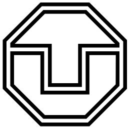


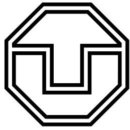
TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Rechnernetze

Master-Arbeit

ENTWICKLUNG AUTOMATISIERTER TESTS ZUR ÜBERWACHUNG DER INTEGRATION UND PERFORMANCE VON FRONTEND UND BACKEND FÜR DIE PLATTFORM AMCS

Samuel Knobloch



TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Rechnernetze

Master-Arbeit

ENTWICKLUNG AUTOMATISIERTER TESTS ZUR ÜBERWACHUNG DER INTEGRATION UND PERFORMANCE VON FRONTEND UND BACKEND FÜR DIE PLATTFORM AMCS

Samuel Knobloch
Matrikelnummer: 3913057
Immatrikulationsjahr: 2014

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

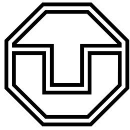
Betreuer

Dr.-Ing. Iris Braun
Dr.-Ing. Tenshi Hara

Betreuender Hochschullehrer

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Eingereicht am: 18. August 2017



TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl für Rechnernetze

AUFGABENSTELLUNG FÜR DIE ANFERTIGUNG EINER MASTER-ARBEIT

Name: Samuel Knobloch
Matrikelnummer: 3913057
Immatrikulationsjahr: 2014

Titel: **Entwicklung automatisierter Tests zur Überwachung der Integration und Performance von Frontend und Backend für die Plattform AMCS**

ZIELE DER ARBEIT

Am Lehrstuhl Rechnernetze wurden in den vergangenen Jahren mehrere prototypische Untersuchungen zum Einsatz technischer Werkzeugen im Lehrbetrieb durchgeführt. Unter anderem wurden Werkzeuge in Vorlesungen und Übungen getestet; es wurden für den jeweiligen Einsatz valide Ergebnisse gewonnen. Am Ende der bisherigen Entwicklungen steht eine komplexe Anwendungsarchitektur und -plattform mit der Prototypanwendung Auditorium Mobile Classroom Service (AMCS).

Da es sich um eine über mehrere Iterationen gewachsene Plattform handelt, gibt es hinreichenden Bedarf für Optimierungen. Beispielsweise gibt es derzeit nicht-deterministische Load-Spitzen und Fehler, die unbedingt untersucht werden müssen. Ein weiteres Problem stellt die Erweiterbarkeit von AMCS dar. Mangels integrierter, automatischer Tests ist die parallele Weiterentwicklung von Frontend und Backend schwierig.

Im Rahmen dieser Master-Arbeit sollen verschiedene Testszenarien entwickelt und umgesetzt werden. Schwerpunkte dabei sind Performance-Tests, speziell Load-Tests für das Backend, sowie automatisierte Testszenarien für das Zusammenspiel von Frontend und Backend anhand vordefinierter Use-Cases. Für die jeweiligen Anwendungsfälle sind verschiedene Tools anhand zu definierender Kriterien miteinander zu vergleichen. Darauf basierend soll eine Auswahl für die Integration und Umsetzung getroffen werden.

SCHWERPUNKTE DER ARBEIT

- Analyse und Auswahl bestehender Ansätze und Lösungen
- Definieren von Anforderungen
- Definieren von Bewertungskriterien für die Anforderungserfüllung
- Konzept zur automatisierten Durchführung von Tests in AMCS
- prototypische Umsetzung des Konzepts
- Evaluation und Auswertung der Ergebnisse

Betreuer: Dr.-Ing. Iris Braun
Dr.-Ing. Tenshi Hara

Ausgehändigt am: 10. März 2017
Einzureichen am: 18. August 2017

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Betreuender Hochschullehrer

INHALTSVERZEICHNIS

Abbildungsverzeichnis iii

Tabellenverzeichnis v

Quelltextverzeichnis vii

1. Einleitung 1
   1.1. Motivation 1
   1.2. Zielsetzung 1
   1.3. Struktur der Arbeit 2

2. Tests und Testautomatisierung 3
   2.1. Einführung und Definition 3
   2.2. Performance-Test 4
      2.2.1. Begriffe 4
      2.2.2. Vorgehensweisen und Modelle 5
   2.3. Web-Test 5
      2.3.1. Begriffe 5
      2.3.2. Headless Browser 6

3. Performance-Tests 7
   3.1. Auswahlkriterien 7
   3.2. Gatling 8
   3.3. JMeter 11
   3.4. Weitere Tools 14
   3.5. Zusammenfassung 15

4. Web-Tests 17
   4.1. Auswahlkriterien 17
   4.2. Maven und Selenium 18
   4.3. WebdriverIO 19
   4.4. Protractor 21
   4.5. Test-Frameworks: Jasmine und Mocha 23
   4.6. Weitere Tools 23
   4.7. Zusammenfassung 24

5. AMCS - Auditorium Mobile Classroom Service 25
   5.1. Projektübersicht 25

5.2. Backend . . . 25
5.2.1. Kriterien für Performance-Tests . . . 26
5.2.2. Durchführung der Performance-Tests . . . 27
5.3. Web-Applikation . . . 27
5.3.1. Kriterien für Web-Tests . . . 27
5.3.2. Durchführung der Web-Tests . . . 28

6. Test-Szenarien **29**
6.1. Performance-Tests . . . 29
6.1.1. Use-Cases . . . 29
6.2. Frontend-Tests . . . 30
6.2.1. Use-Cases . . . 30

7. Implementierung **33**
7.1. Performance-Tests . . . 33
7.2. Frontend-Tests . . . 40

8. Auswertung der Ergebnisse **47**
8.1. Performance-Tests . . . 47
8.1.1. Login mit neuem Account . . . 47
8.1.2. Login mit existierendem Account und Laden von Veranstaltungen . . . 48
8.1.3. Workflow: Login und Laden aller Fragen einer Veranstaltungen . . . 50
8.2. Web-Tests . . . 53
8.2.1. Login und Logout mit einem Testnutzer . . . 53
8.2.2. Lifecycle eines Kurses . . . 54
8.2.3. Lifecycle einer Veranstaltung . . . 55
8.2.4. Verwalten von Fragen im Veranstaltungskontext . . . 56
8.2.5. Beantworten einer Frage . . . 58

9. Zusammenfassung und Ausblick **61**
9.1. Performance-Tests . . . 61
9.2. Web-Tests . . . 62
9.3. Ausblick . . . 62

A. Literaturverzeichnis **vi**

ABBILDUNGSVERZEICHNIS

3.1. Gatling: Recorder Konfiguration 9
3.2. Gatling: Recorder Aufzeichnung 10
3.3. Gatling: Beispielauswertung 10
3.4. JMeter: Beispielkonfiguration Thread Group 12
3.5. JMeter: Oberfläche 13
3.6. JMeter: Beispiel Graph-Auswertung 13

5.1. AMCS Systemarchitektur 26

7.1. Performance-Test: Szenario-Übersicht 35
7.2. Performance-Test: HTTP Request Defaults 36
7.3. Performance-Test: CSV Data Set Config 36
7.4. Performance-Test: HTTP Header Manager für Login-Anfrage 36
7.5. Performance-Test: HTTP Request für Login-Anfrage 37
7.6. Performance-Test: JSON Extractor für JWT 37
7.7. Performance-Test: HTTP Header Manager für folgende Anfrage 37
7.8. Performance-Test: HTTP Request für Kurseinschreibung 38
7.9. Performance-Test: JSON Extractor für Kurs-ID 38
7.10. Performance-Test: HTTP Request für Abfragen aller Veranstaltungen 39

8.1. Performance-Test 1: Ergebnisse der Anfragen 47
8.2. Performance-Test 1: Antwortzeiten 48
8.3. Performance-Test 2: Auswertung der Anfragen 49
8.4. Performance-Test 2: Anstieg Antwortzeit bei 1000 Nutzern 49
8.5. Performance-Test 3: Auswertung der Anfragen 51
8.6. Performance-Test 3: Anstieg Antwortzeit bei 1000 Nutzern 52

TABELLENVERZEICHNIS

3.1. Vergleichskriterien für Performance-Test-Tools 15

4.1. Vergleichskriterien für Web-Test-Tools 24

7.1. Implementierung: Ermittelte REST-Anfragen 33
7.2. Implementierung: Ermittelte Header-Parameter 33

8.1. APDEX-Werte Performance-Test 2 48
8.2. APDEX-Werte Performance-Test 2P 50
8.3. APDEX-Werte Performance-Test 3 51
8.4. APDEX-Werte Performance-Test 3P 53
8.5. Evaluation: Test-Szenario 1 Tabellenübersicht 54
8.6. Evaluation: Test-Szenario 2 Tabellenübersicht 55
8.7. Evaluation: Test-Szenario 3 Tabellenübersicht 56
8.8. Evaluation: Test-Szenario 4 Tabellenübersicht 58
8.9. Evaluation: Test-Szenario 4 Tabellenübersicht 59

QUELLTEXTVERZEICHNIS

4.1. Code-Beispiel: Maven und Selenium 18
4.2. Code-Beispiel: WebdriverIO 20
4.3. Code-Beispiel: Protractor 22

7.1. Ruby: Anlegen der Testnutzer 34
7.2. Ruby: Anlegen von Kurs, Veranstaltungen und Fragen 34
7.3. Ruby: Generieren von Beispielantworten 35
7.4. Installation von node.js und Protractor 40
7.5. Installation von Xvfb Darstellungsabhängigkeiten 40
7.6. Installation von Chrome 40
7.7. Starten der Headless-Umgebung 41
7.8. JavaScript: Login als Administrator 41
7.9. JavaScript: Logout 41
7.10. TypeScript: Vorarbeiten und Prüfen von Button 42
7.11. TypeScript: Prüfen des Formulars auf Vollständigkeit 42
7.12. TypeScript: Übergabe fehlerhafter Daten 43
7.13. TypeScript: Übergabe korrekter Daten 43
7.14. TypeScript: Öffnen der Kursseite 44
7.15. TypeScript: Editieren von Kursdaten 44
7.16. TypeScript: Löschen des Kurses 45

9.1. Headless-Chrome Startparameter 62

1. EINLEITUNG

Moderne Web-Applikationen bestehen nicht mehr nur aus einer einfachen Server-Anwendung, die vom Nutzer über das „ Request-Response-Prinzip" abgefragt werden kann. Durch die Entwicklung unterschiedlicher Technologien werden heute dynamische und asynchron kommunizierende verteilte Systeme in einer Multi-Schicht-Architektur eingesetzt, die von einer Vielzahl von Geräten aus abrufbar sind. Durch diese Auftrennung in unterschiedliche Teilanwendungen für beispielsweise Frontend und Backend besteht die Notwendigkeit der Überwachung der Integration der Systeme und der Last sowie Lastverteilung in skalierten Anwendungen.
In dieser Arbeit soll der Hauptfokus auf der Entwicklung und Umsetzung unterschiedlicher Tests liegen, um mögliche auftretende Problemstellungen sowohl in Integration als auch Last frühzeitig erkennen und beheben zu können.

1.1. MOTIVATION

Die Plattform AMCS wird an der TU Dresden mit dem Ziel entwickelt, Interaktionen zwischen Dozenten und Studenten während einer Veranstaltung und darüber hinaus zu ermöglichen und zu fördern. Die aktuell im Einsatz und der Weiterentwicklung befindliche Version 3 fasst die bisher gesammelten Erfahrungen und Ideen zusammen und behebt in vorangegangenen Versionen aufgetretene Probleme und Engpässe. Da es das Ziel ist, die Plattform nicht nur für einzelne Lehrveranstaltungen einzusetzen, sondern für alle Veranstaltungen an der TU Dresden verfügbar zu machen, besteht die Notwendigkeit, die Leistungsfähigkeit und Skalierbarkeit der Anwendung über Performance-Tests zu ermitteln und automatisiert überprüfbar zu machen. In der Vergangenheit aufgetretene Systemabstürze durch Überlast sollen dadurch frühzeitig erkannt und vermieden werden.

Da auch das Web-Frontend einer stetigen Weiterentwicklung unterliegt, besteht auch hier die Notwendigkeit, die korrekte Funktionsweise von Basisfunktionalitäten und die Integration mit der Backend-Anwendung zu gewährleisten. Etwaige Inkompatibilitäten durch neue Entwicklungen sollen durch dem Einsatz von Web-Tests schon frühzeitig erkannt und vermieden werden.

1.2. ZIELSETZUNG

Der Fokus dieser Arbeit liegt auf der Auswahl von Werkzeugen zur Umsetzung der unterschiedlichen Performance- und Web-Tests. Dazu sollen verschiedene Werkzeuge betrachtet

und anhand definierter Kriterien miteinander verglichen werden. Weiterhin sollen Test-Szenarien definiert, prototypisch implementiert und die Ergebnisse ausgewertet werden.
Die Auswahl der Szenarien erfolgt anhand von Erfahrungswerten aus vergangenen Veranstaltungen, die bereits hohe Lasten oder eine Überlastung des Systems erzeugt haben sowie an oft genutzten Basis-Abläufen. Als Beispiel für ein hohe Last erzeugendes Szenario sei die Abfrage einer vollständigen Lehrveranstaltungsevaluation durch eine größere Anzahl von parallelen Nutzern genannt.

Im Endergebnis soll eine erweiterbare Basis an ausführbaren Tests vorhanden sein, um kontinuierlich die Performance und Integrität des Systems zu überprüfen. Auch sollen auftretende Engpässe identifiziert und mögliche Lösungsansätze definiert werden.

1.3. STRUKTUR DER ARBEIT

Beginnend mit dem Kapitel „Tests und Testautomatisierung“ wird eine Einführung in das Thema Tests gegeben und unterschiedliche Möglichkeiten zum Testen aufgeführt. Weiter werden Begriffe aus den Bereichen Performance- und Web-Test anhand von Definitionen erklärt. Anschließend werden im Kapitel „Performance-Tests“ Kriterien für den Vergleich von Werkzeugen für Lasttests definiert und zwei ausgewählte Werkzeuge miteinander verglichen. Das darauf folgende Kapitel „Web-Tests“ beschäftigt sich analog zum vorherigen mit unterschiedlichen Ansätzen für das Testen von Web-Anwendungen und vergleicht ausgewählte Werkzeuge und Frameworks anhand von definierten Kriterien. Beide Kapitel enden mit einer Auswahl eines passenden Werkzeugs für die Implementierung.

Im folgenden Kapitel „AMCS - Auditorium Mobile Classroom Service“ wird die Plattform AMCS näher vorgestellt und auf Besonderheiten von Backend und Frontend eingegangen. Fokus liegt dabei auch darauf, Kriterien für die Test-Durchführungen in beiden Bereichen zu definieren. Das Kapitel „Test-Szenarien“ beschreibt basierend auf den zuvor definierten Kriterien unterschiedliche Use-Cases für Performance- und Web-Tests mit den zu verwendenden Daten und Parametern. Diese werden, exemplarisch an je einem Beispiel beschrieben, im Kapitel „Implementierung“ umgesetzt. Neben der eigentlichen Umsetzung werden auch notwendige Vorarbeiten aufgeführt und Fehlerquellen erläutert.

Das nachfolgende Kapitel „Auswertung der Ergebnisse“ beschreibt grob für jeden durchgeführten Test den Ablauf und wertet die ermittelten Ergebnisse aus. Speziell für Performance-Tests werden hierbei die Werte zweier unterschiedlicher Ausführungsumgebungen gegenübergestellt und die Leistungsfähigkeit entsprechend bewertet. Abschließend werden im Kapitel „Zusammenfassung und Ausblick“ die wesentlichen Inhalte der Arbeit noch einmal zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

2. TESTS UND TESTAUTOMATISIERUNG

2.1. EINFÜHRUNG UND DEFINITION

In der Entwicklung von komplexen Softwaresystemen spielen Tests eine wichtige Rolle, um die korrekte Funktion von einzelnen Komponenten und der Funktionsweise des Systems selbst zu prüfen und dadurch die Qualität der Software überwachen zu können. Aufbauend auf diesen Tests einzelner Komponenten ist es möglich, das Zusammenspiel mit anderen, auch verteilten, Systemen zu überwachen und eine Integration zu gewährleisten. Ein Beispiel für solch ein Zusammenspiel sind Tests, die den Zugriff auf ein Backend-System über ein Frontend oder allgemein eine GUI[1] anhand von vordefinierten Szenarien prüfen.

Unter Testautomatisierung oder auch automatisierten Tests versteht man allgemein das automatische Ausführen von vordefinierten Tests zu bestimmten Zeitpunkten oder nach Aktionen, beispielsweise dem Hinzufügen neuer Komponenten zu einem System.
WhatIs definiert dies wie folgt:

> »Automated software testing is a process in which software tools execute pre-scripted tests on a software application before it is released into production.« [1]

In Zeiten der agilen Softwareentwicklung und nur kurzen Entwicklungszyklen haben automatisierte Tests eine neue Wichtigkeit erhalten, um die Qualität der Software zu überwachen und schon früh Rückmeldung über Fehler oder Inkompatibilitäten zu geben. Dieses ständige und regelmäßige Überprüfen der Software nach erfolgten Anpassungen wird als Continuous Integration (CI) bezeichnet.

Testautomatisierung lässt sich auf verschiedenen Ebenen umsetzen. Den größten Bereich umfasst dabei das Testen von isolierten Komponenten oder einzelnen Methoden (Unit-Test), darauf aufbauend Adapter- und Integrationstests des Systems. Die dritte und im Vergleich kleinste Stufe der Automation umfasst das Testen der Oberfläche im Zusammenspiel mit dem darunter liegenden System. Diese Aufteilung in verschiedene Teststufen bezeichnet man als Agile Test-Pyramide [2]. Erweiterbar ist dieses System durch manuelle explorative Tests.
Das Hauptziel ist dabei immer eine verbesserte Qualität der Software durch frühzeitiges Erkennen und Beheben von Fehlern.

Automatisierte Tests sind jedoch nicht als Ersatz für manuelles Testen zu verstehen. Sie erleichtern die Prüfung sich wiederholender Abläufe und können einen Überblick über den Zustand des Systems abbilden. Neben dem Aspekt verbesserter Qualität der Software spielt

[1] GUI - Graphical User Interface

auch der finanzielle Faktor eine Rolle, denn manuelles Testen nach jeder Anpassung am System erfordert Zeit und ist kostenintensiv. Durch die Automatisierung von immer wieder auszuführenden Standard-Tests kann in kürzerer Zeit geprüft werden, ob eine Anpassung am System nicht gewünschtes Verhalten hervorruft oder nicht.
Ein weiterer Vorteil von automatisierten Tests ist, dass diese nicht *betriebsblind* werden, während dies bei manuellem Test durchaus passieren kann. Prüft man immer wieder die gleichen Abläufe manuell, erreicht man unter Umständen einen Punkt, an dem bestimmte Abfragen aus Routine geprüft werden und dadurch Flüchtigkeitsfehler unterlaufen können.

Tests lassen sich für verschiedene Bereiche umsetzen. Diese Arbeit beschäftigt sich im Weiteren mit **Performance-Tests** sowie **Web-Tests** im Rahmen der Plattform AMCS.

2.2. PERFORMANCE-TEST

2.2.1. BEGRIFFE

PERFORMANCE

Unter Performance wird in der Informatik die Leistungsfähigkeit eines Rechners oder Systems verstanden. Sie kann über verschiedene Parameter bestimmt werden wie Datendurchsatz, Antwortzeit oder auch die maximal mögliche Anzahl an Anfragen, die ein System parallel abarbeiten kann. Eine Art von Parametern ist dabei zumeist abhängig von weiteren, beispielsweise kann die Antwortzeit nicht ohne die Betrachtung der parallel an das System gestellten Anfragen und die dadurch erzeugte Systemlast für die Performance-Bewertung herangezogen werden. [3, Kapitel 2.1]

LASTTEST

Mittels Lasttests ist es möglich, das Verhalten eines Systems bei einer hohen Anzahl paralleler Anfragen zu analysieren und zu bewerten. Dafür erzeugt man eine hohe Systemlast durch sich ständig wiederholende Abfragen an kostenintensive Ressourcen durch einen Nutzer oder durch die Simulation vieler paralleler Nutzer, die auf die gleiche Ressource zugreifen. Zur Bewertung des Systems wird die Zeit gemessen, die für Nutzeranfragen benötigt wird. Die Nutzeranzahl bleibt über die Dauer des laufendes Tests konstant. [3, Kapitel 2.1.2][4]

Man unterscheidet zwischen Last- und Performance-Tests. Performance-Tests oder auch -Messungen wiederholen einzelne Anfragen oder Testfälle und prüfen dadurch die Performanzeingeschaften und die Skalierbarkeit der getesteten Funktionen, während Lasttests einen Workflow prüfen und die Durchführung konkreter Vorgänge im System simulieren oder mit Testdaten tatsächlich durchführen. [5]

STRESSTEST

Von einem Stresstest spricht man, wenn ein System über eine als sicher definierte Grenze hinweg beansprucht wird. Bei dieser Art von Tests geht es darum zu prüfen, ob es zu Fehlverhalten, Dateninkonsistenzen oder Abstürzen kommt und das System überhaupt noch ansprechbar ist. Weiterhin kann geprüft werden, ob und wie sich ein System von einer Überlastung erholt und wieder in einen Normalzustand übergehen kann.

2.2.2. VORGEHENSWEISEN UND MODELLE

BENCHMARKS

Als Benchmarks bezeichnet man einheitliche Mess- und Bewertungsverfahren zur Bewertung und des Tests der Leistungsfähigkeit von Hardware in einem definierten Bereich. Über die dadurch ermittelten Ergebnisse ist ein Vergleich mit anderen Systemen möglich.
Im Rahmen dieser Arbeit werden Benchmarks nicht weiter betrachtet.

BLACK-BOX, WHITE-BOX, GREY-BOX

Im Gegensatz zu Benchmarks wird über die Box-Verfahren das System selbst anhand von Testdaten geprüft. Testet man Systeme, ohne die internen Strukturen zu kennen oder Wissen darüber, wie eingegebene Daten verarbeitet werden, spricht man von *Black-Box-Tests*. Man erhält einen allgemeinen und groben Überblick über die Dauer bestimmter Anfragen, kann im Detail jedoch nicht definieren, wo Zeit verloren geht. [3, Kapitel 2.3.2]

Bei *White-Box-Tests* hingegen kennt man die Systemstrukturen und arbeitet mit diesem Wissen. Ziel ist es, das Verhalten und die Ausgaben einzelner Softwarekomponenten gegen definierte Kriterien zu prüfen. Man unterscheidet dabei zwischen kontrollfluss- (Modul- und Integrationstests) und datenflussabhängigen (Unit- und Variablentests) Kriterien. [3, Kapitel 2.3.3]

Grey-Box-Tests sind eine vor allem bei Web-Applikationen oft genutzte Kombination aus den Vorteilen von Black- und White-Box-Tests. Man übergibt dem System Testdaten, kennt dabei die internen Strukturen des verarbeitenden Systems und kann dadurch potentielle Schwachstellen identifizieren. [3, Kapitel 2.3.4]

2.3. WEB-TEST

2.3.1. BEGRIFFE

WEB-TEST, FRONTEND-TEST

Unter *Web*- oder auch *Frontend-Tests* versteht man Tests speziell für Web-Applikationen. Mithilfe dieser Tests ist es möglich, Sicherheitsaspekte der Applikation, Grundfunktionalitäten und auch die Zugänglichkeit und Erreichbarkeit zu prüfen. Über die Simulation verschiedener Umgebungen wie Desktop oder Smartphone ist auch die Responsivität[2] der Oberfläche in Hinblick auf mögliche Darstellungsfehler testbar. In Kombination mit Lasttests lässt sich weiterhin das Verhalten der Web-Applikation bei einer hohen Anzahl von Anfragen prüfen. [6]
Durch die schnelle Weiterentwicklung im Bereich der Webtechnologien gibt es eine Vielzahl an unterschiedlichen Möglichkeiten und Werkzeugen für die Durchführung von Web-Tests.

