HOCHSCHULE MITTWEIDA
University of Applied Sciences

BACHELORARBEIT

Herr
Alexander Feilke

Anwendung von Continuous Integration am Enjine.io Framework

Mittweida, Oktober 2022

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Anwendung von Continuous Integration am Enjine.io Framework

Autor:
Alexander Feilke

Studiengang:
Angewandte Informatik

Seminargruppe:
IF19wS-B

Erstprüfer:
Prof. Dr.-Ing. Uwe Schneider

Zweitprüfer:
Dr. rer. nat. Rico Beier-Grunwald

Einreichung:
Mittweida, 24.10.2022

Verteidigung/Bewertung:
Mittweida, 24.10.2022

Faculty of **Applied Computer Sciences and Biosciences**

BACHELOR THESIS

Application of Continuous Integration on the Enjine.io Framework

Author:
Alexander Feilke

Course of Study:
Applied Computer Science

Seminar Group:
IF19wS-B

First Examiner:
Prof. Dr.-Ing. Uwe Schneider

Second Examiner:
Dr. rer. nat. Rico Beier-Grunwald

Submission:
Mittweida, 24.10.2022

Defense/Evaluation:
Mittweida, 24.10.2022

Bibliografische Beschreibung:

Feilke, Alexander:
Anwendung von Continuous Integration am Enjine.io Framework. – 2022. – 47 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2022.

Referat:

Die tägliche Arbeit von Softwareentwicklern ist es, Software so zu schreiben, dass sie auch in Zukunft so schnell und flexibel entwickelt werden kann wie am ersten Tag. Dafür stehen ihnen zahlreiche Methoden und Tools zur Unterstützung zur Verfügung. Jedoch stellt die Einrichtung und erstmalige Anwendung solcher Hilfsmittel oftmals eine Hürde dar, vor allem, wenn man selbst in diesem Bereich noch keine Erfahrung gesammelt hat.
In dieser Arbeit wird dieser Ansatz durch die Entwicklung und Anwendung einer statischen Code-Analyse und Modultests auf eine bestehende NodeJS-Software verfolgt. Diese Software ist Teil einer übergeordneten Projektinfrastruktur, auf deren Komponenten diese Testverfahren später ebenfalls angewendet werden sollen. Für die Evaluation werden etablierte Tools zur Implementierung und Automatisierung der Testprozesse ausgewählt. Der Vergleich dieser Tools erfolgt dann auf der Grundlage gewichteter Kriterien die anhand einer subjektiven Einschätzung bewertet werden.
Zunächst werden die Tools JSLint, JSHint und ESLint für die statische Codeanalyse evaluiert. Dann werden Unit-Tests entworfen und definiert und anschließend mit den Unit-Test-Frameworks Mocha, Jest und Vitest implementiert. Schließlich werden die Tests mit einem CI-Tool automatisiert. Aus der Vielzahl an Plattformen wurden BitBucket Pipelines, CirleCI und Buddy als Testobjekte ausgewählt. Es stellte sich heraus, dass eine vielzahl projektspezifischer Faktoren bei der Auswahl der CI Tools eine Rolle spielen. Die Evaluierung der Tools lieferte eine solide Grundlage für weitere Tests und damit Vertrauen und Sicherheit in die Zukunft von EnjineIO.
Diese Arbeit ist besonders für Softwareentwickler interessant, die noch keine Erfahrung mit Softwaretests gemacht haben und einen Einblick in dieses Thema erhalten möchten. Zudem dient sie dazu, einen Einblick in die Besonderheiten der genannten Softwaretest-Tools zu erhalten, wenn ein Team den Wechsel auf eines davon plant.

Inhaltsverzeichnis

Ergänzende Verzeichnisse

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

BB . . . . . BitBucket

CCI . . . . . CircleCI

CD . . . . . Continuous Deployment

CI . . . . . Continuous Integration

CJS . . . . . CommonJS

CLI . . . . . Command Line Interface (dt. Kommandozeilen-Schnittstelle)

ES . . . . . ECMA-Script

ESM . . . . . ECMA-Script Module

NPM . . . . . Node Package Manager

RBT . . . . . Risk-Based Testing (dt. Risikobasiertes testen)

Vorwort

Diese Arbeit wäre nicht ohne David Hurren entstanden, Mitgründer und Inhaber von droidscript.org und der App 'DroidScript' mit der ich meine Leidenschaft in der Softwareentwicklung gefunden habe. Durch dessen Community und insbesondere dem Forumsmitglied Steve Garman konnte ich meine Programmierfähigkeiten stetig erweitern. Das hat mich zudem motiviert, mich meinerseits durch zahlreiche Zuarbeiten, Unterstützung im Forum sowie einiger Praktika in die Entwicklung von DroidScript einzubringen.
Heute bin ich als Entwickler fester Bestandteil des aufstrebenden Projektes 'EnjineIO', welches auf den Prinzipien von DroidScript aufbaut. Dieses Projekt inspirierte mich dazu mein Portfolio um das Testen von Software zu erweitern.
Mein großer Dank gebührt ebenfalls meinen Betreuern Prof. Dr.-Ing. Uwe Schneider und Dr. rer. nat. Rico Beier-Grunwald, die mich tatkräftig durch ihre zahlreichen Hinweise und Anmerkungen während der Erstellung dieser Arbeit unterstützt haben. Zudem konnte ich das durch sie vermittelte Wissen aus den Studienmodulen Betriebssysteme und Softwaretechnik gut zur Anwendung bringen.
Ebenso danke ich meinen Eltern und Komilitonen, die sich die Zeit genommen haben meine Arbeit nicht nur einmal durchzulesen und einige Wogen in meiner Formulierung und Grammatik zu glätten.

Mittweida, im Oktober 2022
Alexander Feilke

1 Einleitung

Der Prozess Softwareentwicklung unterliegt einem stetigen Wandel an Anforderungen und neuen Funktionalitäten. Mit jeder neuen Funktion wird die Komplexität der Software etwas erhöht, da neue Abhängigkeiten und Nebeneffekte entstehen. Deshalb wird es für Entwickler mit fortschreitender Entwicklungszeit zunehmend schwieriger, den Überblick über die Software zu wahren. Wenn bei der Entwicklung keine Regeln beachtet werden, passiert es daher häufig, dass mit der steigenden Komplexität des Codes seine Wartbarkeit und damit die Produktivität der Entwickler rapide fällt. In extremen Fällen kann das bis zum vollständigen Erliegen der Entwicklung führen, was ein kostenintensives Umschreiben oder den Verlust des Produktes nach sich zieht. [4, S. 4 ff.]

Die Entwicklungsgeschwindigkeit auch über lange Zeiträume hinweg konsistent zu halten und die die Qualität und Agilität des Codes sicherstellen sind Kernaspekte der agilen Softwareentwicklung. Dafür stehen den Entwicklern heutzutage eine Reihe an Methoden und Hilfsmitteln zur Verfügung, von denen viele in Robert C. Martins Buch 'Clean Code' [4] erläutert werden. Diese sollten jedoch bereits frühzeitig in ein Projekt integriert werden, da bereits bestehende Komplexität nicht ohne Weiteres aufgelöst werden kann.

1.1 Zielstellung

Das Ziel dieser Arbeit ist die Anwendung von 'Continuous Integration' in der frühen Entwicklungsphase eines neuen Projekts. Zu diesem Zweck muss zunächst eine Reihe von Tests definiert, implementiert und anschließend automatisiert werden. Der Prozess endet damit, eine ausführbare Version der Software auf einen internen Server hochzuladen und für weitere Abnahmetests bereitzustellen. Für jede dieser Teilaufgaben gibt es bereits zahlreiche branchenübliche Werkzeuge, von denen hier nur eine Auswahl betrachtet wird. Diese werden dann genauer evaluiert und eines davon ausgewählt. Ziel ist es, die Entwicklung in Zukunft schneller und sicherer voranzutreiben und gleichzeitig die Qualität der Software zu sichern.

1.2 Fachlicher Kontext

Die nicht gewinnorientierte Organisation 'droidscript.org' hat ein Projekt entwickelt, das den Einstieg in die Programmierung erleichtern möchte. 'DroidScript' [6] ist eine Android App, mit der man in wenigen Zeilen JavaScript eigene Android-Anwendungen erstellen kann. Dabei unterstützt es den Gebrauch von HTML(5), NodeJS, WebGL/PixiJS, einem Cloud-Service, zahlreichen Plugins und vielen weiteren Features.

1.2.1 EnjineIO

Seit einiger Zeit befindet sich ein neues Projekt 'EnjineIO' [7] in der Entwicklung, welches auf den Grundprinzipien von DroidScript aufbaut, jedoch eine fortgeschrittenere Nutzergruppe ansprechen. Dabei unterstützt es im unterschied zu DroidScript viele verschiedene Plattformen durch eine plattformübergreifende Serverkomponente. Zu dieser kann sich dann eine

Browser-basierte Entwicklungsumgebung (IDE) verbinden, in der der Anwender seinen Code schreibt, debuggt, ausführt, sowie seine Dateien verwaltet und Zugriff auf Dokumentation und weitere Ressourcen hat.

1.2.2 Projektstruktur

EnjineIO besteht aus mehreren Komponenten mit speziellen Aufgaben. Diese sind in separaten Repositories der Quellverwaltung angelegt und werden von verschiedenen Mitgliedern des EnjineIO Teams entwickelt.

Core-Framework
: NodeJS Basisframework für sämtliche grundlegenden Funktionen wie Basisklassen, Programmereignisse, Ein-/Ausgabe und Dateisystemzugriff

UI-Framework
: Auf ReactJS basierende Bibliothek für sämtliche Grafikkomponenten

<OS>-Framework
: Frameworkerweiterung für Betriebssystem-spezifische Implementationen

IDE-Server
: Server-Komponente, welche die IDE über das Netzwerk bereitstellt
Übernimmt die interne Verwaltung und Ausführung von Projekten und verarbeitet IDE-Interaktionen des Anwenders

IDE-Server-GUI
: Grafisches Konfigurationstool des IDE-Servers
sonst nur über Konsolenargumente einstellbar

IDE
: Browser-basierte Entwicklungsumgebung zum Verwalten, Entwickeln und Debuggen von Anwendungen sowie zum Erstellen ausführbarer Dateien
Stellt Editor, Dokumentation, News und Erweiterungen zur Verfügung.

IDE-Extensions
: Mitgelieferte EnjineIO-Standarderweiterungen
Sind durch den Anwender individuell anpassbar, um bspw. Prozesse zu automatisieren

Manager
: Grafisches Management-Programm, welches sich durch ein Systray-Menü mit verfügbaren Server-Instanzen verbinden kann und deren Status überwacht
Fungiert durch eine integrierte IDE und Server-Komponente als EnjineIO-Komplettpaket

1.2.3 Vorarbeit

Im Rahmen einer vorangehenden Praktikumsarbeit wurde diese Serverkomponente inklusive dem Kern-Framework (weiter)entwickelt. Dafür wurde die Sprache JavaScript in dessen Desktop-Variante NodeJS verwendet. Sie unterstützt die Entwicklung, Ausführung und Kompilierung von Grafiklosen Konsolenanwendungen durch das NodeJS Paketierungstool pkg [8] und regulären Desktopanwendungen durch die Verwendung des Electron-Frameworks [9] und des Electron-Builders. [10] Der Anwender soll dabei in jeder Umgebung (Grafik- oder Konsolenanwendung, beim Debuggen oder paketiert) der Spezifizierung entsprechend gleiche Bedingungen vorfinden und möglichst nicht auf unerwartetes Verhalten oder Fehler stoßen.

2 Grundlagen

Das folgende Kapitel soll dem Leser einen groben Überblick über Continuous Integration und gängige Testverfahren in der Softwareentwicklung verschaffen. Auf der Grundlage dieser Vorkenntnisse wird im nächsten Kapitel ihre praktische Anwendung durchgeführt.

Das Konzept des Software Testens ist bereits seit mehreren Jahrzehnten bekannt und von vielen professionellen Entwicklern als notwendig anerkannt. Es dient zum einen dazu bestehende Funktionalität zu erhalten und damit Fehlverhalten zu erkennen [5, S. 7] und zum anderen, die Verwendung der Software und deren Bestandteile zu dokumentieren.

Continuous Integration dagegen ist verglichen mit Softwaretests eine Methode, die sich erst seit einigen Jahren in der Entwicklerszene etabliert hat. [2, S. 7] Es hilft Entwicklungsteams, automatisierte und isolierte Softwaretests durchzuführen und dadurch schnelles Feedback zu den neuesten Änderungen am Code zu erhalten. Die Ersteinrichtung nimmt oft viel Zeit in Anspruch, zahlt sich aber durch weniger und leichter zu behebende Fehler und eine höhere Softwarequalität aus.

2.1 Continuous Integration

Ein bekanntes Phänomen der Softwareentwicklung ist, dass ein Programm auf dem lokalen Rechner des Entwicklers perfekt funktioniert, jedoch nicht bei dem des Kunden oder schlicht in einer leicht veränderten Umgebung. [1, S. 30] Die Ursache sind häufig menschlicher Natur und in der manuellen Ausführung begründet, wie beispielsweise fehlende Dateien, Unterschiede in der Laufzeitumgebung oder vergessene Einrichtungsschritte.

2.1.1 Ziel

Ein populäres Verfahren der agilen Softwareentwicklung welches dieses Problem lösen soll ist 'Continuous Integration' (CI). Dieses automatisiert das Prüfen der Qualität und Funktionalität der Software nach jeder Änderung im Quellcode. Dabei sollen die Tests unter immer gleichen, fest definierten Bedingungen durchgeführt werden. [1, S. 28 f.] Nach der Durchführung der Tests wird dann ein Testbericht generiert und an betroffene Instanzen verschickt (z. B. per E-Mail an die Entwickler). Nach einem erfolgreichen Testdurchlauf kann die Software dann freigegeben werden. (siehe Abbildung 2.1)

CI können weitere Schritte folgen, um das Produkt dem Anwender bereitzustellen. Dabei bezeichnet 'Continuous Delivery' das automatisierte Installieren der Software auf den Produktionsserver. [2, S. 18 ff.] Es übernimmt Aufgaben wie das Erstellen von Backups, Kopieren von Dateien und Anwenden von Konfigurationsschritten. 'Continuous Deployment' stellt einen Zwischenschritt vor Continuous Delivery dar und testet den Bereitstellungsprozess bei jeder Änderung auf einem produktionsähnlichen Server. [2, S. 21] Durch diesen Schritt können Änderungen bereits vor der Freigabe an die Kunden getestet werden.

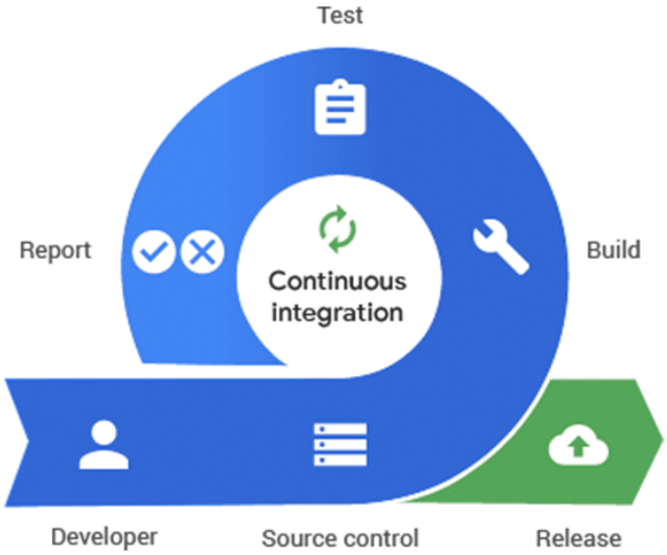

Abbildung 2.1: CI Zyklus[1]

[1] Quelle: https://codefluegel.com/blog/die-kontinuierliche-qualitaetssteigerung-continuous-integration/ vom 09.05.2019, Zugriff 19.10.2022

2.1.2 Aufbau

CI besteht meist aus sich ähnlich verhaltenden Bausteinen, die den Ablauf der CI-Kette definiert. Im Folgenden werden Bezeichnungen von CircleCI [11] und BitBucket [12] mit den Abkürzungen 'CCI' und 'BB' zugeordnet. Zur Vereinheitlichung werden im weiteren Verlauf der Arbeit werden jedoch hauptsächlich die Bezeichnungen von BitBucket verwendet, sofern der Abschnitt keiner speziellen CI-Lösung zugeordnet ist.

Script Die kleinste Einheit stellen einzelne Konsolen-Befehle dar, die zu Scripts (BB) oder Commands (CCI) zusammengefasst werden. Sie können logische Ausdrücke enthalten, die anhand von Eigenschaften oder übergebenen Argumenten verschiedene Befehle ausführen. Sie werden allesamt in der gleichen Umgebung nacheinander ausgeführt. Schlägt ein Befehl unerwartet fehl, wird der CI-Prozess so schnell wie möglich abgebrochen.

Step Den Skripten übergeordnet sind CI Steps (BB) bzw. Jobs (CCI). Sie stellen einen einzelnen CI-Prozess dar, der in einer vorgegebenen Umgebung ausgeführt wird. Diese können individuell an bestimmte Ressourcenklassen, Umgebungsvariablen, Auslöser und andere Eigenschaften gebunden sein. Ein Step verfügt zudem meist über einen Cache[2], der vor jeder Ausführung geladen und zu einem beliebigen Zeitpunkt gesichert werden kann. Während der Ausführung können verschiedene Statistiken über Ausführungszeit, Ressourcenverbrauch und Testergebnisse gesammelt und anschließend in einem Testbericht ausgewertet werden.

