OPC-UA Clients

```
1  /**
2   * initializes the OPC-UA Client and it's session with Ip-address
       and Port
3   * @param ip as string
4   * @param port as number
5   * @returns the session that is created
6   */
7  export async function initClient(ip: string, port: number) {
8    client = OPCUAClient.create({
9      endpointMustExist: false,
10   });
11
12   await client.connect(`opc.tcp://${ip}:${port}`);
13   session = await client.createSession();
14
15   return session;
16 }
```

disconnect

Diese Funktion hat die Aufgabe, die bestehende Session beziehungsweise die Verbindung zum Server ordnungsgemäß und sicher zu schließen. Dabei wird sichergestellt, dass alle verwendeten Ressourcen des OPC-UA Clients korrekt freigegeben werden und keine offenen Verbindungen bestehen bleiben.

Listing 26: Funktion zum Entbinden

```
/**
 * Closes the local session and connection of the OPC-UA client
 */
export async function disconnect() {
  if (client && session) {
    await session.close();
    await client.disconnect();
  }
}
```

getRootStructure

Diese Funktion iteriert über die ersten zwei Ebenen des OPC-UA-Serverbaums und gibt diese zurück. Dabei wird überprüft, ob die Ebenen *Objects* und *Types* vorhanden sind. Falls dies der Fall ist, werden auch deren untergeordnete Knoten in die Rückgabe aufgenommen.

Der Rückgabewert ist daher eine Map, die für jeden Schlüssel ein Array von Strings enthält.

Listing 27: getRootStructure Funktion

```
/**
 * Returns the structure directly (depth of 2) under root of the
    OPC-UA Server
 * @returns the structure as a Map
 * @example { 'Objects': ['DeviceSet', 'NetworkSet'], 'Types': ...
    }
 */
export async function getRootStructure(): Promise<{ [key: string]:
   string[] }> {
  let browseResult: BrowseResult = await
     browsePathToNodeId('/Objects');

  let children: { [key: string]: string[] } = {};

  if (browseResult === null) {
    return null
  } else {
    if (!children["Objects"]) {
      children["Objects"] = [];
    }
    for (const ref of browseResult.references) {
      children["Objects"].push(ref.browseName.name);
    }
```

```
20    }
21
22    browseResult = await browsePathToNodeId('/Types');
23
24    if (browseResult === null) {
25      return null
26    } else {
27      if (!children["Types"]) {
28        children["Types"] = [];
29      }
30      for (const ref of browseResult.references) {
31        children["Types"].push(ref.browseName.name);
32      }
33    }
34
35    return children;
36  }
```

getNodeId

Bei dieser Funktion handelt es sich um eine Suchfunktion innerhalb des OPC-UA-Baums. Sie zählt zu den zentralen Funktionen, da es mithilfe der Namen der Knoten sowie deren übergeordneter Elemente möglich ist, die entsprechende NodeId zu ermitteln.

Dadurch kann schrittweise tiefer in die Struktur des OPC-UA-Baums navigiert werden.

Listing 28: getNodeId Funktion

```
1  /**
2   * Gets the nodeId of a child-node
3   * @param parent the parent of the node that is to be found
4   * @param node child-node
5   * @returns string which includes the nodeId
6   */
7  async function getNodeId(parent: string, node: string):
       Promise<NodeId> {
8    const browseResult: BrowseResult = await
         browsePathToNodeId(parent);
9
10   const ref = browseResult.references.find(r => r.browseName.name
       === node);
11   if (ref) {
12     return ref.nodeId;
13   }
14   return null
15 }
```

findProtocolName

Diese Funktion wird direkt nach dem Verbindungsaufbau einmalig aufgerufen. Sie stellt ebenfalls eine Verbindung zum Client her, gibt jedoch keine Session zurück, sondern ermittelt den Namen des auf dem Server verwendeten Standards.

Dabei wird zunächst eine Verbindung zum Server aufgebaut. Kann diese nicht innerhalb von fünf Sekunden erfolgreich hergestellt werden, wird eine Exception ausgelöst.

Bei erfolgreicher Verbindung wird das *NamespaceArray* des Servers ausgelesen. In diesem Array ist unter anderem der verwendete OPC-UA- beziehungsweise Euromap-Standard des Geräts enthalten. Standardmäßig befindet sich dieser an der vorletzten Position. Falls kein entsprechender Eintrag gefunden wird, wird der Wert *"not found"* zurückgegeben.

Listing 29: findProtocolName Funktion

```
/**
 * Finds the opc-ua standard and fullname of a server and returns it
 * @returns opc-ua-standard/device-fullname as string
 */
export async function findProtocolName(): Promise<string> {
  if (!session && !client) {

    try {
      session = await withTimeout(initClient("192.168.110.250", 4840), 5000);
    } catch (e) {
      if (client) {
        try {
          await client.disconnect();
        } catch (_) { }
      }
      throw e;
    }
  }

  const uris: string[] = await session.readNamespaceArray();

  let splitted: string[];
  let foundProtocol: string;

  for (let i = 0; i < uris.length; i++) {
    splitted = uris[i].split("/");

    let possibleProtocol: string = splitted[splitted.length - 2];

    for (let j = 0; j < supportedProtocols.length; j++) {
      if (supportedProtocols[j] === possibleProtocol &&
          possibleProtocol !== undefined) {
        foundProtocol = possibleProtocol;
      }
```

```
34        }
35      }
36      if (!foundProtocol) {
37        foundProtocol = "not found"
38      }
39      return foundProtocol + "/" + (await getFullDeviceName());
40    }
```

getFullDeviceName

Diese Funktion gibt den vollständigen Gerätenamen des Servers zurück. Hierfür wird die Funktion *getIdentityAttribute()* aufgerufen, um den entsprechenden Identity-Knoten des Servers zu ermitteln und dessen Wert auszulesen.

Falls kein Wert gefunden werden kann, liefert die Funktion den String *"not found"* zurück.

Listing 30: getFullDeviceName Funktion

```
1  /**
2   * extracts the fullname of an opc-ua server
3   * @returns the fullname of a opc-ua server as string
4   */
5  async function getFullDeviceName(): Promise<string> {
6    return await getIdentityAttribute("DeviceClass")
7  }
```

browsePathToNodeId

Diese Funktion stellt eine der zentralen Komponenten der OPC-UA-Clients dar. Mithilfe dieser Funktion können Entwickler von einem beliebigen Startknoten aus zu einem beliebigen Pfad innerhalb des OPC-UA-Baums navigieren. Dabei ist zu beachten, dass die Navigation ausschließlich in absteigender Richtung erfolgt, also nur entlang der untergeordneten Knoten. Aus diesem Grund kann der Funktion jederzeit ein beliebiger Startknoten übergeben werden.

Die Funktionsweise basiert darauf, dass alle Bestandteile des angegebenen Pfades iterativ durchlaufen werden. Für jedes Pfadsegment wird überprüft, ob der entsprechende untergeordnete Knoten im Server existiert. Ist dies der Fall, wird die Navigation zum nächsten Pfadsegment fortgesetzt. Andernfalls wird eine Fehlermeldung ausgelöst.

Listing 31: Browse Funktion

```
/**
 * function that browses to the given path from the given startNode and returns it's BrowseResult (all children)
 * @param path a string path which beginns with / @example '/Objects/DeviceSet'
 * @param startNode optional - the node where the search should beginn. Ifnothing given: search starts from root
 * @returns The children of the last part of the path via BrowseResult
 */
export async function browsePathToNodeId(
  path: string,
  startNode: NodeIdLike = resolveNodeId("RootFolder")
): Promise<BrowseResult | null> {
  const parts = path.split("/").filter(Boolean);
  let currentNode: NodeIdLike = startNode;

  for (const part of parts) {
    const browseResult: BrowseResult = await session.browse({
      nodeId: currentNode,
      browseDirection: BrowseDirection.Forward,
      includeSubtypes: true,
      nodeClassMask: NodeClassMask.Object | NodeClassMask.Variable,
      resultMask: ResultMask.BrowseName | ResultMask.DisplayName,
    });

    const next = browseResult.references?.find(
      (ref) => ref.browseName.name === part
    );

    if (!next) {
      console.warn(`Could not find '${part}' under node ${currentNode}`);
      return null;
    }

    currentNode = next.nodeId;
  }

  const finalResult = await session.browse({
    nodeId: currentNode,
    browseDirection: BrowseDirection.Forward,
    includeSubtypes: true,
    nodeClassMask: NodeClassMask.Object | NodeClassMask.Variable,
    resultMask: ResultMask.BrowseName | ResultMask.DisplayName | ResultMask.NodeClass,
  });

  return finalResult;
}
```

findHasComponentChildByBrowseName

Diese Funktion hat die einfache Aufgabe zu überprüfen, ob ein übergeordneter Knoten den entsprechenden untergeordneten Knoten enthält. Dabei wird innerhalb des OPC-UA-Baums nach einem Kindknoten mit einem bestimmten Namen gesucht.