FUNKTIONSTESTS, AKZEPTANZTESTS

Analog zu *Unit-Tests* im Backend ist es auch im Frontend möglich, die korrekte Funktionsweise von einzelnen Methoden anhand übergebener Testdaten zu überprüfen, beispielsweise die Validierung von Eingaben oder die Konvertierung von Daten in Objekte.

[2] Responsivität: Im Kontext von Webdesign die automatische Anpassung und Skalierung an das Ausgabegerät.

Für *Akzeptanztests* gibt es verschiedene Definitionen. *Software-Testing-Fundamentals* schreibt dazu:

> »**acceptance testing**: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.« [7]

Im Kontext von Web-Applikationen nutzt man Akzeptanztests zur Überprüfung von Nutzereingaben und um zu garantieren, dass Abläufe wie die Führung durch Formulare oder die Navigation über mehrere Unterseiten hinweg für den Nutzer immer abschließbar und verständlich sind. Oder mit anderen Worten, die Applikation sich den Erwartungen entsprechend verhält. Akzeptanztests werden oftmals auch in Verbindung mit *Plausibilitätstests* genutzt, um Nutzereingaben zu prüfen.

Weitere Testmöglichkeiten sind *Visual regression tests*, um die gerenderten Elemente einer Website auf Vollständigkeit zu prüfen sowie Tests der *Erreichbarkeit* und *Performance* der gesamten Anwendung. [8]

2.3.2. HEADLESS BROWSER

Als *Headless Software* bezeichnet man Anwendungen, die ohne eine graphische Nutzeroberfläche (GUI) auskommen. Auf Browser bezogen bedeutet dies, dass Webseiten aufgerufen und verarbeitet werden können, ohne sie dem Nutzer tatsächlich anzuzeigen. [9] Die Darstellung kann dabei auf einem virtuellen Display im Arbeitsspeicher umgesetzt werden. Im Kontext von Web-Tests werden Headless Browser für automatisierte Tests beispielsweise durch *Selenium* in einer *Jenkins*-Umgebung genutzt.

Moderne Browser wie *Google Chrome* oder *Mozilla Firefox* stellen Headless-Varianten für die Nutzung in Testumgebungen bereit, am Beispiel von Chrome über den *Chromedriver*. Weiterhin seien *HtmlUnit*, *PhantomJS* oder der *NodeJS headless browser* als weitere Optionen genannt. [10]

3. PERFORMANCE-TESTS

3.1. AUSWAHLKRITERIEN

Performance-Tests sind ein wichtiges Werkzeug, um die Grenzen eines Systems auszuloten und sicherzustellen, dass es auch unter hoher Last stabil und fehlerfrei funktioniert. Es existieren verschiedene Tools für unterschiedliche Anwendungsfälle. Um diese miteinander vergleichen zu können, sind einige Merkmale und Kriterien erforderlich.

PROTOKOLL- UND FORMATUNTERSTÜTZUNG

Einer der wichtigsten Punkte ist, dass das Tool die vom zu testenden System genutzten Protokolle und Antwortformate unterstützt und verarbeiten kann. Beispielsweise unterstützen nicht alle Tools Web-Sockets. Einige Tools können nur auf API-Tests mit SOAP oder REST spezialisiert sein, andere wiederum bieten eine Bandbreite an unterstützten Technologien. Über *JMeter* ist unter anderem ein Testen von Datenbanken möglich, während *Loader.io* nur reine HTTP/S-Anfragen unterstützt.

NUTZERSIMULATION

Ein wichtiges Kriterium ist die Unterstützung von mehreren verschiedenen virtuellen Nutzern und deren Eigenschaften. Kann das Tool nur einen Nutzer für die Anfragen nutzen oder mehrere, auch mit Daten aus externen Quellen wie einer CSV-Datei? In diesem Zusammenhang ist es auch wichtig zu wissen, ob die einzelnen Nutzer in einer gesamten Session arbeiten oder ob jeder Nutzer in einer separaten Session mit eigenen Daten arbeitet. Auch relevant ist, ob das Tool alle Nutzer gleichzeitig oder über einen definierten Zeitraum verteilt ausführen kann. Der zweite Fall führt zu einem realistischerem Verhalten und ist daher zu bevorzugen.

Zu beachten ist dabei, dass mit steigender Anzahl von virtuellen Nutzern auch die Logs und Auswertungen größer werden. Das Tool muss mit diesen anfallenden Datenmengen umgehen können.

TESTERSTELLUNG UND WORKFLOWS

Workflows, oder auch vordefinierte Arbeitsabläufe, sollen ein realistisches Abfrageverhalten durch einen Nutzer nachstellen. Ein wichtiges Kriterium ist daher, ob das Tool die Aufzeichnung von Interaktionen oder auch die manuelle Erstellung von Abläufen unterstützt und Abhängigkeiten zwischen einzelnen Anfragen abbilden kann. Beispielsweise sei die notwendige Authentifizierung an einer API genannt, um weitere Abfragen durchführen zu können.

Da die manuelle Erstellung komplexerer Tests recht zeitaufwendig werden kann, ist eine Möglichkeit zur Aufzeichnung durch ein Tool als Vorteil anzusehen.

VERTEILTES TESTEN

Es gibt Fälle, in denen eine einzelne Testinstanz nicht ausreicht, um eine geforderte hohe Last durch parallele Nutzer auf dem Zielsystem zu erzeugen. Für diesen Fall ist es erforderlich, dass mehrere Instanzen zur Lasterzeugung miteinander kombiniert werden können, um das gesetzte Ziel zu erreichen. Es gibt Tools, die dies über eine Cluster- oder auch Master-Slave - Konfiguration ermöglichen. Wichtig bei verteiltem Testen ist, dass die anfallenden Daten und Ergebnisse der einzelnen Instanzen sinnvoll aggregiert und ausgewertet werden können. Für *Gatling* beispielsweise muss dies manuell durchgeführt werden.

AUFBEREITUNG DER ERGEBNISSE

Um das zu testende System bewerten zu können, müssen die Testergebnisse analysiert und ausgewertet werden. Ein notwendiger Punkt für das Tool ist daher die Bereitstellung der Testergebnisse in verschiedenen Formen wie Berichten, Graphen und Statistiken für eine weitere Verarbeitung und Analyse. Wünschenswert ist auch eine Möglichkeit, Rohdaten zur Nutzung in anderen Analysewerkzeugen extrahieren zu können.

EINRICHTUNG UND BEDIENBARKEIT

Ein wichtiges Kriterium ist, wie aufwendig oder kompliziert die Einrichtung des Tools und die Einarbeitung in die Syntax ist. Gute Tools bieten eine umfassende und ausführliche Dokumentation und einfach zu erweiternde Beispiele und erleichtern somit den Einstieg in die Erstellung von den eigenen Anforderungen entsprechenden Tests.

3.2. GATLING

Gatling [11] ist ein in Scala geschriebenes Framework der *Gatling Corp* für Last- und Stress-Tests. Es steht unter einer Apache 2.0 Lizenz und ist als Open Source verfügbar. Die erste stabile Version wurde im Jahr 2012 veröffentlicht, die hier betrachtete Version 2.2.5 erschien im April 2017 und ist für viele Plattformen erhältlich. Gatling stellt einige Erweiterungen für die Integration in bestehende Testprozesse und Werkzeuge wie Jenkins bereit. Für den Enterprise-Bereich gibt es eine kostenpflichtige Erweiterung „*Frontend Gatling FrontLine*“ für Live-Monitoring während der Testausführung und der Integration in Cloud-Umgebungen wie die „Amazon Web Services“ (AWS).

PROTOKOLL- UND FORMATUNTERSTÜTZUNG

Gatling unterstützt HTTP und HTTPS für Anfragen, weiterhin können *Web-Sockets*, *Server-Side-Events* und *Java-Message-Service* genutzt werden. Das Ausfüllen und Absenden von Formularen ist ebenso möglich wie die Verarbeitung verschiedener Antwortformate wie HTML, XML oder JSON. JavaScript und darauf basierende dynamische Inhalte kann Gatling jedoch nicht verarbeiten.

NUTZERSIMULATION

Gatling bietet verschiedene Möglichkeiten zur Simulation unterschiedlicher Nutzerzahlen zu bestimmten Zeitpunkten[3]. Der einfachste Fall ist eine feste Anzahl von Nutzern über den gesamten Testzeitraum hinweg. Weiterhin können eine konstante oder ansteigende Anzahl an Nutzern über einen Zeitraum verteilt, neue Nutzer pro Zeiteinheit oder Nutzer verteilt anhand der „Heaviside-step-function" konfiguriert werden. Jeder einzelne virtuelle Nutzer arbeitet dabei in einer eigenen Session unabhängig von den anderen Nutzern.

TESTERSTELLUNG UND WORKFLOWS

Um Tests zu erstellen bietet Gatling die Möglichkeit, Interaktionen im Browser über einen Rekorder aufzuzeichnen und abzuspeichern (siehe Abbildung 3.1). Dafür muss der Datenverkehr über einen durch Gatling gestarteten Proxy geleitet werden.

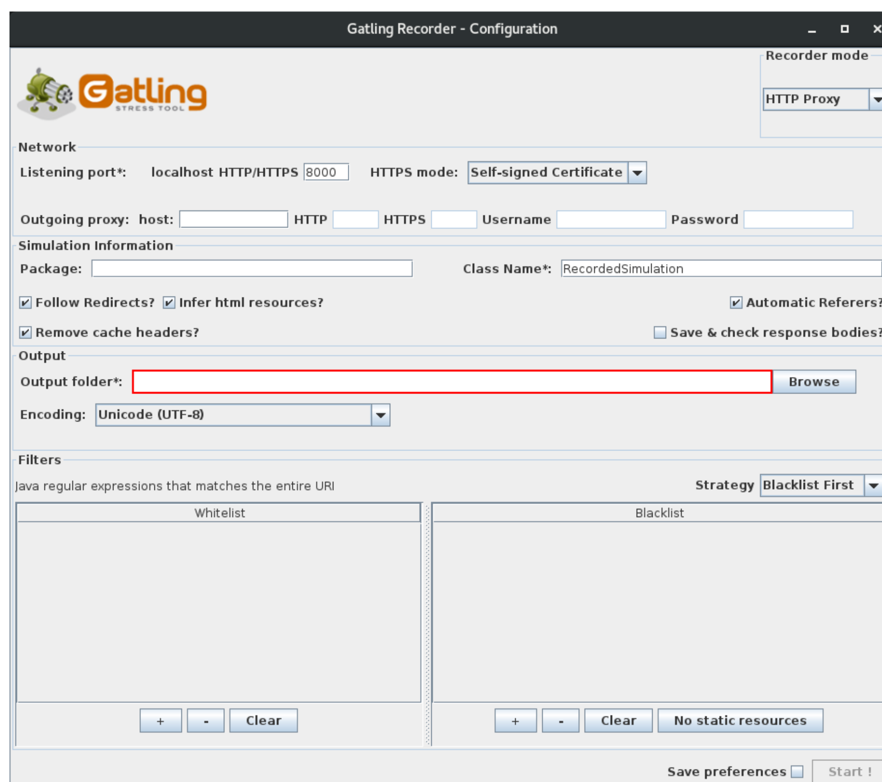


Abbildung 3.1.: Gatling: Recorder Konfiguration

Im folgenden Schritt (Abbildung 3.2) werden die Interaktionen im Browser inklusive der Wartezeiten aufgezeichnet. Das Ergebnis wird in einer Scala-Klasse abgespeichert und kann manuell editiert werden. Dafür stellt Gatling eine *Domain-Specific-Language* (DSL) bereit.

Gatling bietet die Möglichkeit, Serverantworten aus vorangegangenen Anfragen zu parsen und wiederzuverwenden. Dadurch ist die Abbildung von komplexeren Workflows möglich.

VERTEILTES TESTEN

In einige Situationen kann die vorhandene Netzwerk- oder Systemkapazität nicht ausreichen, um eine gewünschte hohe Last zu erzeugen. Für diesen Fall ist es möglich, Gatling-Instanzen

[3] http://gatling.io/docs/current/general/simulation_setup/

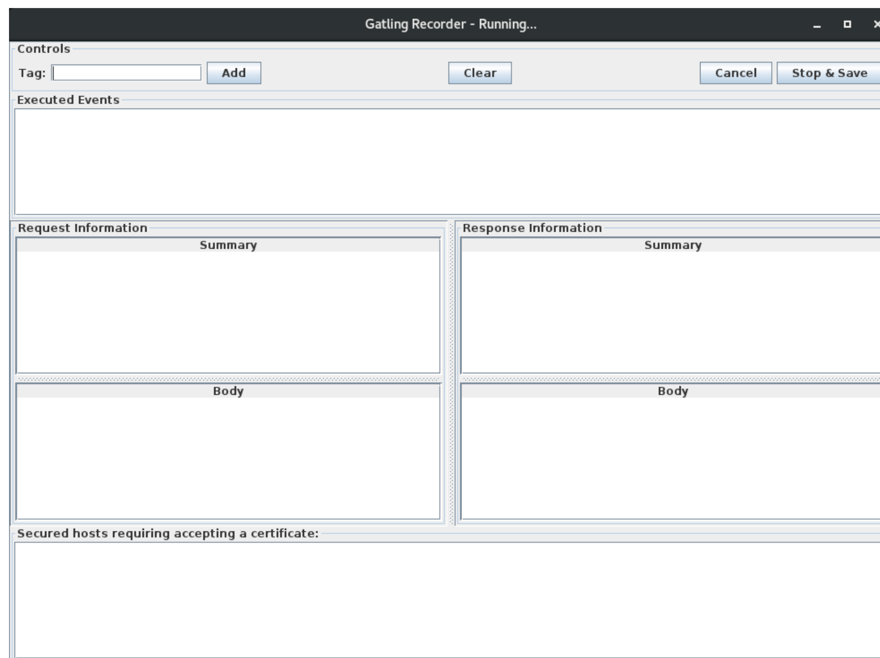


Abbildung 3.2.: Gatling: Recorder Aufzeichnung

auf verschiedenen Systemen zu installieren und entsprechend zu konfigurieren. Die Testergebnisse der einzelnen Instanzen können gebündelt ausgewertet werden[4]. Die Konfiguration muss aktuell manuell vorgenommen werden, da Gatling bisher keinen Cluster-Modus unterstützt.

AUFBEREITUNG DER ERGEBNISSE

Nach Ablauf der Tests erstellt Gatling verschiedene Übersichten und Diagramme (siehe Abbildung 3.3) sowohl für den gesamten Test als auch die einzelnen Unteraufrufe. Nutzt man mehrere Testsysteme, müssen die jeweiligen Ergebnisse vor der Erstellung der Auswertungen zentral gebündelt werden.

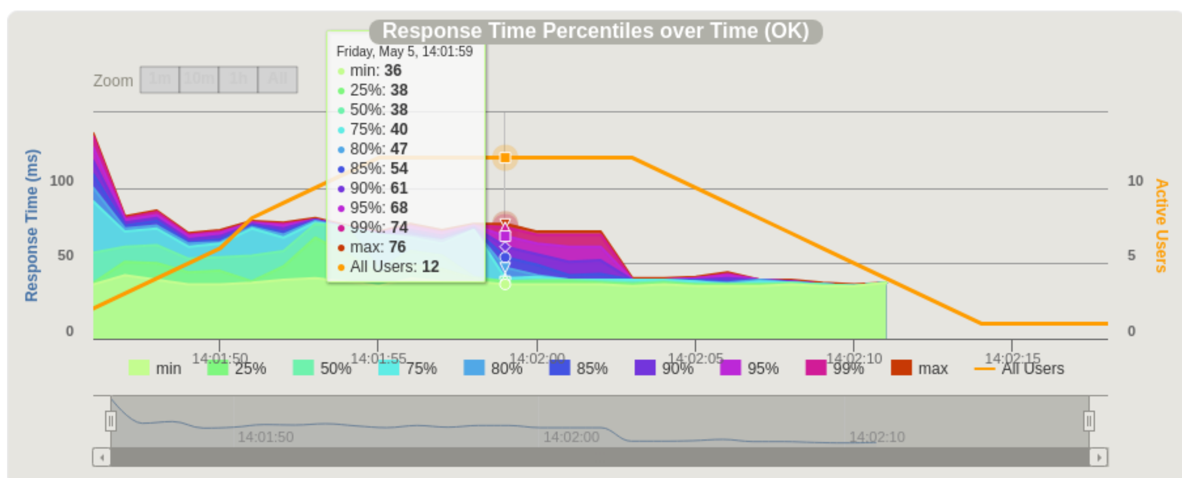


Abbildung 3.3.: Gatling: Beispielauswertung

[4] http://gatling.io/docs/current/cookbook/scaling_out/

EINRICHTUNG UND BEDIENBARKEIT

Gatling wird gepackt als ZIP-Archiv für Windows und Linux-Systeme angeboten. Die Einrichtung besteht aus dem Entpacken des Archivs und der Proxy-Konfiguration beispielsweise direkt im Browser. Für die Ausführung der Oberfläche des Rekorder und dem Start der Test-Suite werden sowohl Batch- als auch Shell-Skripte bereitgestellt. Die Konfiguration ist über die Oberfläche hinaus manuell über die Gatling-DSL möglich, eigene Tests oder Erweiterungen für bestehende können ohne tiefer gehende Scala-Kenntnisse erstellt werden. Weiterhin bietet die Website eine umfassende Dokumentation und Implementationsbeispiele. Für die Auswertung der Ergebnisse generiert Gatling HTML-Dateien, die mit einem Browser aufrufbar sind.

3.3. JMETER

JMeter [12] ist ein in Java geschriebenes Tool der *Apache Software Foundation* für Performance-Messungen und Lasttests. Es steht ebenso wie Gatling unter einer Apache 2.0 Lizenz und ist als Open Source verfügbar. Die erste stabile Version wurde bereits im Jahr 1998 veröffentlicht, die hier genutzte Version 3.2 erschien im April 2017 und ist für alle gängigen Plattformen erhältlich. Mit *JMeter Plugins* gibt es ein von JMeter unabhängiges Projekt zur Bereitstellung vieler verschiedener Plugins. Weiterhin stellt JMeter eine voll funktionsfähige IDE für die Erstellung und Konfiguration von komplexen Tests bereit und bietet eine gute Integration in CI-Systeme wie Jenkins oder Gradle.

PROTOKOLL- UND FORMATUNTERSTÜTZUNG

JMeter bietet Unterstützung für viele Protokolle, Formate und Architekturen. Unterstützt werden unter anderem HTTP und HTTPS, SOAP- und REST-Architekturen oder auch Mail-Protokolle und Datenbanken. Über Plugins kann Unterstützung für Web-Sockets und weitere Protokolle nachinstalliert werden. Möglichkeiten zur Verarbeitung verschiedener Antwortformate wie HTML, JSON oder XML sind gegeben. Dynamische Inhalte und JavaScript werden nicht unterstützt, da JMeter wie auch Gatling auf Protokollebene arbeitet.

NUTZERSIMULATION

JMeter bietet verschiedene Möglichkeiten zur Simulation unterschiedlicher Nutzerzahlen zu bestimmten Zeitpunkten[5]. Dafür arbeitet JMeter mit *Thread Groups*. Der einfachste Fall ist, mit einer festen Anzahl an Nutzern direkt zu Testbeginn zu arbeiten. Die zweite Möglichkeit ist, über einen definierten Zeitraum hinweg dem System immer mehr Nutzer hinzuzufügen, bis eine definierte Anzahl erreicht ist. JMeter erstellt dabei für jeden simulierten Nutzer einen separaten Thread und lässt in diesem die konfigurierten Tests ablaufen. Weiterhin ist es möglich, die maximale Laufzeit eines Threads und eine Startverzögerung zu konfigurieren.

Die Abbildung 3.4 zeigt eine Beispielkonfiguration mit 500 virtuellen Nutzern, die schrittweise innerhalb von 10 Sekunden gestartet werden und die konfigurierten Testschritte ausführen.

[5] http://jmeter.apache.org/usermanual/test_plan.html#thread_group

Abbildung 3.4.: JMeter: Beispielkonfiguration Thread Group

TESTERSTELLUNG UND WORKFLOWS

Um Tests zu erstellen bietet JMeter ein Plugin für *Mozilla Firefox* an, um Interaktionen im Browser aufzuzeichnen und abzuspeichern. Eine weitere Möglichkeit ist die Konfiguration eines Proxy, über den der Datenverkehr geleitet wird. Die Aufzeichnung der Interaktionen wird in diesem Fall durch JMeter selbst durchgeführt. In beiden Fällen können die aufgezeichneten Workflows manuell editiert werden.

Die dritte und umfangreichste Möglichkeit ist das manuelle Erstellen von Workflows über die Oberfläche (siehe Abbildung 3.5). JMeter bietet viele verschiedene Objekte für *Requests*, *Controller*, *Listener* und weitere Konfigurationselemente, um komplexe Vorgänge abbilden zu können. Über Parametrisierung können Serverantworten aus vorangegangenen Anfragen verarbeitet und wiederverwendet werden. Das Nachstellen von Nutzerverhalten zur Lasterzeugung ist dadurch recht einfach möglich.

Die Testerstellung ist hierbei, entweder über den Rekorder oder die Oberfläche, GUI-basiert möglich. Weitere Kenntnisse in der Programmierung sind hierfür nicht erforderlich.

VERTEILTES TESTEN

JMeter bietet die Möglichkeit, mehrere Systeme zu koppeln, um beispielsweise einen Stress-Test mit sehr hoher Last durchzuführen. Unterschieden wird dabei in eine Master- und mehrere Slave-Instanzen, die von dem Master angesteuert werden[6]. Für die Ausführung auf den Slaves bietet JMeter einen Server-Modus, in dem auf RMI-Eingaben (*Remote-Method-Invocation*) über den Master gewartet wird, um das Zielsystem zu testen. Die Ergebnisse der einzelnen Instanzen werden von der Master-Instanz aggregiert und nach Beendigung aller Tests ausgewertet.