Pipeline Die logische Abfolge von Steps wird als Workflow (CCI) oder Pipeline (BB) bezeichnet. Die Steps können dabei sequentiell oder parallel verknüpft werden. Ein Workflow kann manuell durch die UI oder eine API-Anfrage, oder automatisch durch einen Auslöser wie das hochladen neuer Commits oder einen Timer gestartet werden. Bei fehlschlagen einer Pipeline sollte das Entwicklungsteam entsprechend benachrichtigt werden, damit das Problem so schnell wie möglich behoben werden kann.

2.1.3 Ablauf

Der CI-Ablauf startet meist bei dem hochladen des Quellcodes in ein Quellverwaltungssystem wie Bitbucket. (siehe Abbildung 2.1) Darauf reagiert der CI-Server und erstellt daraufhin eine neue Ausführungsumgebung, oft eine Docker-Instanz oder eine Virtuelle Maschine (VM) eines beliebigen Betriebssystems. Diese bezieht dann die neueste Version aus der Quellverwaltung und führt anschließend eine Reihe an Schritten aus (Workflow), um die Software zu Testen und Bereitzustellen. Im Fehlerfall wird der Workflow abgebrochen und eine entsprechende Nachricht an die Entwickler bzw. das Quellverwaltungssystem herausgegeben.

2.1.4 Nutzen

CI und CD Hilft dabei, den qualitativen Zustand einer Software messbar zu machen. Es reduziert das Fehlerrisiko von unerkannten Abhängigkeiten und Dokumentiert im gleichen Zuge alle notwendigen Einrichtungsschritte. [1, S. 47] Es steigert die Zuversicht in die Soft-

[2] Speichert einen Zustand oder das Ergebnis einer Operation zwischen, um in zukünftigen Anfragen an Rechenleistung einzusparen. Reine Downloads profitieren bei CI nicht von Caches, wenn er herunter und wieder hochgeladen werden muss.

ware, da jede Änderung am Code automatisch in einer fest definierten Umgebung getestet wird. [2, S. 29] Zudem kann die Software bei erfolgreicher Ausführung sofort als fertige Anwendung in der zentralen Quellverwaltung bereitgestellt werden. Der komplette Vorgang ist somit unabhängig vom Entwickler, der verwendeten Entwicklungsumgebung, lokal installierter Programme oder Konfigurationen und sogar dem Betriebssystem und der Hardware auf der der Code entwickelt wird.

2.1.5 Anwendungsfälle

Es gibt verschiedene CI-Dienste mit unterschiedlichem Einrichtungsaufwand. Open-Source Projekte Nutzen meist Online CI-Dienste wie beispielsweise Travis CI, CircleCI oder Buddy. [1, S. 266] Diese können mit verschiedenen Online-Quellverwaltungsdiensten wie GitHub, Gitlab oder Bitbucket verknüpft werden und reagieren automatisch auf Änderungen des Quellcodes im Repository (Commits). Manche Quellverwaltungen bieten auch eine eigene CI-Lösung an wie z. B. Bitbucket Pipelines oder Gitlab CI.

Lokale CI-Installation Es gibt jedoch auch CI-Implementationen wie Jenkins, die auf einem privaten Server installiert und eingerichtet werden können. Diese Variante wird aufgrund des hohen Aufwandes eher von Unternehmen mit einem etablierten Projekt und Entwicklungsteam und bei sicherheitskritischen Aspekten angewendet. [1, S. 266]

Eigene CI-Lösung Für noch speziellere Anforderungen ist es auch möglich komplett auf eine konkrete CI-Lösung zu verzichten und nur basierend auf Containern, VMs oder eigenen Computern einen CI-Server zu schaffen der anschließend eine Reihe an Skripten ausführt. Das Entwicklungsteam von VSCode führt beispielsweise regelmäßig Performance-Tests auf einem nur dafür vorgesehenen ThinkPad aus.[3] Diese erfordern jedoch auch einen höheren Einrichtungsaufwand als etablierte CI-Lösungen mit diversen Integrationsmöglichkeiten.

[3] CovalenceConf 2019: Visual Studio Code - The First Second: https://youtu.be/r0OeHRUCCb4?t=688

2.2 Softwaretests

Softwaretests dienen dazu, die Funktionalität und Qualität einer Software zu dokumentieren und nachzuweisen. Softwaretests können wie in Abbildung 2.2 grob in funktionale und nicht-funktionale Tests unterteilt werden. Funktionale Tests überprüfen dabei die Korrektheit der Software auf verschiedenen Softwareebenen. Nichtfunktionale Tests überprüfen Qualitative Aspekte wie Performance, Sicherheit, Stabilität oder Robustheit.[4]

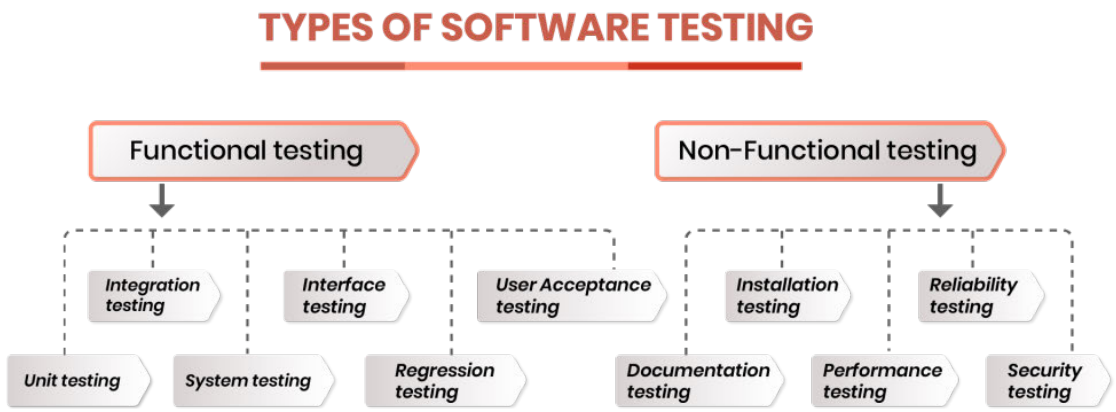

Abbildung 2.2: Arten von Softwaretests[5]

Softwaretests können auch in verschiedene Teststufen eingeteilt werden, die nacheinander durchlaufen werden. Ein verbreitetes Modell ist dabei die 'Test-Pyramide'. In Abbildung 2.3 werden die Tests dabei sortiert nach der Anzahl in mehrere Gruppen unterteilt.

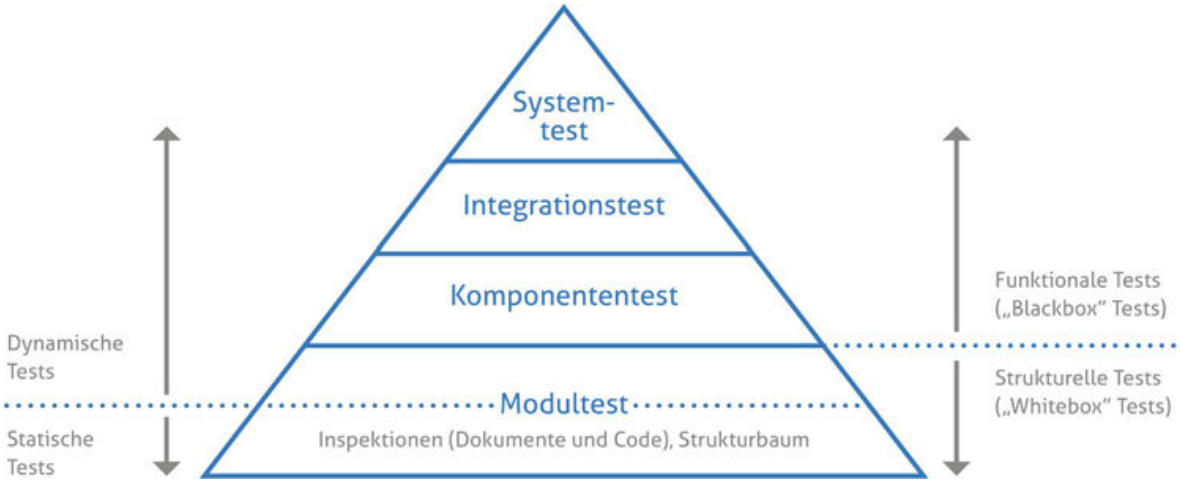

Abbildung 2.3: Test-Pyramide[6]

Im Folgenden werden einige Testarten näher beschrieben und auf deren Eigenschaften und Anwendungspraktiken eingegangen.

[4] siehe: [3, S. 72 f.]
[5] Quelle: https://medium.com/codex/types-of-testing-de4cdd98df77 vom 24.07.2021, Zugriff 12.08.2022
[6] [Abb. 9.4 3, S. 78]

2.2.1 Statische Quellcodeanalyse

Die wohl offensichtlichste und einfachste Methode um Code zu überprüfen sind Tests, die nur anhand des bestehenden Quellcodes Probleme identifizieren. Solche Tests überprüfen an erster Stelle vor allem die Richtigkeit der Syntax, aber auch abstraktere Problemquellen, die sich nicht durch die Syntax sondern in der Semantik aufzeigen. Dabei kann der Code auch auf potentielle Bugs und qualitative Schwachstellen (sog. Code-Smells) und Sicherheitslücken überprüft werden. [1, S. 200] Das Resultat soll dabei ein Test sein, bei dem man nach erfolgreicher Ausführung Vertrauen in die Qualität des entstandenen Codes hat, der dann in die nächste Testphase überführt werden kann.

Einheitliche Standards Ein wichtiger Aspekt für eine langfristige Entwicklung ist dabei ein einheitlicher Code-Stil. [4, S. 90] Ohne definierte Regeln für die Entwicklung wird der Code mit der Zeit stetig unübersichtlicher, und die Wartbarkeit sinkt. Oft zeigen sich diese Probleme erst in einer fortgeschrittenen Entwicklungsphase, in der dann ein zeit- und kostenintensives Refactoring notwendig ist. Um das zu vermeiden, sollten von Anfang an Standards definiert werden, an die sich alle Entwickler halten.

Linter Für Tools, die solche Codeanalysen durchführen, hat sich der Begriff "Linter" etabliert. Linter können aufgrund ihrer Geschwindigkeit schon live während der Entwicklung eingesetzt werden. Entwicklungsumgebungen wie IntelliJ oder Visual Studio unterstützen diese Funktion bereits nativ bzw. durch Erweiterungen wie z. B. bei VS Code. Dadurch werden dem Entwickler Syntaxfehler gemeldet während er schreibt und sparen somit Zeit beim Debuggen.

2.2.2 Unit Tests

Die nächste Stufe von Softwaretests sind Unit tests. Diese testen einzelne Funktionen auf deren Korrektheit und insbesondere all ihre Edge Case. Sie sollten den größten Anteil an Tests ausmachen, da sie sehr schnell in der Ausführung sind und Probleme genau lokalisieren können. Dafür sind Unit Tests jedoch nicht in der Lage festzustellen, ob das Programm als Ganzes funktioniert.

Zeitpunkt Unit Tests sollten parallel während der Entwicklung geschrieben und zusammen mit der statischen Codeanalyse ausgeführt werden. Dadurch werden Fehler sofort erkannt und der geschriebene Code wird von Natur aus 'testbar'. Das heißt Funktionen werden kurz und einfach gehalten[7], haben wenige Parameter und möglichst nur eine Verschachtelung oder Schleife, um dem *do-one-thing*-Prinzip treu zu bleiben. Im Nachhinein ist es meist schwierig Tests zu den geschriebenen Funktionen zu kreieren, was zu 'Legacy Code' führt.[8]

[7] *"Functions should be small. They should do one thing only and do it well."* [vgl. 4, S. 34 f.]
[8] *"Legacy code is simply code without tests"* [5, S. 16]

Test Driven Development

Deshalb ist es eine gute Praxis die Tests schon vor der Implementierung der eigentlichen Funktion zu schreiben. Diese Herangehensweise wird 'Test Driven Development' genannt. Dabei muss der Entwickler bereits vor der Implementierung das Verhalten der Funktion genau (erst formell, dann durch den Test) spezifizieren, bevor er es versucht mit minimalen Aufwand umzusetzen. (siehe 'The Three Laws of TDD' [4, S. 122])

Man dokumentiert also durch die Tests eine Reihe an Anforderungen, die eine Funktion erfüllen soll. Ändern sich diese, werden die Tests entsprechend angepasst und mit ihnen die Code-Implementation.

Modultests

Komponententests bzw. Modultests sind eine Variante von Unit-Tests, werden aber nach ihnen angewandt. Sie testen ein ganzes logisches Modul mit Echtzeitdaten auf deren Funktionalität.

2.2.3 Integration Tests

Die nächste Stufe sind Integrationstests. Sie testen das Zusammenspiel mehrerer voneinander abhängiger Module, die zuvor bereits entsprechende Unit- bzw. Modultests bestanden haben. Dabei wird die Integration aller übrigen Softwareschichten mit der 'Bottom-Up'- oder 'Top-Down'-Strategie in eine Richtung, oder mit dem 'Sandwich'-Ansatz von beiden Seiten getestet. Wichtig dabei ist in jedem Fall, die Module jeweils klar abzugrenzen und deren Abhängigkeiten zu beachten.

2.2.4 System Tests

An oberster Stelle der Test-Pyramide stehen System- bzw. End-To-End-Test. Diese simulieren die Anwendersicht auf die Software und testen Schritte, die auch der Anwender bei der Benutzung gehen würde. Diese sind jedoch oft in der Ausführung sehr zeitaufwändig, weshalb sich in der Realität oft nur auf wesentliche bzw. kritische Aspekte beschränkt wird.

2.2.5 UI Tests

UI Tests sind eine spezielle Form von Softwaretests, die sich nur auf grafische Oberflächen konzentriert. Dabei wird getestet, ob grafische Elemente korrekt angezeigt und nicht überdeckt werden. Weiterhin kann das korrekte Verhalten auf bestimmte Events wie z. B. Mausklicks oder (Falsch)Eingaben überprüft werden.

Für diese Art von Tests sind oft separate Test-Tools vonnöten, die eine grafische Umgebung simulieren können. Aufgrund des hohen Automatisierungsaufwandes werden solche Tests jedoch oft auch nur manuell auf ausgewählte, essentielle Elemente durchgeführt. (siehe Abschnitt 2.2.6)

2.2.6 Praktische Test-Strategien

In der Praxis werden häufig noch weitere Methoden angewandt, um Kosten zu sparen, Termine einzuhalten oder aufgrund mangelnder Erfahrung. Deshalb werden Kompromisse eingegangen und Prioritäten gesetzt, um dennoch eine möglichst hohe Sicherheit garantieren zu können.

Code-Coverage

Code-Coverage ist ein Mittel um festzustellen, wie hoch der Anteil an Code ist, der von den funktionalen Tests erreicht wird. Dabei wird gemessen, welche Teile des Codes während der Ausführung der Tests erreicht wurden. In der Praxis ist jedoch eine Coverage von 100% meist gar nicht möglich, weshalb individuelle Schwellwerte angewendet werden (Richtlinie 80%). Jedoch ist es ein gutes Mittel um festzustellen, welche Teile des Codes noch nicht abgedeckt werden. Die am häufigsten verwendeten Coverage-Arten sind folgende:

Statement Coverage welche Programm-Anweisungen wurden erreicht

Line Coverage welche Quellcode-Zeilen wurden erreicht
(eine Zeile Code kann mehrere Anweisungen enthalten)

Function Coverage welche Funktionen wurden aufgerufen

Branch Coverage welche Abzweigungen wurden im Programm
durch Bedingungen oder Schleifen genommen

Charakterisierungs-Tests

Wenn Legacy Code verändert werden sollen, muss man feststellen können, ob der Rest der Software wie gewohnt funktioniert. Michael C. Feathers schlägt dabei vor, das Verhalten der Software durch sog. Charakterisierungs-Tests festzuhalten. [5, S. 185 ff.] Dabei soll explizit nicht aktiv nach Fehlern gesucht und diese behoben werden. Stattdessen sollen die Tests dementsprechend angepasst werden, was der Code tatsächlich macht. Somit errichtet man ein System mit dem man später Fehler beheben kann, ohne das Verhalten der Software an anderer Stelle unbemerkt zu verändern.

Risk Based Testing

In der Realität ist es oft jedoch wegen Zeitdruck und finanziellen Gründen nicht möglich, umfassende Tests für alle Komponenten zu schreiben. Dafür gibt es den sogenannten 'Risk-Based Testing' (RBT)-Ansatz. Dieser kann ebenfalls auf Software angewendet werden, die keine automatisierten Tests beinhaltet und ist insbesondere für Teams geeignet, die noch keine Erfahrung mit Softwaretests gemacht haben.[9]
Beim RBT wird zunächst analysiert, welche Funktionen kritisch für die Ausführung sind, welche Features am häufigsten benutzt werden, bzw. wie 'sichtbar' Probleme an einer bestimmten Stelle für den Anwender wären. Anschließend werden die entsprechenden Tests nach Priorität sortiert und implementiert.