Wird ein entsprechender Knoten gefunden, so wird dessen *NodeId* zurückgegeben. Falls kein passender Knoten existiert, liefert die Funktion den Wert *null* zurück.

Diese Funktion stellt somit eine grundlegende Hilfsfunktion dar, die insbesondere bei der Navigation innerhalb des OPC-UA-Baums verwendet wird.

Listing 32: HasChild Funktion

```
/**
 * finds the child of the current component and returns it's NodeId
 * @param session current session
 * @param parentNodeId parent
 * @param browseName the name of the child to be found
 * @param type optional - what kind of child it should be looked for. if nothing given: 'HasComponent'
 * @returns NodeId or null
 */
export async function findHasComponentChildByBrowseName(
  session: ClientSession,
  parentNodeId: NodeId,
  browseName: string,
  isProperty: boolean = false
): Promise<NodeId | null> {

  const browseResult = await session.browse({
    nodeId: parentNodeId,
    browseDirection: BrowseDirection.Forward,
    nodeClassMask: 0,
    resultMask: 63,
  });

  const references: ReferenceDescription[] = browseResult.references;

  return references.find((reference) => {
    if (isProperty) {
      return reference.browseName.name === browseName
    } else {
      return (reference.browseName.name === browseName && reference.browseName.name !== "PropertyType")
    }
  })?.nodeId;
}
```

4.4 Testumgebung

Im folgenden Kapitel wird die Entwicklung der Testumgebung beschrieben.

4.4.1 Umstieg von Jasmine auf Vitest

Wie bereits erwähnt, wurde zu Beginn des Projekts das Testing-Framework Jasmine eingesetzt. Dieses erwies sich jedoch als deutlich herausfordernder als ursprünglich angenommen. Bereits nach der initialen Projektaufsetzung zeigte sich, dass Jasmine für die Anforderungen des Systems nur eingeschränkt geeignet ist, da gestartete Tests ihre Ergebnisse erst gesammelt am Ende eines Testlaufs zurückliefern.

Aus diesem Grund versuchte das Projektteam, die ausgewählten Tests einzeln auszuführen, um eine dynamische Auswertung zu ermöglichen. Dieser Ansatz stellte sich jedoch insbesondere bei mehreren, komplexeren und zeitintensiven Testfällen als zu langsam heraus.

Ein weiterer Ansatz bestand darin, Jasmine über einen Konsolenbefehl im sogenannten sequenziellen Modus auszuführen. Zwar funktionierte dieser Ansatz auf Konsolenebene zufriedenstellend, jedoch bestand keine geeignete Möglichkeit, auf die einzelnen Testergebnisse programmgesteuert zuzugreifen.

Nach einer etwa zweiwöchigen Evaluationsphase einigte sich das Projektteam gemeinsam mit dem Auftraggeber darauf, ein alternatives Testing-Framework zu suchen. In weiterer Folge wurde Jasmine durch Vitest ersetzt. Ein wesentlicher Vorteil von Vitest besteht neben der Unterstützung eines funktionierenden sequenziellen Modus auch in der einfachen Integration, da es sich mit dem Befehl *npm install -D vitest* schnell in das bestehende Projekt einbinden lässt.

4.4.2 Umsetzung

OPC-UA-Helper

Die einzelnen Testfälle sind bei den meisten OPC-UA- beziehungsweise Euromap-Standards sehr ähnlich. Beispielsweise erfolgt das Auslesen des Namespace-Arrays bei den Standards Euromap 81.1, 82.2 und 82.3 auf identische Weise, wobei sich lediglich die erwarteten Werte innerhalb des Arrays unterscheiden.

Aus diesem Grund existiert diese Datei. In ihr werden Funktionen definiert, die wiederholt von den implementierten Vitest-Tests verwendet werden können.

Tests-Lifecycle

In dieser Klasse, die das Reporter-Interface von Vitest implementiert, werden die benötigten Lifecycle-Funktionen definiert. **onTestCaseResult()**
Diese Lifecycle-Funktion wird aufgerufen, sobald ein Test ein Ergebnis liefert. Dabei wird zunächst überprüft, ob das Ergebnis gültig ist, also den Status *'passed'* oder *'failed'* aufweist.

Ist dies der Fall, wird das Ergebnis in das vom Frontend benötigte Format überführt. Anschließend wird es mithilfe der eindeutigen *RESULT_LISTENER_ID* über den entsprechenden ContextBridge-Kanal an das Frontend zurückgegeben.

Listing 33: onTestCaseResult

```
async onTestCaseResult(testCase: TestCase) {
        if (testCase.result().state === 'passed' ||
           testCase.result().state === 'failed') {
            const result: Result = formatTestCaseResult(testCase)

            this.event.reply(RESULT_LISTENER_ID, { result })
        }
};
```

onTestCaseReady()
Diese Funktion wird vor dem Start jedes Testfalls ausgeführt. Dabei wird der vollständige Pfad der Tests an das Frontend übermittelt.

Zur Sicherstellung der Korrektheit dieses Pfades wird zusätzlich die Funktion *testIsIncluded()* aufgerufen, um zu überprüfen, ob der entsprechende Testfall tatsächlich vom Frontend angefordert wurde.

Listing 34: onTestCaseReady

```
async onTestCaseReady(testCase: TestCase) {
        const path: string = formatTestPath("plain > " +
           testCase.fullName)[0];

        if(this.testIsIncluded(path)){
            this.event.reply(NEXT_LISTENER_ID, { path })
        }
}
```

onFinished()
Diese Funktion wird ausgeführt, sobald alle Tests abgeschlossen sind. In diesem Fall wird ein leeres Testergebnis an das Frontend zurückgesendet, das den Namen *"finished"* trägt.

Listing 35: onFinished

```
async onTestCaseResult(testCase: TestCase) {
        if (testCase.result().state === 'passed' ||
           testCase.result().state === 'failed') {
            const result: Result = formatTestCaseResult(testCase)

            this.event.reply(RESULT_LISTENER_ID, { result })
        }
};
```

TestRunner

Diese Datei ist für die Verwaltung sämtlicher Tests zuständig. Sie startet die Testläufe, liefert die vorhandenen Tests an das Frontend zurück und formatiert alle Daten in die Baumstruktur, auf der das Frontend aufbaut.

runVitest

Dies ist eine der zentralen Funktionen der Anwendung, da sie einen Testrun initiiert. Vorab muss eine Instanz des zuvor beschriebenen *LiveTestReporter* erstellt werden.

Anschließend werden die übergebenen Testnamen in das von Vitest erwartete Format überführt, sodass sie in der von Vitest bereitgestellten Funktion *startVitest* verwendet werden können. Diese Funktion benötigt darüber hinaus den Pfad zur Vitest-Konfigurationsdatei, das Verzeichnis, in dem sich die Testfälle befinden, sowie den zuvor erstellten Reporter.

Darüber hinaus hat das Projektteam einen optionalen HTML-Reporter implementiert, der eine unmittelbar im Web darstellbare Übersicht des Testruns liefert. Hierfür muss ein Ausgabepfad angegeben werden. Der Parameter *watch*, der auf *false* gesetzt ist, steuert, ob die Informationen zusätzlich in der Konsole ausgegeben werden sollen.