[6] http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html

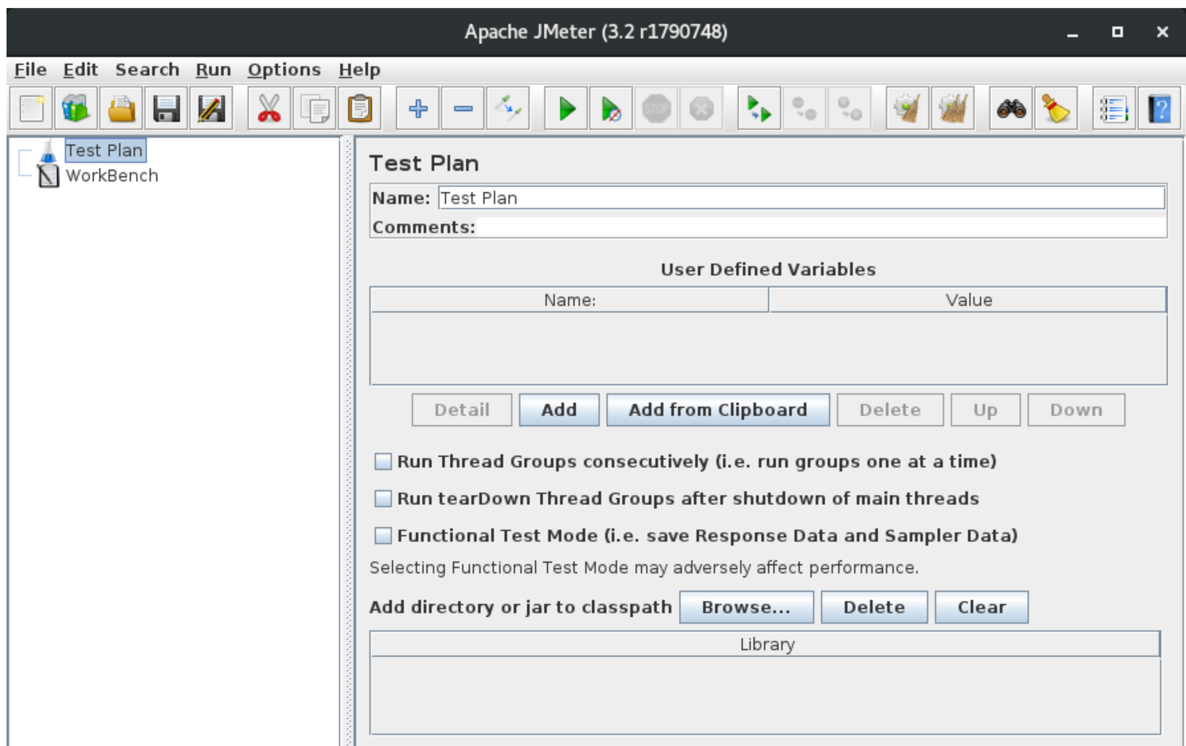


Abbildung 3.5.: JMeter: Oberfläche

AUFBEREITUNG DER ERGEBNISSE

Über *Listener* bietet JMeter eine Anzahl an verschiedenen Darstellungsmöglichkeiten für Testergebnisse an. Es können diverse Ergebnisgraphen (siehe Abbildung 3.6), verschiedene Reports für Einzelaufrufe oder Routen, Analysen über Antwortzeiten und weitere Module genutzt werden, um die Ergebnisse aussagekräftig zu präsentieren. Über *JMeter Plugins* erhält man Zugriff auf einige weitere Möglichkeiten zur Auswertung.
Nach Durchführung einer Test-Suite erstellt JMeter eine HTML-Auswertung, die über den Browser aufrufbar ist.

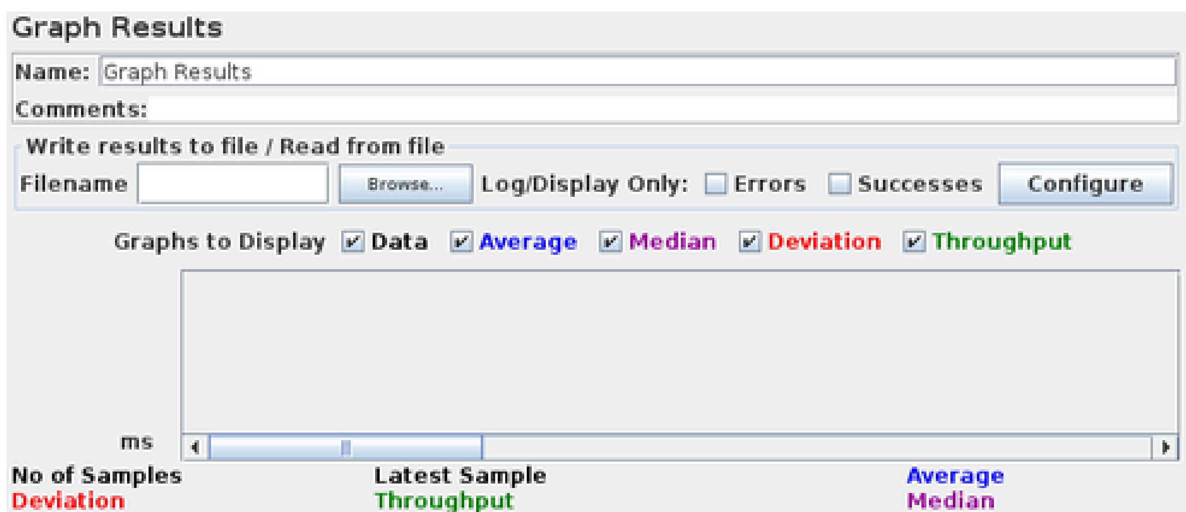


Abbildung 3.6.: JMeter: Beispiel Graph-Auswertung

EINRICHTUNG UND BEDIENBARKEIT

JMeter wird gepackt für alle gängigen Plattformen angeboten. Die Einrichtung besteht auch hier aus dem Entpacken des Archivs und der Proxy-Konfiguration im Browser, möchte man die Rekorderfunktion nutzen. Vorausgesetzt wird für die in dieser Arbeit genutzte Version eine *Java 8*-Installation. Es werden verschiedene Batch- und Shell-Skripte für die einzelnen Umgebungen wie die Entwicklungsoberfläche oder den Server-Modus bereitgestellt.

Die Einarbeitung in JMeter kann je nach Komplexität des abzubildenden Workflows etwas komplizierter sein aufgrund der Mächtigkeit und des Umfangs, den das Tool bietet. Über die Website werden jedoch eine umfassende Dokumentation und mehrere Implementationsbeispiele angeboten, um den Einstieg zu erleichtern.

3.4. WEITERE TOOLS

Im Bereich der Performance-Tests gibt es viele weitere Tools für unterschiedliche Anwendungszwecke. Einige davon werden nun, unterteilt in Kommerziell und Open-Source, aufgelistet [13].

KOMMERZIELL / CLOSED-SOURCE

- Cloud Test (SOASTA)
- Loader.io (SendGrid Labs)
- Silk Performer (Borland)
- Visual Studio (Microsoft)

OPEN-SOURCE

- OpenSTA
- Siege
- loadUI (SmartBear Software)

3.5. ZUSAMMENFASSUNG

Im Folgenden werden *Gatling* und *JMeter* tabellarisch anhand der definierten Kriterien miteinander verglichen und eine Auswahl für die durchzuführenden Tests getroffen.

Kriterium	Gatling	JMeter
Protokoll- und Formatunterstützung	Alle notwendigen Formate direkt unterstützt	Alle notwendigen Formate direkt oder über Plugins unterstützt
Nutzersimulation	Diverse Konfigurationsmöglichkeiten Sofortige Anzahl, über Zeitraum hinweg und weitere	Eingeschränkte Konfigurationsmöglichkeiten Sofortige Anzahl oder über Zeitraum hinweg
Testerstellung und Workflows	Rekorder-Funktion, manuelles Editieren, Workflows möglich	Rekorder-Funktion, Workflows möglich, mächtige GUI
Verteiltes Testen	Über manuelle Konfiguration möglich, keine offizielle Unterstützung	Master-Slave-Konfiguration möglich, offiziell unterstützt
Aufbereitung der Ergebnisse	Umfangreiche Auswertung in HTML Cluster-Auswertung nur manuell möglich	Verschiedene Reports zur Auswahl (Graphen, Statistiken etc.) Umfangreiche Auswertung in HTML Cluster-Auswertung automatisiert möglich
Einrichtung und Bedienbarkeit	Einfache Installation Umfangreiche Dokumentation GUI für Testerstellung und Rekorder Test-Klassen über DSL anpassbar	Einfache Installation Umfangreiche Dokumentation Sehr mächtige GUI für Testerstellung und Rekorder

Tabelle 3.1.: Vergleichskriterien für Performance-Test-Tools

Beide Tools sind grundsätzlich im gleichen Maße für die Durchführung der geplanten Performance-Tests geeignet. Gatling bietet durch seine DSL einen einfachen Einstieg, JMeter bietet eine umfangreiche und mächtige GUI für die Testerstellung. Weiterhin können beide Tools in eine CI-Umgebung wie Jenkins integriert und damit in eine automatisierte Testkette eingebunden werden.

Für die prototypische Implementation und Durchführung der Tests wurde sich für JMeter aufgrund des größeren Umfangs und der GUI zur Testerstellung entschieden.

4. WEB-TESTS

4.1. AUSWAHLKRITERIEN

Web-Tests sind ein wichtiges Werkzeug, um zum einen die Integration von Frontend und Backend zu prüfen, und zum anderen die Funktionalität der Seite selbst in unterschiedlichen Browsern zu gewährleisten. Es existieren verschiedene Tools, die unterschiedliche Ansätze und Aufgaben übernehmen und daher nicht für jeden Anwendungsfall geeignet sind. Es gibt reine Test-Frameworks zum Erstellen von Tests, und weiterhin kombinierte Tools bestehend aus einem Test-Runner wie *Selenium* und einem Test-Framework. Die meisten aktuellen Tools setzen auf diese Kombination.

VERARBEITUNG DER INHALTE

Moderne Webseiten bestehen nicht nur aus statischen Inhalten, sondern werden durch dynamisch generierte und dem Nutzerverhalten angepasste Elemente und Inhalte erweitert und verändert. Viele aktuelle JavaScript-Frameworks wie *Ember.js* oder *Angular* setzen dabei auf das Prinzip der *Single-Page-Application* (SPA). Dabei besteht die Website für den Browser aus nur einer Seite, deren Inhalt dynamisch über JavaScript angepasst wird.
Ein wichtiges Kriterium ist daher, dass das Tool mit asynchron erstellten Inhalten umgehen und entsprechende DOM-Events (*Document Object Model*) verarbeiten kann.

INTEGRATION IN CI-SYSTEME

Tests werden nicht zur einmaligen Ausführung erstellt, sondern sollen die Anwendung im Gesamten sowohl regelmäßig als auch bei erfolgten Anpassungen überprüfen. Um dies zu erleichtern, ist eine einfache Integrationsmöglichkeit in CI-Systeme wie Jenkins und die Auswertung der Testergebnisse durch diese erforderlich und als ein Bewertungskriterium anzusehen.

AUFBEREITUNG DER ERGEBNISSE

Um Fehlerquellen bei fehlgeschlagenen Tests schnell identifizieren zu können, müssen die Testergebnisse analysiert werden. Eine visuelle Darstellung des Fehlers als Screenshot ist ebenso wünschenswert. Notwendige Eigenschaften des Tools sind daher die Aufbereitung der Ergebnisse in lesbarer Form und die automatische Erstellung von Screenshots im Fehlerfall.

EINRICHTUNG UND BEDIENBARKEIT

Auch bei Tools für Web-Test ist ein wichtiger Aspekt, wie viel Zeit erforderlich ist, um sich in das Tool einzuarbeiten und erste Ergebnisse zu erzielen. Da der Fokus auf der Entwicklung von Tests liegt, sollten Tools den Einstieg erleichtern, intuitiv zu bedienen sein und nur einen geringen Overhead mitbringen. Eine umfangreiche Dokumentation und verständliche Anwendungsbeispiele sind daher wichtige Kriterien für die Bewertung.

4.2. MAVEN UND SELENIUM

Maven [14] ist ein in Java geschriebenes Projekt- oder Build-Management-Tool der *Apache Software Foundation* und kann, beispielsweise in Verbindung mit Selenium, dazu genutzt werden, um automatisierte Builds und Frontendintegrationstests durchzuführen. Die erste Version erschien im Jahr 2004, die hier betrachtete Version 3.5 wurde im April 2017 für alle gängigen Plattformen veröffentlicht. Maven nutzt das *Project-Object-Model* (POM), repräsentiert in einer zentralen XML-Datei, um alle wichtigen Abschnitte im Lebenszyklus eines Projekts vom Build über Reporting und Dokumentation zu steuern und zu verwalten. Über eine Schnittstelle ist Maven um zusätzliche Funktionen erweiterbar, die Plugins werden als *Mojo* (Maven (plain) old Java Object) bezeichnet. Jede Erweiterung definiert ein oder mehrere *Goals* (ausführbare Kommandos in Maven) und implementiert dafür ein von Maven bereitgestelltes Java-Interface.

VERARBEITUNG DER INHALTE

Mit Maven selbst lassen sich keine Tests erstellen, sondern nur vorhandene ausführen. Für die Umsetzung eines Web-Tests wird daher das Selenium-Framework benötigt. Die Tests selbst werden in Java erstellt und über einen Maven-Build ausgeführt.
Selenium bietet eine leistungsstarke API zur Verarbeitung von HTML-Inhalten. Nachteilig ist, dass Selenium nicht für dynamische Inhalte konzipiert wurde und es daher, als einfachste Lösung, notwendig ist, an verschiedenen Stellen im Test einen *Sleep*-Befehl zu hinterlegen, damit die asynchronen Inhalten geladen werden können. Ohne diese Wartebefehle würden die Tests fehlschlagen. Adressiert wurde dieses Problem in Test-Lösungen neuerer JavaScript-Frameworks wie *Ember.js*, welche dies durch ein auf Selenium aufsetzendes und erweiterndes Overlay lösen. [15]

Der folgende Codeausschnitt zeigt ein kleines Beispiel für einen in Java erstellten Test mit Selenium. In einer Before-Anweisung wird zuerst der zu nutzende Browser definiert (in diesem Fall Google Chrome) und danach die zu testenden Website (Startseite von Google) aufgerufen. Der eigentliche Test prüft zuerst den Seitentitel gegen einen Erwartungswert und führt danach eine Keywortsuche aus. Über die *submit*-Routine kann das Formular direkt abgeschickt werden, ohne noch extra den Button aus dem HTML filtern zu müssen. Nach dem Absenden wird für 10 Sekunden gewartet und danach erneut der Seitentitel geprüft. Erwartet wird, dass der Titel nun mit dem zuvor eingegebenen Suchbegriff beginnt.
Über Annotationen wie *@Before* können Methoden in Hooks gruppiert werden.

```
1 @Before
2 public void openBrowser()
3 {
4   baseUrl = "https://www.google.com";
5   driver = new ChromeDriver();
6   driver.get(baseUrl);
7 }
```

```
8
9  @Test
10 public void testPageTitle() throws IOException
11 {
12   assertEquals("Page title equals Google", "Google", driver.getTitle());
13   WebElement searchField = driver.findElement(By.name("q"));
14   searchField.sendKeys("Selenium");
15   searchField.submit();
16   assertTrue("Page title now starts with search string",
17     (new WebDriverWait(driver, 10)).until(new ExpectedCondition() {
18       public Boolean apply(WebDriver d) {
19         return d.getTitle().toLowerCase().startsWith("selenium");
20       }
21     })
22   );
23 }
```

Quelltext 4.1: Code-Beispiel: Maven und Selenium

INTEGRATION IN CI-SYSTEME

Sowohl Maven als auch Selenium sind einzeln in Jenkins oder andere CI-Systeme integrierbar. Durch die Einbindung weiterer Plugins lassen sich automatisierte Reports und Statistiken zur Testauswertung generieren. Nutzt man Maven als Build-Tool, dann entfällt die separate Integration von Selenium, dies wird direkt von Maven übernommen.

AUFBEREITUNG DER ERGEBNISSE

Selenium selbst bietet keine Möglichkeit zur Erstellung von Test-Reports. Es gibt jedoch verschiedene Plugins, die diese Aufgabe übernehmen können. Eine Möglichkeit ist die Nutzung der in Java geschriebenen Erweiterung *TestNG*, um eine Zusammenfassung zu erstellen. Eine zweite Möglichkeit ist *JUnit*, ebenfalls in Java geschrieben. JUnit ermöglicht die Erstellung von unter anderem HTML-Reports.
Eine dritte Möglichkeit bietet die Bibliothek *Extent*. Ebenfalls in Java geschrieben ermöglicht dieses Plugin die Erstellung von ausführlichen HTML-Reports mit Screenshots aufgetretener Fehler. [16]

EINRICHTUNG UND BEDIENBARKEIT

Maven bietet über das Apache-Projekt Zugriff auf eine sehr ausführliche und umfangreiche Dokumentation für die Installation und das Arbeiten mit dem Management-Tool. Angebotene Plugins und die Hauptelemente werden in der Dokumentation einzeln aufgeführt und die jeweiligen Nutzungsmöglichkeiten beschrieben. Die meisten aktuellen IDEs, wie beispielsweise *NetBeans*, bieten eine direkte Integration von Maven und die Konfiguration von Tests mit Selenium. Trotzdem ist der Einstieg in Maven nicht trivial und erfordert einige Einarbeitungszeit sowohl in Maven als auch Selenium selbst.

4.3. WEBDRIVERIO

WebdriverIO [17] ist ein Browser-Automation-Tool mit integriertem Test-Runner auf Basis von Selenium. Es unterstützt sowohl *Test-Driven-* (TDD) als auch *Behavior-Driven-Development* (BDD) mit gängigen Frontend-Test-Frameworks wie *Jasmine* oder *Mocha*. Die erste lauffähige Version des Projekts erschien im Mai 2012 unter dem Namen *WebdriverJS* und wurde

2014 umbenannt. Die hier betrachtete Version 4.8 wurde im April 2017 veröffentlicht und steht unter der MIT-Lizenz. WebdriverIO lässt sich einfach in ein vorhandenes Projekt integrieren. Je nach genutztem Werkzeug für die Projektverwaltung, beispielsweise *grunt* oder *gulp*, gibt es entsprechende Plugins zur Einbindung.

Neben WebdriverIO gibt es noch zwei weitere ähnliche Frameworks beziehungsweise Tools, *WD.js* und *selenium-webdriverjs*. Beide arbeiten nach dem gleichen Prinzip wie WebdriverIO und auf Basis von Selenium, befinden sich dabei aber in unterschiedlich weit vorangeschrittenen Entwicklungsstadien und unterscheiden sich weiterhin in der genutzten Syntax für die Erstellung von Tests.

VERARBEITUNG DER INHALTE

WebdriverIO bietet über eine eigene API umfassende Unterstützung zur Verarbeitung von statischen und dynamisch generierten Inhalten. Über Anweisungen wie *pause()* oder verschiedene *waitFor*()*-Methoden können Tests unterbrochen werden, bis asynchrone Inhalte fertig geladen und verarbeitbar sind. WebdriverIO adressiert damit das Hauptproblem von SPA und macht diese leicht testbar.
Tests werden nicht in Java, sondern JavaScript erstellt und sind damit einfach in existierende Projekte einbindbar. Das folgende Beispiel zeigt in komprimierter Form wieder den Aufruf von Google mit anschließender Suche. WebdriverIO arbeitet viel mit Selektoren, ein guter Quellcode der zu testenden Seite mit IDs und CSS-Klassen an den entsprechenden Elementen ist daher von Vorteil.

```
1  var webdriverio = require('webdriverio');
2  var options = { desiredCapabilities: { browserName: 'chrome' } };
3  var client = webdriverio.remote(options);
4
5  client
6    .init()
7    .url('https://www.google.com')
8    .setValue('#q','Selenium')
9    .click('#btnG')
10   .getTitle().then(function(title) {
11     console.log('Page title starts with search string: ' + title);
12   })
13   .end();
```

Quelltext 4.2: Code-Beispiel: WebdriverIO

INTEGRATION IN CI-SYSTEME

WebdriverIO bietet eine enge Integration in Jenkins[7]. In Kombination mit *JUnit* ist somit eine einfache Auswertung und Fehlersuche möglich. Neben Jenkins werden noch verschiedene Integrationsmöglichkeiten in Services wie *BrowserStack*, *Sauce* und weitere angeboten. Lokal erstellte Tests können somit einfach auf einer großen Anzahl von Plattformen und Browsern ausgeführt werden, um größtmögliche Kompatibilität zu gewährleisten.

[7] http://webdriver.io/guide/testrunner/jenkins.html

AUFBEREITUNG DER ERGEBNISSE

WebdriverIO bietet eine große Anzahl von Möglichkeiten zur Auswertung der Testergebnisse. Von *JUnit* über *Spec* bis zur Integration in *TeamCity* oder einfaches JSON sind verschiedene Reports möglich. Für die gewünschte Integration werden jeweils Plugins bereitgestellt[8].

EINRICHTUNG UND BEDIENBARKEIT

Für die Installation wird ein npm-Plugin[9] bereitgestellt. Über eine Konfigurationsdatei sind Anpassungen am Test-Runner und der Test-Ausführung allgemein möglich, beispielsweise kann der zu verwendende Browser konfiguriert werden. Für einen schnellen Einstieg stellt WebdriverIO eine umfangreiche API-Dokumentation und einen Developer-Guide bereit. Weiterhin wird für jede unterstützte Integration von *Reporters*, *Services* und anderen Erweiterungen jeweils ein eigenes Plugin zur Installation bereitgestellt und die Einrichtung und Nutzung kurz erläutert.

4.4. PROTRACTOR

Protractor [18] ist ein Test-Framework und Browser-Automation-Tool für *Angular* von *Google* und unter einer OpenSource-Lizenz veröffentlicht. Die ersten Versionen erschienen im Sommer 2013, seitdem unterliegt das Projekt einer stetigen Weiterentwicklung. Die hier verwendete Version 5.1.2 wurde im Mai 2017 unter der MIT-Lizenz veröffentlicht.
Protractor basiert, ähnlich wie WebdriverIO, auf Selenium-WebDriverJS und erweitert den Runner und die API um weitere Funktionen und Besonderheiten, um SPA und Angular-Anwendungen im Besonderen besser verarbeiten zu können. Obwohl in der Hauptsache für *Angular* und *AngularJS* entwickelt, kann es mit wenigen Anpassungen auch für andere JavaScript-Frameworks wie *Ember.js* verwendet werden.