[9] siehe: https://www.guru99.com/risk-based-testing.html

3 Toolvergleich

Im folgenden Kapitel werden nun für jeden Teil der CI-Kette verschiedene Tools miteinander verglichen, um jeweils das am besten auf die Projektinfrastruktur passende auszuwählen. Für eine einheitliche Testbasis wird dabei vorrangig die EnjineIO Server-Komponente als Testobjekt verwendet.

Diese Arbeit kategorisiert die Vergleichskriterien gleichgewichtet in Funktionalität und Anwendung. Diese werden in den entsprechenden Abschnitten jeweils in spezifischer gewichtete Anforderungen aufgeteilt. Die Bewertung der Tools erfolgt anhand von subjektiven Einschätzungen die sich nach dem Maßstab aus Tabelle 3.1 richten. Nach dieser wird dann das passendste Tool für die zukünftige Entwicklung ausgewählt.

Wertung	Interpretation
0%	Nicht vorhanden
20-40%	Vorhanden, mit erheblichen Mängeln
50%	Grundlegend vorhanden
60-80%	Erwartungen erfüllt
100%	Erwartungen übertroffen

Tabelle 3.1: Interpretation der Bewertungen

3.1 Anforderungen

Da EnjineIO nicht auf einer gewinnorientierten Basis entwickelt wird, setzt das Team vorrangig auf freie Software. Die neu integrierten Tools sollten dementsprechend aktuell, gut dokumentiert und von einer etablierten Community verwendet, getestet und unterstützt werden. Weiterhin sollen sie mit den bereits verwendeten Tools und Frameworks kompatibel sein, sowie durch eine möglichst hürdenfreie Einrichtung und Anwendung überzeugen.

3.2 Statische Analysetools

Um eine statische Quellcodeanalyse durchzuführen wird ein Tool benötigt, welches die Programmiersprache in ihrer verwendeten Version bis zu einem gewissen Grad versteht. Der Linter muss wenigstens die Korrektheit der Syntax überprüfen können. Weiterhin soll durch den Linter ein einheitlicher Code-Stil vorgegeben und auf eventuelle Problemquellen aufmerksam gemacht werden können. Zudem wäre es wünschenswert, wenn durch den Linter ebenfalls die Formatierung definiert und überprüft werden kann. Außerdem hat sich im Verlauf der Arbeit als Bonus herausgestellt, wenn das Tool einfache Fehler selbstständig beheben kann.

3.2.1 Anforderungen

Die Tools selbst sollten jeweils möglichst einfach anzuwenden sein und in einem Grade konfigurierbar, dass die Stilvorgaben den Vorstellungen der Entwickler entsprechend angepasst werden können. Die Dokumentation der Optionen sollte dementsprechend vollständig, übersichtlich und gut erklärt sein.

Die Vergleichskriterien werden für die statische Quellcodeanalyse nach Tabelle 3.2 gewichtet.

Funktionalität		**Anwendung**	
Formatierung	20	Installation	10
einh. Stil	30	Ausführung	30
Probleme	40	Konfiguration	40
Problemerkennung	10	Dokumentation	20
Summe	100	Summe	100

Tabelle 3.2: Anforderungswichtung der Linter

3.2.2 JSLint

Das erste Programm, das einem bei der Suche nach JavaScript Lintern vorgeschlagen wird, ist das Code-Quality-Tool JSLint. [13] Programmiert wurde es von Douglas Crockford, einem Mitentwickler der JavaScript-Sprache und des JSON-Formates und Autor von 'How JavaScript Works'. Aus der offiziellen Hilfeseite: *"JSLint [...] looks for problems in JavaScript programs. [...] JSLint defines a professional subset of JavaScript. [...] JSLint will hurt your feelings."*

Dokumentation Die angewendeten Regeln werden auf dessen Hilfeseite[10] erklärt und begründet, weshalb diese für angemessen erachtet werden. Quellen berichten, dass JSLint nicht ohne weiteres zufriedenzustellen ist. "*I do not think I have ever had a perfect score on that thing.*" [2, S. 153] Es ist also fraglich, ob mit JSLint tatsächlich zuverlässig die Qualität des Codes gesichert werden kann.

Installation Die Pakete aus dem NPM-Registry[11] sind leider veraltet und die offizielle VSCode-Erweiterung ist nicht konfigurierbar. Um die aktuellste JSLint-Version zu installieren, muss man dem Quickstart[12] folgen und den Quellcode manuell herunterladen.

[10] https://www.jslint.com/help.html
[11] offizielle, freie Datenbank von JavaScript-Bibliotheken
[12] https://github.com/jslint-org/jslint#quickstart-install

Ausführung JSLint kann in der Konsole mit einer einzelnen zu prüfenden Datei oder einem Ordner als Argument ausgeführt werden: `$ node test/jslint.mjs src/`

Mit dieser Methode wird das Ordnerverzeichnis jedoch nicht rekursiv durchsucht, und es können ebenfalls keine Konfigurationen vorgenommen werden. Dafür muss ein eigenes Hilfsskript geschrieben werden (siehe Quelltext 3.1). Dieses importiert die Funktionalität aus dem Linter und übergibt diesem einige Optionen.

```
import fs from 'fs/promises';
import jslint from './jslint.mjs';

if (process.argv.length <= 2) throw 'expected ...files argument';
// JSLint options
const options = { node: true, single: true, long: true, unordered: true, for: true };

for (const file of process.argv.slice(2))
{
    const source = await fs.readFile(file, 'utf8');
    const result from file = await jslint(source, options);
    if (result from file.warnings.length > 0) continue;

    const warnings = result from file.warnings
        .map(({ formatted message }) => formatted message);
    console.error(`\x1b[1mjslint ${file}\x1b[22m\n${warnings.join('\n')}`);
    process.exit code = 1;
}
```

Quelltext 3.1: JSLint Konsolen-Wrapper

Konfiguration JSLint kann neben der oben beschriebenen Variante durch sog. Code-Direktiven konfiguriert werden. Diese werden direkt durch Kommentare in den Quellcode geschrieben, die der Linter erkennt und dementsprechend reagiert. Es sind nicht viele Optionen verfügbar, da die meisten Regeln direkt im Linter kodiert wurden. Die übrigen erlauben es vorwiegend Warnungen über die Benutzung einiger Schlüsselwörter und Operatoren zu deaktivieren.

Ergebnis JSLint hat hauptsächlich Warnungen über Einrückung, Variablenbenennungen, fehlende oder unnötige Kommas oder Semikolons und weggelassene Klammern bei Pfeilfunktionen und bei kurzen Kontroll-Audrücken ausgelöst. Zudem schien es trotz der vielen unterstützten ECMA-Script 6 Features nicht mit dem OOP-Klassenkonzept mit dem `class`-Schlüsselwort zurechtzukommen. Da viele der Warnungen nicht deaktivierbar und das Tool auch sonst nur begrenzt konfigurierbar ist, war es für diese Arbeit leider nicht brauchbar. Dieses Ergebnis spiegelt sich auch in der Wertung in Tabelle Tabelle 3.3 wieder.

Funktionalität		**Anwendung**	
Formatierung	10/20	Installation	0/10
einh. Stil	20/30	Ausführung	5/30
Probleme	20/40	Konfiguration	10/40
Problemerkennung	0/10	Dokumentation	15/20
Summe	50/100	Summe	30/100

Tabelle 3.3: Anforderungswertung von JSLint

3.2.3 JSHint

JSHint [14] ist eine deutlich nachsichtigere Version von JSLint. [2, S. 153] Es soll JavaScript-Entwicklern dabei helfen komplexe Programme zu schreiben, ohne sich über Tipp- und Sprachfehler Gedanken machen zu müssen. Es unterstützt verschiedene Ausführungsumgebungen, einschließlich modernem ECMA-Script.

Installation und Ausführung JSHint ist mit einem Node-Paketmanager aus dem Npm-Registry installierbar. Nach der Konfiguration der Umgebung kann es dann als Konsolenanwendung verwendet werden:

```
$ yarn add --dev jshint
$ echo '{"esversion":8,"node":true}' > .jshintrc
$ yarn jshint src/
```

Quelltext 3.2: JSHint Installation

Konfiguration JSHint wird durch eine .jshintrc Datei im JSON-Format konfiguriert. Es bietet eine überschaubare Optionenauswahl, von denen die wichtigsten im Beispiel Quelltext 3.3 zu sehen sind. Die Dokumentation unterteilt die Regeln in 'Enforcing'-, 'Relaxing'- und 'Environment'-Optionen und beschreibt jeweils kurz deren Anwendung und Auswirkung.

```
{
    "esversion": 8, "node": true,
    "maxparams": 3, "maxdepth": 3,
    "shadow": true, "nonew": true,
    "unused": true, "undef": true,
    "latedef": "nofunc"
}
```

Quelltext 3.3: JSHint Konfiguration

Ergebnis JSHint unterstützt alle essenziellen Funktionen, um den Code in der verwendeten Umgebung zu analysieren. Es half dabei, den Code an einigen Stellen lesbarer zu gestalten und machte auf potenziell unsichere Variablenzugriffe aus Funktionen aufmerksam. Es war zudem in der Lage, nicht verwendete Variablen und Argumente zu erkennen und damit tatsächlich einen Fehler zu finden. Jedoch konnten absichtlich nicht verwendete Variablen (z. B. Funktionsargumente, die nur der Vollständigkeit halber bei Callback-Funktionen angegeben wurden) nicht ignoriert werden. Weiterhin war es recht aufwändig die vielen vergessenen Semikolons nachzutragen, die in dem Projekt konsistent verwendet werden sollten. An dieser Stelle wäre ein Tool, welches einfache Fehler automatisch beheben kann, sehr hilfreich gewesen. Die daraus resultierende Wertung in Tabelle 3.4 spiegelt diese Schwächen wieder.

Funktionalität		**Anwendung**	
Formatierung	10/20	Installation	8/10
einh. Stil	20/30	Ausführung	25/30
Probleme	30/40	Konfiguration	20/40
Problemerkennung	0/10	Dokumentation	10/20
Summe	60/100	Summe	53/100

Tabelle 3.4: Anforderungswertung von JSHint

3.2.4 ESLint

ESLint [15] ist ein Tool das von JSLint und JSHint inpiriert wurde. Unabhängig von ESLint wurde ein ähnliches Tool namens JSCS[13] entwickelt, welches seit April 2016 mit ESLint fusionierte.[14] Beide Tools verfolgten das Ziel neuartige JavaScript-Linter basierend auf einem AST (Abstract Syntax Tree) zu entwickeln. ESLint fokussierte sich dabei anfangs auf durch Plugins erweitere Regeln, und JSCS auf die automatische Behebung von Fehlern. Durch die Zusammenführung existiert nun ein Tool mit den Vorteilen beider Seiten.

Installation ESLint kann ebenfalls aus dem NPM-Registry installiert werden. Zusätzlich kann ein interaktives Initialisierungstool verwendet werden, um die erste Einrichtung zu vereinfachen. Es lässt den Nutzer die verwendete Umgebung spezifizieren, fragt einige grundlegende Formatierungsregeln ab (siehe Quelltext 3.4) und speichert diese in einer `.eslintrc` Datei ab (siehe Quelltext 3.6).

```
$ yarn add --dev jslint
$ yarn jslint --init # installs @eslint/config
You can also run this command directly using 'npm init @eslint/config'.
✓ How would you like to use ESLint? · style
# ...
✓ How would you like to define a style for your project? · prompt
✓ What format do you want your config file to be in? · YAML
# ...
Successfully created .eslintrc.yml file in /home/.../ide-server
```

Quelltext 3.4: ESLint Installation

Ausführung Die verfügbaren Standard-Regeln bieten bereits ein sehr großes Spektrum zur Erkennung möglicher Logikfehler, Vereinheitlichung des Code-Stils und natürlich der Formatierung. Bereits die erste Ausführung ohne zusätzliche Konfiguration zeigte 603 Probleme auf, von denen 581 als automatisch behebbar aufgeführt wurden. Die Fehler bestanden dabei beinahe ausschließlich aus Einrückung, inkonsistenten Anführungszeichen, fehlenden Semikolons sowie unbenutzten und undefinierten Variablen. Hauptsächlich letztere blieben nach der automatischen Fehlerbehebung übrig (siehe Quelltext 3.5). Undefinierte Variablen wurden meist entweder durch das Electron- oder Enjine-Framework vorgegeben, oder es fehlte die Deklaration mit dem Schlüsselwort `const` `var` oder `let`. Unbenutzte Variablen traten meist bei überschüssigen Imports oder Callback- bzw. Funktionsargumenten auf.

```
$ eslint src/
# ...
× 603 problems (603 errors, 0 warnings)
  581 errors and 0 warnings potentially fixable with the `--fix` option.

$ eslint src/ --fix
# ...
× 22 problems (22 errors, 0 warnings)
```

Quelltext 3.5: Erste ESLint Ausführung

[13] https://jscs-dev.github.io/
[14] https://eslint.org/blog/2016/04/welcoming-jscs-to-eslint/

Konfiguration Für die Konfiguration wurde das weniger Steuerzeichen-lastige `YAML`-Format anstatt `JSON` verwendet. Das vereinfacht das Modifizieren und Schreiben neuer Regeln (siehe Quelltext 3.7)

```
{
    "env": {
        "commonjs": true,
        "es2021": true,
        "node": true
    },
    "extends": "eslint:recommended",
    "parserOptions": {
        "ecmaVersion": "latest"
    },
    "rules": {
        "indent": ["error", 4],
        "linebreak-style": ["error", "unix"],
        "quotes": ["error", "single"],
        "semi": ["error", "always"]
    }
}
```

Quelltext 3.6: .eslintrc im JSON Format

```
env:
  commonjs: true
  es2021: true
  node: true
extends: eslint:recommended
parserOptions:
  ecmaVersion: latest
rules:
  indent: [error, 4]
  linebreak-style: [error, unix]
  quotes: [error, single]
  semi: [error, always]
```

Quelltext 3.7: .eslintrc im YAML Format

Dokumentaiton Der Dokumentation wurden noch viele weitere Regeln übernommen, die nicht durch die `eslint:recommended` Gruppe angewendet wurden. Die Dokumentation ist hierbei unterteilt in 'Possible Problems', 'Suggestions' und 'Layout & Formatting' unterteilt. Zu jeder Regel ist neben einer kurzen Erklärung eine weitere Detail-Seite verlinkt, in der der Nutzen, den Effekt, korrekte und Falsche Codebeispiele bei verschiedenen Optionen der Regel, sowie Limitierungen und wann sie nicht verwendet werden sollte beschrieben wird.

Erweiterungen Weiterhin konnten den Standardregeln noch weitere Regeln durch externe Pakete hinzugefügt werden. Dazu gehörten die Npm-Pakete `eslint-plugin-node`[15] und `eslint-plugin-security`[16], die sich jeweils auf speziellere Aspekte konzentrieren. Es ist somit sogar möglich eigene Regeln zu schreiben und anwenden zu lassen.

Ein Plugin welches ebenfalls in Betracht gezogen wurde war `eslint-plugin-unicorn`.[17] Dieses erweitert die Standardregeln um viele restriktivere Stilrichtlinien. Ein Teil davon waren jedoch bereits in ESLint integriert, oder wurden als nicht essentiell für das Projekt erachtet, sodass es nicht mit aufgenommen wurde.

Ergebnis ESLint überzeugte durch seine Nutzerfreundlichkeit, eine hohe Konfigurierbarkeit und eine übersichtliche Dokumentation mit detaillierten Beschreibungen und Beispielen für alle Optionen. Durch Erweiterungen konnten zusätzlich weitere Aspekte wie Sprachversions-Konformität und Code-Sicherheit abgedeckt werden, und die Definition eigener Regeln ist dem Anwender ebenfalls vorbehalten.

[15] https://www.npmjs.com/package/mysticatea/eslint-plugin-node
[16] https://www.npmjs.com/package/eslint-plugin-security
[17] https://www.npmjs.com/package/eslint-plugin-unicorn

Angesichts dessen blieben bei der Anwendung keine Wünsche offen, was sich in der Bewertung in Tabelle 3.5 widerspiegelt.

Funktionalität		**Anwendung**	
Formatierung	20/20	Installation	10/10
einh. Stil	30/30	Ausführung	30/30
Probleme	40/40	Konfiguration	40/40
Problemerkennung	10/10	Dokumentation	20/20
Summe	100/100	Summe	100/100

Tabelle 3.5: Anforderungswertung von ESLint

3.2.5 Resultat

Viele Regeln hatten keine Auswirkung auf den bestehenden Code, halfen jedoch dabei, die Formatierung und den Code-Stil in der Zukunft auch für andere Entwickler fest zu definieren.

Regeln die eine Änderung bewirkt haben waren beispielsweise solche für Variablenbenennung, -Redeklaration, implizite Typenkonvertierungen, Gleichheits-Operatoren[18] und die einheitliche Verwendung von bestimmten Funktionen wie `return` und `throw`.

Aus den Gesamtwertungen in Tabelle 3.6 geht ESLint als klarer Sieger hervor, und wird von nun an in die Entwicklung als fester Bestandteil integriert. Es ist vorstellbar diesen Test als Akzeptanztest für jede folgende Codeänderung zu etablieren. Mehr dazu in Abschnitt 3.5.