Listing 36: runVitest

```
/**
 * runs a list of tests on the OPC-UA client with the given ip and port
 * @param event Electron.IpcMainEvent | is needed to return the value through the right contextBridge
 * @param testNames string[] | the names of all the tests that need to be run
 * @param ip string | ip address of the OPC-UA client
 * @param port number | port of the OPC-UA client
 */
export async function runVitest(event: Electron.IpcMainEvent, testNames: string[], ip: string, port: number) {
  process.env.IP_ADDRESS = ip;
  process.env.PORT = port + "";

  const reporter = new LiveTestReporter(event, testNames);
  const testNamePattern = testNames.map(name => `($${name}$)`).join('|');

  const vitest = await startVitest('test', [
    '--config', configPath,
    '--root', path.resolve(projectRoot, 'libs/electron-core/vitest/packages/test-runner'),
    '--run',
  ], {
    testNamePattern: testNamePattern,

    reporters: [reporter, 'html'],
    outputFile: path.resolve(projectRoot, 'vitest-output/vitest-results.html'),

    watch: false,
  });
}
```

getTestCases

Diese Funktion hat die Aufgabe, alle implementierten Testfälle zu suchen und an das Frontend zurückzugeben. Hierfür wird die Funktion *execa()* verwendet, mit der Entwickler beliebige Konsolenbefehle oder Dateien ausführen können. In diesem Fall wird der Befehl *'vitest list'* übergeben.

Für die Ausführung werden Npx, der Pfad zur Konfigurationsdatei sowie das Verzeichnis, in dem sich die Testfälle befinden, benötigt.

Die Funktion *execa()* liefert als Rückgabewert einen String, aus dem das Projektteam anschließend die korrekte Baumstruktur der Tests aufbaut. Dies erfolgt mittels der Funktion *buildTreeFromPaths*. Bevor diese aufgerufen werden kann, muss zuvor *formatTestPath* ausgeführt werden. Diese Funktion passt das Format der Testpfade an, sodass es dem vom Frontend

erwarteten Format *'TestPackage/Übertest/Test1'* entspricht, da Vitest die Pfade ursprünglich im Format *'TestPackage > Übertest > Test1'* zurückliefert.

Listing 37: getTestCases

```
/**
 * runs a cmd command to return all found tests
 * @returns TreeNode<TreeNodeData>[]
 */
export async function getTestCases():
   Promise<TreeNode<TreeNodeData>[]> {
  const { stdout } = await execa('npx', [
    'vitest',
    'list',

    '--config', configPath,
    '--root', path.resolve(projectRoot,
       'libs/electron-core/vitest/packages/test-runner'),
  ])

  let testListString: string = stdout
  let testList: string[] = formatTestPath(testListString);
  const testTree = buildTreeFromPaths(testList);
  return testTree;
}
```

buildTreeFromPaths

Aufgrund des verzögerten Umstiegs von Jasmine auf Vitest war diese Funktion bereits im Frontend implementiert. Sie wurde jedoch auch ins Backend integriert, da es ein zentrales Ziel des Projektteams ist, das Frontend möglichst wenig mit logischer Funktionalität zu belasten.

Zu Beginn der Funktion werden Leerzeichen sowie abschließende Schrägstriche ('/') von den Testpfaden entfernt. Grundsätzlich werden alle Testpfade iterativ durchlaufen. Für jede Ebene wird überprüft, ob der entsprechende Knoten bereits im Baum existiert. Falls dies nicht der Fall ist, wird ein neuer Knoten daraus erstellt.

Abschließend wird geprüft, ob es sich bei dem aktuellen Pfadsegment um das letzte Segment des Pfades handelt. Wenn nicht, wird die Iteration auf die nachfolgenden Knoten fortgesetzt.

Listing 38: buildTreeFromPaths

```
// Helper function: Converts a list of path strings like
// ["Math/test1", "Math/test2", "OPC-UA/Connection/connection"]
// into a nested TreeNode[] structure
function buildTreeFromPaths(paths: string[]):
    TreeNode<TreeNodeData>[] {
  const rootNodes: TreeNode<TreeNodeData>[] = [];

  for (const fullPath of paths) {

    // Remove trailing slashes and trim whitespace (e.g. "Math /
       test1 /" -> "Math/test1")
    const cleanPath = fullPath.trim().replace(/\/+$/, '');
    if (!cleanPath) continue;

    const parts = cleanPath.split('/');

    // Start at the root level of the tree
    let currentLevel = rootNodes;

    for (let i = 0; i < parts.length; i++) {
      const part = parts[i];

      // Try to find an existing node with the same label at the
         current level
      let existingNode = currentLevel.find(n => n.label === part);

      // If it doesn't exist, create and add a new node
      if (!existingNode) {
        existingNode = {
          label: part,
          data: {
            description: '',
            status: Status.Pending,
            response: '',
            failureMessage: '',
          },
          children: [],
        };
        currentLevel.push(existingNode);
      }

      // If this is not the last part, go deeper into the tree
      if (i < parts.length - 1) {
        if (!existingNode.children) {
          existingNode.children = [];
        }
        currentLevel = existingNode.children;
      }
    }
  }
  return rootNodes;
}
```

4.4.3 Tests schreiben

Für die Testfälle ist ein korrekter Aufbau beziehungsweise eine klare Struktur essenziell, damit mithilfe von *vitest list* die gewünschte „Ordnerstruktur“ der Tests erzeugt werden kann. Diese Struktur wird folgendermaßen umgesetzt:

Im obersten *describe*-Block wird der Name des jeweiligen Standards angegeben, für den die Tests gelten. Eine Ausnahme bilden die sogenannten allgemeinen Tests, welche Fälle überprüfen, die für alle Geräte identisch sind.

Auf der nächsten Ebene gibt es zwei Möglichkeiten: Entweder wird ein weiterer *describe*-Block verwendet, um ein neues „Testpaket“ zu definieren, oder es wird direkt ein *it*-Block geschrieben. Letzterer stellt die unterste Ebene eines Testzweigs dar.

Innerhalb eines *it*-Blocks wird der eigentliche Testfall implementiert und mit einem eindeutigen Namen versehen. Dieser Name muss zusätzlich in einer Shared-Datei eingetragen werden, damit das Frontend jederzeit auf die exakten Testbezeichnungen zugreifen kann.

Grundsätzlich muss jeder erwartete Wert mithilfe der *expect*-Funktion überprüft werden, damit Vitest ein entsprechendes Testergebnis erzeugen kann. Diese Funktion liefert eine Assertion zurück, an die weitere Überprüfungsfunktionen wie *.not()* oder *.toBe()* angehängt werden können.

Listing 39: Beispiel Test

```
describe("Euromap 82.1", () => {
    it(testNames.NAMESPACES.name + " TCD", async () => {
        const arrayFromServer: string[] = await
           getNameSpaceArray();
        expect(arrayFromServer.length).not.toBe(0)

        const exists = arrayFromServer.some(elem => elem ===
           'http://opcfoundation.org/UA/PlasticsRubber/TCD/' ||
           elem === 'http://www.euromap.org/euromap82_1/')

        expect(exists).toBe(true);
    })
    it(testNames.INTERFACETYPE.name + " TCD", async () => {
        await expectInterfaceType("1012");
    });

    describe("Identification TCD Test Pack", async () => {
        it(testNames.DEVICECLASS.name + " TCD", async () => {
            await expectIdentityAttribute("DeviceClass",
               "Temperature Control Device");
        })
    })
})
```

5 Ergebnis

In diesem Kapitel werden die erzielten Ergebnisse der Diplomarbeit dargestellt. Dabei wird zunächst der finale Stand des Backends beschrieben, gefolgt von der Präsentation des Frontend-Endergebnisses, um einen Überblick über die umgesetzten Funktionen sowie die praktische Umsetzung der entwickelten Anwendung zu geben.

5.1 Backend Endergebnis

Im folgenden Abschnitt wird der Endstand des Backends ausführlich beschrieben. Dabei wird insbesondere erläutert, welche Daten die ContextBridge in Abhängigkeit der übergebenen Parameter zurückliefert.

5.1.1 ContextBridge: testHandler

Funktionen für die Testverwaltung.

Funktion: runTests

Diese Funktion erwartet den exakten Namen eines Testfalls, der sowohl im Shared-File hinterlegt ist als auch vom Backend über die Funktion *listAll()* bereitgestellt wird. Dabei ist es wichtig, dass keine Testpfade übergeben werden, sondern ausschließlich die Namen der einzelnen Testfälle.

Funktion: setTestResultListener

Diese Funktion erwartet, dass ihr eine Callback-Funktion übergeben wird. Diese wird jedes Mal ausgeführt, sobald ein Testfall abgeschlossen ist.