VERARBEITUNG DER INHALTE

Protractor bietet eine Unterstützung für viele verschiedene Test-Frameworks. Neben Jasmine als voreingestelltem Standard können Mocha, Cucumber und weitere Frameworks konfiguriert und den eigenen Anforderungen entsprechend angepasst werden. Im Abschnitt 4.5 wird noch etwas näher auf Jasmine und Mocha eingegangen.
Um auch Anwendungen zu testen, die nicht auf Angular basieren, werden Methoden zur Deaktivierung der entsprechenden Routinen angeboten. Das folgende Beispiel fragt wieder die Google-Website an und prüft vor und nach einer Suche den Seitentitel. Damit dies funktioniert, muss vor dem Aufruf der Seite *waitForAngularEnabled(false)* ausgeführt werden. Die Syntax der Tests ähnelt dabei der von WebdriverIO.

Ein großer Unterschied ist hierbei die Herangehensweise an die Testerstellung. Jasmine als Standard ist ein BDD-Framework, was sich in der Syntax widerspiegelt. Die erstellten Tests haben den Anspruch, einfach gelesen und verstanden werden zu können. Über beispielsweise *beforeEach*- oder *afterEach*-Anweisungen können Abläufe definiert werden, die bei jedem Einzeltest auszuführen sind. Im folgenden Beispiel wird die Angular-Routine deaktiviert, gefolgt vom Aufruf der Startseite für die einzelnen Tests.

---

[8]Beispiel: `http://webdriver.io/guide/reporters/junit.html`
[9]npm - Node.js package manager

```
describe('Simple Google Search', function() {
  beforeEach(function() {
    browser.waitForAngularEnabled(false);
    browser.get('http://www.google.com');
  });

  it('has default title', function() {
    expect(browser.getTitle()).toEqual('Google');
  });

  it('contains keyword in title', function() {
    element(by.name('q')).sendKeys('Selenium');
    element(by.name('btnG')).click();

    browser.driver.sleep(1000);
    expect(browser.getTitle()).toMatch(/(Selenium)/);
  });
});
```

Quelltext 4.3: Code-Beispiel: Protractor

INTEGRATION IN CI-SYSTEME

Da auch hier Selenium zugrunde liegt, ist Protractor einfach in eine Jenkins- oder andere CI-Umgebung integrierbar. Im Abschnitt 7.2 ist die Einrichtung und Konfiguration im Detail beschrieben. Wie auch WebdriverIO kann Protractor in Services wie BrowserStack [10] integriert werden, um eine breite Palette an Plattformen und Browsern testen zu können.

AUFBEREITUNG DER ERGEBNISSE

Je nach genutztem Test-Framework gibt es verschiedene Plugins zur Erstellung von XML- oder HTML-Reports. Oft genutzt wird dabei *jasmine-html-reporter* in Kombination mit *JUnit*. Weiterhin ist es recht einfach möglich, im Fehlerfall einen Screenshot der aktuellen Seite zu erstellen und in den Test-Report einzubetten. Wird ein CI-System genutzt, ist über dieses der automatisierte Versand der Reports möglich.

EINRICHTUNG UND BEDIENBARKEIT

Protractor sowie alle erforderlichen Erweiterungen lassen sich einfach über einen Paketmanager wie *yarn* oder *npm* installieren. Über die Website erhält man Zugriff auf eine umfangreiche API-Dokumentation, verschiedene Tutorials und eine Auflistung der häufigsten Fragen. Auch erklärt sind die notwendigen Schritte für die Konfiguration eines alternativen Test-Frameworks, sollte nicht Jasmine genutzt werden.
Da Protractor *TypeScript* unterstützt, ist der Einstieg in das Schreiben von Tests recht einfach, auch die Syntax ist oft selbsterklärend. Erfahrungen mit anderen Test-Frameworks wie *Rspec* sind von Vorteil, da es viele Ähnlichkeiten im Aufbau und der Beschreibung von Tests gibt.

[10] https://www.browserstack.com/automate/protractor

4.5. TEST-FRAMEWORKS: JASMINE UND MOCHA

Jasmine [19] (in Version 2.6.4) und *Mocha* [20] (in Version 3.4.2) sind beides BDD-Frameworks für *Node.js*, die in Kombination mit verschiedenen Tools für Integrationstests eingesetzt werden. Neben diesen beiden Frameworks gibt es noch einige andere wie *Cucumber* oder *Serenity/JS*, im Rahmen dieser Arbeit werden diese aber nicht näher betrachtet.

Jasmine ist ein umfassendes Framework mit einfach zu erlernender Syntax. Bereits enthalten sind eine Assertion- beziehungsweise Expectation-Library zur Abbildung von Erwartungswerten und eine Möglichkeit zur Erstellung von Mocks und Stubs[11]. Durch verschiedene *waitFor*()*-Methoden ist auch das Testen von asynchronen Funktionen ohne Probleme möglich. Nicht möglich ohne die Nutzung von weiteren Plugins ist die Erstellung einer Fake-API zur Beantwortung von AJAX-Anfragen, ohne dabei ein real existierendes Backend zu benötigen. [21, Abschnitt 4]
Wenn man Tests erstellen möchte, ohne sich vorher noch viele Gedanken über die zu verwendende Assertion- oder Stub-Erweiterung machen zu müssen, ist Jasmine die richtige Wahl für einen schnellen Einstieg.

Mocha ist ein einfaches Framework, welches dem Nutzer die Wahl überlässt, welche Erweiterungen für die Testerstellung benötigt werden. Im Grundsetup sind weder eine Assertion-Library noch die Möglichkeit zur Erstellung von Stubs enthalten. Für fast jeden Anwendungsfall gibt es mehrere Plugins, aus denen man eine Wahl treffen kann. Für Assertions und Expectations kann man unter anderem zwischen *Chai*, *expect.js* oder *better-assert* wählen. Empfohlen wird Chai, die Syntax ist dabei ähnlich der von Jasmine. Für Mocks und Stubs wird das Plugin *Sinon* angeboten, die Syntax ist hierbei vereinfacht im Vergleich zu Jasmine. Auch mit Sinon möglich ist die Erstellung einer Fake-API, um Anfragen beantworten zu können. Für Reporting und zur Abbildung der Code-Abdeckung durch Tests kann man unter anderem das Plugin *Instanbul* nutzen. [22]
Für Nutzer, die Flexibilität als ein wichtiges Kriterium betrachten und nur die notwendigen Komponenten im Framework haben wollen, ist Mocha die bessere Wahl. Von der Syntax sind sich beide Frameworks sehr ähnlich, daher ist auch ein Umstieg nicht sehr kompliziert.

4.6. WEITERE TOOLS

Auch im Bereich der Web-Tests gibt es noch eine Vielzahl weiterer Tools und Test-Frameworks zur Umsetzung und Integrationsmöglichkeiten zur Automatisierung. Folgend ist eine kleine Auswahl an Beispielen gelistet.

TOOLS, FRAMEWORKS

- Cucumber (Cucumber Limited)
- Selenide
- TestNG
- WD.js

CI-SYSTEME

- Bamboo (Atlassian)
- BrowserStack
- TeamCity (JetBrains)
- Travis CI (Travis CI GmbH)

[11] Mock, Stub: Fake-Objekte mit vordefinierten Rückgabewerten für die Testausführung, um nicht auf realen Objekten oder Systemen arbeiten zu müssen.

4.7. ZUSAMMENFASSUNG

Das für Web-Tests passende Tool zu finden ist immer eine von den Anforderungen abhängige Entscheidung. Die näher vorgestellten Tools *Maven*, *WebdriverIO* und *Protractor* werden tabellarisch anhand der definierten Kriterien miteinander verglichen und danach eine Auswahl für die Implementierung und Durchführung der geplanten Test getroffen.

Kriterium	Maven	WebdriverIO	Protractor
Verarbeitung der Inhalte	Reines Build-Tool Verarbeitung nur direkt über Selenium möglich Einarbeitung in zwei Tools notwendig Probleme mit asynchronen Inhalten	Statische und dynamische Inhalte möglich Asynchrone Inhalte verarbeitbar Diverse Frameworks nutzbar Nutzung von CSS-Selektoren	Statische und dynamische Inhalte möglich Asynchrone Inhalte verarbeitbar Diverse Frameworks nutzbar Nutzung von CSS-Selektoren
Integration in CI-Systeme	Leicht über Plugins integrierbar	In diverse CI-Systeme integrierbar Plugins für Konfiguration verfügbar	In diverse CI-Systeme integrierbar Plugins für Konfiguration verfügbar
Aufbereitung der Ergebnisse	Keine Reports direkt über Selenium möglich Ausführliche Reports über Plugins möglich	Verschiedene Systeme zur Auswahl Pro Report-Möglichkeit Plugin vorhanden Screenshots	Je nach Framework verschiedene Report-Möglichkeiten Screenshots
Einrichtung und Bedienbarkeit	Einfache Installation Umfangreiche Dokumentationen Tests werden in Java erstellt	Einfache Installation Umfangreiche Dokumentation Tests werden in JavaScript erstellt	Einfache Installation Umfangreiche Dokumentation Tests werden in JavaScript oder TypeScript erstellt

Tabelle 4.1.: Vergleichskriterien für Web-Test-Tools

WebdriverIO und Protractor sind beide gut geeignet für das Testen von *Single-Page-Applications*. Beide setzen mit einem eigenen Overlay auf Selenium auf und beheben dadurch unter anderem Probleme mit asynchronen Inhalten. Weiterhin wird die Selenium-API um neue Funktionen erweitert. Protractor bietet dabei den Vorteil, direkt für die Nutzung mit Angular konzipiert zu sein. Daher ist es nicht erforderlich, explizite Wartebefehle in Tests einzufügen. Das Framework kümmert sich selbst im Hintergrund darum.
Auch die Integration in CI-Umgebungen wie Jenkins ist bei allen Tools kein Problem.

Für die prototypische Implementation und Durchführung der Tests wurde sich für Protractor mit voreingestelltem Test-Framework Jasmine entschieden. Ausschlaggebend ist, dass das zu testende AMCS-Frontend mit Angular entwickelt wird. Protractor wurde für genau diesen Anwendungsfall geschaffen und fügt sich daher nahtlos in das bestehende Projekt ein.

5. AMCS - AUDITORIUM MOBILE CLASSROOM SERVICE

5.1. PROJEKTÜBERSICHT

AMCS ist ein aktuelles Forschungsprojekt der Professur für Rechnernetze der Fakultät Informatik in Kooperation mit der Professur für die Psychologie des Lehrens und Lernens der Fachrichtung Psychologie an der TU Dresden. Es wurde entwickelt, um den Dozenten Möglichkeiten zur Interaktion mit den Studenten während einer Veranstaltung zu geben. AMCS unterstützt darüber hinaus auch die Vor- sowie Nachbereitung von Veranstaltungen für Studenten. In AMCS sind verschiedene Ansätze aus bereits existierenden Audience-Response-Systemen (ARS) vereint, welche durch zusätzliche Funktionalitäten erweitert werden. [23]

Das System besteht aus einer Backend-Applikation und unterschiedlichen Apps für Web, Smartphones und Präsentationssoftware. Für die zu entwickelnden Web-Tests relevant ist die Web-Applikation, andere Apps werden in dieser Arbeit nicht weiter betrachtet.

5.2. BACKEND

Das Backend bildet den Kern von AMCS und besteht aus einer *Rails*-Applikation mit Erweiterungen für *Web-Sockets* (*ActionCable* und *Redis*) und Jobs (*Sidekiq*). Für die Persistierung der Daten kommt dabei eine *PostgreSQL*-Datenbank zum Einsatz. ActionCable nutzt das Prinzip der *Message-Queues*, welche über Redis bereitgestellt werden. Für die Kommunikation nach außen nutzt Rails den *Puma*-Webserver, welcher über ein Unix-Socket mit einem *nginx*-Webserver verbunden ist und von diesem Anfragen weitergeleitet bekommt. Die Komponenten und Kommunikationsflüsse sind schematisch in Abbildung 5.1 nachgebildet. Das Backend ist als reine *REST-API* konzipiert und liefert auf Anfragen nur JSON-Antworten aus. Puma arbeitet mit Threads und ist für viele parallele Anfragen geeignet, was eine wichtige Voraussetzung für den Einsatz in einer Umgebung mit hohen Nutzerzahlen ist.

In der Entwicklung und zur Qualitätssicherung wird in verschiedenen Umgebungen gearbeitet. Anpassungen und neue Features werden in der *Development*-Umgebung entwickelt und über Funktions- sowie Integrationstests geprüft. Fertige Entwicklungen werden auf der *Staging*-Umgebung eingespielt und dort nochmals auf korrekte Funktion und App-Integration geprüft. In unregelmäßigen Abständen werden Releases der umgesetzten Features erstellt und auf der *Production*-Umgebung eingespielt. Um Fehler, die in der *Production*-Umgebung auftreten, nachvollziehen zu können, ist die Konfiguration der *Staging*-Umgebung weitestge-

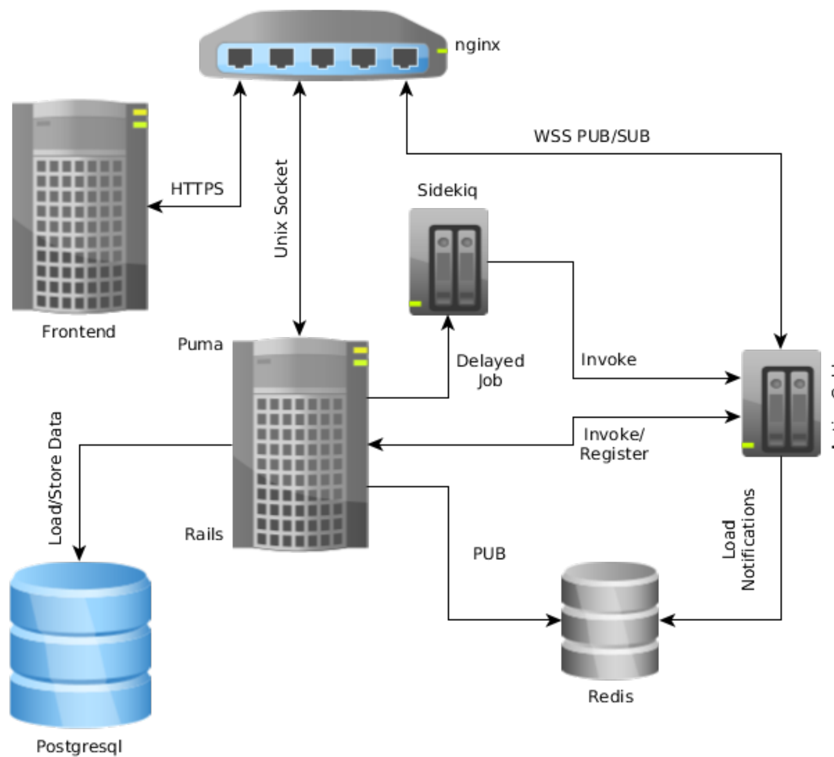


Abbildung 5.1.: AMCS Systemarchitektur

hend identisch mit *Production*, Unterschiede gibt es hauptsächlich in der genutzten Hardware. Hinzu kommt eine *Test*-Umgebung, in welcher die bereits existierenden Tests regelmäßig ausgeführt werden. Für diese CI-Tests wird bereits Jenkins genutzt.

Das Projekt befindet sich in einer Test- und Evaluierungsphase, daher sind die Nutzerzahlen aktuell nicht sehr hoch. Perspektivisch soll es für alle Lehrstühle verfügbar gemacht werden, wodurch sich eine wesentlich höhere Anzahl an parallelen Veranstaltungen mit zum Teil mehreren 100 Teilnehmern pro Veranstaltung ergeben können. Das System muss in der Lage sein, diese kurzfristig auftretenden hohen Lasten verarbeiten zu können. Hinzu kommt eine ständige Web-Socket-Verbindung pro Nutzer, über die Echtzeit-Informationen verschickt werden können.

5.2.1. KRITERIEN FÜR PERFORMANCE-TESTS

Mit Performance-Tests soll die Stabilität des Backend geprüft werden, Sicherheitsaspekte werden im Rahmen dieser Arbeit nicht behandelt. Aktuell läuft die Anwendung auf nur einer Instanz, für die Zukunft vorgesehen ist jedoch eine Verteilung auf mehrere Instanzen für Load-Balancing und die Fähigkeit, auch mit sehr hohen Nutzerzahlen umzugehen. Eine Instanz besitzt typischerweise vier echte CPU-Kerne mit 8 GiB Arbeitsspeicher. Pro Kern ist ein Puma-Worker mit bis zu 16 Threads konfiguriert, um die HTTPS-Anfragen zu verarbeiten. Die Performance-Tests sollen zum einen einzelne API-Ressourcen prüfen, zum anderen komplexere Workflows im System nachstellen. Nachgewiesen werden soll weiterhin erwartbares Verhalten und deren Ergebnisse unterschiedlicher Anfragen an das System.

Folgende Anforderungen lassen sich nun für Performance-Tests definieren:

- Nachbilden von realistischem Nutzerverhalten
- Prüfen, welche Abfragen kostenintensiv sind oder Engpässe verursachen können
- Simulieren von hoher Last in kurzer Zeit
- Plausibilität der Ergebnisse und Vergleich mit Erwartungswerten

Um die Testergebnisse bewerten zu können, wird eine Usability-Metrik benötigt, innerhalb derer die Antwortzeiten des Systems akzeptabel sind. Grundsätzlich soll das System bei Anfragen schnellstmöglich ein Ergebnis liefern. Für die Definition einer Metrik wird sich am Modell von *Jakob Nielsen* [24] orientiert.

- 0.1 - 0.5s: Akzeptable Antwortzeit für einfache Anfragen (Einzelabfragen)
- 0.5 - 2.0s: Akzeptable Antwortzeit für umfangreichere Anfragen (Listenabfragen)
- 2.0 - 5.0s: Akzeptable Antwortzeit für kostenintensive und komplexe Anfragen (Auswertungen)

Weiterhin wird für eine allgemeine Request-Bewertung der *APDEX* [25] (Application Performance Index) herangezogen. APDEX definiert einheitliche Standardmethoden zur Bewertung der Performance von Anwendungen zu Vergleichszwecken. Dabei werden Messwerte, Testergebnisse und die erwarteten Ergebnisse in sogenannte Nutzerzufriedenheit umgewandelt. Je mehr sich der Wert 1.0 annähert, desto besser ist dabei das Ergebnis.

5.2.2. DURCHFÜHRUNG DER PERFORMANCE-TESTS

Die Performance-Tests werden als Grey-Box-Test durchgeführt. Stufenweise ist dabei eine Lasterhöhung über die Gesamtzahl der parallelen Nutzer durchzuführen. Durchgeführt werden sollen die Tests mit 100, 250, 500 und 1000 parallelen Nutzern über einen definierten Zeitraum hinweg.

Es werden einfache Tests zur Überprüfung der Plausibilität von Ergebnissen durchgeführt. Weiterhin wird versucht, die Performance des Systems unter Normallast sowie einer Lastspitze gegenüberzustellen. Als Testumgebung steht eine virtuelle Maschine von Hetzner mit 2 GiB Arbeitsspeicher und zwei CPU-Kernen zur Verfügung. Puma ist mit acht Threads pro Worker konfiguriert. Um Referenzwerte zu erhalten werden die Tests auch einmal auf dem Produktivsystem ausgeführt.

5.3. WEB-APPLIKATION

Das Frontend besteht aus einer mit *Angular* erstellten *Single-Page-Application* und läuft vollständig im Browser des Nutzers. Datenanfragen werden dabei asynchron an das Backend gestellt und die Ergebnisse je nach Kontext für eine bestimmte Dauer im Browser-Cache abgespeichert. Anfragen der Web-Applikation werden von nginx verarbeitet und entsprechend der angeforderten Route entweder direkt an die REST-Schnittstelle oder die Schnittstelle für Web-Sockets weitergeleitet. Gearbeitet wird dabei mit dem JSON-Format für Requests an und Responses von der Schnittstelle.

5.3.1. KRITERIEN FÜR WEB-TESTS

Über Web-Tests soll das Zusammenspiel zwischen Frontend und Backend über Integrations- beziehungsweise End-to-End-Tests (e2e) geprüft werden. Sicherheitsaspekte werden im Rahmen dieser Arbeit nicht behandelt. Mit den Tests sollen der korrekte Seitenaufbau und verschiedene Workflows, die regelmäßig ausgeführt werden, getestet und gegen erwartetes Verhalten geprüft werden.

Durch die Nutzung von Angular mit hoher Asynchronität und den Eigenschaften der Seite als SPA ergeben sich folgende Anforderungen an Web-Tests:

- Verarbeitung statischer und dynamischer Inhalte
- Parsen von HTML und JSON-Responses
- Umgang mit Angular und SPA im Allgemeinen
  - Warten auf vollständigen Seitenaufbau und asynchrone Inhalte
  - Behandeln und Verarbeiten Angular-spezifischer Elemente und Attribute
- Abbilden komplexer Workflows (BDD)

5.3.2. DURCHFÜHRUNG DER WEB-TESTS

Im Rahmen dieser Arbeit werden keine Unit-, sondern nur e2e-Integrationstests erstellt.
Die Web-Tests werden zuerst auf der Staging-Umgebung durchgeführt und das System auf korrekte Funktion überprüft. Es werden einfache Tests wie der Login in das System mit unterschiedlichen Nutzern und die folgende Prüfung der Rechte durchgeführt, darauf aufbauend komplexere Workflows wie das Anlegen eines Kurses mit Veranstaltungen und den unterschiedlichen Fragetypen. Nach jedem Absenden der Daten an die Schnittstelle werden die Antworten gegen Erwartungswerte sowie der Seitenaufbau geprüft.
Ziel ist es, jede Hauptfunktionalität in einem eigenen Test abzubilden und automatisiert testen zu lassen. Die automatisierte Ausführung soll dabei über Jenkins-CI erfolgen. Die vollständige Test-Suite wird auch für eine regelmäßige Ausführung auf dem Produktivsystem konfiguriert, um dort die Integrität und korrekte Funktion zu gewährleisten. Folgende Hauptfunktionen sind vorhanden:

- Login mit unterschiedlichen Nutzern
- Anlegen, Editieren und Löschen eines Kurses
- Anlegen, Editieren und Löschen einer Veranstaltung
- Anlegen, Editieren und Löschen unterschiedlicher Fragetypen in jedem Kontext (Kurs, Veranstaltung, Folie)
  - FreetextQuestion
  - GroupedQuestion
  - ScaleQuestion
  - SingleChoiceQuestion
  - MultipleChoiceQuestion
  - StudySingleChoiceQuestion
  - StudyMultipleChoiceQuestion
- Anlegen und Löschen eines Templates
- Anlegen und Löschen unterschiedlicher Nachrichtentypen
- Anlegen und Löschen eines Nutzers
- Klonen eines Kurses
- Export eines Kurses
- Beantworten einer Frage

Im Rahmen dieser Arbeit werden prototypisch die folgenden Hauptfunktionen abgebildet:

- Login mit unterschiedlichen Nutzern
- Anlegen, Editieren und Löschen eines Kurses
- Anlegen, Editieren und Löschen einer Veranstaltung
- Anlegen, Editieren und Löschen unterschiedlicher Fragetypen im Veranstaltungskontext
- Beantworten einer Frage

6. TEST-SZENARIEN

6.1. PERFORMANCE-TESTS

Anhand der in Kapitel 5.2.1 definierten Anforderungen werden nun verschiedenen Use-Cases zum Testen des Systems erstellt. Eine Umsetzung der Use-Cases mit den generierten Ergebnissen dieser Arbeit befindet sich in einem dem AMCS-Projekt übertragenem Repository.