Tool	**Funktionalität**	**Anwendung**
JSLint	50/100	30/100
JSHint	60/100	53/100
ESLint	100/100	100/100

Tabelle 3.6: Gesamtwertung der Linter

[18] In JavaScript wird bei dem aus 'C' bekannten Gleichheits-Operator '==' eine implizite Typenkonvertierung (Type Coercion) durchgeführt, die zu oft angeprangerten Verwirrungen führt. Stattdessen sollte bis auf wenige Ausnahmen der Strict-Equal Operator '===' verwendet werden.

3.3 Unit Tests

Der nächste Schritt ist die Implementierung der Unit Tests in die EnjineIO Serverkomponente. Da die Software jedoch schon existiert aber keine Tests beinhaltet, ist das Schreiben von Unit Tests auf der untersten Funktionsebene nicht sinnvoll. Aufgrund der begrenzten Zeit ist es zudem wünschenswert durch wenige Tests eine möglichst hohe Coverage und damit Vertrauen in die Software zu schaffen. Deshalb sollen die Tests auf einer möglichst anwendernahen Ebene geschrieben werden.

3.3.1 Vorüberlegungen

Für die Unit Tests wird die Server-Komponente als Testobjekt verwendet. Dabei sollen vor allem die vom Anwender am häufigsten Funktionen getestet werden, sodass möglichst schnell eine hohe Coverage der wichtigsten Funktionen erreicht wird. Dies sind zum einen die Konsolen (CLI) -Argumente, über die der Anwender das Serverprogramm extern konfigurieren kann. Zum anderen ist es die REST-API des Servers, über die die Eingaben des Anwenders intern von der IDE an die Server-Komponente weitergeleitet und verarbeitet werden. Außerdem liegt dem ein großer Anteil an Dateisystemarbeit zugrunde, da sowohl Projekte, Bibliotheken und Erweiterungen der IDE darüber verwaltet werden.

Für den Toolvergleich ist es nicht notwendig eine Test-Suite mit vollständiger Test-Abdeckung zu entwickeln. Es reicht aus sich auf einige wesentliche Funktionen zu beschränken, die den Implementierung und die Arbeitsweise der Test-Tools demonstrieren. Dafür wurden einige Tests ausgewählt, die grundlegende Funktionen der Server-Komponente repräsentieren.

CLI Tests

Beim Test der Konsolenargumente kommt es hauptsächlich auf das korrekte Einlesen der Argumente und deren Auswirkung an. Dafür werden einige grundlegende Ausgabefunktionen, sowie das Verhalten der komplexeren Broadcast-Option getestet.

1. Argument-Verarbeitung
   a) Args-Objekt
   Werden die CLI-Optilen korrekt eingelesen, einschließlich positionaler Argumente?
   b) Standard(Fehler)ausgabe
   Funktionieren die Standardausgabe-Funktionen des Servers erwartungsgemäß?
   c) Debug-Level
   Werden die Ausgabemodi `--verbose` und `--quiet` korrekt beachtet?
2. Argument-Funktionen
   a) Minimale Konfiguration
   keine Ausgabe, wenn alle Server-Funktionen deaktiviert sind.
   b) Logfile Umleitung
   mit der `--logfile`-Option werden alle Ausgaben in einen Dateistream geschrieben.
   c) Version
   Gibt die `--version`-Option die Versionsnummer im gewünschten Format aus? Das Programm wird danach vorzeitig beendet.

d) Hilfe
Gibt die `--help`-Option eine Beschreibung zu jedem Befehl aus? Das Programm wird danach vorzeitig beendet.

e) Broadcast
Die Server-Komponente bietet eine `--broadcast`-Option an, die die Verbindungsinformationen des Servers über das lokale Netzwerk broadcastet.
Wird dafür udp4-Socket mit bestimmten Parametern konfiguriert, gestartet und die Informationen gesendet? Enthält die Ausgabe die Internet-Adresse der geöffneten Socket?

API Tests

Bei der REST-API wird zunächst die Erreichbarkeit getestet. Danach sind die wichtigsten Aktionen des Anwenders der Login-Vorgang, sowie das Ausführen einer Anwendung.

1. REST-Anfragen
   a) Dummy
   Antwortet der Server?
2. Login
   a) Pre-Login
   Sind vor dem Login nur bestimmte Endpunkte erreichbar? (`dummy`, `login` und `info`)
   b) Login
   Wird der `login`-Endpunkt mit falschem und korrektem Passwort korrekt behandelt?
   c) Post-Login
   Sind nach dem Login alle Endpunkte erreichbar?
3. App-Ausführung
   a) App ausführen
   Ruft der `run`-Endpunkt die interne Funktion zum Starten einer App korrekt auf?
   b) App ohne installiertes Electron ausführen
   wie a), erwartet aber zusätzlich eine automatischen Nachinstallation von Electron.

Dateisystem Tests

Die am häufigsten benutzten Endpunkte sind Dateisystem-Operationen. Dabei wird ein minimales In-Memory Dateisystem zur Beschleunigung verwendet. (siehe Abschnitt 3.3.2) Von diesem werden grundlegende Lese- und Schreiboperationen von Dateien und Ordnern getestet, sowie komplexere Vorgänge wie das Erstellen eines neuen Projektes.

1. Auflistung
   a) Ohne/Root Pfad
   Überprüft die alleinige Existenz der EnjineIO-Ordner im Home-Pfad
   b) Pfad außerhalb Home
   Pfade die nicht im Home-Ordner liegen, resultieren in einen 'access denied' Fehler.
   c) Ungültiger Pfad
   Pfade, die zu keiner gültigen Datei oder Ordner führen, resultieren in einem 'invalid path' Fehler.

2. Dateierstellung
   a) Datei Speichern
   Das Speichern einer Datei wird durch einen Upload simuliert. Wird die Datei mit korrektem Inhalt am richtigen Pfad gespeichert?
   b) Projekt anlegen
   Überprüft, ob beim anlegen eines neuen Projektes die Templates korrekt kopiert und umbenannt werden.

3.3.2 Implementierung

Tests werden üblicherweise in sog. Test-Suites aufgeteilt, die jeweils Aspekte einer bestimmten Komponente testen. Jeder Test wird dabei nach dem *Build-Operate-Check*-Prinzip [4, S. 127] strukturiert. Zuerst werden die Test-Daten generiert, dann wird eine Operation angewandt und danach wird das Ergebnis überprüft.

CLI Tests

Um die Kommandozeilen-Argumente zu testen wurden diese in zwei Gruppen unterteilt. Einige werden bereits zur Initialisierung angewendet, und andere haben erst einen Effekt, nachdem das Hauptprogramm ausgeführt wurde. Von beiden werden jeweils die Standardausgaben überprüft und bei letzteren zusätzlich die korrekte Benutzung interner Funktionen.

Initialisierung Vor jedem Test wird die Umgebung auf eine minimale Konfiguration eingestellt. Das heißt, dass die Prozessargumente zurückgesetzt und alle Funktionen deaktiviert werden, die die Server-Komponente standardmäßig bereitstellt. Anschließend werden einzelne Prozess-Argumente hinzugefügt, um jeweils eine spezifische Option zu testen.

Validierung Durch sog. Spione (engl. spies) werden Aufrufe einer Funktion überwacht und damit Aufrufanzahl, Reihenfolge und Funktionsargumente validiert. Um die Standardausgaben zu prüfen, werden diese vor jedem Test in separate Zeichenketten umgeleitet.

Beschleunigung Optional kann man originale Implementationen durch einen Dummy ersetzen (engl. implementation mock), wenn diese bereits durch einen Unit Test getestet wurde oder aus einer vertrauenswürdigen externen Quelle stammt. Außerdem können zeitaufwändige Operationen aus dem Test ausgeschlossen werden, um die Laufzeit weiter zu verbessern.

API Tests

Der Enjine-Server ist in der Server-Komponente ein eigenständiges Modul, welches im Hintergrund von der Express-Bibliothek Gebrauch macht. Um den Server zu testen wird für jeden Test eine neue Instanz des Servers erstellt.

Express-Endpunkt-Tests Express-Server empfangen normalerweise Anfragen über einen lokalen Netzwerk-Port. Um die Tests effizienter auszuführen, kann sich die Test-Bibliothek direkt in die Express-Instanz einklinken. Anschließend können die Endpunkte durch diese mit direkten URL-Anfragen getestet und deren Antwort und Nebeneffekte validiert werden.

Validierung Die Server-Komponente antwortet auf API-Anfragen mit einem JSON-Objekt. In Kombination mit dem Header *Content-Type: application/json* ist dieses direkt im Express-Antwort-Objekt verfügbar. Dieses wird dann im Test zusammen mit eventuellen Konsolenausgaben validiert.

Dateisystem Tests

Einen großen Anteil der API nehmen Dateioperationen ein. Sie stellt Funktionen zum Auflisten, Hochladen, Erstellen, Löschen und Umbenennen von Dateien und Ordnern bereit, sowie das Anlegen neuer Projekte aus Vorlagen.

Simulationstools Zum simulieren des Dateisystems im Arbeitsspeicher wurde fs-mock [16] und memfs [17] verwendet. Im Gegensatz zu memfs überschreibt fs-mock die interne NodeJS Dateisystem-Bibliothek fs. memfs bietet stattdessen ein fs-ähnliches Interface an, welches anstelle von fs benutzt werden kann.

Initialisierung Das Dateisystem wird vor jedem Test in einen minimalen Stand gebracht, der mindestens den 'Temp'-Pfad des Betriebssystems enthält, den Pfad zum internen Server-Ordner und einen 'EnjineIO'-Ordner für Projekte des Anwenders. Individuell können jedem Test weitere benötigte Dateien und Ordner hinzugefügt werden.

Validierung Um die entstehende Dateistruktur nach der Operation zu validieren wird der anwenderspezifische 'Home'-Pfad zunächst in einen einheitlichen umbenannt und für den Test irrelevante Ordner entfernt. Anschließend wird das Dateisystem in eine JSON-Struktur umgewandelt und dann mit einem Snapshot-Test[19] validiert.

3.3.3 Resultat

Durch die so entwickelten Tests konnte die Server-Komponente an einigen Stellen verbessert und robuster gestaltet werden. Zum Beispiel wurde der Login-Prozess robuster und für mehrere gleichzeitig laufende Server-Instanzen überarbeitet. Weiterhin wurden bei manchen Sonderfällen neue Fehlerbeschreibungen hinzugefügt, Code vereinheitlicht sowie einige Fehler behoben, die unvollständigen manuellen Tests geschuldet waren.

Testbarkeit

Die Server-Komponente musste an einigen Stellen verändert werden, um den Code testen zu können, d. h. testbar zu machen. Einher gingen einige Umstrukturierungen, wie das Trennen der Initialisierung und Konfiguration des Programms vom Starten des Servers.

[19] Speichert einen Wert bei der ersten Ausführung in eine separate Datei, welcher dann bei jeder folgenden Ausführung mit dem neuen verglichen wird. Findet häufig bei komplexen Datenstrukturen wie DOM-Strukturen Anwendung

Exportieren von Hauptfunktionen Außerdem muss sowohl die Server-Komponente als auch das dahinterliegende Server-Modul Funktionen exportieren, die von den Tests verwendet werden können. Dazu gehören solche zum Erstellen, Starten und Beenden einer jeweiligen Server-Instanz. Außerdem müssen Objekte und Funktionen, deren Verhalten überwacht oder überschrieben werden sollen ebenfalls exportiert werden. (siehe Absatz 3.3.2 Beschleunigung)

Dynamische Abhängigkeiten Ein Problem, welches sich durch Zeitmessungen der Tests bemerkbar machte, waren dynamische Abhängigkeiten. Teilweise wurde die Ausführungszeit durch Skripte verfälscht, die erst nach Anfrage einer bestimmten Operation zur Laufzeit des Servers geladen wurden. Deswegen wurden dynamische Abhängigkeiten aus vielen Stellen der Server-Komponente entfernt, bzw. in einem zeitunkritischen Dummy-Test geladen.

Ausgaben Zuletzt wurden alle Ausgaben des Servers durch ein zentrales Modul geleitet, welches für die Tests überschrieben werden kann, ohne die gesamte Ausgabe der Tests zu blockieren. So kann jeder Test aus dem gleichen Prozess heraus gestartet werden, anstatt für jeden Test einen neue Prozess-Instanz der gesamten Server-Komponente zu erstellen.

3.4 Unit Test Frameworks

Ein Test-Framework stellt ein standardisiertes Interface für die Definition und Ausführung von Softwaretests bereit.

3.4.1 Anforderungen

Die beschriebenen Tests dienen gleichfalls als Dokumentation zur Benutzung der Software. Deshalb sollten sie gut strukturierbar und intuitiv zu lesen sein.

Testautomatisierung Für Continuous Integration ist dabei speziell die Automatisierung von Tests von Interesse. Um die Ausführungszeit der CI-Pipeline zu minimieren, müssen Fehler möglichst frühzeitig erkannt werden. Deshalb soll das Framework die Tests schnell und effizient ausführen, wenn sinnvoll durch Parallelisierung unterstützt. Die Abdeckung der Tests wird dabei durch ein Code-Coverage-Tool überprüft und somit nicht erreichte Teile des Codes zu identifizieren und die Test ggf. entsprechend zu erweitern.

Geschwindigkeit Um die Tests auf einer hohen Ebene in einer passablen Zeit durchführen zu können, müssen Schnittstellen zu Komponenten außerhalb der Software simuliert werden. Dazu gehört in diesem Fall vorrangig das Dateisystem und das Netzwerk. Das Dateisystem ist eine limitierte Ressource, auf die vergleichsweise nur langsam zugegriffen werden kann. Zudem variiert der Aufbau je nach Betriebssystem und Anwender. Deshalb sollen die Tests mit einem simulierten Dateisystem im Arbeitsspeicher und standardisierten Pfaden durchgeführt werden. Ähnlich verhält es sich mit dem Netzwerk. Die Server-Endpunkte sollen möglichst direkt und ohne den Gebrauch des Netzwerk-Interfaces getestet werden.

Die Vergleichskriterien der Unit-Test-Tools werden nach Tabelle 3.7 gewichtet.

Funktionalität		**Anwendung**	
Module Mocks	40	Einrichtung	20
Lesbarkeit	40	Bedienung	30
Coverage	10	Geschwindigkeit	30
Parallelität	10	Dokumentation	20
Summe	100	Summe	100

Tabelle 3.7: Anforderungswichtung der Unit Test Frameworks

3.4.2 mocha

Das erste Test-Framework, was an dem Projekt ausprobiert wurde, war 'Mocha'. [18] Mocha ist ein Framework, welches meist nur zusammen mit anderen Modulen verwendet wird. Hier wird die oft in Kombination genutzte 'Chai-Assertion-Library' [19] verwendet, sowie 'Chai-Spies' [20] zum Abfangen von Funktionsaufrufen, das auf Supertest basierende 'Chai-Http' [21] zum Testen des Express-Servers, 'mock-fs' [16] für ein simuliertes Dateisystem und 'Snap-Shot-It' [20] zum Erstellen von Snapshot-Tests. Für Coverage Analysen muss ein externes Tool wie z. B. das Istanbul-basierte 'nyc' [21] verwendet werden.

[20] https://github.com/chaijs/chai-spies
[21] https://github.com/chaijs/chai-http

Syntax Die Chai-Bibliothek stellt verschiedene Methoden zum beschreiben von Assertions bereit. Die verbreitetste ist dabei die expect-Form, die zunächst den zu prüfenden Wert übergeben bekommt, gefolgt von einer Kette an Wörtern, die die erwartete Bedingung beschreibt:

```
expect(variable).to.equal(value);
```

Diese können sogar Logisch durch `and`, `or` und `not` verknüpft werden. Somit können gut lesbare Sätze formuliert werden, die die zu testenden Variablen beschreiben.

Dateisystem Tests In Mocha wurde die Bibliothek mockfs verwendet, um ein Dateisystem im Arbeitsspeicher zu simulieren. Es genügt ein simpler Aufruf der Funktion, um die internen Dateisystem-Funktionen durch mockfs zu überschreiben. Optional kann eine Dateistruktur im JSON-Format übergeben werden, die den initialen Zustand des Dateisystems beschreibt (siehe Quelltext 3.8).

```
// test suite
describe('server fs', function()
{
    // override fs before each test, and restore afterwards
    beforeEach(() => mockfs({ [session.user.dir]: {} }));
    afterEach(() => mockfs.restore());

    // test
    it('list /', async () =>
    {
        // action
        const res = await chaiReq.get('/ide?cmd=list&dir=/');

        // result assertion
        expect(res).to.have.status(200);
        expect(res.body).to.eql({ status: 'ok', list: [] });

        // fs assertion
        const vol = unifyRestoreMockfs();
        matchSnapshot(vol);
    });
});
```

Quelltext 3.8: Mock-fs Hooks

Snapshot-Tests Das Listing zeigt auch, wie die resultierende Dateisystem-Struktur validiert wird. Dazu kopiert die Funktion unifyRestoreMockfs zunächst das aktuelle Dateisystem als JSON-Objekt und gibt dieses in allgemeingültige Pfade umgewandelt zurück. Außerdem wird das interne FS-Modul zum Original zurückversetzt, damit der anschließende Snapshot-Test auf das reale Dateisystem zugreifen kann. Die Umwandlung in ein testbares JSON Objekt war dabei etwas aufwändig, da mock-fs darin auch alle Metainformationen der Dateien speichert. Diese müssen herausgefiltert werden, um lediglich die Struktur und den Inhalt zu testen.

