BACHELORARBEIT

Zum Thema

Evaluation und Umsetzung von Performance-Tests

mit Hilfe von CodeceptJS

Vorgelegt an der

[redacted]

Von: [redacted]

[redacted]

[redacted]

Matrikelnummer: [redacted]

Studiengang: Praktische Informatik

Studienrichtung: Technik

Praxispartner: dotSource GmbH

Goethestraße 1

07743 Jena

Gutachter der

Dualen Hochschule Gera-Eisenach: [redacted]

Gutachter des Praxispartners

(Betreuer i.S.v. § 20 (1) DHGEPrüfO): [redacted]

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH. Die Arbeit ist nur Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Abstract

Studien zeigen, dass Absprungraten von Webseitennutzern mit der Dauer von Ladezeiten ansteigen. Dies kann einen potenziellen Umsatzverlust für den Seitenbetreiber bedeuten. Ladezeiten können durch unterschiedliche Faktoren beeinflusst werden. Ein relevanter Faktor bei Onlineshops kann dabei die Anzahl der zeitgleich eintretenden Kunden sein. Zu viele Nutzer können zu einer Überlast des Webservers und somit zu höheren Antwortzeiten führen. Um solche Probleme frühzeitig zu erkennen, werden Lasttests an Servern durchgeführt, um die Auswirkungen von hohen Nutzerzahlen auf Seitenladezeiten zu messen. Diese Arbeit beschäftigt sich daher mit der Konzeption, Entwicklung und Durchführung von Lasttests unter der Verwendung des Frontend-Testing Frameworks CodeceptJS. Jedoch war es bislang unbekannt, ob und inwiefern sich Lasttests mit Hilfe dieses Tools umsetzen lassen. Um dieses Ziel zu erreichen, wurde ein Testkonzept aufgestellt, welches alle benötigten Testabläufe definiert. Anschließend wurden Testskripte erstellt, welche anhand eines Benutzermodells Webseitenbesucher simulieren und deren Auswirkungen auf einem Testserver messen. Dieser Versuch konnte aufzeigen, dass es durchaus möglich ist, Lasttest mit CodeceptJS aufzusetzen. Jedoch benötigt es einen hohen Aufwand, alle Funktionalitäten von XLT nachzubilden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CJS	CodeceptJS
DS	dotSource GmbH
OS	Online Shop
PLT	Performance- und Lasttests
VM	Virtuelle Maschine
VN	Virtueller Nutzer
XLT	Xceptance Load Test

1 Die Bedeutung von Performance- und Lasttests

1.1 Auswirkungen von Serverlasten auf Webnutzer

Das Internet entwickelt sich seit seiner Erfindung unaufhaltsam fort. Jährlich nimmt die Anzahl von Internetanschlüssen und Nutzer weltweit stark zu. Aufgrund der rasant steigenden Verfügbarkeit von schnellen Breitbandverbindungen tolerieren immer weniger Internetnutzer langsam arbeitende Webseiten. Bei nicht responsiven Anwendungen kann sich beim Nutzer Frust aufbauen und ihn dazu veranlassen, die jeweilige Webseite zu verlassen. Ein solcher Vorfall kann für den Seitenbetreiber einen Umsatzverlust bedeuten. Absprungraten sind eine der wichtigsten Metriken zur Messung des Erfolges einer Webseite. Studien von Google LLC haben deutlich gezeigt, dass die Absprungraten von Webnutzern mit der Ladezeit der Webseiten stark steigen:

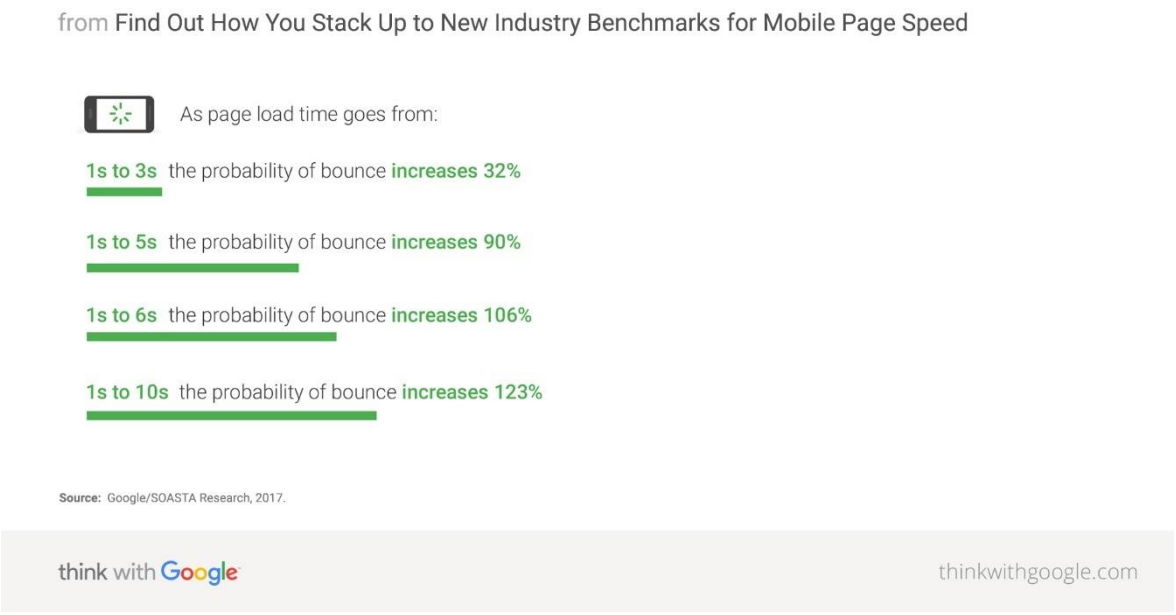

Abb. 1: Auswirkung von Ladezeiten auf Absprungraten[1]

Wie in dieser Abbildung dargestellt, können bereits wenige Sekunden eine signifikante Auswirkung auf die Nutzererfahrung haben.

[1] [An18]

Häufige Ursachen für Leistungseinbußen von Webseiten sind zum Beispiel aufwändige Anfragen, Bugs oder Überlastungen der Server durch hohe Benutzerzahlen. Mit dem Wachstum an Menschen, die ihre Tätigkeiten online vornehmen, steigt die Anzahl von Anfragen an Webservern und zwangsläufig auch die Auslastung dieser. Bei zu hoher Belastung eines Webservers können Anfragen nicht mehr zeitnah beantwortet werden, was zu den unerwünschten höheren Ladezeiten für den Nutzer führt.

Aus diesem Grund sollten Webseitenbetreiber immer sicherstellen, dass ihre Seiten stets schnell Inhalte liefern und Webserver mit unerwarteten Lastspitzen umgehen können. Ein geeignetes Mittel zur Überprüfung von Antwort- und Ladezeiten ist das Durchführen von Performance- und Lasttests (PLT). Bei PLT geht es um den Einsatz von Software um Leistungsmetriken für eine Webapplikation aufzuzeichnen und zu bewerten. Sie dienen zum rechtzeitigen Erkennen und Beheben von Problemen, die durch eine höhere Auslastung des Servers hervortreten können. Insbesondere gravierende Fehler können zu Ausfällen von Webservern führen, was sinkende Kundenzufriedenheit zur Folge haben kann. Dies würde folglich Umsatzraten negativ beeinflussen. PLT werden in zwei Arten unterteilt:

Performancetests überprüfen die Leistungsfähigkeit, Stabilität oder Skalierbarkeit der Webapplikation. Sie sollten idealerweise nach jeder Änderung des Quellcodes durchgeführt werden, um zu versichern, dass die Qualität der Serverleistung nicht unter Neuerungen leidet. Sie helfen dabei sicherzustellen, dass die Benutzer auf jeder Plattform auf alle Funktionalitäten einer Webapplikation zugreifen können. Unterschiedliche Computer, Betriebssysteme oder Browser können Ladezeiten und weitere Leistungsmerkmale beeinflussen. Performancetests sind wichtig, um potenzielle Probleme auf verschiedenen Plattformen rechtzeitig zu erkennen.

Lasttests überprüfen das System auf Antwort- und Ladezeiten unter hoher Auslastung. Dies kann beispielsweise erzielt werden, indem eine große Anzahl an Nutzern mit verschiedenen Transaktionen auf dem System simuliert werden. Sie helfen bei der Entscheidung, über welche Ressourcen der Webserver verfügen sollte und welche Richtlinien zur Skalierung eingehalten werden müssen. Bei einem Lasttest werden keine Funktionstests durchgeführt. Das heißt, es wird nicht geprüft ob z.B. das Anklicken einer Schaltfläche auf einer Webseite die tatsächlich gewünschte Funktion durchführt.

1.2 Aktuelle Lasttest Werkzeuge im Einsatz

Derzeit verwendet die dotSource GmbH (DS) das Testautomations- und Lasttestwerkzeug Xceptance Load Test (XLT)[2] für die Überprüfung der Systemleistung ihrer Kundenwebshops. XLT bietet eine breite Palette zur Erstellung und Durchführung von Tests verschiedener Arten. Das Tool verfügt über eingebaute Funktionalitäten zur Ausführung von Tests auf verteilten Systemen und zeichnet zahlreiche Metriken über die Leistung der laufenden Maschinen auf. Des Weiteren ist XLT auf Analytik spezialisiert und besitzt stark ausgeprägte Reporting Funktionen, welche viele Daten in übersichtliche Graphen darstellen können. Der große Umfang an Test- und Reportmöglichkeiten von XLT bringt jedoch einen erheblichen Nachteil mit sich. Es ist ein komplexes Programm mit vielen Dateien, Konfigurationseinstellungen und Erweiterungen. Es ist daher sehr zeitaufwendig aufzusetzen und Schulungen für neue Mitarbeiter darin durchzuführen. Darüber hinaus sind unerwartete Probleme bei einem solch komplexen Programm schwieriger zu behandeln. Diese Aufwände reduzieren unter Umständen die Produktivität des jeweiligen Entwicklers und folglich auch des Firmenumsatzes.