Innerhalb dieser Funktion wird das Testergebnis übergeben, sodass es weiterverarbeitet werden kann.

Diese ContextBridge bleibt durchgehend aktiv, bis sie entweder explizit über die *stop*-Funktion der ContextBridge geschlossen wird oder der entsprechende Vorgang abgeschlossen ist.

Funktion: setOnTestReadyListener

Ähnlich wie die zuvor beschriebene Funktion erwartet auch diese eine Callback-Funktion. Der wesentliche Unterschied besteht darin, dass hier die Tests übergeben werden, die gerade gestartet werden.

Dies ermöglicht es beispielsweise, für die entsprechenden Testfälle eine Ladeanimation zu setzen.

Diese ContextBridge bleibt durchgehend aktiv, bis sie entweder explizit über die *stop*-Funktion der ContextBridge geschlossen wird oder der entsprechende Vorgang abgeschlossen ist.

Funktion: stop

Diese Funktion erwartet keine Eingabeparameter und liefert keine Rückgabe. Bei ihrem Aufruf werden der zuletzt genannte *setTestResultListener* sowie der *setOnTestReadyListener* geschlossen.

Funktion: listAll

Diese ContextBridge erwartet ebenfalls keine Eingabeparameter. Ihre Aufgabe besteht darin, alle implementierten Testfälle in der bereits mehrfach erwähnten Baumstruktur an das Frontend zurückzugeben.

Dabei wird die hierarchische Struktur der Tests berücksichtigt, sodass das Frontend die Testfälle in der gleichen Ordnung erhält, wie sie im Backend organisiert sind. Auf diese Weise kann das Frontend jederzeit auf die vollständige Übersicht der verfügbaren Tests zugreifen, ohne eigene Logik zur Ermittlung der Testhierarchie implementieren zu müssen.

5.1.2 ContextBridge: deviceHandler

Funktionen für die Client-Server-Kommunikation.

Funktion: getDevice

Diese ContextBridge erwartet die IP-Adresse und den Port eines Geräts als feste Eingabeparameter. Zusätzlich können optional ein Benutzername und ein Passwort übergeben werden, da einige Geräte eine Authentifizierung verlangen.

Als Rückgabewert liefert die Funktion den Namen des Geräts sowie den zugehörigen OPC-UA- bzw. Euromap-Standard in Form eines Strings. Diese Informationen ermöglichen es dem Frontend, das Gerät korrekt zu identifizieren und den passenden Standard für die weitere Kommunikation oder Testausführung zu berücksichtigen.

Funktion: startPing

Bei dieser ContextBridge handelt es sich um eine Schnittstelle zum Starten des *PingWorker*. Sie erwartet keine Eingabeparameter und liefert keinen Rückgabewert.

Funktion: stopPing

Diese ContextBridge dient dazu, den *PingWorker* zu beenden, der zuvor über die Funktion *startPing* gestartet wurde. Sie erwartet keine Eingabeparameter und liefert ebenfalls keinen Rückgabewert.

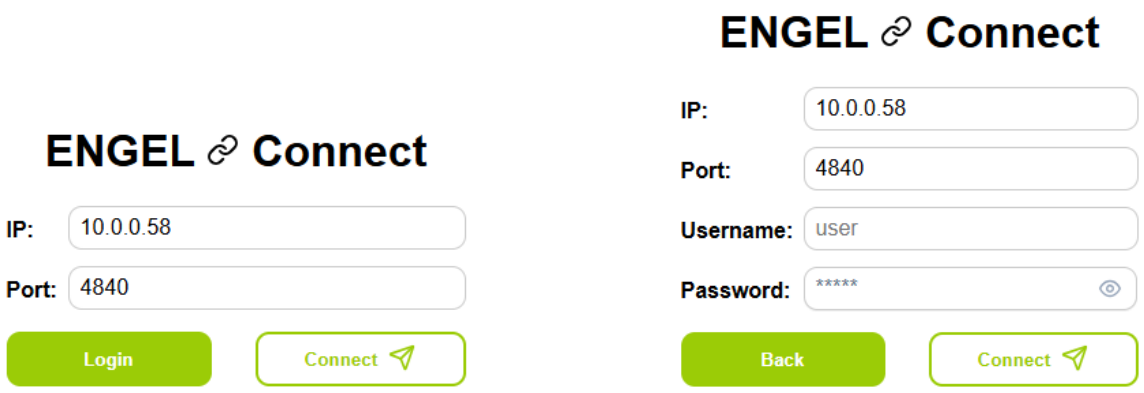
Funktion: onStatusChange

Diese ContextBridge erwartet eine Callback-Funktion als Übergabeparameter. Ihr wird der Verbindungsstatus übergeben, der ausschließlich die Werte *'online'* oder *'offline'* annehmen kann.

5.2 Frontend Endergebnis

Im folgenden Abschnitt wird das finale Ergebnis der Frontend-Implementierung dargestellt. Dabei werden die einzelnen Ansichten der Anwendung sowie deren Umsetzung im Vergleich zum ursprünglichen -Entwurf beschrieben. Ziel ist es, die praktische Umsetzung der Benutzeroberfläche sowie wesentliche Anpassungen und Erweiterungen im Entwicklungsprozess nachvollziehbar darzustellen.

5.2.1 Startseiten



(a) Startseite ohne Login (b) Loginseite

Abbildung 23: Vergleich beider Verbindungsmöglichkeiten

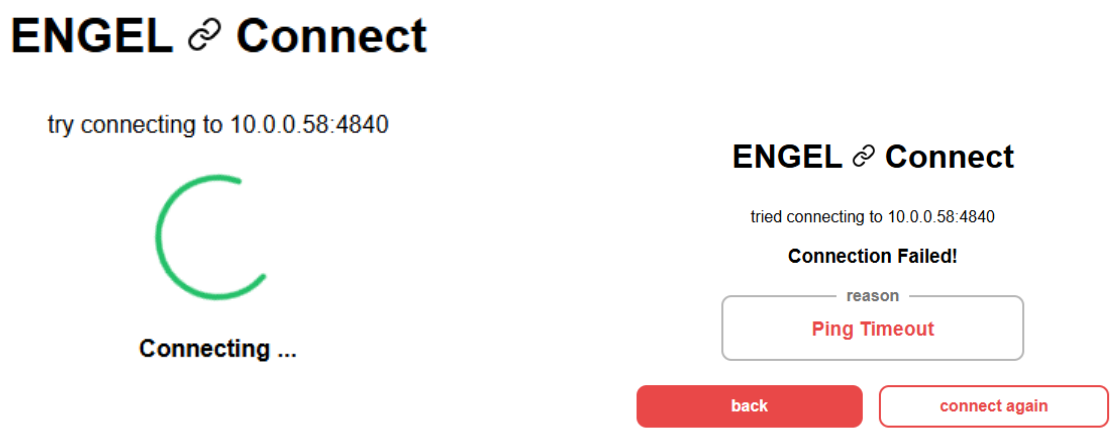
Wie bereits im Figma-Entwurf (23) konzipiert, existiert eine Startseite mit sowie eine ohne Anmeldung. Beide Varianten wurden nahezu unverändert in Angular umgesetzt.

Die einzige wesentliche Abweichung besteht darin, dass die Eingabefelder für Benutzername und Passwort in der finalen Implementierung unterhalb der übrigen Elemente angeordnet wurden und nicht — wie im Figma-Design vorgesehen — rechts daneben.

5.2.2 Verbindungsseiten

Auch in diesem Fall wurde — bis auf kleinere Anpassungen — das im Figma-Entwurf (16) festgelegte Design weitgehend beibehalten.

Für die Ladeanimation wurde aus Gründen der effizienteren Entwicklung der vorgefertigte Progress-Spinner von PrimeNG verwendet. Dies stellt zwar eine Abweichung vom ursprünglichen Design dar, wirkt jedoch bei nüchterner Betrachtung optisch ansprechender und professioneller.