Ausführung Mocha zeichnet sich durch seine hohe Geschwindigkeit in der Testausführung aus. Die komplette entstandene Test-Suite wird innerhalb von ca. 650ms abgearbeitet (mit minimaler Ausgabe sogar nur 380ms) und einer Gesamtlaufzeit des Prozesses von 1,8 bzw. 1,5 Sekunden. Es ist auch möglich die Tests durch die '`--parallel`'-Option auf mehrere Prozesse aufzuteilen. Das Verlangsamt die Gesamtlaufzeit allerdings um ca. eine halbe Sekunde.

Ausgabe Die Ausgabe des Frameworks wird durch den Test-Reporter bestimmt. Der Standard-Reporter gibt der Reihe nach den Namen jeder Suite und dementsprechend eingerückt die Tests und deren Status aus (siehe Quelltext 3.9). Ist ein Test fehlgeschlagen, wird in den meisten Fällen der erwartete und der tatsächliche Wert ausgegeben, komplexere Objekte im Diff-Format. Zum Schluss wird die Gesamtlaufzeit der Tests ausgegeben.

```
$ TEST=1 mocha --recursive

  cli pre startup
    ✓ stdout
    ✓ log info
    ✓ log silent
    ✓ log verbose
    # ...

34 passing (646ms)
Done in 1.76s.
```

Quelltext 3.9: Mocha Output

Die Coverage-Ausgabe von nyc veranschaulicht dem Anwender für jede Quelldatei aufgeschlüsselt nach Statement, Branch, Function und Line-Coverage, welche Code-Teile durch die Tests erreicht wurde. (siehe Quelltext 3.10)

```
---------------------|---------|----------|---------|---------|------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|------------------
All files            |   56.79 |    45.16 |   49.75 |    61.3 |
 src                 |   85.71 |    76.92 |   68.75 |   91.45 |
  ide.js             |   81.73 |    77.41 |   84.61 |   84.84 | 49-50,117,130-134
  main.js            |   89.43 |    76.19 |   57.89 |      98 | 38,205
 src/commands        |    34.5 |    28.09 |   28.57 |   37.85 |
  cmd ide.js         |   55.55 |    44.15 |   52.17 |   63.55 | 126-134,167-258
  cmd sys.js         |     100 |      100 |     100 |     100 |
  cmd usr.js         |   10.08 |        0 |       0 |   11.32 | 33-231
 # ...               |         |          |         |         |
---------------------|---------|----------|---------|---------|------------------
```

Quelltext 3.10: nyc Coverage Output

Resultat Mocha überzeugt bereits wegen seiner hohen Ausführungsgeschwindigkeit. Es kann zudem durch seine Modularität vielfältig eingesetzt und mit verschiedensten Bibliotheken kombiniert werden. Einher geht aber ein größerer Einrichtungsaufwand, was auch die Wertung in Tabelle 3.8 widerspiegelt. Die Chai-Bibliothek erlaubt viele Freiheiten in der Formulierung der Tests und stellt von vorn herein viele hilfreiche Überprüfungsmethoden bereit. Durch die Erweiterungen chai-spies und chai-http konnten problemlos Funktionsaufrufe überwacht und überschrieben, sowie der Express-Server getestet werden. Durch mock-fs war es sehr einfach möglich, das Dateisystem zu simulieren. Die Umwandlung in eine überprüfbare Struktur war jedoch etwas aufwändig. Das Coverage-Tool musste ebenfalls extern nachinstalliert werden, funktionierte daraufhin aber einwandfrei.

Funktionalität		Anwendung	
Module Mocks	30/40	Einrichtung	5/20
Lesbarkeit	40/40	Bedienung	20/30
Coverage	10/10	Geschwindigkeit	30/30
Parallelität	10/10	Dokumentation	20/20
Summe	90/100	Summe	75/100

Tabelle 3.8: Anforderungswertung von Mocha

3.4.3 Jest

Jest [22] ist ein Framework, welches viele häufig verwendete Features eines Test-Frameworks in sich integriert hat. Es unterstützt Mocking, Spione und Snapshot-Tests ohne extra Bibliothek, kann ohne zusätzliche Abhängigkeiten Coverage-Analysen durchführen und führt Tests standardmäßig in parallelen Prozessen aus. Zusätzliche wurden nur memfs [17] für Dateisystem-Mocks, und Supertest [23] für Express-Server Tests benötigt.

Syntax Die Jest-Tests wurden basierend auf den existierenden Mocha-Test umgeschrieben. Die Assertion-API unterscheidet sich zu Chai dadurch, dass sie einzelne Funktionen zum Überprüfen bereitstellt, anstatt von verketteten Wörtern. Die Umwandlung war dabei relativ unkompliziert, bis auf den Test auf die Existenz bestimmter Eigenschaften in einem Objekt, der einen kleinen Umweg über eine Jest-Util Funktion benötigte.

Dateisystem Tests Die Umwandlung von mock-fs zu memfs gestaltete sich etwas schwieriger. Jest bietet eine Methode, um gesamte Module durch ein eigenes zu ersetzen. Da memfs ein dem internen NodeJS fs-Modul ähnliches Objekt bereitstellt, bot es sich an, dieses dadurch zu ersetzen. Das Problem dabei ist jedoch, dass das fs-Modul nur für zukünftige Modul-Imports im Import-Cache[22] ersetzt wird. Das Server-Modul wird allerdings global importiert und daraus Instanzen erstellt, sodass es notwendig war, die fs-Methoden manuell vor und nach jedem Tests durch die des memfs-Moduls zu ersetzen. Daraufhin funktionierten die Dateisystem-Tests in den meisten Fällen. Manchmal schien jedoch eine Race-Condition[23] aufzutreten, in der zufällig ein interner Fehler im Server oder aufgrund fehlender Dateien auftrat und der Test deshalb fehlschlug.

Ausführung Jest stellte sich als um einiges langsamer als mocha heraus. Das liegt darin begründet, dass Jest den Quellcode zunächst zu regulärem JavaScript transformieren lässt,[24] u. a. um erweiterte Sprach-Features wie Typescript in Coverage-Analysen unterstützen zu können. Die Ausführung aller Tests benötigte mit Jest ca. 4 Sekunden, der gesamte Prozess noch etwa eine Sekunde länger.

Ausgabe Jest zeigt im Unterschied zu mocha den Fortschritt jeder Test-Datei, und nicht die der Tests selbst an. Somit ist die Struktur der Tests nur durch die Dateistruktur, und nicht durch die Deklaration der Suites bestimmt. Nur im Fehlerfall wird der genaue Pfad zu dem spezifischen Test ausgegeben. Leider schienen die Angaben der Zeilennummer nicht mit

[22] Importierte Module werden nach dem ersten laden in einem Cache gespeichert, sodass zukünftige Modul-Imports sofortige Zugriff darauf haben
[23] (Fehler-)Zustand, der abhängig vom zeitlichen Verhalten bestimmter Einzeloperationen eintritt
[24] https://jestjs.io/docs/next/code-transformation

dem tatsächlichen Code übereinzustimmen. Die Coverage-Ausgabe unterschied sich wenig zu der von nyc bei mocha, außer dass sich ebenfalls Zeilennummern und einige Prozentwerte unterschieden. Beides könnte an der Code-Transformation liegen, die Jest vor der Ausführung der Tests durchführt.

Resultat Jest ist ein sehr einsteiger-freundliches Framework, welches kaum zusätzliches Setup benötigt und sogar das Coverage-Tool integriert hat. Allerdings ist es bedeutend langsamer als die Alternative Mocha gewesen. Dafür führt es die Tests von vorn herein parallel aus. Die Test-API ist nicht so flexibel in der Beschreibung der Tests wie chai, was jedoch bis auf eine Testart die nicht direkt verfügbar war keinen spürbaren Nachteil hatte. Daraus ergab sich die Wertung in Tabelle 3.9. Die Dateisystemtests bereiteten bei der Einrichtung kleine Schwierigkeiten, jedoch gab memfs direkt eine testbare JSON-Struktur zurück. Die Server-Endpunkt-Tests konnten recht einfach zu Supertest umgewandelt werden, da chai-http ebenfalls darauf basiert.

Funktionalität		**Anwendung**	
Module Mocks	20/40	Einrichtung	15/20
Lesbarkeit	35/40	API-Bedienung	25/30
Coverage	10/10	Geschwindigkeit	5/30
Parallelität	8/10	Dokumentation	20/20
Summe	73/100	Summe	65/100

Tabelle 3.9: Anforderungswertung von Jest

3.4.4 vitest

Vitest [24] ist ein sehr neues, aber vielversprechendes Test-Framework des Vite-Frameworks. Es ist kompatibel zu Jest, da es dieselbe Assertion-API verwendet, mit einigen Erweiterungen von Chai und TinySpy für Mocks. Tests werden ebenfalls standardmäßig parallel ausgeführt, jedoch optional auch sequentiell. Die eingebaute Coverage-Option benötigt das extern installierte Coverage-Tool 'c8'. [25] Zudem erwartet es, dass die Tests ES-Module (ESM) sind. Zuvor wurden die Tests für Mocha und Jest in CommonJS (CJS) geschrieben.

ESM-First Der Hauptunterschied zu CJS ist die Abhängigkeiten-Auflösung, die zuvor synchron und mit einem veränderbaren Cache einherging, in ESM's jedoch asynchron abläuft und unveränderbare, isolierte Module importiert. Das ist vor allem für Webseiten relevant, die Abhängigkeiten möglichst effizient laden können sollen. ESM's werden dementsprechend bereits in vielen Browsern unterstützt, angewendet und sollen zukünftig Standard für die NodeJS Entwicklung werden. Im Desktop-Bereich ist dieser Umstieg jedoch weniger Relevant. Glücklicherweise ist es möglich, aus ES-Modulen CJS Module zu importieren, weshalb durch Wrapper-Module relativ einfach beide Varianten angeboten werden können.

ESM-Konvertierung Das Umschreiben der Tests zu ESM stellte vor allem wegen der isolierten, unveränderlichen Module ein Problem dar. Da in den Tests Funktionen und Objekte zugunsten der Laufzeit modifiziert werden, behinderte diese ESM-Eigenschaft die Tests. Aus diesem Grund wurden viel mehr Objekte aus der Server-Komponente heraus exportiert.

Dadurch bekamen die Tests Zugriff auf dessen Instanzen und Abhängigkeiten, anstatt diese selbst zu importieren. Die Syntax für Imports ist leicht verschieden, stellte aber kein Problem dar, solange diese alle in den globalen Teil an den Anfang des Skriptes verschoben wurden.

Dateisystem-Tests Hier wurde zunächst wieder zurück auf das Modul mock-fs ausgewichen, da die verwendete Methode zur Funktionsüberschreibung nun den Upload-Endpunkt des Express-Moduls störte. Die Ursache wurde erst später entdeckt, da scheinbar eine dynamische Abhängigkeit durch das Überschreiben des Dateisystems nicht geladen werden konnte. Diese wurde dann manuell dem virtuellen Dateisystem hinzugefügt und dadurch das Problem behoben, sodass sowohl memfs als auch mock-fs anwendbar waren.

Ausführung Vitest unterschied sich in der Geschwindigkeit eines gesamten Testdurchlaufs Ausführung nicht von Jest. Laut Anwenderberichten zeigt sich Vitests Stärke in einem schnelleren Watch-Mode[25] aufgrund von intelligenten Code-Abhängigkeitsanalysen.

Resultat Bei Vitest war das größte Problem die Konvertierung der Tests in ES-Module, was auch die Wertung in Tabelle 3.10 zeigt. Ansonsten wurde die Test-API von Jest vollständig unterstützt, sodass die Tests kaum umgeschrieben mussten. Sogar besagte fehlende Testart (siehe Absatz 3.4.3 Syntax) wurde in Vitest direkt unterstützt. Die Geschwindigkeit eines kompletten Testdurchlaufs unterschied sich ebenfalls nicht. Das Coverage-Tool musste wie bei mocha nachinstalliert werden, konnte dann aber direkt durch eine Option angewendet werden. Das Dateisystem konnte sowohl mit mock-fs als auch mit memfs mit einem kleinen zusätzlichen Einrichtungsschritt mit bereits beschriebenen kleineren Schwierigkeiten.

Funktionalität		**Anwendung**	
Module Mocks	30/40	Einrichtung	5/20
Lesbarkeit	40/40	API-Bedienung	25/30
Coverage	10/10	Geschwindigkeit	5/30
Parallelität	10/10	Dokumentation	20/20
Summe	90/100	Summe	55/100

Tabelle 3.10: Anforderungswertung von Vitest

3.4.5 Resultat

Die Gesamtwertung in Tabelle 3.11 zeigt, dass alle vorgestellten Tools ihre Stärken und Schwächen haben. Mocha war sehr schnell, funktioniert jedoch nur mit einer Reihe verschiedener Erweiterungen die es nochmals auszuwählen und in sie einzuarbeiten gilt. Jest bietet dafür ein Komplettpaket an Funktionen, ist jedoch um ein vielfaches langsamer als Mocha. Vitest ist sehr ähnlich zu Jest und füllt einige seiner Lücken, dafür ist jedoch die Verwendung von ES-Modulen sehr unpraktisch und inkonsistent zum Rest des Projektes.

Ausschlaggebend für die spätere Ausführung in der CI Pipeline ist jedoch die Geschwindigkeit. Zudem ist der Einrichtungsaufwand für die zukünftige Anwendung nicht wesentlich hinderlich. Im Gegenteil kann es sogar förderlich sein die Möglichkeit zu haben, die passendere Erweiterung für sein Projekt zu finden, solange man die Zeit dazu hat. Deshalb wurde sich im

[25] Modus, in dem das Test-Framework dauerhaft aktiv ist und auf Änderungen im Quellcode wartet, um dann alle durch die Änderung beeinflussten Tests zu starten

Tool	Funktionalität	Anwendung
Mocha	90	75
Jest	73	65
Vitest	90	55

Tabelle 3.11: Gesamtwertung der Unit Test Tools

Rahmen der CI für Mocha entschieden. Es ist jedoch dennoch denkbar, dass kompliziertere Test außerhalb von CI zukünftig auch mit einem anderen Framework möglich wären. Die finale Entscheidung darüber wird erst mit den UI Tests fallen.

3.5 CI Automation

In den vorherigen Abschnitten wurden Tools für die einzelnen Schritte untersucht. Diese führten verschiedene Tests durch, um die Qualität der Software zu gewährleisten. Der letzte Schritt zu CI ist nun die automatische Ausführung dieser Test-Schritte in einer unabhängigen Umgebung.

Ausgangssituation EnjineIO verwendet aktuell Bitbucket als primären Quellverwaltungsservice. Dieses unterstützt CI durch die sog. Bitbucket Pipelines. Es sind aber auch andere CI-Tools mit Bitbucket verknüpfbar. Deshalb werden in dieser Arbeit mit Buddy und CircleCI auch zwei weitere Tools untersucht, die mit Bitbucket kompatibel sind.

3.5.1 Vorüberlegungen

Für die Auswahl der CI Tools gibt es viele Einflussfaktoren. Zunächst sollten sie eine Integration für Bitbucket anbieten. Da EnjineIO zunächst ohne zusätzliche Kosten auskommen soll gilt es dann die kostenlosen Angebote zu vergleichen und wie hoch der Einrichtungsaufwand bei einer vorgegebenen Pipeline ist.

Private Abhängigkeiten Den Zugriff auf das Repository lässt sich meist bereits durch die CI-Integrierung einrichten. Die Server-Komponente macht jedoch zusätzlich von weiteren (privaten) Repositories Gebrauch. Spätestens dann muss man diesen einen sog. Access-Key in Bitbucket hinzufügen der der Pipeline Lesezugriff gewährt (siehe Abbildung 3.1). Dafür wird mithilfe des Konsolen-Tools 'ssh-keygen' ein neues SSH-Schlüsselpaar generiert (siehe Quelltext 3.11). Der öffentliche SSH-Schlüssel wird als Access-Key der privaten Repositories hinterlegt und der private der Pipeline hinzugefügt.

```
# generate keys (on local machine)
# generate id rsa key pair (without passphrase!)
$ ssh-keygen <<< ''
# print public key for bitbucket
$ cat id rsa.pub
# print base64 encoded private key for pipeline var
$ base64 -w0 id rsa
# save host fingerprints
$ ssh-keyscan bitbucket.org > known hosts

# decode keys (in pipeline)
# write private key with 600 permissions
$ (umask 177; base64 -d <<< "$SSH KEY" > ~/.ssh/id rsa)
# append host fingerprints
$ cat known hosts >> ~/.ssh/known hosts
```

Quelltext 3.11: Bash-Befehle zur Erstellung und Einrichtung von BitBucket Access-Keys

Ablauf Nach der Einrichtung wird die eigentliche Pipeline ausgeführt. Diese beginnt meist mit dem Herunterladen der aktuellsten Projektversion aus der Quellverwaltung und anschließend mit der Auflösung und Installation aller (öffentlichen und privaten) Abhängigkeiten. Bei diesem Schritt wird meist ein Cache eingesetzt, um die Installation der Abhängigkeiten zu beschleunigen.