Zur Vorbereitung der Tests wird ein vollständiger Beispiel-Kurs mit Veranstaltungen und Fragen sowie Testnutzern angelegt. Im Abschnitt 7.1 ist das Vorgehen näher beschrieben.

6.1.1. USE-CASES

LOGIN MIT NEUEM ACCOUNT

Zu prüfen ist der Anmeldevorgang und das erwartete Ergebnis mit einem neuen Account. Im zweiten Durchlauf werden die gleichen Anmeldedaten verwendet und mit dem Ergebnis der ersten Anmeldung verglichen. Das System erstellt im ersten Durchlauf einen neuen Account und antwortet mit einem JWT[12] zur Authentifizierung und dem HTTP-Status 200. Der zweite Durchlauf antwortet mit einem aktualisierten Token zur Authentifizierung und dem HTTP-Status 200. Folgende Daten werden verwendet:

- Username: perflogin
- Password: perflogin
- Wartezeit: 5s

LOGIN MIT EXISTIERENDEM ACCOUNT UND LADEN VON VERANSTALTUNGEN

Zu prüfen ist der Anmeldevorgang mit Einschreibung in einen Kurs und nachfolgender Abfrage aller Veranstaltungen aus diesem. Der Test wird mehrfach mit steigender Anzahl an Nutzern über einen definierten Zeitraum hinweg ausgeführt. Folgende Daten werden verwendet:

- Username: perftest0001-perftest1000
- Password: perftest
- Course-PIN: PERFTEST
- Zeitraum: 30s

[12] JWT - JSON Web Token (`https://tools.ietf.org/html/rfc7519`)

WORKFLOW: LOGIN UND LADEN ALLER FRAGEN EINER VERANSTALTUNG

Zu prüfen ist ein generischer Workflow, bei dem Nutzer nach erfolgtem Login eine aktive Vorlesung aufrufen und die Fragen dieser laden. Der Test wird mehrfach mit einer steigenden Anzahl an Nutzern über einen definierten Zeitraum hinweg ausgeführt. Folgende Daten werden verwendet:

- Username: perftest0001-perftest1000
- Password: perftest
- Course-PIN: PERFTEST
- Lecture-Name: PERFTEST-DATA 1
- Zeitraum: 60s

6.2. FRONTEND-TESTS

Anhand der in Kapitel 5.3.1 definierten Anforderungen werden nun verschiedenen Use-Cases zum Testen der Anwendung erstellt. Eine prototypische Umsetzung der Use-Cases befindet sich in einem dem AMCS-Projekt übertragenem Repository.

Zur Vorbereitung der Tests wird eine virtuelle Displayumgebung für die Ausführung von Headless-Browsern konfiguriert. Im Abschnitt 7.2 ist das Vorgehen näher beschrieben.

6.2.1. USE-CASES

LOGIN UND LOGOUT MIT EINEM TESTNUTZER

Zu prüfen ist der Aufruf der Startseite gefolgt vom Öffnen des Login-Formulars und der Eingabe von Testdaten. Als Negativ-Test wird im ersten Versuch ein inkorrektes Passwort angeben und geprüft, ob eine Fehlermeldung erscheint. Im zweiten Versuch wird das richtige Passwort übergeben. Nach erfolgreichem Login wird geprüft, ob die dem Nutzer zugewiesene Rolle korrekt im Menü hinterlegt und der Logout-Link vorhanden ist. Nun wird ein neuer Tab geöffnet und die Startseite erneut aufgerufen. Es wird geprüft, ob das JWT korrekt im Browser-Storage hinterlegt ist und von den einzelnen Tabs darauf zugegriffen werden kann, also kein erneuter Login erforderlich ist. Wie im vorherigen Testschritt werden die zugewiesene Rolle und der Logout-Link geprüft. Im letzten Schritt wird der Logout-Link geklickt und zur Startseite geleitet. Die genutzten Testdaten lauten wie folgt:

- Username: e2estudent
- Password: e2estudent
- Role: Student

LIFECYCLE EINES KURSES

Mit diesem Test ist der vollständige Lifecycle eines Kurses abzubilden und die Einzelschritte zu testen. Ausgehend von einem Login als Administrator wird ein neuer Kurs angelegt. Es werden die Formularfelder auf Vollständigkeit geprüft und mit Testdaten befüllt. Nach der Erstellung wird der neue Kurs in der Übersicht gesucht, zur Kursseite gewechselt und geprüft, ob ein Button für das Editieren vorhanden ist. Bei Erfolg wird das Editierformular geöffnet und die Beschreibung angepasst. Nach Absenden der Änderung wird geprüft, ob die neue Beschreibung übernommen wurde. In der letzten Teststufe wird der Button für das Löschen des Kurses auf der Kursseite gesucht und der Löschvorgang ausgeführt. Nach erfolgreicher Bestätigung wird auf die Startseite geleitet und dort nochmals die Kursübersicht geprüft.

Ist der Kurs nicht mehr vorhanden, war die Durchführung erfolgreich. Für die Durchführung werden folgende Testdaten genutzt:

- Username: e2eadmin
- Password: e2eadmin
- Role: Admin
- Course-Name: FRONTEND_CI_E2E
- Course-Description: E2E
- Course-Description2: E2E-EDIT
- Course-Pin: E2EE2E
- Course-Owner: e2electurer

LIFECYCLE EINER VERANSTALTUNG

Analog zum Lifecycle eines Kurses wird hier eine Veranstaltung abgebildet und die Einzelschritte getestet. Ausgehend von einem Login als Lecturer und dem Startpunkt Kursseite wird eine neue Veranstaltung angelegt. Zu prüfen ist im ersten Schritt das Vorhandensein eines Buttons zum Anlegen einer Veranstaltung. Bei Erfolg wird dieser geklickt, die Formularfelder auf Vollständigkeit geprüft und mit Testdaten befüllt. Im ersten Versuch werden fehlerhafte Daten eingegeben und Pflichtfelder ausgelassen, um die Fehlerbehandlung zu prüfen. Im zweiten Versuch werden die Daten korrekt und vollständig übergeben. Nach der Erstellung wird die neue Veranstaltung direkt aufgerufen und die übergebenen Daten dabei auf Vorhandensein geprüft. Nun wird der Zustand der Veranstaltung auf *before* gesetzt und die entsprechende Status-Meldung überprüft. Im folgenden Schritt wird auf das Vorhandensein eines Editierbutton geprüft und dieser angeklickt. Im Editierformular wird der Veranstaltungstitel angepasst und abgeschickt. Anschließend wird geprüft, ob der neue Titel übernommen wurde.

In der letzten Teststufe wird der Button für das Löschen der Veranstaltung auf der Seite gesucht und der Löschvorgang ausgeführt. Nach erfolgreicher Bestätigung wird auf die Kursseite geleitet und dort nochmals die Übersicht der Veranstaltungen geprüft. Ist die Veranstaltung nicht mehr vorhanden, war die Durchführung erfolgreich. Die folgenden Testdaten werden verwendet:

- Username: e2electurer
- Password: e2electurer
- Role: Lecturer
- Lecture-Name: FRONTEND_CI_VLE2E
- Lecture-Description: E2E
- Lecture-Description2: E2E-EDIT
- Lecture-Start: 2050-01-01
- Lecture-Duration: 90
- Slides: 1

VERWALTEN VON FRAGEN IM VERANSTALTUNGSKONTEXT

Geprüft werden soll das Anlegen, Editieren und Löschen von verschiedenen Fragetypen im Veranstaltungskontext. Dabei wird sich auf den Zeitraum *during* (nur während einer Veranstaltung beantwortbar) der Fragen beschränkt. Ausgehend von einem Login als Lecturer und dem Startpunkt Veranstaltungsseite wird prototypisch eine Frage mit zwei Antwortmöglichkeiten angelegt (SingleChoiceQuestion).

Es wird auf die Unterseite zur Fragenübersicht gewechselt und dort nach dem Button für die Erstellung einer Frage gesucht und dieser geklickt. Bei Erfolg wird das Erstellungsformular geöffnet und die erforderlichen Formularfelder nach Wahl des Fragetyps und Kontext auf Vollständigkeit geprüft. Im ersten Versuch werden Daten eingebenen, jedoch keine Antwortoptionen definiert und nach dem Absenden geprüft, ob Fehlermeldungen vorhanden sind. Folgend werden die Daten vollständig eingeben und das Formular abgeschickt. Nach erfolgreicher Erstellung wird die Frage in der Übersicht gesucht und die Daten überprüft, innerhalb des Containers auf der Seite wird weiterhin der Button zum Editieren der Frage gesucht. Ist

der Editierbutton vorhanden, wird ein Klick-Event ausgelöst und das Editierformular geöffnet. Hier wird der Name angepasst und das Formular abgeschickt. Anschließend wird die Frage nochmals in der Übersicht gesucht und geprüft, ob die Anpassung erfolgreich war.

In der letzten Teststufe wird der Button für das Löschen der Frage im Container gesucht und der Löschvorgang ausgeführt. Nach erfolgreicher Bestätigung wird geprüft, ob die Frage noch in der Übersicht vorhanden ist. Ist die Frage nicht mehr vorhanden, war die Durchführung erfolgreich. Die Daten aus den vorherigen Tests werden wiederverwendet und um folgende ergänzt:

- Question-Name: SCQ
- Question-Name2: SCQ-EDIT
- Question-Formulation: Select A or B.
- Question-Position: 10
- Choice-Formulation1: Choice-A
- Choice-Formulation2: Choice-B
- Choice-Position1: 1
- Choice-Position2: 2

BEANTWORTEN EINER FRAGE

Geprüft werden soll das Beantworten einer Frage und das Verhalten im Frontend. Je nach Fragetyp und gewählter Antwortmöglichkeit sind auch zwei Versuche möglich, wenn der erste nicht korrekt war. Es wird sich in diesem Test auf Veranstaltungsfragen im Zeitraum *during* beschränkt.

Ausgehend von einem Login als Student und dem Startpunkt Veranstaltungsseite wird prototypisch eine Frage beantwortet (SingleChoiceQuestion). Nach Aufruf des Startpunkts wird nach vorhandenen Fragen gesucht und geprüft, ob sie beantwortbar sind und die vorgegebenen Daten beinhalten. Nach dem Absenden einer Frage wird diese nicht mehr in der Übersicht als aktiv angezeigt. Es ist zu prüfen, dass wiederholtes Beantworten nicht möglich ist. Nach Beantwortung der Frage wechselt der Testnutzer in die Auswertungsübersicht und wählt den entsprechenden Kurs und die Vorlesung aus. Es wird geprüft, dass die beantwortete Frage in der Übersicht gelistet ist. Neben den Daten der vorhandenen Tests werden die folgenden ergänzt:

- Username: e2estudent
- Password: e2estudent
- Role: Student
- Choice-Selection: A

7. IMPLEMENTIERUNG

Basierend auf den in Kapitel 6 erstellten Szenarien sollen in diesem Abschnitt die Schritte zur Implementierung der einzelnen Tests erläutert werden. Für jeden Teilbereich wird je ein Test gewählt und das Vorgehen beschrieben.

7.1. PERFORMANCE-TESTS

Das Vorgehen bei der Erstellung der Performance-Tests wird am Beispiel vom zweiten Szenario Login mit existierendem Account und Laden von Veranstaltungen beschrieben.

VORARBEITEN

Um ein möglichst realistisches Verhalten des Tests zu erzeugen müssen die gleichen Anfragen an die API gesendet werden wie bei einem regulären Seitenaufruf. Um alle erforderlichen Anfragen zu ermitteln wurde der Test schrittweise über die Website ausgeführt und alle dabei abgeschickten Anfragen geloggt und ausgewertet. Für den Test relevant sind nur REST-Anfragen, Ressourcen wie CSS, HTML und andere sind hier vernachlässigbar.

Method	Ressource
POST	/api/auth/authenticate
POST	/api/courses/enroll
GET	/api/courses/COURSEID/lectures

Tabelle 7.1.: Implementierung: Ermittelte REST-Anfragen

Header-Parameter	Wert
Content-Type	application/json
X-AMCS-API	3
Authorization	JWT

Tabelle 7.2.: Implementierung: Ermittelte Header-Parameter

Tabelle 7.1 listet die ermittelten Anfragen an die API, die zweite Tabelle 7.2 die notwendigen Header-Parameter für die Kommunikation mit Angabe des Inhaltstypen, der genutzten

API-Version und nach dem Login auch zur Authentifizierung für folgende Anfragen.

Weiterhin müssen zur Vorbereitung der Tests die erforderlichen Testdaten und Testnutzer generiert werden. Das dafür erforderliche Script wurde in *Ruby* geschrieben und kann über die *Rails-Console* auf den entsprechenden Systemen geladen und ausgeführt werden. Nachfolgend werden die einzelnen Schritte an kurzen Code-Ausschnitten grob erläutert.

```
1 @perflec = User.create(username: 'perflec', password: 'perflec',
     permission_level: 1)
2
3 1000.times do |i|
4   User.create(username: "perftest#{(i+1).to_s.rjust(4, '0')}", password: '
       perftest')
5 end
```

Quelltext 7.1: Ruby: Anlegen der Testnutzer

Im ersten Schritt werden ein Nutzer *perflec* mit der Rolle *Lecturer* und 1000 Nutzer mit der Rolle *Student* angelegt. Der erstellte *Lecturer* wird in den nachfolgenden Schritten zur Erstellung von Kurs, Veranstaltungen und Fragen weiterverwendet.

```
6  @perf_course = Course.create(user: @perflec, name: 'PERFTEST', description:
      'PERFTEST', pin: 'PERFTEST')
7
8  3.times do |i|
9    l = Lecture.create(course: @perf_course, user: @perflec, name: "PERFTEST-
       DATA#{i+1}", description: "PERFTEST-DATA#{i+1}", state: 0, start:
       Date.today())
10   l.create_slides(10)
11 end
12 @perf_lecture = @perf_course.lectures.order(name: :asc).first
13 @perf_lecture.state = 2
14 @perf_lecture.save
15
16 10.times do |i|
17   sscq = Question.create(context: @perf_lecture, user: @perflec, title: '
       StudySingleChoiceQuestion', formulation: 'Some random text', position
       : i+1, type: 'StudySingleChoiceQuestion')
18
19   Choice.create(question: sscq, formulation: 'Choice A', position: 1,
       is_correct: false, feedback: 'A not correct')
20   Choice.create(question: sscq, formulation: 'Choice B', position: 2,
       is_correct: true, feedback: 'B correct')
21   Choice.create(question: sscq, formulation: 'Choice C', position: 3,
       is_correct: false, feedback: 'C not correct')
22   Choice.create(question: sscq, formulation: 'Choice D', position: 4,
       is_correct: false, feedback: 'D not correct')
23 end
```

Quelltext 7.2: Ruby: Anlegen von Kurs, Veranstaltungen und Fragen

Im Ausschnitt 7.2 werden nacheinander ein Kurs und drei dazugehörende Veranstaltungen angelegt. Für die erste Veranstaltung werden nachfolgend 10 Fragen des Typs *StudySingleChoiceQuestion* mit je vier Antwortmöglichkeiten erstellt, von denen nur eine korrekt ist. Der Status dieser Veranstaltung wird davor noch auf *during*, also aktiv, gestellt, damit die Testnutzer auf die Fragen zugreifen können.

```ruby
24 @perf_question = @perf_lecture.questions.order(position: :asc).first
25 User.where('username LIKE ?', 'perftest%').in_batches(of: 250).each.
     with_index do |group, index|
26   choice = @perf_question.choices.where(position: index+1).first
27   group.each do |user|
28     Answer.create(user: user, answerable: choice)
29   end
30 end
```

Quelltext 7.3: Ruby: Generieren von Beispielantworten

Im letzten Schritt wird die erste erstellte Frage der Test-Veranstaltung geladen. Für diese werden die zuvor erstellten Testnutzer in Blöcken von 250 Datensätzen geladen und pro Antwortmöglichkeit 250 Antworten generiert. Jeder geladene Block entspricht dabei einer Antwortmöglichkeit.
Das Script kann in einem Aufruf komplett ausgeführt werden, die Daten sind danach dauerhaft im System für Tests vorhanden.

TEST-IMPLEMENTIERUNG: LOGIN

In JMeter werden zusammengehörende Tests und Abfragen in einer *Thread Group* zusammengefasst. In dieser befinden sich neben Request-Objekten verschiedene Konfigurationen und Voreinstellungen, die entweder global und nur für einen bestimmten Request angewandt werden. Abbildung 7.1 zeigt die vollständige Struktur des Szenarios.

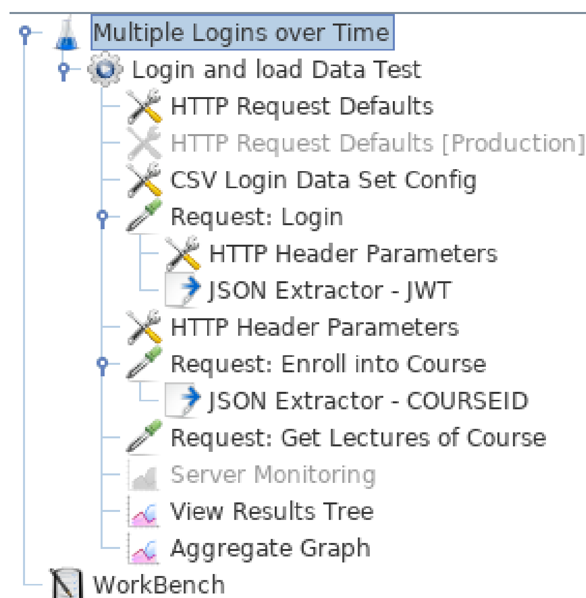


Abbildung 7.1.: Performance-Test: Szenario-Übersicht

Die erstellte Thread Group wird mit den zu erzeugenden Testnutzern und der *Ramp-Up-Period* (Zeitraum, über den immer mehr Nutzer schrittweise das System unter Last setzen und die Thread Group abarbeiten) vorkonfiguriert, im ersten Use-Case 100 Nutzer über 30 Sekunden.
Über ein *HTTP Request Default*-Objekt (siehe Abbildung 7.2) werden zu Beginn global das verwendete Protokoll und die Basis-URL für alle Anfragen definiert. Zur Verbesserung der Antwortzeiten wird anstatt einer URL direkt die IP des Servers angegeben, damit keine extra DNS-Auflösung erforderlich ist.

Basic | Advanced

Web Server

Protocol [http]: https   Server Name or IP: 78.47.103.210

Abbildung 7.2.: Performance-Test: HTTP Request Defaults

Um die unterschiedlichen Nutzer abbilden zu können, wird mit einem vorbereiteten CSV-Datensatz gearbeitet, der die 1000 Login-Einträge der zuvor angelegten Student-Testnutzer beinhaltet. JMeter geht diesen Datensatz bei Ausführung von oben nach unten durch und nutzt für jede ausgeführte Thread Group einen anderen Login. Abbildung 7.3 zeigt die Konfiguration des CSV-Objektes.

CSV Data Set Config

Name: CSV Login Data Set Config

Comments:

Configure the CSV Data Source

Filename:	logins.csv
File encoding:	
Variable Names (comma-delimited):	USERNAME,PASSWORD
Ignore first line (only used if Variable Names is not empty):	True
Delimiter (use '\t' for tab):	;
Allow quoted data?:	False
Recycle on EOF ?:	True
Stop thread on EOF ?:	False
Sharing mode:	All threads

Abbildung 7.3.: Performance-Test: CSV Data Set Config

Für die Login-Anfrage werden drei Objekte benötigt. Die Anfrage selbst wird über ein *HTTP Request*-Objekt abgebildet, weitere Parameter werden über das *HTTP Header Manager*-Objekt konfiguriert. In Abbildung 7.5 sind die aufgerufene Route und die übermittelten Daten im HTTP-Body abgebildet. Erkennbar ist die Nutzung von den im CSV-Objekt vordefinierten Variablen *${USERNAME}* und *${PASSWORD}* für die Login-Daten. Die zur Kommunikation mit der API erforderlichen Header-Parameter sind in Abbildung 7.4 gezeigt.

Headers Stored in the Header Manager

Name:	Value
Content-Type	application/json
X-AMCS-API	3

Abbildung 7.4.: Performance-Test: HTTP Header Manager für Login-Anfrage

Basic | Advanced

Web Server

Protocol [http]: Server Name or IP:

HTTP Request

Method: POST Path: /api/auth/authenticate

Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive Use multipart/form-data for POST

Parameters | Body Data | Files Upload

```
1 {
2   "auth": {
3     "username": "${USERNAME}",
4     "password": "${PASSWORD}"
5   }
6 }
```

Abbildung 7.5.: Performance-Test: HTTP Request für Login-Anfrage

Die Auswertung der Server-Antwort wird über einen *JSON Extractor* umgesetzt. Mit diesem Objekt lässt sich ein JSON-String parsen und einzelne Werte über die Anfragesyntax auslesen. Abbildung 7.6 zeigt das Auslesen des Token für die weitere Request-Authentifizierung in eine Variable namens *${JWT}*.

JSON Extractor

Name: JSON Extractor - JWT

Comments:

Apply to:

○ Main sample and sub-samples ◉ Main sample only ○ Sub-samples only ○ JMeter Variable

Variable names	JWT
JSON Path expressions	$.token
Match No. (0 for Random)	
Compute concatenation var (suffix _ALL)	☐
Default Values	NOT_FOUND

Abbildung 7.6.: Performance-Test: JSON Extractor für JWT

TEST-IMPLEMENTIERUNG: KURSEINSCHREIBUNG

Um sich in einen Kurs einzuschreiben muss eine POST-Anfrage mit der entsprechenden PIN für den gewünschten Kurs an die API geschickt werden. Abgebildet wird dies im Szenario wieder durch ein HTTP Request-Objekt (siehe Abbildung 7.8).

Headers Stored in the Header Manager

Name:	Value
Content-Type	application/json
X-AMCS-API	3
Authorization	${JWT}

Abbildung 7.7.: Performance-Test: HTTP Header Manager für folgende Anfrage

Zuvor werden die Parameter in einem global definierten HTTP Parameter Manager um das JWT erweitert (siehe Abbildung 7.7).

Basic | Advanced
Web Server
Protocol [http]: Server Name or IP:
HTTP Request
Method: POST Path: /api/courses/enroll
Redirect Automatically | Follow Redirects | Use KeepAlive | Use multipart/form-data for POST
Parameters | Body Data | Files Upload

```
1 {
2   "pin": "PERFTEST"
3 }
```

Abbildung 7.8.: Performance-Test: HTTP Request für Kurseinschreibung

Die JSON-Antwort wird auch hier wieder durch einen JSON Extractor weiterverarbeitet. Um die Veranstaltungen zu laden muss die ID des Kurses aus dem Datensatz extrahiert werden. Im Test wird dafür die Variable *${COURSEID}* genutzt. Abbildung 7.9 zeigt das dazugehörende Objekt.