(a) Loading Screen (b) Verbindung Fehlgeschlagen

Abbildung 24: Zwischenseiten beim Laden und bei fehlgeschlagenen Verbindungen

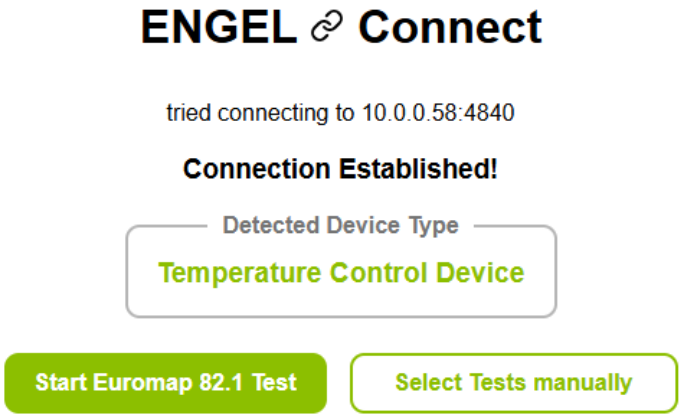


Abbildung 25: Seite Verbindung erfolgreich

5.2.3 Testauswahlseite

Das im Figma-Entwurf (18) definierte Design wurde in Angular entsprechend umgesetzt. Wie in Abbildung (26) ersichtlich ist, funktioniert das geplante Auswahlkonzept innerhalb der Baumstruktur wie vorgesehen.

Zur technischen Realisierung kam die „TreeSelect“-Komponente mit zusätzlich konfigurierten Checkboxen zum Einsatz. Dadurch konnte eine erhebliche Zeitersparnis erzielt werden, da eine eigenständige Implementierung dieser Funktion mit deutlich höherem Entwicklungsaufwand verbunden gewesen wäre.

Aufgrund dieser gewonnenen Entwicklungszeit war es möglich, ein zusätzliches Feature zu implementieren. Dabei handelt es sich um die Statusanzeige neben dem „Run All“-Button (siehe Abbildung 27). In einem Intervall von fünf Sekunden wird überprüft, ob weiterhin eine aktive

Verbindung zur Maschine besteht. Andernfalls könnte es vorkommen, dass der Benutzer noch Tests auswählt, obwohl die Verbindung zum Gerät bereits unterbrochen wurde.

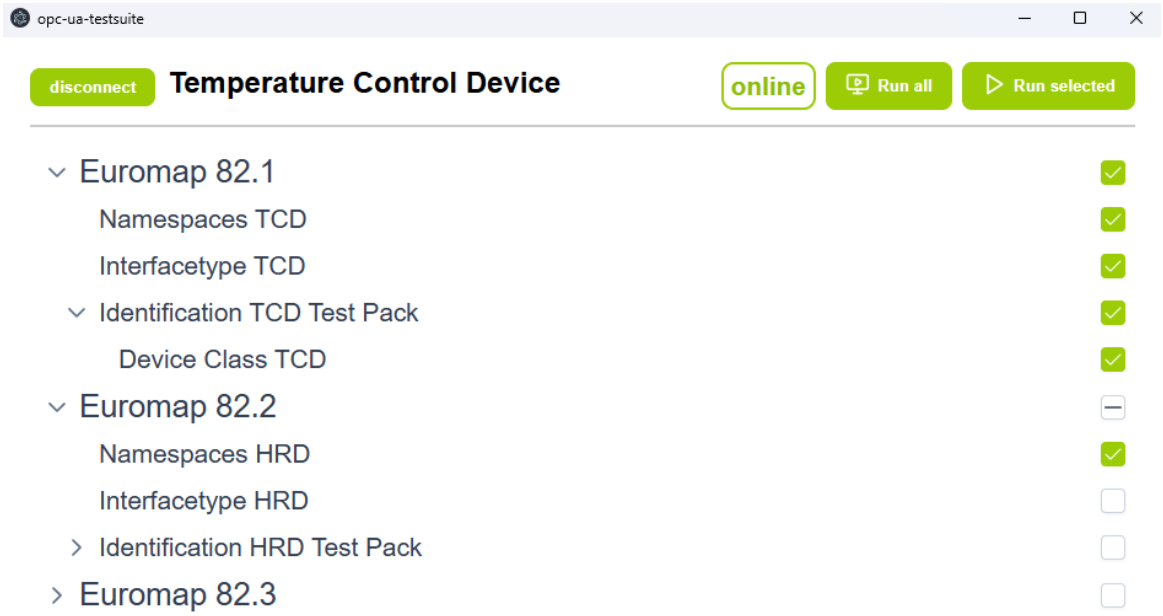


Abbildung 26: Seite zur manuellen Testauswahl

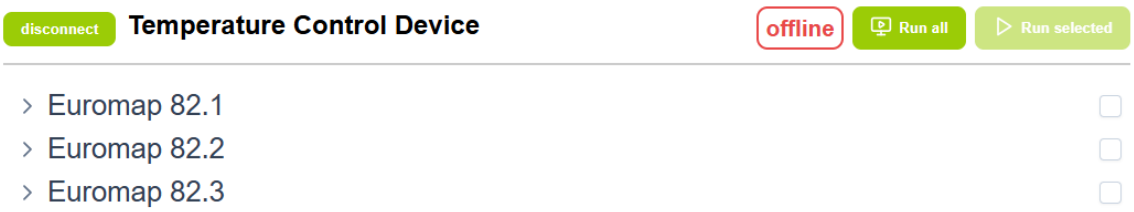


Abbildung 27: Live-Überwachung, ob die Verbindung mit der Maschine noch besteht

5.2.4 Testergebnisseiten

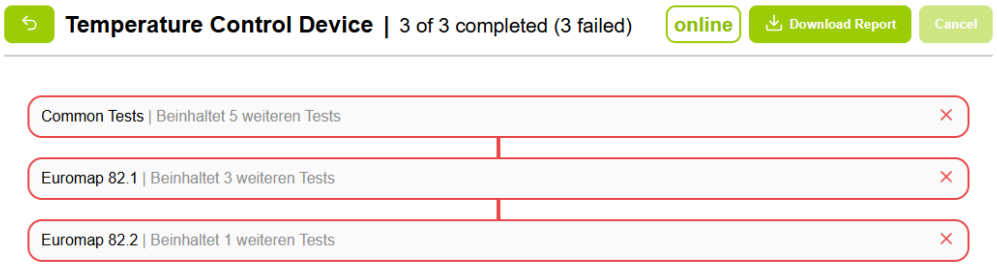


Abbildung 28: Übersichtsseite aller Testergebnisse

Auch in diesem Abschnitt wurde das im Figma-Entwurf (19) festgelegte Design weitgehend beibehalten. Zusätzlich wird hier erneut eine Statusanzeige dargestellt, die anzeigt, ob weiterhin eine Verbindung zum Gerät besteht.

Darüber hinaus wurde rechts daneben ein Button integriert, über den der Benutzer die Testergebnisse herunterladen kann. Dabei wird eine JSON-Datei generiert, in der die vollständige Baumstruktur der Tests gespeichert ist.

Eine weitere Funktion, die erst im Zuge der Implementierung ergänzt wurde, ist die Möglichkeit, laufende Tests abzubrechen. Dies ist insbesondere dann relevant, wenn zunächst nicht bekannt war, welches Gerät verbunden ist und daher der „Run All“-Test gestartet wurde. In einem solchen Fall müsste ansonsten gewartet werden, bis sämtliche Tests abgeschlossen sind, obwohl bereits nach einem Teil der Tests — beispielsweise nach einem Drittel — ausreichend Informationen vorliegen.

Als weitere Erweiterung gegenüber dem ursprünglichen Figma-Entwurf wird zudem angezeigt, wie viele Untertests ein Testpaket beinhaltet.

Navigiert der Benutzer innerhalb der Baumstruktur durch einzelne Testpakete, ergibt sich eine Ansicht wie in Abbildung (29) dargestellt. Wie in der Breadcrumb-Leiste im oberen Bereich ersichtlich ist, wurde in diesem Beispiel zunächst „Common Tests“ und anschließend das „Identification Test Pack“ ausgewählt. Diese Ansicht bleibt für alle möglichen Navigationspfade konsistent. Zudem ist ersichtlich, dass keine weiteren Testpakete vorhanden sind, da hier die Blätter der Baumstruktur erreicht wurden.