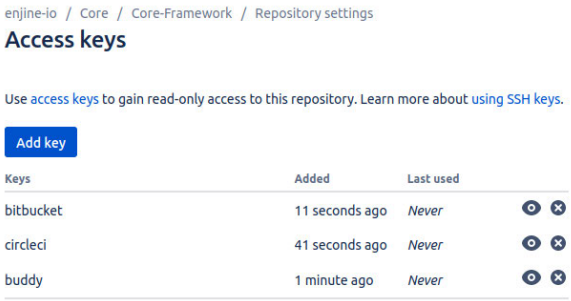

Abbildung 3.1: SSH-Keys für Lesezugriff auf Repositories

Upload Nach erfolgreicher Ausführung aller Tests wird die Software als ausführbare Datei zum Download bereitgestellt. Dafür bietet Bitbucket in der Repository-Übersicht eine Sektion für Downloads, die mithilfe einer REST-API verwaltet wird. Für den Zugriff muss man zunächst in den Profileinstellungen des Bitbucket-Accounts ein sog. App-Passwort erstellen (siehe Abbildung 3.2) und diesem Schreibrechte auf Repositories gewähren. Dieses kann dann in der Pipeline für die Authentifizierung verwendet werden.

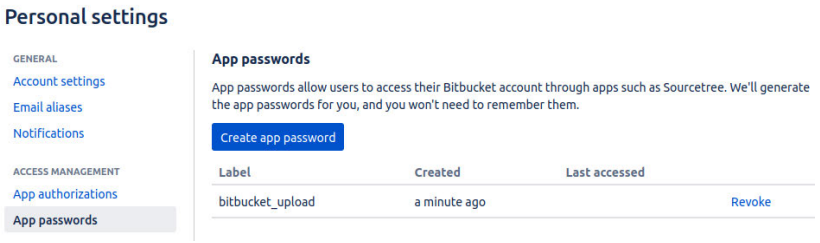

Abbildung 3.2: Einrichtung eines BitBucket App-Passwortes mit Schreibrechten

Quelltext 3.12 zeigt den Hochladeprozess unter Verwendung des Datentransferprogramms 'curl'. CI-Umgebungsvariablen stellen dabei weitere Parameter ein:

USER Bitbucket-Nutzername

ACCESS_TOKEN generiertes App-Passwort des Bitbucket-Nutzers

WORKSPACE Name des Bitbucket-Projekts in dem sich das Repository befindet

REPO_SLUG URL-freundlicher Name oder UUID des Repositories

GLOB Glob-Pattern der hochzuladenden Dateien (Aufruf: `$ GLOB=dist/* ./upload.sh`)

```
#!/bin/bash

# upload url
api_host="https://$USER:$ACCESS_TOKEN@api.bitbucket.org"
api_url="$api_host/2.0/repositories/$WORKSPACE/$REPO_SLUG/downloads"

# file list
files=($GLOB)
flist=${files[*]/#/-F files=@}

# curl upload command
curl -f -X POST $api_url $flist || exit 1
```

Quelltext 3.12: Dateiupload zu Bitbucket Repository Downloads via curl

3.5.2 Anforderungen

Im Folgenden werden die CI-Tools auf Grundlage der zuvor beschriebenen Vorüberlegungen miteinander verglichen. Dabei geht es hauptsächlich um den Einrichtungsaufwand der Bitbucket-Integration, der privaten Zugriffsrechte und der Pipeline-Konfiguration. Dabei spielt abermals die Dokumentation der Tools eine Rolle. EnjineIO setzt zudem vorerst auf frei nutzbare Software, sodass auch die Angebote der freien Nutzungspläne verglichen werden.

Tabelle 3.12 zeigt die Wichtung der Anforderungen und die Free-Plan Elemente. Der Punkt Hilfestellungen fasst dabei Unterstützungen der Online-Tools während der Konfiguration zusammen. Das beinhaltet beispielsweise Vorlagen, Voreinstellungen oder grafische Konfigurationselemente. Caching und SSH-Keys spiegelt wieder, wie einfach die Einrichtung der notwendigen Caches und SSH-Keys war. Der letzte Punkt SSH Debugging gilt speziell für Möglichkeiten sich mit fehlgeschlagenen Containern zu verbinden um die Problemursache festzustellen und sie zu beheben.

Funktionalität	
Hilfestellungen	30
Caching	30
SSH-Keys	30
SSH Debugging	10
Summe	100

Anwendung	
Einrichtung	20
Konfiguration	50
Dokumentation	30
Summe	100

Free-Plan Bestandteile	
Build-Zeit	Laufzeitlimitierung
OS-Support	unterstützte Betriebssysteme
Ressourcen	Hardware & Speicher
Parallel	Simultan ausführbare Jobs

Tabelle 3.12: Anforderungswichtung der CI-Tools

3.5.3 BitBucket Pipelines

Bitbucket bietet durch seine Bitbucket Pipelines eine eigene CI-Lösung an. Diese ist im Online Repository-Übersicht im Untermenü 'Pipelines' auswählbar. Um Pipelines anlegen zu können wird ein Nutzerkonto mit 2-Faktor Authentifizierung benötigt und es muss in den Repository-Einstellungen unter 'Pipelines' die Funktion aktiviert werden. Dort hat man die Möglichkeit eines von verschieden Templates auszuwählen und dieses dann seinen Ansprüchen entsprechend weiter anzupassen. Die Pipelines werden mit einer einzelnen 'bitbucket-pipelines.yml' Datei im Hauptverzeichnis des Repositories konfiguriert. In Abbildung 3.3 ist der Online-Editor zu sehen, in dem man YAML-Vorlagen für weitere parallele, manuelle oder Deployment-Schritte einfügen, sowie schützbare Umgebungsvariablen definieren kann. Außerdem werden sogenannte 'Pipes' für verschiedenste Zielplattformen bereitgestellt, die CI-Schritte wie Hochladen, Bereitstellen von Software oder Funktionalität externer Plattform-Integrationen vorimplementieren. Für weitergehende Schritte steht dem Anwender eine ausführliche Dokumentation zur Verfügung.

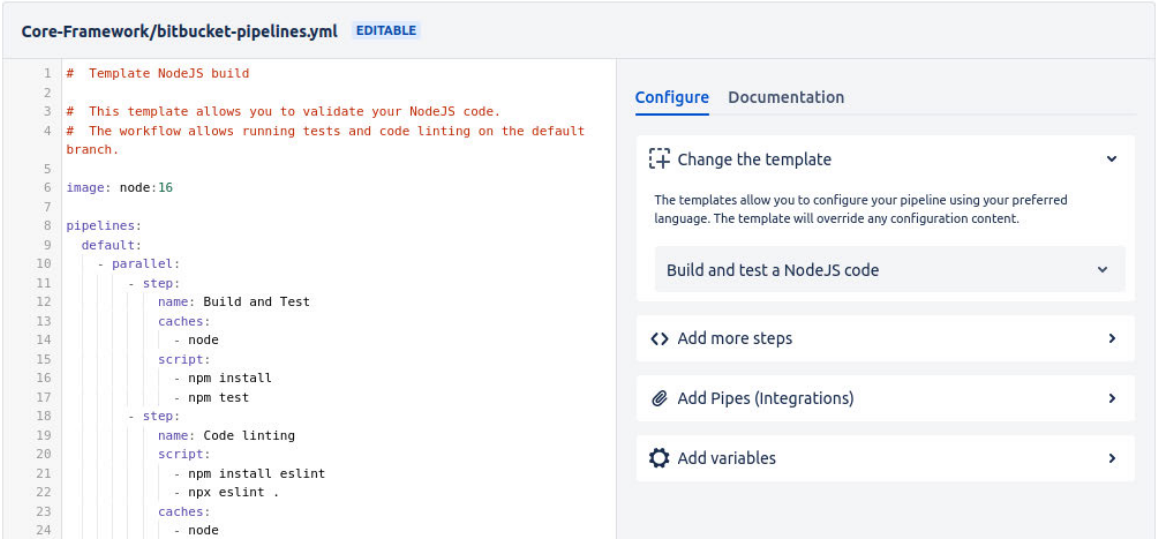

Abbildung 3.3: Bitbucket Pipelinekonfiguration

Konfiguration Die Konfigurationsdatei beginnt mit der Angabe des gewünschten Basis-Docker-Images (in diesem Fall node:16 für die Node.js Version 16). Es folgen Einstellungen zum Klonen des Repositories und Definitionen eigener Cache-Pfade und wiederverwendbaren CI-Steps. Im Anschluss werden die Pipelines kategorisiert nach verschiedene Auslöser-Ereignissen definiert. Es gibt dabei u. A. die Ereignis-Typen 'Default' für alle neuen Commits, 'Branch' und 'Tag' für Commits auf einen bestimmten Branch bzw. mit einem Git-Tag und 'Custom' für rein manuell ausführbare Pipelines in der Online-Ansicht. In der Definition der Pipeline erfolgt u. A. die Angabe des Namens, der Skript-Befehle, der verwendeten Caches und der maximalen Ausführungszeit. Die Konfiguration könnte dann wie in Quelltext 3.13 aussehen.

```yaml
image: node:16
clone: { enabled: true, lfs: false, depth: 1 }

definitions:
  caches: { yarn-global: /usr/local/share/.config/yarn/global }
  steps:
    - step: &test
        name: Install > Lint > Test
        caches: [node, yarn-global]
        script:
          - (umask 177; echo "$SSH KEY" | base64 -d > ~/id rsa)
          - GIT SSH COMMAND='ssh -i ~/id rsa' yarn install
          - yarn test
          - yarn dist
          - GLOB='dist/*' bash upload.sh

pipelines:
  branches: { bb pipes: [step: *test] }
```

Quelltext 3.13: Bitbucket-Pipelines Konfiguration

SSH-Keys Bitbucket Pipelines stellt keine vorgefertigte Lösung für das Anlegen von SSH-Schlüsseln bereit, erklärt in der Dokumentation jedoch die notwendigen Schritte.[26] Nach dem erstellen der Schlüsselpaare in Absatz 3.5.1 Private Abhängigkeiten muss der private

[26] https://support.atlassian.com/bitbucket-cloud/docs/variables-and-secrets/

Schlüssel als gesicherte Variable hinterlegt werden. Dafür wird dieser Base64 kodiert, im Pipeline-Skript dekodiert und in einer Datei abgespeichert. Im Anschluss muss diese Datei noch als SSH-Identität registriert werden, was mit dem Überschreiben des Git-SSH-Befehls gelöst wurde:

```
$ GIT SSH COMMAND='ssh -i ~/id rsa'yarn install
```

Ausführung Die Test-Pipeline wurde hier automatisch für Commits auf den 'bb_pipes' Branch angesetzt. In der Online-Oberfläche sind nach dem Neuladen des Pipeline-Tabs die laufende und alle vergangenen Ausführungen und Logs einsehbar. Die in Abbildung 3.4 gezeigte Übersicht einer Ausführung beschränkt sich auf die Commit-Informationen, einer Schaltfläche zum abbrechen bzw. Neustarten einer Pipeline und einem Konsolenausgabefenster mit gruppierten Befehlsausgaben. In einem zweiten Tab sind optionale in der Pipeline generierte Artefakte herunterladbar. In diesem Fall wurde aber von der Download-Sektion des BitBucket-Repositories Gebrauch gemacht um keinen Extra-Speicher zu verbrauchen (siehe Absatz 3.5.1 Upload).
Ein kompletter Test-Durchlauf mit eslint und mocha dauert in Bitbucket ca. 19 Sekunden. Zum aktuellen Stand werden ca. 8.5MB Cache beansprucht, welcher die Ausführungszeit auf 11 Sekunden reduziert. Einen großen Anteil nehmen dabei jedoch die Einrichtung und Freigabe des Containers vor und nach der Pipeline ein. Die Tests selbst benötigen gerade mal 4 Sekunden. Die Erstellung und Upload der ausführbaren Dateien beansprucht für Linux 30 und für alle alle Betriebssysteme 40 Sekunden.

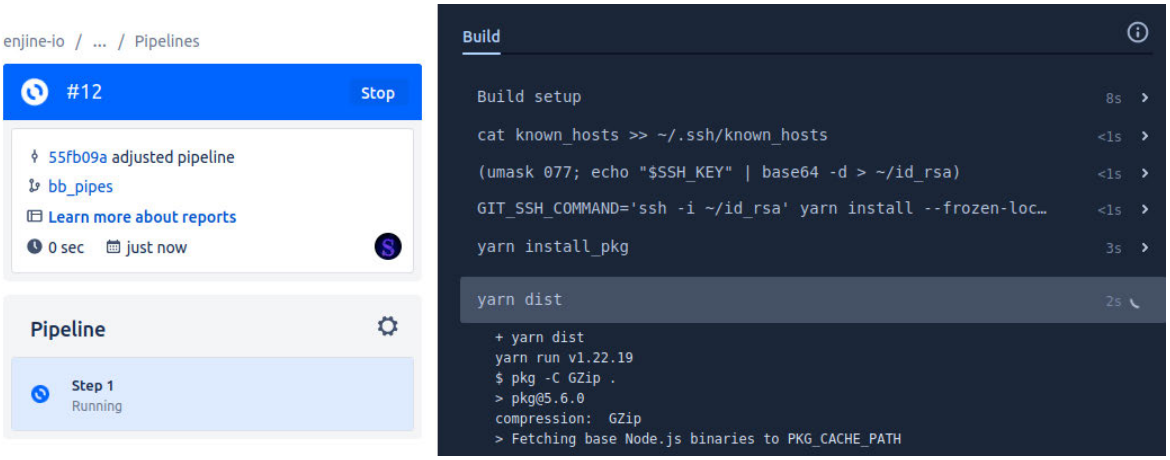

Abbildung 3.4: Bitbucket Pipelineausführung

Free-Plan Bitbucket stellt für jede Projektgruppe einen Nutzungsplan zur Verfügung. In der kostenlosen Variante sind bis zu 5 Nutzer, 1GB Repository-Speicher und 50 kostenlose Pipeline Build-Minuten pro Monat enthalten. Diese stehen für alle Repositories gemeinsam zur Verfügung. Bis zu 10 CI-Steps können parallel in Docker-Containern ausgeführt werden. Jeder Step hat dabei standardmäßig 4GB Arbeitsspeicher zur Verfügung.[27] Zudem müssen in der Pipeline verwendete Caches komprimiert kleiner als 1GB sein. Der kostenpflichtige Standard-Plan ist dennoch erwähnenswert, da er nur 3$ pro Monat kostet. Durch diesen kann man die Build-Zeit auf 2.500 Minuten (42h) erhöhen und die Projektgruppe untersteht keiner Nutzerbegrenzung mehr.

[27] kann auf 8GB erhöht werden, verbraucht aber auch doppelt so viel Build-Zeit

Resultat Bitbucket stellt seinen Nutzern eine grundlegende CI-Umgebung zum Ausprobieren zur Verfügung, was die gute Bewertung in Tabelle 3.13 ebenfalls zeigt. Der Free-Plan stellt zwar nur wenig Ausführungszeit zur Verfügung und begrenzt die Nutzerzahl, ist aber mit noch erschwinglichen 3$ pro Monat deutlich erweiterbar. Die Weboberfläche ist sehr einfach gehalten und stellt keine zusätzlichen Debugging-Möglichkeiten wie z. B. Live-Konsolensitzungen im Container bereit. Die Einrichtung verlief Problemlos und die Konfiguration war bis auf die SSH-Key einbindung relativ einfach gehalten. Die Ausführungszeit wird leider stark von dem Erstellen und Freigeben des Ausführungscontainers beeinflusst, macht jedoch dennoch die Ausführung von grob 100 Tests und Builds im Monat möglich. Da EnjineIO aus noch mehr Komponenten besteht die später ebenfalls getestet werden sollen, müsste früher oder später über die Aufwertung des Plans diskutiert werden.

Funktionalität	
Hilfestellungen	15/30
Caching	25/30
SSH-Keys	15/30
SSH Debugging	0/10
Summe	55/100

Anwendung	
Einrichtung	20/20
Konfiguration	35/50
Dokumentation	20/30
Summe	75/100

Free-Plan	
Build-Zeit	50min - 42h [3$]
OS-Support	Docker
Ressourcen	1GB Cache, 4GB RAM
Parallel	10 Steps

Tabelle 3.13: Anforderungswertung von Bitbucket

3.5.4 CircleCI

CircleCI ist ein Online-CI-Tool, welches sich in Gitlab, GitHub und BitBucket integrieren lässt. Es zeichnet sich durch seine vielzahl an unterstützten Ausführungsumgebungen aus und es gibt auch eine lokal ausführbare Docker-Variante. Um sich anzumelden ist es möglich sich direkt mit seinem BitBucket-Account zu authentifizieren. Allerdings konnte CircleCI dabei nicht auf BitBuckets Projektgruppen zugreifen. Da die hinterlegte Email-Adresse nicht mehrmals verwenden konnte mussten die Account-Daten durch den Support zurückgesetzt werden. Danach konnte ein neuer Zugang per Email registriert und BitBucket im Nachhinein integriert werden.