JSON Extractor
Name: JSON Extractor - COURSEID
Comments:
Apply to:
Main sample and sub-samples | Main sample only | Sub-samples only | JMeter Variable

Field	Value
Variable names	COURSEID
JSON Path expressions	$.course.id
Match No. (0 for Random)	
Compute concatenation var (suffix _ALL)	
Default Values	NOT_FOUND

Abbildung 7.9.: Performance-Test: JSON Extractor für Kurs-ID

TEST-IMPLEMENTIERUNG: ABFRAGEN VON VERANSTALTUNG

Die dritte Anfrage ruft alle zum eingeschriebenen Kurs gehörenden Veranstaltungen ab. Je nach Umfang des Kurses kann die JSON-Antwort mehr oder weniger umfangreich sein. Im hier beschriebenen Szenario würde das Ergebnis ohne weitere Verarbeitung auf der Website angezeigt werden. Abbildung 7.10 zeigt das HTTP Request-Objekt mit dem Parameter ${COURSEID} aus dem vorherigen JSON Extractor.

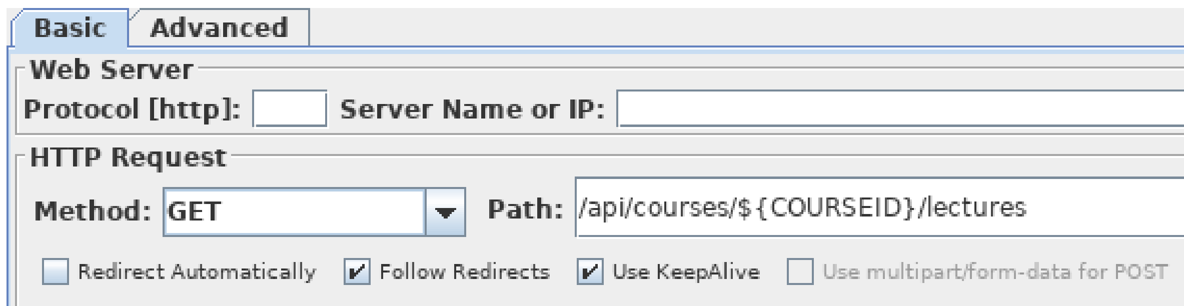


Abbildung 7.10.: Performance-Test: HTTP Request für Abfragen aller Veranstaltungen

FEHLERQUELLEN UND FEHLERVERHALTEN

Ausgeführt wurden die Tests auf dem Staging-System mit zwei CPU-Kernen und zwei GiB Arbeitsspeicher. Aufgrund dieser hardware-seitigen Limitierung ist die tatsächliche Leistungsfähigkeit der API nur bis zu einem bestimmten Punkt hin testbar. Konnten Anfragen aufgrund eines Timeouts nicht verarbeitet werden, antwortet der Server mit einem Fehlercode.

Da die einzelnen Anfragen im Szenario aufeinander aufbauen, hat ein Fehlschlag von beispielsweise dem Einschreiben in den Kurs direkte Auswirkungen auf die Abfrage der Veranstaltungen. Gleiches gilt, wenn schon der Login aufgrund einer Überlast nicht verarbeitet werden kann. Durch diese Iteration und dem Weiterreichen des ersten Fehlers schlagen alle drei Abfragen fehl und damit das gesamte Szenario.

7.2. FRONTEND-TESTS

Das Vorgehen bei der Erstellung der Web-Tests wird am Beispiel vom Szenario **Lifecycle eines Kurses** beschrieben. Der Test ist dabei ähnlich einem Wasserfallmodell aufgebaut, spätere Teststufen erfordern den Erfolg aller vorherigen. Weiterhin zu erwähnen ist, dass die Tests als End-2-End implementiert und die einzelnen Testschritte dabei nach dem Ansatz *behavior-driven* umgesetzt werden. Dies bedeutet, dass je nach Verhalten und den Elementen auf der Seite bestimmte Schritte ausgeführt werden können oder fehlschlagen. Ein anderer Ansatz ist *state-drive*, man arbeitet hierbei mit unterschiedlichen Zuständen anstatt dem Verhalten.

VORARBEITEN

Die Umsetzung der einzelnen Tests erfolgt mit Protractor. In den folgenden Schritten wird die Einrichtung der Systemumgebung beschrieben, um Frontend-Tests mit einem Headless-Browser durchzuführen.

Alle Befehle sind als *root* oder mit *sudo*-Rechten auszuführen.
Im ersten Schritt werden *node.js* in Version 7 und das Package *Protractor* installiert sowie Selenium vorbereitet. Es ist darauf zu achten, das Global-Flag „**-g**“ zu verwenden, um die Installation allen Systemnutzern zugänglich zu machen.

```
1 $ curl -sL https://deb.nodesource.com/setup_7.x | bash -
2 $ apt-get install -y nodejs
3 $ npm install protractor -g
4 $ webdriver-manager update
```

Quelltext 7.4: Installation von node.js und Protractor

Xvfb [26] steht für *X virtual framebuffer* und ist ein im Speicher laufender Display-Server für unixoide Betriebssysteme wie Linux. Mit diesem ist es möglich, Anwendungen wie Browser ohne die Notwendigkeit eines physische vorhandenen Displays ausführen zu können. Selenium in Verbindung mit Jenkins greift für die Ausführung von Oberflächentests darauf zurück.

```
1 $ apt-get install libxpm4 libxrender1 libgtk2.0-0 libnss3 libgconf-2-4
2 $ apt-get install xvfb gtk2-engines-pixbuf
3 $ apt-get install xfonts-cyrillic xfonts-100dpi xfonts-75dpi xfonts-base
    xfonts-scalable
4 $ apt-get install imagemagick x11-apps
```

Quelltext 7.5: Installation von Xvfb Darstellungsabhängigkeiten

Als Headless Browser für Tests wird *Google Chrome* verwendet. Möglich wäre beispielsweise auch die Installation von *Mozilla Firefox*.
Folgend ist die Installation und Einrichtung von Chrome beschrieben. Bei der Erstellung des symbolischen Links muss gegebenenfalls der Pfad des Protractor-Moduls und der entsprechenden Chromedriver-Version angepasst werden. Je nach Systemkonfiguration ist die Erstellung des Links auch überspringbar.

```
1 $ wget https://dl.google.com/linux/direct/google-chrome-
    stable_current_amd64.deb
2 $ dpkg -i google-chrome-stable_current_amd64.deb
```

```
3 $ ln /usr/lib/node_modules/protractor/node_modules/webdriver-manager/
    selenium/chromedriver_2.27 /usr/bin/chromedriver
```

Quelltext 7.6: Installation von Chrome

Die folgenden Befehle starten ein virtualisiertes Display auf Port 99 mit einer Auflösung von 1280x1024 Pixel und setzen dieses mittels einer globalen Variable als Standard. Der letzte Befehl startet den über Protractor mitgelieferten *webdriver-manager* im Hintergrund und lenkt die Ausgabe um. Auszuführende Tests laufen nun in dieser virtuellen Umgebung.

```
1 $ Xvfb -ac :99 -screen 0 1280x1024x16 &
2 $ export DISPLAY=:99
3 $ webdriver-manager start /dev/null &
```

Quelltext 7.7: Starten der Headless-Umgebung

TEST-IMPLEMENTIERUNG: LOGIN- UND LOGOUT-ROUTINEN

Login-Routinen sind oft und mit unterschiedlichen Daten benötigte Methoden. Um Dopplungen im Quellcode zu vermeiden (DRY-Prinzip[13]) wurde diese Routine in einen Helper ausgelagert. Helper-Methoden können grundsätzlich die gleichen Funktionen auf einer Website ausführen wie ein normaler Test. Für die Erstellung eines Kurses werden *Admin*-Rechte benötigt und daher die entsprechende Login-Routine aufgerufen.

```
1 module.exports.loginAsAdmin = function() {
2   browser.get('/');
3 
4   element(by.css('.loginButton')).click();
5   element(by.id('username')).sendKeys(browser.params.login.e2eadmin);
6   element(by.id('password')).sendKeys(browser.params.login.e2eadminpw);
7   element(by.css('.loginSubmitButton')).click();
8 };
```

Quelltext 7.8: JavaScript: Login als Administrator

Die Routine beginnt mit dem Aufruf der Startseite gefolgt von einem Klick auf den Login-Button und der Eingabe der erforderlichen Daten. Zur Wiederverwendung und Wartbarkeit wurden Parameter wie Login-Daten global vordefiniert und nicht fest im Quellcode verankert.

Wie auch die Login-Routine wird der Logout oft genutzt und daher in einen Helper ausgelagert. Die entsprechende Methode sieht aus wie folgt:

```
1 module.exports.logout = function() {
2   browser.sleep(500);
3   element(by.css('.logoutButton')).click();
4 };
```

Quelltext 7.9: JavaScript: Logout

[13] DRY - Don't Repeat Yourself

Vor der Ausführung wird für 500 Millisekunden gewartet, erst danach wird auf den entsprechenden Link geklickt. Durch die Wartezeit wird sichergestellt, dass die Website vollständig geladen hat und das Element tatsächlich vorhanden ist.

TEST-IMPLEMENTIERUNG: FORMULAR ZUR KURSERSTELLUNG

Zu Beginn des Tests werden Hooks definiert, die vor und nach dem Test durchzuführen sind. Dies kann einmalig für alle Teststufen über *beforeAll* beziehungsweise *afterAll* definiert werden, alternativ gibt es die Möglichkeit, Hooks für jede Teststufe auszuführen (*beforeEach* beziehungsweise *afterEach*).
Vor dem Test soll zuerst ein Nutzer mit der Rolle *Admin* eingeloggt und nach Durchführung des Tests wieder ausgeloggt werden. Das Keyword *await* gibt dabei an, dass die auszuführende Anfrage asynchron laufen kann und auf die Beendigung dieser zu warten ist, bevor mit dem Test fortgefahren wird.

```
1  import {browser, element, by, By, $, $$, ExpectedConditions} from '
       protractor';
2
3  describe('ManageCourse test - Start page', () => {
4    beforeAll(async function() : Promise<any> {
5      await protractor.loginHelpers.loginAsAdmin();
6    });
7
8    afterAll(async function() : Promise<any> {
9      await protractor.loginHelpers.logout();
10   });
11
12   it('has course add button', () => {
13     expect<any>(element(by.css('.createCourse')).isPresent()).toBe(true);
14   });
```

Quelltext 7.10: TypeScript: Vorarbeiten und Prüfen von Button

Mit einer *it*-Anweisung gibt man zu prüfende Kriterien an, damit eine Teststufe erfolgreich ist oder fehlschlägt. In diesem Fall wird geprüft, ob das über die CSS-Klasse *.createCourse* definierte Element auf der aktuellen Seite vorhanden ist. War der Test erfolgreich, wird ein Klick auf das Element ausgeführt, worauf sich das Formular zum Anlegen eines Kurses öffnet. Das Formular wird nachfolgend auf Vollständigkeit überprüft.

```
15
16     describe('click on Course add button', () => {
17     beforeAll(async function() : Promise<any> {
18       await element(by.css('.createCourse')).click();
19     });
20
21     it('opens form with required fields', () => {
22       expect<any>(element(by.id('courseName')).isPresent()).toBe(true);
23       expect<any>(element(by.id('courseDescription')).isPresent()).toBe(
             true);
24       expect<any>(element(by.id('coursePin')).isPresent()).toBe(true);
25       expect<any>(element(by.id('courseOwner')).isPresent()).toBe(true);
26       expect<any>(element(by.css('.createCourseSubmitButton')).isPresent())
             .toBe(true);
27     });
```

Quelltext 7.11: TypeScript: Prüfen des Formulars auf Vollständigkeit

Da dieser Test in *TypeScript* (ein JavaScript-Dialekt) geschrieben ist, können verschiedene Vorteile genutzt werden. TypeScript arbeitet mit Promise-Objekten, um Ergebnisse von Anfragen vorab bereitzustellen. Die Angabe von *expect<any>* gibt beispielsweise an, dass jeder Promise-Typ für die Durchführung akzeptabel ist, in diesem Fall vom Typ *Boolean* durch die Prüfung, ob das gesuchte Element vorhanden ist.

TEST-IMPLEMENTIERUNG: KURSERSTELLUNG MIT FEHLERHAFTEN DATEN

Der darauf folgende Testschritt prüft das Verhalten der Anwendung bei Übergabe von fehlerhaften Daten. Dazu werden schlicht nicht alle erforderlichen Felder ausgefüllt und dadurch ein Validierungsfehler verursacht. Das erwartete Verhalten ist dabei, dass das Formular nicht abgeschickt wird und offen bleibt. Geprüft wird dies durch nochmaliges Suchen des Buttons zum Absenden. Anschließend wird für eine Sekunde gewartet, um etwaige Hintergrundprozesse beenden zu lassen.

```
28
29      describe('submit invalid course data', () => {
30        beforeAll(async function() : Promise<any> {
31          element(by.id('courseName')).sendKeys(browser.params.course.name);
32          element(by.cssContainingText('option', browser.params.login.
                e2electurer)).click();
33
34          element(by.css('.createCourseSubmitButton')).click();
35        });
36
37        it('does not create a course, form still open', () => {
38          expect<any>(element(by.css('.createCourseSubmitButton')).isPresent
                ()).toBe(true);
39          browser.sleep(1000);
40        });
41      });
```

Quelltext 7.12: TypeScript: Übergabe fehlerhafter Daten

TEST-IMPLEMENTIERUNG: KURSERSTELLUNG MIT VOLLSTÄNDIGEN DATEN

Der zweite Versuch führt das Anlegen eines Kurses mit vollständigen und korrekten Testdaten aus. Als Nutzer wird dabei ein zuvor angelegter Testnutzer definiert. Die genutzten Daten sind wieder vordefiniert, um statische Inhalte zu verwenden und den Quellcode wartbar zu halten.

```
42
43      describe('submit course data', () => {
44        let createdCourse;
45
46        beforeAll(async function() : Promise<any> {
47          element(by.id('courseName')).clear().sendKeys(browser.params.course
                .name);
48          element(by.id('courseDescription')).sendKeys(browser.params.course.
                description);
49          element(by.id('coursePin')).sendKeys(browser.params.course.pin);
50          element(by.cssContainingText('option', browser.params.login.
                e2electurer)).click();
51
52          await element(by.css('.createCourseSubmitButton')).click();
53        });
54
```

```
55        it('creates the course', async function() : Promise<any> {
56          browser.sleep(500);
57          createdCourse = element(by.cssContainingText('a', browser.params.
                course.pin));
58          expect(await createdCourse.isPresent()).toBe(true);
59        });
```

Quelltext 7.13: TypeScript: Übergabe korrekter Daten

Nach Klick auf den Button zum Absenden wird wieder auf die Ausführung gewartet, bevor das Ergebnis geprüft wird. Um zu gewährleisten, dass die Kursliste aktuell ist, wird zuerst für 500 Millisekunden gewartet und anschließend nach der PIN des neu angelegten Kurses gesucht. Es wird geprüft, dass das entsprechende Container-Element vorhanden ist.

TEST-IMPLEMENTIERUNG: AUFRUF DER KURSSEITE UND EDITIEREN VON DATEN

In diesem Testabschnitt wird auf das zuvor gefilterte Container-Element des neuen Kurses geklickt und zur Kursseite gewechselt. Wieder wird über die entsprechende *async*-Anweisung auf die vollständige Ausführung gewartet, bevor das Ergebnis geprüft wird.

```
60
61        describe('click on course page link', () => {
62          beforeAll(async function() : Promise<any> {
63            await createdCourse.click();
64          });
65
66          it('opens the course page', () => {
67            expect<any>(element(by.cssContainingText('h2', browser.params.
                  course.name)).isPresent()).toBe(true);
68            expect<any>(element(by.css('.editCourse')).isPresent()).toBe(true
                  );
69            expect<any>(element(by.css('.deleteCourse')).isPresent()).toBe(
                  true);
70          });
```

Quelltext 7.14: TypeScript: Öffnen der Kursseite

Auf der Kursseite müssen ein Element mit dem Namen des neuen Kurses sowie Button zum Editieren und Löschen des Kurses vorhanden sein. Gefiltert werden die Button anhand der CSS-Klassen *.editCourse* und *.deleteCourse*.

Nachfolgend wird ein Klick auf den Button zum Editieren des Kurses ausgeführt und auf das Öffnen des Formulars gewartet. Über eine vordefinierte Helper-Methode wird der Inhalt des Feldes der Kursbeschreibung geleert und mit einem Alternativwert befüllt. Das Formular wird anschließend abgeschickt und auf die Ausführung der Anpassung gewartet.

```
71
72          describe('click on edit course link and edit value', () => {
73            beforeAll(async function() : Promise<any> {
74              await element(by.css('.editCourse')).click();
75
76              let description = element(by.id('courseDescription'));
77              protractor.customHelpers.fieldCleaner(description);
78              description.sendKeys(browser.params.course.description2);
79
80              await element(by.css('.editCourseSubmitButton')).click();
```

```
81              });
82
83              it('opens form and updates the course', () => {
84                browser.sleep(500);
85                expect<any>(element(by.cssContainingText('p', browser.params.
                    course.description2)).isPresent()).toBe(true);
86              });
```

Quelltext 7.15: TypeScript: Editieren von Kursdaten

Nach dem Absenden werden 500 Millisekunden abgewartet, um Hintergrundprozesse zur Aktualisierung der Seite abschließen zu lassen. Anschließend wird nach der angepassten Kursbeschreibung auf der aktuellen Seite gesucht und im Erfolgsfall zum letzten Testschritt geleitet.

TEST-IMPLEMENTIERUNG: LÖSCHEN DES KURSES

Im letzten Testschritt wird ein Klick auf den Button zum Löschen des Kurses ausgeführt und gewartet, bis sich das Overlay zur Bestätigung geöffnet hat. In diesem wird auf den Bestätigungsbutton geklickt und auf die Ausführung der Anweisung gewartet.

```
87
88              describe('click on delete course link', () => {
89                beforeAll(async function() : Promise<any> {
90                  await element(by.css('.deleteCourse')).click();
91                  await element(by.cssContainingText('button', 'Confirm')).
                      click();
92                });
93
94                it('deletes the course and relocates to start page', () => {
95                  browser.sleep(500);
96                  expect(browser.getCurrentUrl()).toMatch('/');
97                  expect<any>(element(by.cssContainingText('a', browser.params.
                      course.pin)).isPresent()).toBe(false);
98                });
99              });
100           });
101         });
102       });
103     });
104   });
```

Quelltext 7.16: TypeScript: Löschen des Kurses

Das erwartete Verhalten ist, dass der Kurs gelöscht und der Nutzer zurück zur Startseite geleitet wird. Nach einer Wartezeit von 500 Millisekunden wird geprüft, ob die aktuelle Seite der Startseite entspricht und ob das Container-Element des Kurses noch vorhanden ist. Wurde es nicht gefunden, war der Test erfolgreich.
Abschließend wird der *afterAll*-Hook ausgeführt und der aktuelle Nutzer ausgeloggt.

NOTWENDIGE ANPASSUNGEN, FEHLERQUELLEN UND FEHLERVERHALTEN

An vielen Elementen auf den einzelnen Seiten fehlten eindeutige Attribute wie eine ID oder CSS-Klasse, um einfach über entsprechende Selektoren darauf zugreifen zu können. Im Rahmen der Implementierung der einzelnen Tests wurde der Quellcode des Web-Frontends an-

gepasst und um fehlende Attribute erweitert.

Tests können fehlschlagen, wenn erforderliche Elemente nicht auf der aktuellen Seite gefunden wurden. Normalerweise sollte dies durch die *sleep*-Anweisungen verhindert werden, jedoch kann es unter Umständen dazu kommen, dass Antworten der API oder asynchrone Verarbeitungen im Hintergrund länger benötigen als angenommen. Dadurch schlägt ein Test dann fehlt.
Weiterhin bauen auch hier die einzelnen Teststufen aufeinander auf und bedingen sich daher. Schlägt ein Test fehl, hat das in den meisten Fällen direkte Auswirkungen auf alle nachfolgenden Teststufen. Kann beispielsweise schon der Login am Anfang nicht durchgeführt werden, schlägt das gesamte Test-Szenario fehl.
Ursprünglich war geplant, die Tests modular aufzubauen und nicht als Wasserfall-Modell. Dies ist jedoch in dieser Form nicht mit Jasmine möglich und wurde daher verworfen.

8. AUSWERTUNG DER ERGEBNISSE

Nach der Implementierung der zuvor entwickelten Test-Szenarien für Performance- und Web-Tests werden diese ausgeführt und die ermittelten Ergebnisse ausgewertet. In diesem Kapitel werden die durchgeführten Tests kurz erläutert und die Ergebnisse interpretiert. Insbesondere erfolgt eine Gegenüberstellung der Ergebnisse bei Performance-Tests mit unterschiedlich hohen Lasten und den Auswirkungen auf das Gesamtsystem sowie ein Vergleich mit den ermittelten Werten bei Ausführung auf dem Produktivsystem.

8.1. PERFORMANCE-TESTS

8.1.1. LOGIN MIT NEUEM ACCOUNT

Durchgeführt wurde eine Loginanfrage mit neuen Daten, gefolgt von einer Pause von 5 Sekunden und einer Wiederholung der Anfrage. Das erwartete Ergebnis ist jeweils ein JWT für die Authentifizierung nachfolgender Anfragen. Durch die Pause zwischen den Anfragen ändert sich der Zeitstempel und dadurch das JWT selbst.

Abbildung 8.1.: Performance-Test 1: Ergebnisse der Anfragen

Erkennbar ist eine Änderung des dritten Teils des JWT (dem Payload), in dem neben Informationen zum Nutzerkonto auch ein aktueller Zeitstempel und ein Ablaufdatum enthalten ist.

Die Abbildung 8.2 zeigt die Antwortzeiten beider Anfragen in Millisekunden. Hierbei ist erkennbar, dass die erste Anfrage mit 434ms etwas mehr Zeit benötigt, da der Login vor der JWT-Erstellung angelegt werden muss. Die zweite Anfrage beinhaltet nur einen Datenbank-Lookup mit nachfolgender JWT-Erstellung und benötigt 327ms.

Im Ergebnis ist festzustellen, dass sowohl der Login mit bisher unbekannten Daten als auch mit bestehenden ein gleiches und erwartbares Resultat erzeugt.