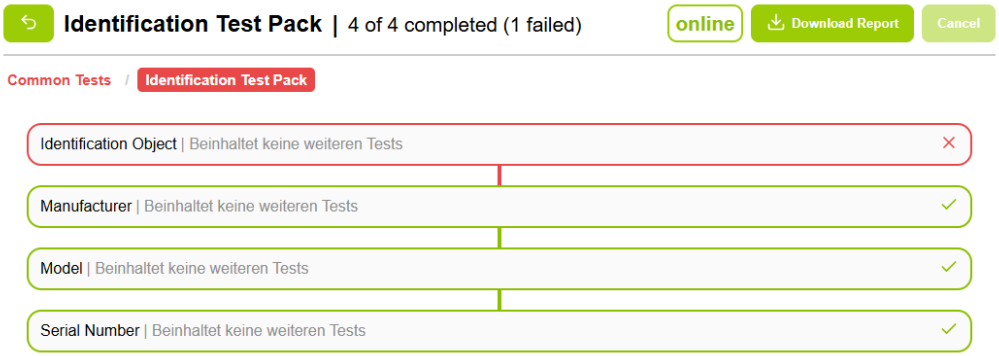


Abbildung 29: Unterseiten der Tests

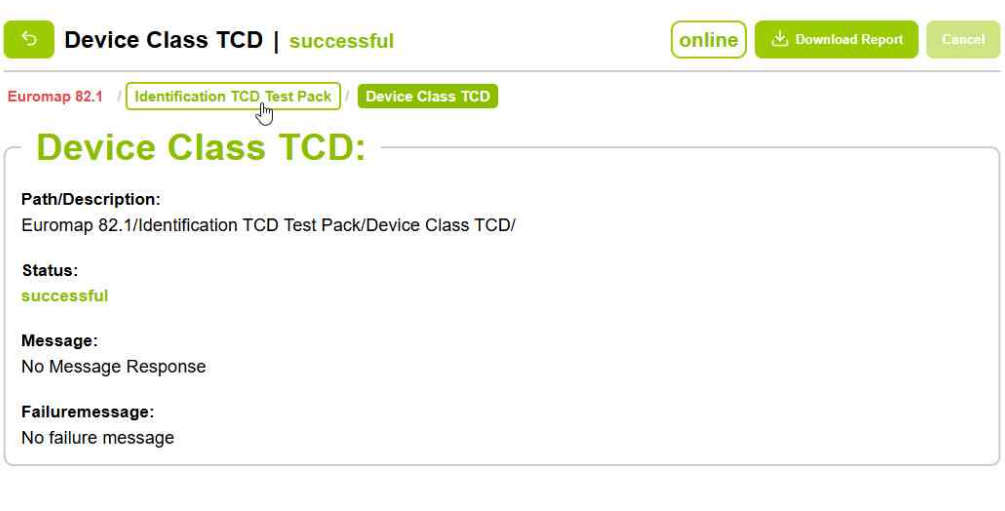


Abbildung 30: Breadcrump-Navigation in der Test-Baumstruktur

Abbildung 31: Detailseite der Testergebnisse

Wählt der Benutzer einen Test aus, der keine weiteren Untertests enthält, gelangt er zu einer Detailansicht, wie in Abbildung (31) dargestellt.

In diesem Bereich wurde bewusst vom ursprünglichen Figma-Design (Abbildungen 20 und 21) abgewichen. Dies ist einerseits auf die Einführung der Baumstruktur zurückzuführen, die zu Beginn der Konzeption noch nicht vorgesehen war. Ursprünglich war geplant, beim Anklicken eines Tests sämtliche Untertests als Liste mit entsprechenden Rückgabewerten darzustellen. Da die Untertests jedoch selbst wiederum aus weiteren Tests bestehen können, war eine zusätzliche Navigationsebene erforderlich, um diese hierarchische Struktur übersichtlich abzubilden.

Die Navigation erfolgt nun so lange innerhalb der Baumstruktur, bis ein Blattknoten erreicht wird. Erst dann wird die Detailansicht geöffnet. Diese ermöglicht eine deutlich präzisere Darstellung der Testergebnisse. Ein weiterer Grund für die Anpassung des Designs liegt darin, dass die ausgelesenen Fehlermeldungen teilweise sehr umfangreich sind, wie beispielsweise in Abbildung 21 ersichtlich. Das gewählte Layout bietet ausreichend Platz, um diese detaillierten Rückmeldungen strukturiert darzustellen.

Zur besseren Orientierung innerhalb der Baumstruktur wurde unterhalb der oberen Leiste zusätzlich eine Breadcrumb-Navigation eingeführt. Diese zeigt den aktuell gewählten Navigationspfad an. Befindet sich der Benutzer beispielsweise im Pfad „Euromap 82.1 / Identification TCD Test Pack / Device Class TCD“ (siehe Abbildung 30), kann er über die Breadcrumb-Leiste direkt zu einer beliebigen übergeordneten Ebene zurückkehren. Alternativ ermöglicht der „Back“-Button links oben eine schrittweise Navigation zur jeweils vorherigen Ebene.

Darüber hinaus visualisiert die Breadcrumb-Leiste mithilfe der Farbkennzeichnung auch den Status der übergeordneten Tests. So kann beispielsweise das übergeordnete Testpaket „Euromap 82.1“ als fehlgeschlagen markiert sein, während das darunterliegende „Identification TCD Test Pack“ erfolgreich abgeschlossen wurde.

6 Resümee

Die Zusammenarbeit zwischen Frontend- und Backend-Entwicklung stellt in Softwareprojekten eine zentrale Herausforderung dar, insbesondere wenn beide Bereiche parallel entwickelt werden sollen. Ziel ist es, Arbeitsabläufe so zu koordinieren, dass keine unnötigen Abhängigkeiten entstehen und beide Teilbereiche möglichst effizient voranschreiten können. Im Rahmen des Projekts konnte das Team wertvolle Erfahrungen in diesem Bereich sammeln, insbesondere im Umgang mit unerwarteten technischen Herausforderungen. Verzögerungen im Backend, bedingt durch ein nicht wie vorgesehen funktionierendes Testframework und den daraus resultierenden Umstieg auf eine alternative Lösung, machten deutlich, wie wichtig eine flexible Planung sowie frühzeitige Abstimmung zwischen den Teilbereichen ist. Diese Erkenntnisse trugen dazu bei, das Verständnis für effiziente Entwicklungsprozesse nachhaltig zu stärken.

Ein weiterer wesentlicher Aspekt im Projektverlauf war die konsequente Einhaltung von Qualitätsstandards im Entwicklungsprozess. Besonderes Augenmerk lag auf einer strukturierten Arbeitsweise im Umgang mit Pull Requests sowie auf einer sorgfältigen Überprüfung des Codes. Dabei wurde nicht nur die funktionale Korrektheit berücksichtigt, sondern auch die Einhaltung von Naming Conventions und einheitlichen Strukturvorgaben. Im Vergleich zu bisherigen Schulprojekten erfolgte diese Kontrolle deutlich strenger und orientierte sich an den unternehmensinternen Richtlinien der ENGEL Austria GmbH. Für das Projektteam stellte dies eine neue und praxisnahe Erfahrung dar.

Die Anwendung der Scrum-Methodik im Unternehmensumfeld erwies sich als besonders wertvoll. Durch tägliche Abstimmungen im Rahmen von Daily Meetings erhielt das Team kontinuierlich Feedback von erfahrenen Mitarbeiterinnen und Mitarbeitern. Dadurch konnten Fehler frühzeitig erkannt und behoben werden. Insbesondere bei grundlegenden Problemen wurde zeitnah Unterstützung geleistet, wodurch verhindert werden konnte, dass sich fehlerhafte Ansätze weiter verfestigen.

Neben organisatorischen Aspekten konnten auch auf technischer Ebene neue Erkenntnisse gewonnen werden. Im Zuge des Projekts wurde eine neue Art der Kommunikation zwischen Frontend und Backend kennengelernt, insbesondere durch den Einsatz der sogenannten ContextBridge. Diese ermöglicht eine strukturierte und zugleich sichere Interaktion zwischen den verschiede-

nen Anwendungsebenen und stellte somit einen wichtigen Bestandteil der Systemarchitektur dar.

Darüber hinaus bot die praktische Arbeit im Unternehmen wertvolle Einblicke in reale Entwicklungsprozesse. Neben technischen und organisatorischen Vorgaben spielte auch die soziale Komponente eine bedeutende Rolle. Der regelmäßige Austausch im Team, gemeinsame Pausen sowie die tägliche Zusammenarbeit trugen wesentlich zu einem produktiven Arbeitsumfeld bei. Insgesamt stellte die Mitarbeit in einem Unternehmen eine neue Erfahrung dar, da zuvor lediglich vereinzelt im Rahmen von Schulprojekten mit externen Partnern zusammengearbeitet wurde.