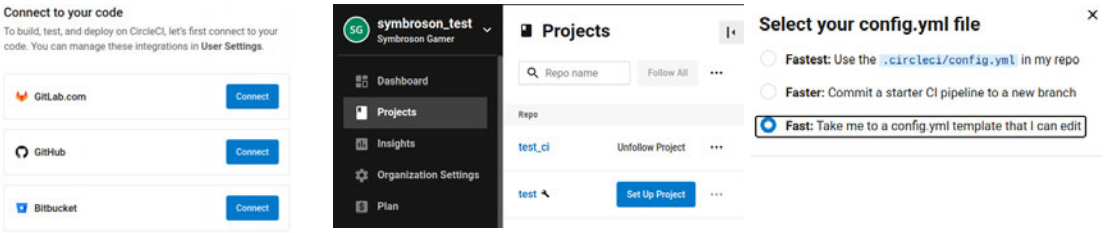

Abbildung 3.5: CircleCI Einrichtung

Einrichtung Nach der Anmeldung wählt man die Projektgruppe aus in der man Arbeiten möchte und wird dann zur entsprechenden Repository-Übersicht weitergeleitet. Man kann Repositories 'folgen' und somit die CI-Ausführung einschalten. Dieser Einrichtungsvorgang ist in Abbildung 3.5 nachvollziehbar. Ein neues Repository kann bei der Ersteinrichtung aus einem von vielen Templates erstellt werden. Für NodeJS gibt es dafür 3 verschiedene Templates, wobei das einfachste einen Node-Orb mit vordefinierten Test-Jobs nutzt und das Fortgeschrittene Docker-Images mit selbst definierten Jobs. In den Projekteinstellungen können neue Umgebungsvariablen definiert und SSH-Keys für die Benutzung in den CircleCI Jobs angelegt werden.

Konfiguration CircleCI wird mit einer YamL-Datei am Projektpfad '.circleci/config.yml' Konfiguriert. In der obersten Ebene der Datei wird zunächst die Versionsnummer definiert, gefolgt von den Workflows, Jobs und Commands die den CI-Ablauf spezifizieren. Jedem Job muss eine Ausführungsumgebung durch einen Orb[28] oder einem Docker-Image zugeordnet werden. In den Workflows wird durch die Angabe von Abhängigkeiten die Ausführungsreihenfolge der Jobs festgelegt. Wenn ein Job nicht direkt von einem anderen abhängig ist, führt CircleCI diesen parallel aus. Eine etwas gekürzte Beispielkonfiguration wird in Quelltext 3.14 gezeigt.

```yaml
version: 2.1
orbs: { node: circleci/node@5.0.2 }

workflows:
  build:        # workflow name
    jobs:
      - install # execute 'install' job
      - test: { requires: [install] }  # requires 'install'
      - build:
          requires: [install] # requires 'install' -> parallel to 'test'
          filters: { branches: { only: build }} # only on 'build' branch

jobs:
  install:            # job-name
    executor: node/16 # orb-defined executor
    steps: [add ssh key, checkout, install, persist to workspace]
    # persist to workspace: store folder structure
  build:
    executor: node/16
    steps:
      - attach workspace: { at: ~/ } # restore folder structure from 'install'
      - run: { command: yarn dist:l }
      - store artifacts: { path: dist/ } # upload binaries

commands:
  add ssh key: # command name
    steps:
      - run: { command: cat known hosts >> ~/.ssh/known hosts }
      - add ssh keys:
          fingerprints: [ 81:46:c9:85:2b:6e:21:d2:c8:45:d7:06:24:14:c7:69 ]
  install:
    steps: # install dependencies via cache
      - restore cache: { keys: <pattern> }
      - run: { command: yarn install --frozen-lockfile }
      - save cache: { key: <pattern>, paths: [./node modules] }
```

Quelltext 3.14: Gekürzte CircleCI Beispielkonfiguration

[28] entspricht einem Docker-Image bei Docker, mit zusätzlichen Job-Definitionen für CircleCI

SSH-Keys Das CircleCI Online-Tool stellt eine Möglichkeit bereit die SSH-Schlüssel über das Online-Tool zu erfassen. Dieser kann optional einem bestimmten Host zugeordnet werden (siehe Abbildung 3.6). Es ist zusätzlich notwendig den Schlüssel mithilfe des Fingerprints in der jeweiligen Job-Konfiguration zu laden (siehe Quelltext 3.14 'add_ssh_key').

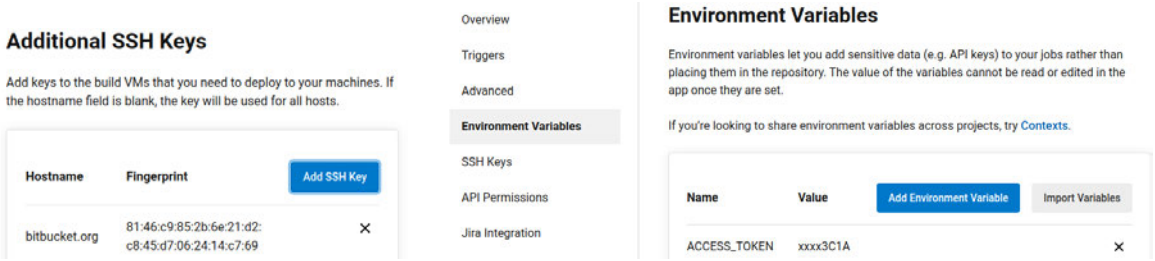

Abbildung 3.6: CircleCI SSH-Keys und Variablen

Ausführung Sofern die Projektüberwachung in der Online-Oberfläche eingeschalten ist werden alle Workflows bei jeder Änderung ausgeführt. Nur auf dem Job-Level sind Filter Konfigurierbar wie in Quelltext 3.14 unter 'workflows' beispielhaft zu sehen ist. Es wird auch ein CI-Prozess gestartet wenn ein Commit auf einem Branch ohne Konfiguration, was zwangsläufig in einen Fehler resultiert (der in manchen Fällen auch Ausführungszeit verbraucht hat). Im CircleCI Dashboard kann man laufende CI-Prozesse live verfolgen und die Details zu beendeten Workflows einsehen. Ist ein Job in einem Workflow fehlgeschlagen steht die Option zur Verfügung diesen Job mit SSH-Zugang neu zu starten (siehe Abbildung 3.7). Während der Job läuft kann man sich auf Kosten der Ausführungszeit mit einem Programm (z. B. einer SSH-Konsole) mit dem Container Verbinden und das Problem analysieren. Der Job wird erst terminiert, wenn der Job in der Weboberfläche des Jobs manuell abgebrochen wird.

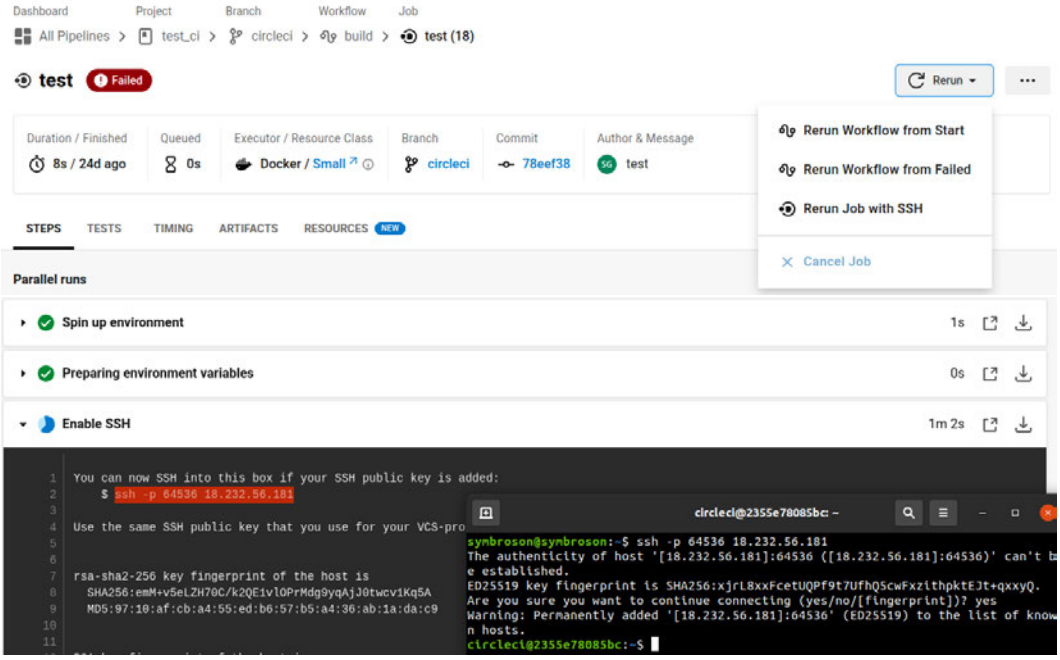

Abbildung 3.7: CircleCI SSH Session

Free-Plan CircleCI baut auf ein Credit-System die von verschiedenen Funktionen zu bestimmten Raten verbraucht werden. So hat jede Ausführungsumgebung eine festgelegte Grundrate und zusätzlich mehrere Ressourcenklassen mit mehr RAM und einem schnelle-

ren Prozessor.[29] In der kostenlosen Version von CircleCI werden jedem Monat 30.000 Credits bereitgestellt. Den niedrigsten verbrauch hat dabei die kleine Docker-Ressourcenklasse mit 5 Credits pro Minute. Damit lassen sich bis zu 100 Stunden Ausführungszeit im Monat beanspruchen. Die nächst teureren Ausführungsumgebungen Stand Oktober 2022 sind in aufsteigender Reihenfolge Linux VM (bis zu 50h), ARM VM (50h), Windows (12h) und MacOS (10h). Außerdem werden 420 Credits pro Gigabyte verbraucht, wenn das monatliche Limit von 2GB Speicher für Cache, Workspaces und Artifacts oder 1GB gesendete Netzwerkdaten überschritten werden. Für ersteres wurden im Rahmen der Tests knapp 200MB hauptsächlich für Workspace-Caching benötigt. Letzteres ist nur für selbst verwaltete Ausführungsumgebungen relevant, in die Daten aus der Pipeline hochgeladen werden müssen. Außerdem können bis zu 30 Jobs parallel zueinander ausgeführt werden. Das begünstigt zudem den Creditverbrauch, da nur die Gesamtausführungszeit der Workflows zählt.

Lokale Docker-Version

Mit der lokalen Docker-Version von CircleCI ist es nur möglich einzelne CircleCI-Jobs lokal auszuführen und zu Testen, nicht aber den kompletten Workflow. (siehe [1, S. 292]) Sie lässt sich mit dem öffentlichen Docker-Container 'circleci/picard' ausführen. In diesen Container wird dann der Projektordner verlinkt und die SSH-Schlüssel der privaten Abhängigkeiten als Variablen übergeben. Die Konfigurationsdatei muss dabei zunächst durch das Konsolenprogramm in eine ältere Konfigurationsversion umgewandelt werden. Anschließend wird standardmäßig der 'build'-Job ausgeführt. Online-Funktionen wie z. B. Caching, Workspaces und Artifacts stehen dabei leider nicht zur Verfügung.

Ausführung Bei der ersten Ausführung wir der angegebene Container automatisch von Docker heruntergeladen. Danach lädt und validiert CircleCI die Pipeline-Konfiguration und führt dann den angegeben Job aus. Dabei ergab sich zunächst das Problem, dass CircleCI die Konfiguration nicht aus dem Projektordner laden konnte. Stattdessen musste sie mithilfe eines Bash-Skriptes außerhalb des Projektordners im Container verlinkt werden[30] (siehe Quelltext 3.15). Danach funktionierte diese Variante jedoch auch einwandfrei, auch wenn sie im Rahmen dieses Projektes bis auf das grundlegende Testen der CI-Konfiguration eine eher nebensächliche Bedeutung hat. Ein kompletter Test-Durchlauf mit eslint und mocha dauert in CircleCI ca. 20 Sekunden. Zum aktuellen Stand werden ca. 7MB Cache und 39MB Workspace Cache beansprucht die die Ausführungszeit auf 10 Sekunden reduziert. Die Erstellung und Upload der ausführbaren Dateien beansprucht für Linux 17 und für alle alle Betriebssysteme 25 Sekunden.

Resultat CircleCI sticht vor allem durch sein großes Angebot an Ausführungsumgebungen und einem kostenlosen Angebot welches viele Stunden ausführungszeit zulässt. Dahingehend scheint dieses Tool optimal für die Zwecke dieses Projektes geeignet zu sein. Leider ergaben sich während der Anwendung vielerlei Problemquellen die die die Einrichtung erschwerten und sich entsprechend sich auf die Bewertung in Tabelle 3.14 auswirkten.

[29] offizielle Übersicht: https://circleci.com/product/features/resource-classes
[30] Quelle: https://github.com/CircleCI-Public/circleci-cli/issues/413#issuecomment-853747791

```
#!/bin/bash
[ $# = 0 ] && echo "example usage: $0 --job test" && exit

src cfg="$CWD"/.circleci/local config.yml
intermediate cfg="$CWD"/.circleci/processed.yml
docker sock=/var/run/docker.sock

out=`circleci config process $src cfg` || exit
SSHKEY=`base64 ../id rsa`
SSHKEYPUB=`base64 ../id rsa.pub`

echo "$out" > $intermediate cfg
docker run --rm -it \
      -v "$docker sock:$docker sock" \
      -v "$intermediate cfg:/tmp/local build config.yml" \
      --workdir "$CWD" \
      circleci/picard \
      circleci build --config /tmp/local build config.yml \
      -e SSHKEY="$SSHKEY" \
      -e SSHKEYPUB="$SSHKEYPUB" \
      $*
```

Quelltext 3.15: Lokales CircleCI-Ausführungsskript

Webseite Erstens gab es schon während der Anmeldung Probleme mit der Bitbucket-Integration, bei der Zuarbeit des Supports notwendig war. Zweitens wirkte die Webseite an vielen Stellen kompliziert und unübersichtlich. Beispielsweise bei der Nachverfolgung des Credit-Verbrauchs und den Ausführungsstatistiken. Wichtige Bedienfelder wie z. B. der zum Beenden einer Job-SSH-Sitzung waren ohne Kenntnis über die Webseite schwer zu erreichen. Das löste ein gewisses Stresslevel bei der Suche aus, während kontinuierlich Credits verbraucht wurden. Workflows wurden auch ohne existente Konfiguration ausgeführt und verbrauchten dadurch manchmal sogar Ausführungszeit.

Dokumentation Grundlegenden Abläufen wurden in der Dokumentation ähnlich unübersichtlich dargestellt und beinhaltete kompliziert wirkende Beispiele.[31] Aus diesen Gründen zog sich die korrekte Einrichtung verglichen zu den anderen Tools sehr in die Länge. Trotz dessen macht der äußerst großzügige Free-Plan dieses Tool interessant für eine nähere Auswahl.

Funktionalität	
Hilfestellungen	20/30
Caching	25/30
SSH-Keys	20/30
SSH Debugging	5/10
Summe	70/100

Anwendung	
Einrichtung	5/20
Konfiguration	25/50
Dokumentation	15/30
Summe	45/100

Free-Plan	
Build-Zeit	max 100h - min 10h
OS-Support	Docker, Linux, Win, MacOS, ARM
Ressourcen	2GB Storage (max +70), 2GB+ RAM
Parallel	30 Jobs

Tabelle 3.14: Anforderungswertung von CircleCI

[31] vgl. Einführung zur Konfiguration: https://circleci.com/docs/config-intro/

3.5.5 Buddy

Im Unterschied zu den beiden vorherigen Tools kann Buddy wie in Abbildung 3.8 komplett über eine Online-Benutzeroberfläche konfiguriert werden. Es gibt aber noch immer die Möglichkeit eine Konfigurationsdatei zu verwenden. Eine grafische Pipeline-Konfiguration hat den Nachteil, dass diese nicht in der Versionsverwaltung verzeichnet wird und zur Änderung dieser immer Zugriff auf den zugehörigen Buddy-Account benötigt wird. Dafür war die Einrichtung die einfachste von allen.

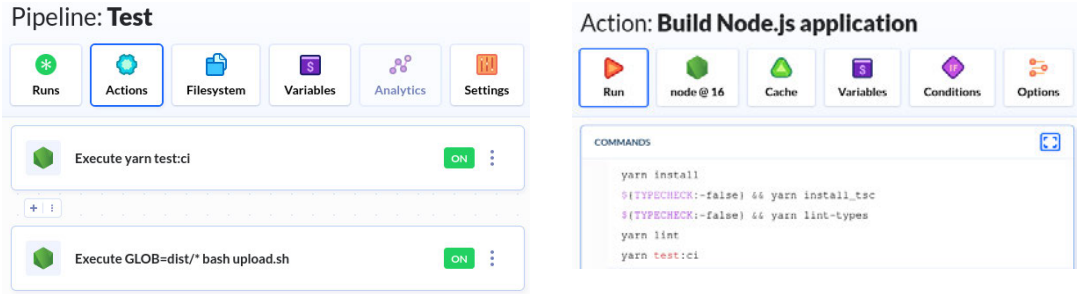

Abbildung 3.8: Buddy Konfiguration

Konfiguration Buddy stellt eine Bitbucket-Registrierung zur Verfügung durch die es Zugriff auf alle Repositories des jeweiligen Accounts erhält. Anschließend legt man eine neue Pipeline für ein Repository an und legt den Namen und ein oder mehrere Auslöserereignisse fest. Letzteres kann manuell durch eine Knopfdruck, wiederholt zu bestimmten Intervallen (auch via Cronjob[32]-Konfigurationen) oder zu neuen Commits mit bestimmten Eigenschaften wie Branch oder Tag-Name.