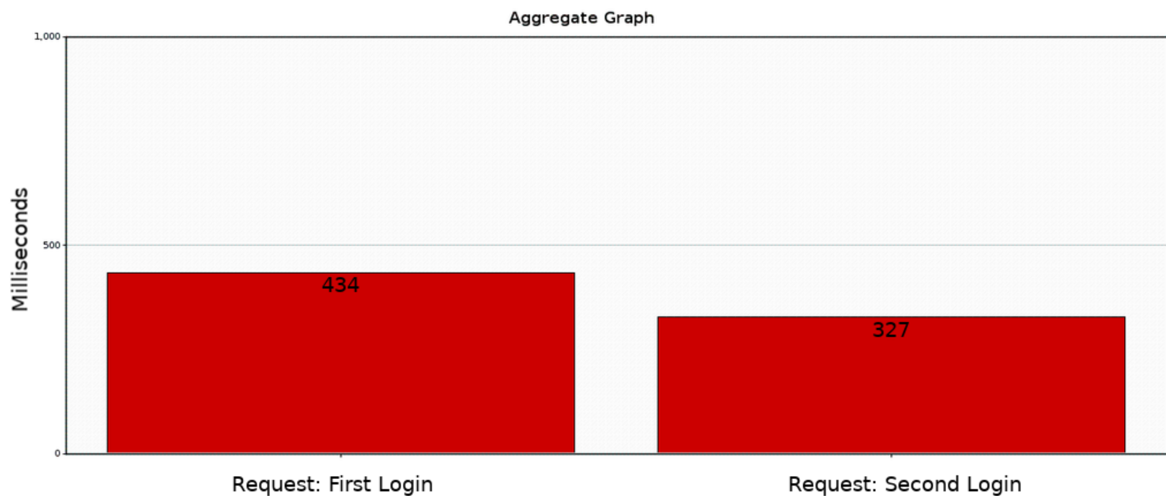


Abbildung 8.2.: Performance-Test 1: Antwortzeiten

8.1.2. LOGIN MIT EXISTIERENDEM ACCOUNT UND LADEN VON VERANSTALTUNGEN

Zur Vorbereitung für diesen Test wurden 1000 Testnutzer und ein Kurs mit Veranstaltungen angelegt. Ausgeführt werden nacheinander drei aufeinander aufbauende Requests. Zuerst wird der Login gesendet, um ein gültiges JWT für die nachfolgenden Requests zu erhalten. Der zweite Request schreibt den Nutzer über eine im JSON-Payload gesendete Pin in den zuvor erstellten Testkurs ein und gibt das entsprechende Kursobjekt zurück. Aus dem Response wird die Kurs-ID ausgelesen und im nächsten Request wiederverwendet, um alle vorhandenen Veranstaltungen aus diesem Kurs zu laden. Folgende Routen werden dabei aufgerufen:

- POST /api/auth/authenticate
- POST /api/courses/enroll
- GET /api/courses/COURSEID/lectures

Der Test wurde mehrfach mit unterschiedlichen Nutzerzahlen ausgeführt, die das System über einen Zeitraum von 30 Sekunden unter Last setzen. Die folgende Tabelle listet die ermittelten APDEX-Werte der Teststufen auf.

	APDEX			
Anzahl Nutzer	100	250	500	1000
Request: Login	0.945	0.940	0.365	0.034
Request: Enroll into Course	1.000	1.000	0.504	0.019
Request: Get Lectures of Course	1.000	1.000	0.479	0.015

Tabelle 8.1.: APDEX-Werte Performance-Test 2

Erkennbar ist eine erhöhte Antwortzeit und ein abfallender APDEX ab 500 Nutzern, es treten jedoch noch keine Fehler oder Verluste auf. Bei einer weiteren Verdopplung der Nutzer auf 1000 sinkt der APDEX auf einen Wert nahe 0 und es sind Fehler und Verluste festzustellen. Die folgenden Statistiken wurden durch JMeter erstellt. Zum Vergleich sind die Werte für 250 und 1000 Nutzer angegeben.

Requests	Executions			Response Times (ms)						
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput
Total	**750**	**0**	**0.00%**	**202.86**	**53**	**930**	**424.90**	**470.80**	**742.82**	**24.49**
Request: Enroll into Course	250	0	0.00%	90.04	53	353	146.80	201.85	323.96	8.38
Request: Get Lectures of Course	250	0	0.00%	97.49	59	473	192.60	225.45	322.00	8.39
Request: Login	250	0	0.00%	421.04	315	930	541.60	673.00	798.78	8.20

Requests	Executions			Response Times (ms)						
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput
Total	**3000**	**150**	**5.00%**	**13111.72**	**80**	**35771**	**28061.50**	**31288.70**	**34809.97**	**37.63**
Request: Enroll into Course	1000	52	5.20%	15134.76	81	35771	29368.90	32742.50	34849.72	13.33
Request: Get Lectures of Course	1000	69	6.90%	11490.61	80	35433	27275.90	31286.50	34646.98	12.83
Request: Login	1000	29	2.90%	12709.77	88	35438	27261.80	29862.45	35099.64	14.97

Abbildung 8.3.: Performance-Test 2: Auswertung der Anfragen

Die Statistik bei 1000 Nutzern zeigt, dass 29 Nutzer überhaupt nicht mehr bedient werden konnten und die Requests verworfen wurden. Durch daraus resultierende Folgefehler und weitere Timeouts wurde eine Fehlerrate von 5% über alle drei Anfragen ermittelt. Mit steigender Ausführungszeit des Tests steigt auch die Antwortzeit von zum Teil wenigen Millisekunden bis zu über einer halben Minute. Der folgende Graph verdeutlicht den Anstieg der Antwortzeiten noch einmal.

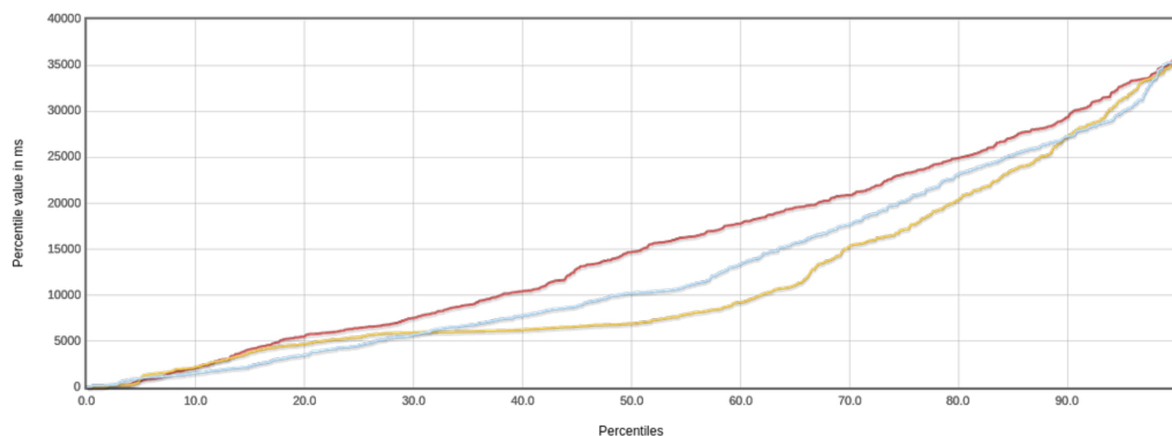


Abbildung 8.4.: Performance-Test 2: Anstieg Antwortzeit bei 1000 Nutzern

Erklärung der Graph-Farben:

- Blau: Login
- Gelb: Einschreiben in den Kurs
- Rot: Laden der Veranstaltungen.

Im Ergebnis ist festzuhalten, dass das genutzte Testsystem bis zu einer Gesamtzahl von 500 parallelen Nutzern mit je drei auszuführenden Requests noch in einem tolerierbaren Bereich arbeitet, jedoch auf Kosten der Antwortzeiten. Die Engstelle liegt hierbei bei Rails mit dem Puma-Webserver, da aufgrund der Systemkonfiguration eine Limitierung der Worker und Threads vorliegt und daher Wartezeiten von Anfragen in Timeouts laufen und schlicht

nicht beantwortet werden können. Die kostenintensivste Ressource in diesem Szenario ist der Login, da neben der Prüfung der Login-Daten auch Zeitstempel aktualisiert werden, bevor das JWT zurückgegeben wird. Die Daten der anderen beiden Abfragen werden datenbankseitig im Cache vorgehalten und müssen nicht bei jeder Anfrage neu erstellt werden. Positiv zu werten ist das Verhalten im Fehlerfall. Bricht ein Request aufgrund eines Timeouts ab, werden von diesem abhängige nicht mehr ausgeführt und schlagen daher in der Teststatistik auch fehl, anstatt eine fehlerhafte Anfrage an das System zu erzeugen.

VERGLEICHSWERTE DES PRODUKTIVSYSTEMS

Der Test wurde mit 500 und 1000 Nutzern durchgeführt. Die APDEX-Werte sinken erst bei der Verdopplung auf 1000 Nutzer um rund 60% in einen nicht akzeptablen Bereich, trotzdem können noch alle Anfragen ohne Fehler beantwortet werden.

	APDEX	
Anzahl Nutzer	500	1000
Request: Login	0.984	0.279
Request: Enroll into Course	1.000	0.318
Request: Get Lectures of Course	1.000	0.268

Tabelle 8.2.: APDEX-Werte Performance-Test 2P

8.1.3. WORKFLOW: LOGIN UND LADEN ALLER FRAGEN EINER VERANSTALTUNGEN

Zur Vorbereitung für diesen Test wurden die bestehenden Testdaten erweitert. Dazu wurde die zuvor erstellte Veranstaltung um 10 Fragen mit je vier Antwortmöglichkeiten ergänzt. Für die erste Frage wurden pro Antwortmöglichkeit je 250 Antworten blockweise aus dem Pool der bereits existierenden Testnutzer ergänzt. Dies stellt ein vereinfachtes Szenario dar, denn nur eine Frage besitzt hierbei Antworten.
Neben den vorhandenen Requests aus Abschnitt 8.1.2 kommen zwei weitere hinzu. Neu ist die Verarbeitung der Veranstaltungsliste. Es wird die erste Veranstaltung herausgefiltert, die ID ausgelesen und aufgerufen. Nachfolgend werden die Fragen der Veranstaltung angefordert. Da die Anfrage ohne weitere Parameter erfolgt, werden nur Fragen für den Zeitraum *during* angefordert. Die folgenden beiden Routen werden dafür verwendet:

- GET /api/lectures/LECTUREID
- GET /api/lectures/LECTUREID/questions

Der Test wurde mehrfach mit unterschiedlichen Nutzerzahlen ausgeführt, die über einen Zeitraum von 60 Sekunden das System unter Last setzen. Auch hier zuerst ein Blick auf die APDEX-Werte der einzelnen Teststufen.

	APDEX			
Anzahl Nutzer	100	250	500	1000
Request: Login	0.950	0.362	0.070	0.033
Request: Enroll into Course	1.000	0.350	0.079	0.038
Request: Get Lectures of Course	1.000	0.312	0.061	0.021
Request: Get Lecture	1.000	0.268	0.042	0.015
Request: Get Questions of Lecture	0.800	0.010	0.000	0.000

Tabelle 8.3.: APDEX-Werte Performance-Test 3

Schon bei 250 Nutzern über den Testzeitraum ist ein starker Anstieg der Ausführungszeiten auf dem Testsystem zu verzeichnen, die APDEX-Werte fallen um über 60%. Um alle Anfragen beantworten zu können wird die doppelte Zeit benötigt im Vergleich zu 100 Nutzern.

Requests	Executions			Response Times (ms)						
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput
Total	**1250**	**0**	**0.00%**	**4640.04**	**61**	**15628**	**10158.00**	**12047.40**	**14609.22**	**12.79**
Request: Enroll into Course	250	0	0.00%	3567.84	61	11235	9311.00	10101.35	11104.29	3.34
Request: Get Lecture	250	0	0.00%	4122.31	67	11469	9336.20	10145.05	11100.20	2.69
Request: Get Lectures of Course	250	0	0.00%	3782.43	71	10983	9083.70	9859.95	10754.70	2.96
Request: Get Questions of Lecture	250	0	0.00%	8438.50	942	15628	13925.70	14606.10	15516.10	2.59
Request: Login	250	0	0.00%	3289.12	344	11067	9301.90	9881.55	10534.91	3.62

Requests	Executions			Response Times (ms)						
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Throughput
Total	**5000**	**1360**	**27.20%**	**20667.32**	**69**	**60527**	**37101.10**	**47912.15**	**55744.49**	**19.53**
Request: Enroll into Course	1000	231	23.10%	16712.91	69	36909	34374.70	35142.35	36434.96	8.09
Request: Get Lecture	1000	341	34.10%	22583.05	80	52876	36974.00	44387.95	48861.57	4.81
Request: Get Lectures of Course	1000	293	29.30%	18606.12	80	36907	33657.60	34860.95	36278.78	6.39
Request: Get Questions of Lecture	1000	364	36.40%	31684.24	80	60527	53975.70	55738.45	59977.76	3.93
Request: Login	1000	131	13.10%	13750.28	83	35759	30411.00	32009.75	35265.62	10.88

Abbildung 8.5.: Performance-Test 3: Auswertung der Anfragen

Das Testsystem ist mit 1000 Nutzern überfordert und kann die ankommenden Anfragen nicht mehr verarbeiten und beantworten. Daraus ergibt ergibt sich eine Fehlerquote von 27.2%. In Abbildung 8.5 sind zum Vergleich wieder die Werte für 250 und 1000 Nutzer angegeben. Deutlich erkennbar ist, dass das Laden der Fragen für eine Veranstaltung kostenintensiv ist und viel Zeit benötigt. Grund dafür ist, dass für die Erstellung der Responses einige Abfragen und Berechnungen für ein Evaluationobjekt notwendig sind. Diese Daten müssen pro Nutzer neu ermittelt werden, da sich die gewählte Antwort unterscheiden kann und somit andere Felder im Response belegt werden.

Der Anstieg der Antwortzeiten über den gesamten Testzeitraum wird in Abbildung 8.4 noch einmal als Graph dargestellt.

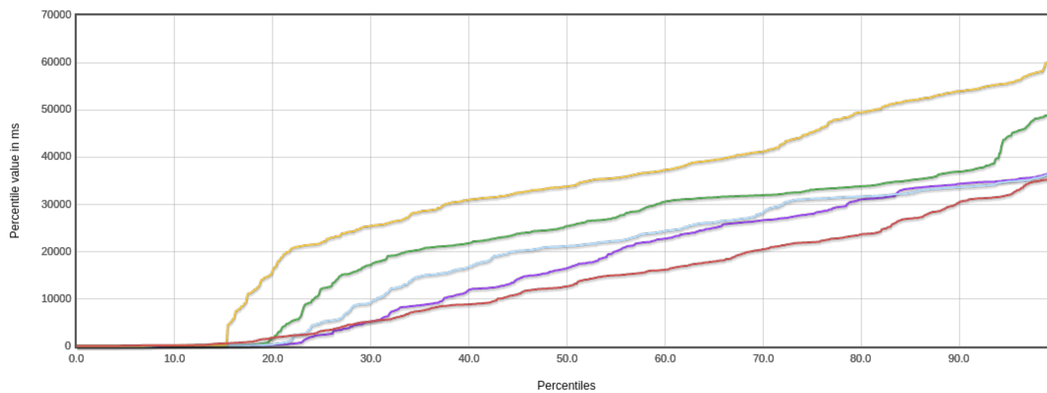


Abbildung 8.6.: Performance-Test 3: Anstieg Antwortzeit bei 1000 Nutzern

Erklärung der Graph-Farben:

- Rot: Login
- Lila: Einschreiben in den Kurs
- Hellblau: Laden der Veranstaltungen.
- Grün: Laden einzelner Veranstaltung
- Gelb: Laden der Fragen

Aus den Ergebnissen dieses Szenarios folgt, dass vor allem die Abfrage von Fragen mit existierenden Antworten hohe Anforderungen an das System stellen kann, je nach Anzahl aktuell aktiver Nutzer. Verarbeiten kann das genutzte Testsystem gerade noch 250 parallele Nutzer mit je fünf auszuführenden Requests, wobei die Antwortzeiten hier schon nicht mehr in tolerierbaren Bereichen liegen. Im Szenario ist eine Frage mit vier Antwortmöglichkeiten und insgesamt 1000 Antworten zu serialisieren, dazu kommen weitere Berechnung für Evaluation und neun einfache Fragen mit auch je vier Antwortmöglichkeiten. Aufsummiert ergeben sich alleine für diese Anfrage pro Nutzer 1050 zu serialisierende Objekte, wobei die Datenbankabfragen automatisch gecached werden. Durch die begrenzten Systemressourcen des Testsystems und der daraus folgenden Konfiguration von Puma liegt auch hier die Engstelle im Rails-Backend, eine weitere Ursache ist das allgemeine Fehlen von Caching-Strukturen für komplexere Response-Sets. Wie im vorherigen Szenario werden auch hier keine Folge-Requests ausgeführt, wenn Abhängigkeiten zuvor schon fehlgeschlagen sind.

VERGLEICHSWERTE DES PRODUKTIVSYSTEMS

Der Test wurde mit 500 und 1000 Nutzern durchgeführt. Durch die kostenintensiven Abfragen der Veranstaltungen und Fragen sinken die APDEX-Werte bei 500 Nutzern schon stark ab. Auch hier ist gut erkennbar, dass die Abfragen selbst auf einem leistungsstärkeren System immer höhere Antwortzeiten generieren und sich Anfragen somit stauen.

	APDEX	
Anzahl Nutzer	500	1000
Request: Login	0.253	0.086
Request: Enroll into Course	0.290	0.095
Request: Get Lectures of Course	0.226	0.059
Request: Get Lecture	0.186	0.055
Request: Get Questions of Lecture	0.0002	0.000

Tabelle 8.4.: APDEX-Werte Performance-Test 3P

Eine Lastspitze mit 1000 Nutzern konnte aber zwar immer noch fehlerfrei vom System verarbeitet werden, jedoch auf Kosten der Antwortzeiten. Es wird deutlich, dass das Rails-Backend in der Leistungsfähigkeit stark von den vorhandenen Systemressourcen und darauf basierenden Konfigurationen für unter anderem Puma abhängig ist.

8.2. WEB-TESTS

8.2.1. LOGIN UND LOGOUT MIT EINEM TESTNUTZER

Durchgeführt wurde das Aufrufen der Startseite, gefolgt vom Öffnen des Login-Formulars und der Eingabe von fehlerhaften und im zweiten Durchlauf korrekten Nutzerdaten. Jede Teststufe prüft dabei die auf der aktuellen Seite vorhandenen Elemente gegen erwartete und, im Falle vom Absenden des Formulars, auf korrektes Verhalten der Anwendung bei der Übergabe von fehlerhaften und korrekten Daten. Nach erfolgreichem Login wurde die Seite in einen neuen Tab geklont und geprüft, ob der Nutzer noch immer eingeloggt ist. Abschließend wurde ein Logout durchgeführt und auf korrekte Weiterleitung geprüft.

Testschritt	Elemente und Verhalten	Status
Aufruf Startseite mit Klick auf Login-Button	Login-Button (.loginButton)	OK
	Eingabefeld Username (#username)	OK
	Eingabefeld Password (#password)	OK
Absenden fehlerhafter Testdaten	Send-Button (.loginSubmitButton)	OK
	Verhalten: Formularvalidierung erzeugt Fehler, Formular nicht sendbar (.alert-danger)	OK
Absenden korrekter Testdaten	Send-Button (.loginSubmitButton)	OK
	Verhalten: Login wird akzeptiert und Nutzer weitergeleitet	OK
	Korrekte Rolle wird im Dashboard angezeigt (Student)	OK
	Logout-Button (.logoutButton)	OK
Klonen der aktuellen Seite	Nutzer weiterhin eingeloggt durch Nutzung von Browser-Storage für JWT	OK
Klick auf Logout-Button	Verhalten: Nutzer wird ausgeloggt	OK

	Weiterleitung auf Startseite mit Login-Button (.loginButton)	OK

Tabelle 8.5.: Evaluation: Test-Szenario 1 Tabellenübersicht

Das Ziel des Tests, fehlerhafte und korrekte Login-Vorgänge abzubilden und zu testen, wurde erreicht. Die erforderlichen Elemente sind auf den jeweiligen Seiten vorhanden und das erwartete Verhalten im Fehlerfall trifft zu.

Da nicht alle Elemente IDs oder eindeutige CSS-Pseudo-Klassen besitzen, wurde der bestehende Quellcode an den entsprechenden Stellen um neue Pseudo-Klassen erweitert. Dies erleichtert die Arbeit mit den Selektoren beim Suchen von Elementen innerhalb der Seite.

8.2.2. LIFECYCLE EINES KURSES

Durchgeführt wurde der Login mit der Rolle *Admin* gefolgt vom Öffnen des Formulars zum Anlegen eines Kurses und der Eingabe der notwendigen Daten, im ersten Versuch mit fehlerhaften. Nachfolgend wird der neu erstellte Kurs in der Übersicht gesucht und aufgerufen. Auf der Kursseite wurde der Editierbutton gesucht und ein Wert des Kurses angepasst. Im letzten Testschritt wurde der Button für das Löschen des Kurses gesucht und der Vorgang ausgeführt.
Jede Teststufe prüft dabei wieder die auf der Seite vorhandenen Elemente gegen erwartete und, im Falle vom Absenden des Formulars, auf korrektes Verhalten der Anwendung bei der Übergabe von fehlerhaften und korrekten Daten.

Testschritt	Elemente und Verhalten	Status
Aufruf der Startseite mit Prüfung auf Button für neuen Kurs	Add-Course-Button (.createCourse)	OK
	Eingabefeld Name (#courseName)	OK
	Eingabefeld Description (#courseDescription)	OK
	Eingabefeld Pin (#coursePin)	OK
	Select-Feld Owner (#courseOwner)	OK
Absenden fehlerhafter Testdaten	Send-Button (.createCourseSubmitButton)	OK
	Verhalten: Formularvalidierung erzeugt Fehler, Formular nicht sendbar und weiterhin offen	OK
Absenden korrekter Testdaten	Send-Button (.createCourseSubmitButton)	OK
	Verhalten: Eingaben werden akzeptiert und der Kurs angelegt	OK
	Kurs erscheint nach kurzer Verzögerung in Kursliste (Suche anhand PIN)	OK
Aufruf des neuen Kurses	Edit-Button (.editCourse)	OK
	Kurs-Titel vorhanden (h2-Tag mit Text)	OK
Editieren und Absenden von Kursdaten	Eingabefeld Description (#courseDescription)	OK
	Edit-Button (.editCourseSubmitButton)	OK

	Verhalten: Beschreibungstext wurde angepasst (p-Tag mit Text)	OK
Löschen des Kurses	Delete-Button (.deleteCourse)	OK
	Verhalten: Abfrage-Overlay zum Bestätigen vorhanden	OK
	Kurs wird gelöscht und auf Startseite geleitet	OK
	Kurs in Kursliste nicht mehr vorhanden	OK

Tabelle 8.6.: Evaluation: Test-Szenario 2 Tabellenübersicht

Das Ziel dieses Szenarios, den vollständigen Kreislauf von der Erstellung über das Aufrufen und Editieren bis hin zum Löschen eines Kurses abzubilden, wurde erreicht. Die erforderlichen Elemente sind auf jeden jeweiligen Seiten vorhanden und das erwartete Verhalten im Fehlerfall trifft zu.