7 Planung und Realisierung

In diesem Kapitel wird die Planung und Realisierung des Projekts näher beschrieben. Dazu werden zunächst die definierten Meilensteine dargestellt, bevor anschließend auf den tatsächlichen Projektverlauf sowie die dabei aufgetretenen Herausforderungen eingegangen wird.

7.1 Meilensteine

Die Umsetzung der Diplomarbeit erfolgte über einen Zeitraum von vier Wochen (20 Arbeitstage) und wurde anhand mehrerer Meilensteine strukturiert. Diese orientierten sich sowohl an den geplanten individuellen Zielsetzungen als auch am tatsächlichen Projektverlauf.

Meilenstein 1: Grundlegendes UI-Design (Tag 1–3) Zu Beginn des Projekts wurde innerhalb der ersten drei Tage ein grundlegendes UI-Design in Figma erstellt. Dieses diente als visuelle Grundlage für die spätere Implementierung im Frontend. Parallel dazu wurden erste architektonische Überlegungen sowie die Projektstruktur definiert. Dieser Meilenstein konnte planmäßig erreicht werden und bildete die Basis für die weitere Entwicklung.

Meilenstein 2: Zugriff auf die Schnittstellen (bis Ende Woche 1) Ein zentraler technischer Meilenstein war der erfolgreiche Zugriff auf die OPC-UA-Schnittstellen. Dieser wurde innerhalb der ersten Woche erreicht. Dabei wurden erste Verbindungen aufgebaut sowie grundlegende Kommunikationsmechanismen implementiert. Dies war eine essenzielle Voraussetzung für die spätere Entwicklung der automatisierten Tests.

Meilenstein 3: Funktionsfähige Angular-Seite (Woche 1–2) Nach der Fertigstellung des Designs wurde innerhalb der folgenden Woche eine erste funktionsfähige Angular-Anwendung umgesetzt. Diese beinhaltete grundlegende UI-Komponenten wie Startscreen, Login, Navigation sowie erste Darstellungen von Testergebnissen. Damit konnte eine visuelle und funktionale Basis geschaffen werden, auf der die weiteren Features aufgebaut wurden.

Meilenstein 4: Ausführbare Testumgebung (bis Ende Woche 3) Bis zum Ende der dritten Woche wurde eine ausführbare Anwendung realisiert, die automatisierte Tests gegen OPC-UA-Schnittstellen durchführen konnte. Im Zuge der Umsetzung wurde das ursprünglich verwendete Testframework (Jasmine) aufgrund technischer Einschränkungen durch Vitest ersetzt. Dieser Meilenstein stellte einen entscheidenden Fortschritt dar, da nun erstmals vollständige Testabläufe inklusive Ergebnisrückgabe funktionierten.

Meilenstein 5: Tests für Basisschnittstellen (Woche 3–4) Aufbauend auf der funktionierenden Testumgebung wurden die Tests für die Basisschnittstellen implementiert. Dabei wurden zentrale Funktionen wie das sequentielle Ausführen der Tests, die Live-Anzeige der Ergebnisse sowie die korrekte Statusvererbung innerhalb der Baumstruktur umgesetzt. Dieser Meilenstein markierte die Fertigstellung der Kernfunktionalität der Anwendung.

Meilenstein 6: Erweiterung durch optionale Schnittstellen und Features (Woche 4) Im letzten Abschnitt des Projekts wurden zusätzliche Funktionen und Erweiterungen umgesetzt. Dazu zählen unter anderem:

- Abbrechen laufender Tests
- Anzeige des Verbindungsstatus (Ping-Funktion)
- Breadcrumb-Navigation zur Baumstruktur
- JSON-Export der Testergebnisse
- Verbesserungen der Benutzeroberfläche

Diese Erweiterungen konnten insbesondere durch eingesparte Entwicklungszeit (z. B. durch Nutzung bestehender Komponenten) realisiert werden und führten zu einer deutlichen Verbesserung der Benutzerfreundlichkeit und Funktionalität.

7.2 Projektverlauf und Herausforderungen

Der Projektverlauf erstreckte sich über einen Zeitraum von vier Wochen und war geprägt von einer iterativen Entwicklung, bei der sowohl konzeptionelle als auch technische Herausforderungen bewältigt werden mussten. Zu Beginn lag der Fokus auf der Einarbeitung in die Aufgabenstellung sowie auf der Analyse der relevanten Technologien, insbesondere im Bereich OPC UA und der zugehörigen Schnittstellenstandards.

In den ersten Tagen wurde ein grundlegendes UI-Design in Figma erstellt, das als Basis für die spätere Implementierung diente. Parallel dazu wurden erste architektonische Entscheidungen

getroffen, unter anderem die Verwendung eines Monorepos sowie die Kombination aus Angular für das Frontend und Electron für die Desktopanwendung. Bereits in dieser frühen Phase traten erste Schwierigkeiten auf, insbesondere bei der Einrichtung der Entwicklungsumgebung und der notwendigen Zugriffsrechte, wodurch sich die initiale Projektstruktur verzögerte.

Nach der erfolgreichen Einrichtung der Grundstruktur konnte mit der Implementierung der ersten Komponenten begonnen werden. Dazu zählten unter anderem der Startbildschirm, grundlegende Eingabemasken sowie erste Verbindungsversuche mit OPC-UA-Geräten. Parallel dazu wurde die Backend-Logik zur Durchführung von Tests entwickelt. Bereits hier zeigte sich, dass die Umsetzung komplexer war als ursprünglich angenommen, insbesondere im Hinblick auf die Kommunikation zwischen Frontend und Backend sowie die Verarbeitung von Testergebnissen.

Eine der größten Herausforderungen im Projekt stellte die Implementierung der Testausführung dar. Ursprünglich war vorgesehen, das Testing-Framework Jasmine zu verwenden. Im Laufe der Entwicklung stellte sich jedoch heraus, dass Jasmine für den vorgesehenen Anwendungsfall nur eingeschränkt geeignet ist. Insbesondere die Tatsache, dass Jasmine Schwierigkeiten mit mehrfachen oder parallelen Testausführungen hatte und globale Instanzen verwaltet, führte zu erheblichen Problemen. Tests konnten nicht zuverlässig mehrfach gestartet werden, und es kam zu Fehlern bei der Wiederverwendung von Testläufen.

Nach intensiver Analyse und mehreren Lösungsversuchen wurde daher entschieden, das Testing-Framework zu wechseln. Es erfolgte ein vollständiger Umstieg auf Vitest, ein moderneres und flexibleres Framework. Dieser Wechsel stellte zunächst einen zusätzlichen Aufwand dar, da Teile der bestehenden Implementierung angepasst werden mussten. Gleichzeitig brachte Vitest jedoch entscheidende Vorteile mit sich, insbesondere eine einfachere Testausführung, bessere Kontrolle über Testläufe sowie eine leichtere Integration in die bestehende Architektur. Durch die Einführung eines eigenen Test-Runners und die Anpassung der ContextBridge konnte eine stabile und performante Testausführung realisiert werden.

Ein weiterer zentraler Punkt im Projektverlauf war die Entwicklung der Baumstruktur für die Tests. Ursprünglich war geplant, Testergebnisse in einer einfachen Listenstruktur darzustellen. Im Laufe der Implementierung zeigte sich jedoch, dass Tests hierarchisch aufgebaut sind und Untertests wiederum eigene Unterstrukturen besitzen können. Dies führte zur Einführung einer rekursiven Baumstruktur sowie entsprechender Navigationsmechanismen im Frontend. Diese Anpassung hatte auch Auswirkungen auf das UI-Design, wodurch vom ursprünglichen Figma-Entwurf teilweise abgewichen werden musste.

Im weiteren Verlauf des Projekts lag der Fokus auf der Stabilisierung und Erweiterung der Anwendung. Es wurden zusätzliche Funktionen implementiert, wie etwa das Abbrechen laufender Tests, die Anzeige des aktuellen Verbindungsstatus sowie die Möglichkeit, Testergebnisse zu exportieren. Auch die Benutzeroberfläche wurde kontinuierlich verbessert, unter anderem durch die Einführung einer Breadcrumb-Navigation zur besseren Orientierung innerhalb der Baumstruktur.