Anschließend wählt man den Projekttyp aus einer Reihe von Vorlagen, in diesem Fall NodeJS. Diese bewirkt, dass in dem Laufzeitcontainer bereits alle entsprechend notwendigen Programme und Tools zur Verfügung stehen. Es sind auch eine Linux und eine Windows VM verfügbar, jedoch konnte insbesondere für letztere keine Dokumentation oder Hinweis zur Installation von NodeJS gefunden werden.
Unter dem Reiter 'Run' kann man dann sein (Bash-)Skript mit den Test- bzw. Build-Anweisungen eintragen. Caching muss in diesem Fall nicht konfiguriert werden. Buddy speichert automatisch das gesamte Dateisystem nach jedem Step im Cache und lädt diesen vor der Ausführung desselben wieder. Dieses Verhalten wird auch durch den besonderen Free-Plan (siehe Absatz 3.5.5 Free-Plan) gestützt. Unter 'Variables' lassen sich neue gesicherte Umgebungsvariablen anlegen. In dem gleichen Reiter können zusätzliche Dateien hochgeladen und auch SSH-Schlüssel für die Privaten Abhängigkeiten hinterlegt werden. Damit ist die Konfiguration bereits abgeschlossen und die CI-Pipeline einsatzbereit.

Auch bei Buddy eine Konfiguration im YAML-Format möglich. Diese kann direkt aus der grafisch konfigurierten Pipeline exportiert werden. Ein Beispiel einer solchen Konfigurationsdatei ist in Quelltext 3.16 zu sehen, wird hier aber aufgrund des Fokusses von Buddy auf grafische Konfiguration nicht näher betrachtet.

[32] siehe https://wiki.ubuntuusers.de/Cron

```yaml
- pipeline: Test Server
  on: EVENT
  events: [{ type: PUSH, refs: [refs/heads/bb pipes] }]
  priority: NORMAL
  fail on prepare env warning: true
  actions:
    - action: Test and Build
      type: BUILD
      execute commands:
        - yarn install --frozen-lockfile
        - yarn lint
        - yarn test
        - yarn dist
      shell: BASH
      variables:
        - { key: id rsa, value: secure!fd9...+FA==, type: SSH KEY, encrypted: true, ... }
        - { key: ACCESS TOKEN, value: secure!MOf...IUA==, type: VAR, encrypted: true }
```

Quelltext 3.16: Exportierte YAML-Konfiguration von Buddy

Ausführung Die Pipeline startet automatisch nach Eintreten des eingestellten Auslöseereignisses, in diesem Fall ein Commit auf dem 'buddy'-Branch. Auch hier kann der aktuelle Ausführungsstatus und Konsolenausgaben von laufenden und abgeschlossenen Pipelines eingesehen werden. Im Fehlerfall ist es sogar möglich, eine Konsolensitzung im Ausführungscontainer zu starten und so das Problem zu analysieren und beheben (siehe Abbildung 3.9). Anschließend kann die Ausführung von dem fehlgeschlagenen Schritt aus fortgesetzt werden. Ein kompletter Test-Durchlauf mit eslint und mocha dauert in Buddy ca. 11 Sekunden. Der Cache beansprucht zum aktuellen Stand ca. 53MB für das Dateisystem und reduziert die Ausführungszeit auf 4 Sekunden. Die Erstellung und Upload der ausführbaren Dateien beansprucht für Linux 12 und für alle alle Betriebssysteme 18 Sekunden.

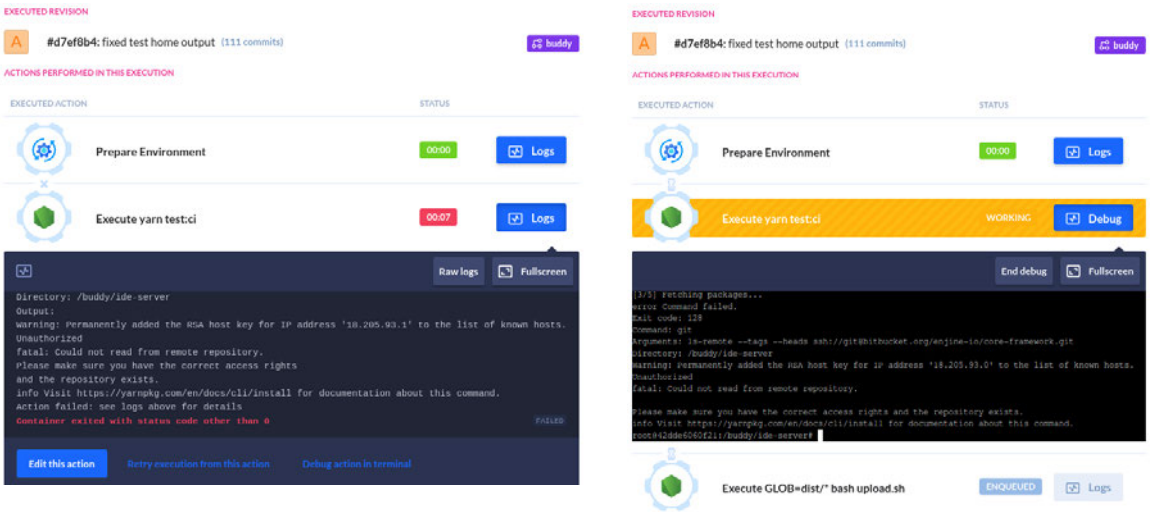

Abbildung 3.9: SSH-Session zum Beheben einer Abhängigkeit

Free-Plan Der Free-Plan von Buddy ist nicht durch Zeit gebunden, sondern stellt 120 Ausführungen pro Monat verteilt auf maximal 5 Repositories zur Verfügung (ca. 4 Ausführungen jeden Tag, weniger als eine pro Repository). Aktionen können in einer Docker, Linux oder Windows-Umgebung ausgeführt werden, jedoch ohne Parallelität. Die Anzahl der Projekte ist auf 5 begrenzt und der Speicherplatz für Caches und Dateisysteme auf 512MB. Letztere können jedoch jederzeit manuell gelöscht werden.

Resultat Buddy macht besonders die sehr einfache und schnelle grafische Einrichtung attraktiv und die unlimitierte Laufzeit einer Pipeline. Vielfältigen Funktionen wie z. B. die Live-Konsolensession in der Online-Ansicht oder die zahlreiche Integrationen von 3rd-Party Software runden dieses Angebot ab. Deshalb fällt die Bewertung in Tabelle 3.15 sehr gut aus. Leider ist der kostenlose Plan durch die begrenzte Nutzerzahl und nur 120 Ausführungen pro Monat stark limitiert und angesichts der anderen zu testenden EnjineIO-Komponenten nicht zukunftstauglich. Für ein einzelnes Projekt wäre die Ausführungszahl ausreichend, für mehre müsste jedoch ein besserer Plan ausgewählt werden. Dieser ist aber bei aktuell 75$ pro Monat vorerst nicht erschwinglich.

Funktionalität	
Hilfestellungen	20/20
Caching	30/30
SSH-Keys	30/30
SSH Debugging	20/20
Summe	100/100

Anwendung	
Einrichtung	20/20
Konfiguration	50/50
Dokumentation	20/30
Summe	90/100

Free-Plan	
Build-Zeit	120 Ausführungen/Monat
OS-Support	Docker, Linux VM, Windows VM
Ressourcen	512MB Cache (löschbar), 1GB RAM
Parallel	keine

Tabelle 3.15: Anforderungswertung von Buddy

3.5.6 Resultat

Von allen Tools stellte sich CircleCI bei weitem als funktionsreichstes heraus und bot zudem den besten Free-Plan an. Leider hat die Registrierung zunächst nicht funktioniert und die Webseite und die Dokumentation sind verglichen mit den anderen Tools sehr komplex und unübersichtlich, sodass CircleCI mehr Einarbeitungszeit benötigt.
Bitbucket Pipelines ist durch die direkte Zugehörigkeit zu Bitbucket die offensichtlichste Wahl. Hat jedoch nur einen Bruchteil der Funktionalität wie CircleCI und stellt erst durch den fortgeschrittenen Plan für 3$ eine für das Projekt akzeptable Ausführzeit bereit.

Buddy war das am einfachsten zu verwendende Tool, schied aber wegen der geringen Ausführungen pro Monat und die begrenzte Nutzerzahl aus. Von den verbleibenden Tools lässt sich allerdings kein eindeutiger Favorit bestimmen. Langfristig ist CircleCI die bessere Wahl sobald die Einstiegshürden überwunden und die Probleme sich durch die Praxis minimieren. Es ist aber auch denkbar zunächst Bitbucket und CircleCI kombiniert anzuwenden. Das eine für die Ausführung der Tests und das andere für das Erstellen und Hochladen von plattformspezifischen ausführbaren Dateien.

Tool	Funktionalität	Anwendung	Free-Plan
BitBucket Pipelines	55	75	50 min/mon - 42 h/mon[3$] 10 Steps parallel
CircleCI	70	45	10-100 h/mon 30 Jobs parallel
Buddy	100	90	120 x/mon keine Parallel

Tabelle 3.16: Gesamtwertung der CI-Tools

4 Fazit

Diese Arbeit hat gezeigt, dass es beim Anwenden von Continuous Integration eine Vielzahl an Faktoren zu beachten gibt. Für jeden Teilschritt müssen genaue Anforderungen spezifiziert werden, nach denen dann eine Vorauswahl und im Anschluss eine detailliertere Evaluation erfolgen kann. Diese Anforderungen sind immer Projektspezifisch und müssen jeweils im Hinblick auf die größere Projektinfrastruktur betrachtet werden. Das macht es zu einer besonderen Herausforderung die richtigen Tools für seine Anwendung zu finden.

In dieser Arbeit wurden für EnjineIO jeweils 3 Tools für die statische Quellcodeanalyse, für Modultests und die Continuous Integration Pipeline miteinander verglichen. Bei der statischen Quellcodeanalyse stellte sich ESLint gegenüber JSLint und JSHint als deutlich überlegen heraus. Den anderen beiden fehlte es erheblich an Funktionalität und Flexibilität. Bei den Modultests tat sich Mocha durch seine Geschwindigkeit hervor was für die spätere Ausführung in der CI Pipeline essenziell für ein schnelles Feedback ist. Jest führte zunächst eine langsame Quellcode-Transformation durch und Vitest nutzte dieselbe Strategie, erforderte jedoch zusätzlich eine Umwandlung der Test-Skripte in ES-Module. Bei den CI Tools war hauptsächlich der Free-Plan ausschlaggebend für die Auswahl. Da EnjineIO Teil einer nichtgewinnorientierte Organisation ist, wäre ein teurer Nutzungsplan für die erste Phase der Projektentwicklung nicht wirtschaftlich. Deshalb fiel die Wahl hier auf CircleCI, welches mit einem großen Funktionsumfang und bis zu 100 Stunden Ausführungszeit pro Monat überzeugte, auch wenn es einige Probleme bei der Einrichtung gab. Buddy war am einfachsten einzurichten aber war wie auch Bitbucket Pipelines wegen seiner Free-Plan Einschränkungen nicht auf mehrere Projekte skalierbar.

Die Zielstellung dieser Arbeit wurde dahingehend erfüllt, dass eine Basis zur Anwendung von Continuous Integration in diesem Projekt geschaffen wurde. Viele Aspekte konnten durch ausgewählte Beispiele abgedeckt, aber nicht vollständig umgesetzt werden. Jedoch ist diese Arbeit als Grundlage für die Skalierung auf weitere Komponenten der EnjineIO Infrastruktur sowie für ähnliche NodeJS-basierte Projekte nutzbar.

4.1 Wertung

Da die Problemstellung sehr von dem konkreten Anwendungsfall abhängig ist, lassen sich nur schwer allgemeingültige Aussagen zu anderen Softwareprojekten formulieren. Die Tools wurden zudem nicht durch objektiv messbare Einflussfaktoren, sondern durch subjektive Eindrücke bewertet. Es konnten auch nur für dieses Projekt relevante Eigenschaften einer ausgewählten Toolmenge betrachtet werden. Deshalb könnten in einem anderen Kontext mit anderen Anforderungen auch ganz andere Ergebnisse entstehen. Zudem hatte der Autor dieser Arbeit keine nennenswerten Vorkenntnisse zum Thema Softwaretests. Deshalb ist es nicht auszuschließen, dass einige Aspekte dieses Themengebietes nicht ausreichend bis gar nicht betrachtet wurden. Die Arbeit stützt sich jedoch auf Erfahrungsberichte und Praktiken anderer professioneller Entwickler und namenhafter Persönlichkeiten aus dem Bereich der Softwareentwicklung.

4.2 Ausblick

Wie bereits erwähnt, sollen die Tests in Zukunft auf weitere Komponenten der EnjineIO Infrastruktur ausgeweitet werden. Dabei sind bisher nicht betrachtete Testarten wie z. B. UI-Tests im UI-Framework vonnöten, deren Umsetzung in dieser Arbeit nicht betrachtet wurde. Außerdem sind vor allem die Modultests aktuell nur exemplarisch umgesetzt und deshalb unvollständig. Deshalb ist es notwendig diese noch weiter auszubauen und ggf. weitere Testenswerte Problemstellen zu identifizieren.

Im künftigen Verlauf sollen die Tests bereits zusammen mit der Implementierung geschrieben werden. Dafür ist es vonnöten, dass auch die anderen Entwickler des EnjineIO Teams auf das Thema Softwaretests aufmerksam gemacht und eingearbeitet werden. Den Testprozess in den Arbeitszyklus aufzunehmen wird durch die bestehende Test- und CI Infrastruktur erleichtert. Dennoch müssen die Entwickler die Tests selbst schreiben. Es bleibt abzuwarten, wie gut sich die in dieser Arbeit ausgewählten Tools auf Dauer bewähren und im Team integrieren lassen.

Literaturverzeichnis

[1] J.-M. Belmont, *Hands-On Continuous Integration and Delivery Build and release quality software at scale with Jenkins, Travis CI, and CircleCI*, eng, 1. 2018, ISBN: 9781789133073.

[2] S. Rossel, *Continuous Integration, Delivery, and Deployment Reliable and faster software releases with automating builds, tests, and deployment*, eng, 1. 2017, ISBN: 9781787284180.

[3] F. Witte, *Testmanagement und Softwaretest Theoretische Grundlagen und praktische Umsetzung* (SpringerLink Bücher), ger, 1. Aufl. 2016. 2016, ISBN: 9783658099640.

[4] R. C. Martin, *Clean code: a handbook of agile software craftsmanship* (Robert C. Martin series), eng. Prentice Hall, 2009, ISBN: 9780132350884 0132350882.

[5] M. C. Feathers, *Working Effectively with Legacy Code* (Robert C. Martin Series), eng. Prentice Hall PTR, 2004, ISBN: 0131177052.

Toolverzeichnis

[6] D. Hurren, *DroidScript*, https://droidscript.org, Zugriff Juli 2022.

[7] D. Hurren, *EnjineIO*, https://enjine.io, Zugriff Juli 2022.

[8] *Node-Pkg*, https://github.com/vercel/pkg, Zugriff Juli 2022.

[9] *Electron*, https://github.com/electron/electron, Zugriff Juli 2022.

[10] *Electron-Builder*, https://github.com/electron-userland/electron-builder, Zugriff Juli 2022.

[11] *CircleCI*, https://github.com/visionmedia/supertest, Zugriff August 2022.

[12] *Bitbucket-Pipelines*, https://bitbucket.org/product/de/features/pipelines, Zugriff August 2022.

[13] D. Crockford, *JSLint*, https://github.com/jslint-org/jslint, Zugriff Juli 2022.

[14] A. Kovalyov, *JSHint*, https://github.com/jshint/jshint, Zugriff Juli 2022.

[15] N. C. Zakas, *ESLint*, https://github.com/eslint/eslint, Zugriff Juli 2022.

[16] T. Schaub, *Mock-FS*, https://github.com/tschaub/mock-fs, Zugriff August 2022.

[17] V. Dalecky, *Memfs*, https://github.com/streamich/memfs, Zugriff August 2022.

[18] T. Holowaychuk, *Mocha*, https://github.com/mochajs/mocha, Zugriff August 2022.

[19] J. Luer, *Chai*, https://github.com/chaijs, Zugriff August 2022.

[20] G. Bahmutov, *Snap-Shot-It*, https://github.com/bahmutov/snap-shot-it, Zugriff August 2022.

[21] B. Coe, *Nyc*, https://github.com/istanbuljs/nyc, Zugriff August 2022.

[22] *Jest*, https://github.com/facebook/jest, Zugriff August 2022.

[23] T. Holowaychuk, *Supertest*, https://github.com/visionmedia/supertest, Zugriff August 2022.

[24] *Vitest*, https://github.com/vitest-dev/vitest, Zugriff August 2022.

[25] B. Coe, *C8*, https://github.com/bcoe/c8, Zugriff August 2022.

Eidesstattliche Erklärung

Hiermit versichere ich – Alexander Feilke – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 24. Oktober 2022	
Ort, Datum	Alexander Feilke