8.2.3. LIFECYCLE EINER VERANSTALTUNG

Durchgeführt wurde der Login mit der Rolle *Lecturer* mit direkter Weiterleitung zur Kursseite gefolgt von der Prüfung auf das Vorhandensein des Buttons für das Hinzufügen einer Veranstaltung. Das Formular zum Anlegen einer Veranstaltung wurde nachfolgend geöffnet und die notwendigen Daten eingegeben, im ersten Versuch dabei bewusst Fehler eingebaut. Beide Versuche zum Anlegen mit fehlerhaften und korrekten Daten werden überprüft. Nach erfolgreichem Anlegen wird zur Seite der neuen Veranstaltung weitergeleitet und der Editierbutton gesucht sowie der Status der Veranstaltung auf *before* angepasst. Nachfolgend wird ein Wert der Veranstaltung angepasst und die Übernahme der Änderung geprüft. Im letzten Schritt wird der Button für das Löschen der Veranstaltung gesucht und der Vorgang ausgeführt.
Analog zum Test für das Kurs-Management werden auch hier die Elemente auf den einzelnen Seiten gegen erwartete geprüft. Auch wird im Falle vom ersten Absenden des Formulars auf korrektes Verhalten der Anwendung bei Übergabe von fehlerhaften sowie korrekten Daten geprüft.

Testschritt	Elemente und Verhalten	Status
Aufruf Kursseite mit Prüfung auf Add-Lecture-Button	Add-Lecture-Button (.createLecture)	OK
Öffnen des Formulars nach Klick auf Add-Lecture-Button	Add-Lecture-Button (.createLecture)	OK
	Eingabefeld Name (#lectureName)	OK
	Eingabefeld Description (#lectureDescription)	OK
	Eingabefeld dp (name:dp)	OK
	Eingabefeld Duration (#lectureDuration)	OK
	Eingabefeld Hour (placeholder:HH)	OK
	Eingabefeld Minute (placeholder:MM)	OK
	Eingabefeld Slide-Count (#lectureSlideCount)	OK
Absenden fehlerhafter Testdaten	Send-Button (.createLectureSubmitButton)	OK

	Verhalten: Formularvalidierung erzeugt Fehler, Formular nicht sendbar und weiterhin offen	OK
Absenden korrekter Testdaten	Send-Button (.createLectureSubmitButton)	OK
	Verhalten: Eingaben werden akzeptiert und die Veranstaltung angelegt	OK
	Weiterleitung zur neuen Veranstaltungsseite	OK
	Edit-Button (.editLecture)	OK
	Status-Button *before* (.setStateBefore)	OK
Klick auf Button für Statusanpassung *before*	Status-Button *before* (.setStateBefore)	OK
	Verhalten: Statusanpassung (.stateHint)	OK
Editieren und Absenden von Veranstaltungsdaten	Eingabefeld (#lectureName)	OK
	Edit-Button (.editLectureSubmitButton)	OK
	Verhalten: Titel wurde angepasst (h2-Tag mit Text)	OK
Löschen der Veranstaltung	Delete-Button (.deleteLecture)	OK
	Verhalten: Abfrage-Overlay zum Bestätigen vorhanden	OK
	Veranstaltung wird gelöscht und zur Kursseite geleitet	OK
	Veranstaltung in Liste nicht mehr vorhanden	OK

Tabelle 8.7.: Evaluation: Test-Szenario 3 Tabellenübersicht

Das Ziel dieses Szenarios, den vollständigen Kreislauf von der Erstellung über das Aufrufen, Wechseln von Status und Editieren bis hin zum Löschen einer Veranstaltung abzubilden, wurde erreicht. Die erforderlichen Elemente sind auf den jeweiligen Seiten vorhanden und das erwartete Verhalten im Fehlerfall trifft zu.

8.2.4. VERWALTEN VON FRAGEN IM VERANSTALTUNGSKONTEXT

Durchgeführt wurde der Login mit der Rolle *Lecturer* und dem Startpunkt Veranstaltungsseite. Es wird geprüft, ob die korrekte Seite geladen ist und ob es einen Reiter für die Verwaltung von Fragen gibt. Zu diesem wird gewechselt und geprüft, ob ein Button für das Hinzufügen neuer Fragen vorhanden ist. Das Formular zum Anlegen einer neuen Frage wird nachfolgend geöffnet und die erforderlichen Daten eingegeben. Im ersten Versuch werden wieder Fehler eingebaut und überprüft, ob die Anwendung erwartungsgemäß reagiert und Fehler anzeigt. Nach dem erfolgreichen Anlegen der Frage wird nach kurzer Verzögerung geprüft, ob die Frage in der Übersicht gelistet ist und einen Editierbutton besitzt. Im Anschluss daran werden Werte der Frage angepasst. Der letzte Schritt prüft das Löschen der erstellten Frage.

Testschritt	Elemente und Verhalten	Status
Aufruf Veranstaltungsseite mit Prüfung auf Reiter und URL	Question-Reiter (#tab-1)	OK
	Aktuelle URL identisch mit gespeicherter Route der Veranstaltung	OK
Wechseln zum Question-Reiter und Öffnen des Formulars nach Klick auf Add-Question-Button	Add-Question-Button (.createQuestion)	OK
	Button Slide-Context (.contextSlide)	OK
	Button Before-Context (.contextBeforeLecture)	OK
	Button During-Context (.contextDuringLecture)	OK
	Button After-Context (.contextAfterLecture)	OK
	Button Course-Context (.contextCourse)	OK
	Eingabefeld Name (#questionTitle)	OK
	Eingabefeld Formulation (#questionFormulation)	OK
	Eingabefeld Position (#questionPosition)	OK
	Button SingleChoiceQuestion (.typeSCQ)	OK
	Button MultipleChoiceQuestion (.typeMCQ)	OK
	Button ScaleQuestion (.typeSQ)	OK
	Button StudySingleChoiceQuestion (.typeSSCQ)	OK
	Button StudyMultipleChoiceQuestion (.typeSMCQ)	OK
	Button FreetextQuestion (.typeFQ)	OK
	Button GroupedQuestion (.typeGQ)	OK
Absenden fehlerhafter Testdaten	Send-Button (.manageQuestionSubmitButton)	OK
	Verhalten: Formularvalidierung erzeugt Fehler, Formular nicht sendbar und bleibt offen	OK
Absenden korrekter Testdaten	Send-Button (.manageQuestionSubmitButton)	OK
	Verhalten: Eingaben werden akzeptiert und die Frage angelegt	OK
	Frage erscheint in *Before*-Liste (h6-Tag mit Text)	OK
	Edit-Button (.editQuestion)	OK
Editieren und Absenden von Frage-Daten	Eingabefeld (#questionTitle)	OK
	Edit-Button (.manageQuestionSubmitButton)	OK

	Verhalten: Titel wurde angepasst (h6-Tag mit Text)	OK
Löschen der Frage	Delete-Button (.deleteQuestion)	OK
	Verhalten: Abfrage-Overlay zum Bestätigen vorhanden	OK
	Frage wird gelöscht	OK
	Frage in Liste nicht mehr vorhanden	OK

Tabelle 8.8.: Evaluation: Test-Szenario 4 Tabellenübersicht

Das Ziel dieses Szenarios, den vollständigen Kreislauf von der Erstellung über das Aufrufen und Editieren bis hin zum Löschen einer Frage abzubilden, wurde erreicht. Die erforderlichen Elemente sind auf jeden jeweiligen Seiten vorhanden und das erwartete Verhalten im Fehlerfall trifft zu.

8.2.5. BEANTWORTEN EINER FRAGE

Durchgeführt wurde ein Login mit der Rolle *Student* und dem Startpunkt Kursseite. Ausgehend von dieser Seite wurde die aktive Veranstaltung aufgerufen und geprüft, ob Fragen zur Beantwortung vorhanden sind und ob es die richtigen Fragen sind. Nachfolgend wird die erste Antwortmöglichkeit der vorhandenen Frage gewählt und abgeschickt. Nach dem Senden wurde geprüft, ob die Frage weiterhin beantwortbar ist oder die entsprechenden Elemente deaktiviert sind. Im Anschluss daran wurde zur Evaluationsübersicht für diese Veranstaltung navigiert und geprüft, ob das dort angezeigte Ergebnis mit der zuvor ausgeführten Antwort übereinstimmt.

Testschritt	Elemente und Verhalten	Status
Aufruf Kursseite, Aufruf der Veranstaltung und Prüfung der URL	Aktive Veranstaltung (.lecture .ribbon-success)	OK
	Aktuelle URL identisch mit gespeicherter Route der Veranstaltung	OK
Prüfen auf vorhandene Frage mit korrekten Daten	Erste Frage (.answerQuestion)	OK
	Titel (.title)	OK
	Formulierung (.formulation)	OK
	Antwortmöglichkeiten (.choices)	OK
	Send-Button (type:submit)	OK
	Anzahl Antwortmöglichkeiten (.choices type:radio)	OK
Auswahl und Absenden einer Antwort	Erste Antwortmöglichkeit wählen (.choices type:radio:first)	OK
	Send-Button (type:submit)	OK
	Verhalten: Erster Radio-Button wird gewählt	OK
	Formular wird abgeschickt	OK
	Send-Button nach Absenden deaktiviert	OK
Navigieren zu Evaluation der Veranstaltung (ohne Prüfung der Schritte)	Fragen vorhanden (.answerQuestion)	OK

Titel der Frage stimmt mit beantworteter überein (.title > h4 mit Text)	OK
Korrekte Antwortmöglichkeit gewählt (.choices type:radio:first)	OK

Tabelle 8.9.: Evaluation: Test-Szenario 4 Tabellenübersicht

Das Ziel dieses Szenarios, die Beantwortung einer Frage mit anschließender Überprüfung der gegebenen Antwort über die Evaluation abzubilden, wurde erreicht. Die erforderlichen Elemente sind auf den jeweiligen Seiten vorhanden, die beantwortete Frage in der Evaluationsübersicht stimmt mit der vorherigen Selektion der Antwortmöglichkeit überein.

9. ZUSAMMENFASSUNG UND AUSBLICK

Im Rahmen dieser Masterarbeit wurden zum einen prototypisch Performance-Tests entwickelt, um die grundsätzliche Leistungsfähigkeit des Backends sowie Engstellen und kostenintensive Anfragen an das System zu ermitteln, zum anderen prototypische e2e-Tests für das Web-Frontend des AMCS-Projekts, um eine Integration von Frontend und Backend zu gewährleisten. Dieses Kapitel fasst die wesentlichen Inhalte zusammen und gibt einen Ausblick auf mögliche Erweiterungen und Anpassungen.

9.1. PERFORMANCE-TESTS

Zu Beginn wurden mehrere Werkzeuge zur Durchführung von Performance-Tests vorgestellt und evaluiert. Basierend auf definierten Kriterien wurde eine Auswahl vorgenommen und JMeter als zu verwendendes Werkzeug festgelegt. Die ausschlaggebenden Vorteile für die Wahl von JMeter liegen in der einfach zu bedienenden GUI und simplen Testerstellung sowie der Möglichkeit, Master-Slave-Konfigurationen für Performance-Test über mehrere Nodes hinweg umsetzen zu können. Für die Durchführung der Performance-Tests wurden mehrere Standard-Workflows und Anfragen evaluiert, die häufig genutzt werden und für die ein entsprechender Test sinnvoll ist. Der Fokus liegt dabei vor allem auf wiederkehrenden Abläufen wie der Abfrage von vielen unterschiedlichen Fragen. Diese Tests sind erweiterbar, auch wurde etwaige Web-Socket-Verbindungen vorerst nicht mit einbezogen.
Die einzelnen Test-Szenarien wurden schrittweise mit unterschiedlichen Nutzerzahlen bis maximal 1000 auf der Staging- und Production-Umgebung ausgeführt und die dabei ermittelten Ergebnisse miteinander verglichen. Zum Einsatz kommt dabei unter anderem die Bewertungsmethode APDEX, um die Nutzerzufriedenheit basierend auf der Antwortzeit bewerten zu können. Im Ergebnis zeigt sich, dass die Production-Umgebung hohe Lasten von 1000 parallelen Nutzer zwar noch fehlerfrei verarbeiten kann, dies jedoch nur auf Kosten einer stark erhöhten Antwortzeit möglich ist. Als Nadelöhr ist bei hohen Lasten immer das Rails-Backend in Verbindung mit dem Puma-Web-Server identifizierbar. Dieses Ergebnis ist für jedes getestete Szenario gleich.

Zurückzuführen sind die erhöhten Antwortzeiten darauf, dass die Systeme aktuell nur aus je einer Instanz bestehen. Weitere Instanzen in einem Load-Balancing-Verbund würden die Lasten besser verteilen und die Antwortzeiten zum Teil stark verringern. Diese Aufteilung auf mehrere Instanzen würde auch vermeiden, dass sich Anfragen gegenseitig behindern oder nicht ausgeführt werden können. Als Beispiel sei der Login genannt, der möglichst unabhän-

gig von der Abfrage von vielen Daten durchführbar sein soll.
Ein weiterer Engpass wurde beim Generieren der JSON-Antworten identifiziert. Für jede Anfrage wird das Ergebnis-Set erneut erstellt wird, es findet kein Caching stattfindet. Über partielles Caching von sich nicht oder nur selten ändernden Daten kann die Antwortzeit auch hier reduziert und das System performanter gemacht werden.

9.2. WEB-TESTS

Für die Durchführung von e2e-Tests gibt es viele unterschiedliche Möglichkeiten. Mit Maven, WebdriverIO und Protractor wurden einige Ansätze näher vorgestellt und die Vor- sowie Nachteile benannt. Um eine Wahl für die Umsetzung der Tests treffen zu können wurden Auswahlkriterien definiert und die vorgestellten Werkzeuge und Frameworks anhand dieser verglichen. Durch die Nutzung des Angular-Frameworks im AMCS-Projekt und die einfachere Syntax und Integration fiel die Wahl auf Protractor in Kombination mit dem Framework Jasmine, welches sehr umfangreich ist und alle benötigten Elemente und Erweiterungen bereits beinhaltet. Basierend auf der Liste aller Hauptfunktionalitäten des Projekts wurde eine Auswahl an umzusetzenden Web-Tests getroffen und diese prototypisch implementiert. Der Fokus bei diesen Tests liegt auf dem korrekten Seitenaufbau, der Funktionsweise und der Integration von Frontend und Backend, nicht auf Unit- oder Methoden-Tests.
Am bestehenden Quelltext wurden kleinere Anpassungen vorgenommen, um den Zugriff auf Elemente und die Arbeit mit Selektoren in den Tests zu vereinfachen und zu optimieren.

Weiterhin wurde das Konzept der Headless-Browser und die dafür erforderliche Systemkonfiguration näher betrachtet und auf einem Testsystem umgesetzt. Während der Bearbeitung des Themas wurde seitens Google eine neue Version des Browsers Chrome [27] veröffentlicht, welche die in Abschnitt 7.2 beschriebenen erforderlichen Schritte obsolet macht. Ab Version 59 ist es möglich, Chrome über einen Parameter im Headless-Mode zu starten, ohne dafür eine virtuelle Display-Umgebung zu benötigen. Die dafür notwendigen Parameter für die Protractor-Konfiguration sind hier kurz aufgeführt.

```
1 capabilities: {
2   'browserName': 'chrome',
3   'chromeOptions': {
4     'args': ['--headless', '--disable-gpu', '--window-size=1280x1024', '
        allow-no-sandbox-job=true', 'no-sandbox']
5   }
6 }
```

Quelltext 9.1: Headless-Chrome Startparameter

9.3. AUSBLICK

In Zukunft können Konzepte für ein Load-Balancing der Anwendung und partielles Caching von Daten entwickelt werden, um eine verbesserte Performance zu erlangen und wiederkehrende Berechnungen nicht pro Anfrage ausführen zu müssen. Basierend auf den entwickelten Tests können weitere hinzugefügt und das System feingranular abgestimmt und angepasst werden.
Um generell die Leistung des Backends zu verbessern kann die Möglichkeit evaluiert werden, die zugrundeliegenden Programmiersprache *Ruby* mit dem Framework *Rails* hin zu einem PHP-basierten Framework anzupassen. Durch Puma als Zwischenstufe zwischen dem

normalen Web-Server und der Anwendung und aus dem genutzten Framework selbst ergibt sich ein Nadelöhr, welches sich beispielsweise durch den Einsatz von PHP mit FastCGI verringern ließe. Dieser Ansatz muss jedoch gut evaluiert und getestet werden.

Im Web-Frontend können die bestehenden Tests erweitert und verfeinert werden, um für e2e eine sehr hohe Testabdeckung zu erreichen. Bisher nicht näher betrachtete Aspekte und Funktionen wie Web-Sockets oder die Benachrichtigungen im Browser können in die Tests integriert und geprüft werden. Auch auf Basis der entwickelten e2e-Tests können weitere Konzepte für Unit- und Modul-Tests erstellt und diese schrittweise implementiert werden. Die entwickelte Test-Suite kann in eine CI-Umgebung wie Jenkins integriert und bei Anpassungen am Quellcode automatisiert ausgeführt werden, um dem Entwickler eine schnelle Rückmeldung über etwaige Fehler zu geben. Weiterhin ist es auch möglich, die entwickelten Performance-Tests regelmäßig und automatisiert über eine CI-Umgebung auszuführen.

A. LITERATURVERZEICHNIS

[1] *What is automated software testing? - Definition from WhatIs.com*. URL: http://searchsoftwarequality.techtarget.com/definition/automated-software-testing (besucht am 17. 04. 2017).

[2] *What is agile test automation pyramid? - Definition from WhatIs.com*. URL: http://searchitoperations.techtarget.com/definition/agile-test-automation-pyramid (besucht am 17. 04. 2017).

[3] Jirk Scholze. »Performancetests und Bottleneck-Analyse in Multischichtarchitekturen«. Diplomarbeit. Freie Universität Berlin, 2008. URL: http://www.performance-test.de (besucht am 21. 04. 2017).

[4] *Lasttest Definition, Ziele, Best Practices und Fehlervermeidung*. URL: https://www.testing-board.com/lasttest-und-performancetest/ (besucht am 21. 04. 2017).

[5] *Lasttest (Computer)*. URL: https://de.wikipedia.org/wiki/Lasttest_(Computer) (besucht am 21. 04. 2017).

[6] *Web testing*. URL: https://en.wikipedia.org/wiki/Web_testing (besucht am 22. 04. 2017).

[7] *Acceptance Testing - Software Testing Fundamentals*. URL: http://softwaretestingfundamentals.com/acceptance-testing/ (besucht am 22. 04. 2017).

[8] Alicia Sedlock. *An introduction to frontend testing | Creative Bloq*. URL: http://www.creativebloq.com/how-to/an-introduction-to-frontend-testing (besucht am 22. 04. 2017).

[9] Andrew Girdwood. *What is a headless browser?* URL: http://blog.arhg.net/2009/10/what-is-headless-browser.html (besucht am 23. 04. 2017).

[10] *TOOLSQA | Free QA Automation Tools Tutorials*. URL: http://toolsqa.com/selenium-webdriver/headless-browser-testing-selenium-webdriver/ (besucht am 23. 04. 2017).

[11] *Gatling Load and Performance testing - Open-source load and performance testing*. URL: http://gatling.io (besucht am 05. 05. 2017).

[12] *Apache JMeter - Apache JMeter™*. URL: http://jmeter.apache.org (besucht am 15. 05. 2017).

[13] *Load testing*.
URL: https://en.wikipedia.org/wiki/Load_testing#Load_testing_tools
(besucht am 15. 05. 2017).

[14] *Maven - Welcome to Apache Maven*.
URL: http://maven.apache.org (besucht am 12. 06. 2017).

[15] Al Scott. *Testing Asynchronous Behavior in JavaScript with Selenium*.
URL: https://spin.atomicobject.com/2015/05/12/testing-asynchronous-behavior-javascript-selenium/ (besucht am 12. 06. 2017).

[16] Harsh S. *3 Techniques to Generate Reports in Selenium Webdriver*.
URL: http://www.techbeamers.com/generate-reports-selenium-webdriver/
(besucht am 12. 06. 2017).

[17] *WebdriverIO - Webdriver bindings for Node.js*.
URL: http://webdriver.io (besucht am 13. 06. 2017).

[18] *Protractor - end-to-end testing for AngularJS*.
URL: http://www.protractortest.org/ (besucht am 19. 06. 2017).

[19] *Jasmine Documentation*.
URL: https://jasmine.github.io/ (besucht am 20. 06. 2017).

[20] *Mocha - the fun, simple, flexible JavaScript test framework*.
URL: https://mochajs.org (besucht am 20. 06. 2017).

[21] David Tang. *Jasmine vs. Mocha, Chai, and Sinon - The JS Guy - David Tang*.
URL: http://thejsguy.com/2015/01/12/jasmine-vs-mocha-chai-and-sinon.html
(besucht am 20. 06. 2017).

[22] Vitalik Zaidman.
An Overview of JavaScript Testing in 2017 – powtoon-engineering – Medium.
URL: https://medium.com/powtoon-engineering/a-complete-guide-to-testing-javascript-in-2017-a217b4cd5a2a (besucht am 20. 06. 2017).

[23] *Projektinformationen — Professur für Rechnernetze — TU Dresden*.
URL: https://tu-dresden.de/ing/informatik/sya/professur-fuer-rechnernetze/forschung/forschungsprojekte/projektinformationen?pID=132
(besucht am 17. 05. 2017).

[24] Jakob Nielsen. *Response Time Limits: Article by Jakob Nielsen*.
URL: https://www.nngroup.com/articles/response-times-3-important-limits/
(besucht am 29. 05. 2017).

[25] *Apdex: Measuring user satisfaction | New Relic Documentation*.
URL: https://docs.newrelic.com/docs/apm/new-relic-apm/apdex/apdex-measuring-user-satisfaction (besucht am 06. 06. 2017).

[26] *XVFB*. URL: https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml
(besucht am 21. 04. 2017).

[27] Eric Bidelman. *Getting Started with Headless Chrome | Web | Google Developers*.
URL: https://developers.google.com/web/updates/2017/04/headless-chrome
(besucht am 26. 07. 2017).

[28] *Testautomatisierung Definition, Tutorial und Artikel*.
URL: https://www.testing-board.com/testautomatisierung/ (besucht am
15. 04. 2017).

[29] *Why Test Automation? Automated Testing Benefits and Tips*. URL:
https://smartbear.com/learn/automated-testing/ (besucht am 17. 04. 2017).

[30] Torsten Horn. *Maven 3.5*.
URL: http://www.torsten-horn.de/techdocs/maven.htm (besucht am 12. 06. 2017).

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle wörtlich oder sinngemäß übernommenen Gedanken und Zitate als solche kenntlich gemacht habe. Ich erkläre ferner, dass ich die vorliegende Arbeit an keiner anderen Stelle als Prüfungsarbeit eingereicht habe oder einreichen werde.

Heidenau, 18.08.2017 Samuel Knobloch