Zusammenfassend lässt sich festhalten, dass der Projektverlauf stark von iterativen Anpassungen geprägt war. Ursprüngliche Konzepte mussten teilweise überarbeitet und an die tatsächlichen Anforderungen angepasst werden. Insbesondere der Wechsel des Testing-Frameworks sowie die Einführung der Baumstruktur stellten wesentliche Wendepunkte dar. Trotz dieser Herausforderungen konnte eine stabile und erweiterbare Lösung entwickelt werden, die den Anforderungen der Praxis gerecht wird.

8 Aufgabenverteilung

Im folgenden Punkt ist festgehalten, wer welche Kapitel der Diplomarbeit verfasst hat. Teilweise wurden Kapitel von mehreren Autoren gemeinsam verfasst, da der jeweilige Bereich in die Zuständigkeit beider Teammitglieder fällt.

Inhaltsverzeichnis Thomas Kastner

Inhaltsverzeichnis Adam Halasz

9 Glossar & Abkürzungsverzeichnis

API application programming interface

bzw. beziehungsweise

IP Internet Protocol

JS JavaScript

JSON JavaScript Object Notation

Monorepo monolithic repository (zentrales Repository für mehrere Projekte)

MVC Model-View-Controller

Nx Build-System und Toolchain für Monorepos

Node.js JavaScript-Laufzeitumgebung

PR Pull Request

REST Representational State Transfer

Scrum agiles Vorgehensmodell der Softwareentwicklung

URL Uniform Resource Locator

ContextBridge Schnittstelle zur sicheren Kommunikation zwischen Frontend und Backend

Port Kommunikationsendpunkt innerhalb eines Netzwerks

Testframework Software zur automatisierten Durchführung von Tests

Mockup visuelle Darstellung eines Designs ohne Funktionalität

Euromap standardisierte Schnittstellen für Maschinen in der Kunststoffindustrie

usw. und so weiter

uvm. und vieles mehr

Literaturverzeichnis

[1] Microsoft, „TypeScript Documentation," https://www.typescriptlang.org/docs/, 2025, Zugriff am 17.03.2026.

[2] Microsoft, „TypeScript Handbook," https://www.typescriptlang.org/docs/handbook/intro.html, 2025, Zugriff am 17.03.2026.

[3] Wikimedia, „TypeScript Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Typescript_logo_2020.svg

[4] Google, „Angular Documentation," https://angular.io/docs, 2025, Zugriff am 17.03.2026.

[5] Google, „Angular Architecture Overview," https://angular.io/guide/architecture, 2025, Zugriff am 17.03.2026.

[6] Wikimedia, „Angular Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Angular_wordmark_gradient.png

[7] npm, Inc. (2026) npx: Execute Node Packages Easily. Zugriff am 24.03.2026. Online verfügbar: https://www.npmjs.com/package/npx

[8] OpenJS Foundation. (2026) Node.js JavaScript Runtime. Zugriff am 24.03.2026. Online verfügbar: https://nodejs.org/

[9] Microsoft, „Azure DevOps Documentation," https://learn.microsoft.com/en-us/azure/devops/, 2025, Zugriff am 17.03.2026.

[10] Microsoft, „Pull Requests in Azure Repos," https://learn.microsoft.com/en-us/azure/devops/repos/git/pull-requests, 2025, Zugriff am 17.03.2026.

[11] Git, „Git Documentation," https://git-scm.com/docs, 2025, Zugriff am 17.03.2026.

[12] Wikimedia, „Git Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Git-logo.svg

[13] Wikimedia, „Azure DevOps Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Microsoft_Azure_Logo.svg

[14] Electron Team. (2026) Electron: Build Cross-Platform Desktop Apps with JavaScript, HTML, and CSS. Zugriff am 24.03.2026. Online verfügbar: https://www.electronjs.org/

[15] The Chromium Authors. (2026) Chromium: Open-source Browser Project. Zugriff am 24.03.2026. Online verfügbar: https://www.chromium.org/

[16] Electron Team. (2026) ContextBridge API Documentation. Zugriff am 24.03.2026. Online verfügbar: https://www.electronjs.org/docs/latest/api/context-bridge

[17] Jasmine Core Team. (2026) Jasmine: Behavior-Driven JavaScript Testing Framework. Zugriff am 24.03.2026. Online verfügbar: https://jasmine.github.io/

[18] Vitest Contributors. (2026) Vitest: A Fast Unit Testing Framework for Vite Projects. Zugriff am 24.03.2026. Online verfügbar: https://vitest.dev/

[19] Microsoft, „Visual Studio Code Documentation," https://code.visualstudio.com/docs, 2025, Zugriff am 17.03.2026.

[20] Wikimedia, „VS Code Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Visual_Studio_Code_1.35_icon.svg

[21] Figma Inc., „Figma Documentation," https://help.figma.com/, 2025, Zugriff am 17.03.2026.

[22] Wikimedia, „Figma Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Figma-logo.svg

[23] PrimeTek, „PrimeNG Documentation," https://primeng.org/, 2025, Zugriff am 17.03.2026.

[24] Wikimedia, „PrimeNG Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://kinsta.com/de/blog/angular-komponentenbibliotheken/

[25] Nrwl. (2025) Nx: Smart, Fast and Extensible Build System. Zugriff am 24.03.2026. Online verfügbar: https://nx.dev

[26] Nx und Contributors. (2025) Monorepo vs Modular Repo. Online verfügbar: https://monorepo.tools/

[27] EUROMAP, „EUROMAP - European Committee of Machinery Manufacturers for the Plastics and Rubber Industries," https://www.euromap.org/, 2025, Zugriff am 17.03.2026.

[28] Wikimedia, „Euromap Logo," 2017, letzter Zugriff am 12.03.2026. Online verfügbar: https://getvectorlogo.com/euromap-european-plastics-and-rubber-machinery-vector-logo-svg/

[29] OPC Foundation, „OPC Unified Architecture (OPC UA) Overview," https://opcfoundation.org/about/opc-technologies/opc-ua/, 2025, Zugriff am 17.03.2026.

[30] Clarify. (2025) OPC-UA Logo. Online verfügbar: https://www.clarify.io/integrations-browse/opc-ua

[31] diagrams.net, „diagrams.net Documentation," https://www.diagrams.net/, 2025, Zugriff am 17.03.2026.

[32] Draw.io Team. (2026) Draw.io Logo. Zugriff am 24.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Diagrams.net_Logo.svg

[33] Microsoft. (2026) Microsoft Paint Logo. Zugriff am 24.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Microsoft_Paint.svg

[34] CAKE.com Inc., „Clockify Documentation," https://clockify.me/help, 2025, Zugriff am 17.03.2026.

[35] Clockify. (2026) Clockify. Zugriff am 24.03.2026. Online verfügbar: https://commons.wikimedia.org/wiki/File:Clockify.png

[36] E. Gamma, R. Helm *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

Abbildungsverzeichnis

Quellcodeverzeichnis

Anhang

A Projektplakat

Abbildung 32: Projektplakat

B Zeitaufzeichnung

Abbildung 33: Zeitaufzeichnung

C AI-Dokumentation

KI-Tool	Verwendete Prompts	Einsatz in der Arbeit	Bemerkungen
ChatGPT (OpenAI)	„Ich schreibe gerade eine Diplomarbeit in Österreich. Ich werde dir einzelne Ausschnitte geben die du dann nach Ausdrucksweise, Grammatik, Rechtschreibung usw. kontrollieren sollst“ „Formuliere diesen Absatz wissenschaftlicher um“	Gesamte Arbeit	Rechtschreibung und Grammatikfehler wurden verbessert sowie auch die Ausdrucksweise
ChatGPT (OpenAI)	„Ich schreibe gerade einen Diplomarbeit an einer österreichischen HTL. Ich habe weiters eine Sprachschwäche, da Deutsch nicht meine Muttersprache ist, und deswegen möchte ich, dass du was ich schreibe grammatikalisch ausbesserst oder verschönerst. Auf jeden Fall darfst du an der Bedeutung/Aussage der Sätze nichts verändern.“	Gesamte Arbeit	Rechtschreibfehler und Grammatikfehler wurden aufgedeckt

Tabelle 1: Dokumentation der Nutzung von KI-Tools und Prompts