Aus diesem Problem hat sich ergeben, dass es sinnvoll wäre, neben diesem Tool eine einfachere Variante bei kleineren Projekten zu verwenden. Dabei richtete man sich nach bereits in der Firma bewährten Werkzeugen. In diesem Zusammenhang wurde CodeceptJS (CSJ) als Lösung in Betracht gezogen. Bereits im Jahr 2019 wurde in der DS bei der Umsetzung von Akzeptanztests ein Umstieg von XLT auf CJS untersucht. Es liegt daher nahe, dieses Framework beizubehalten und wiederzuverwenden, da es sich bereits als nützlich erwiesen hat. CJS hat den Vorteil gegenüber XLT, dass es deutlich einfacher zu verstehen und aufzusetzen ist. Jedoch ist CJS offiziell nur auf Akzeptanztests ausgelegt und bietet daher keine direkt passenden Funktionalitäten zum Ausmessen und Auswerten von Ladezeiten. Es ist daher unklar, ob sich ein solches Framework für PLT eignet.

[2] [Xce22a]

Aus dieser Ungewissheit leitet sich das Thema dieser Arbeit ab. Es wird untersucht ob, und wie sich die Durchführung von PLT mit Hilfe von CSJ sinnvoll umsetzen lässt. Um dieses Ziel zu erreichen, müssen zuerst die funktionellen Möglichkeiten von XLT und CSJ verglichen werden. Anschließend werden Testfälle definiert, welche ein eindeutiges Ergebnis über die Verwendung von CSJ liefern können.

Das Testkonzept wird folgende Inhalte haben:

Schritt	**Beschreibung**
Bestimmen von Leistungskriterien	Erstrangig muss die Webapplikation auf Antwortzeiten, Durchsatzraten und Ressourcenauslastung bei bestimmten Nutzeraktionen überprüft werden.
Testplanung	Für die Untersuchung werden einige Testfälle benötigt. Diese können aus den bisher bestehenden XLT Tests abgeleitet werden. Sie sollten jedoch so ausgelegt werden, dass Veränderungen in der Performance einer Webapplikation genau nachgewiesen und repliziert werden können.
Einrichten der Umgebung	In diesem Abschnitt wird die benötigte Hardware herbeigezogen und die notwendige Software installiert. Konfigurationen über die Zeitplanung und Skalierung der Tests werden hier ebenfalls betrachtet.
Testdurchführung	Die Tests werden auf den jeweiligen Maschinen zu einem bestimmten Zeitpunkt oder auch in regelmäßigen Abständen ausgeführt.
Aufzeichnung und Auswertung von Ergebnissen	Um ein finales Ergebnis über dieses Experiment zu erhalten, müssen sinnvolle Messwerte in einer geeigneten Form aufgezeichnet werden. Nach Auswertung dieser Messwerte lässt sich eindeutig bestimmen, ob CodeceptJS sich tatsächlich für PLT eignet.

Tab. 1: Inhalte des Testkonzepts

Im nächsten Abschnitt werden alle für diese Arbeit relevanten Tools zur Durchführung von PLT genauer beschrieben und evaluiert. Dazu erfolgt eine Konkretisierung der Bestandteile des Testkonzepts. Im dritten Kapitel wird dann der eigentliche Umsetzungsprozess behandelt. Dabei wird zuerst auf die Installation aller benötigter Software sowie deren Konfiguration eingegangen und als zweites der Entwicklungsprozess der Testdateien beschrieben. Im Anschluss darauf werden die Ergebnisse zusammengefasst und interpretiert. Zuletzt erfolgt eine Einschätzung der Ergebnisse aus dieser Arbeit und zeigt deren Nutzen für die Zukunft auf.

Es existieren mehrere Tools welche speziell auf das Durchführen von PLT ausgelegt sind. Diese Arbeit legt das Hauptaugenmerk auf die Untersuchung der Tauglichkeit von CSJ für PLT. Daher ist die Betrachtung oder der Vergleich von anderen Testing Frameworks neben CSJ und XLT nicht Bestandteil dieser Arbeit. Lasttests setzen voraus, dass eine hohe Anzahl von Rechenmaschinen bereitgestellt und koordiniert werden müssen. Versuche in dieser Hinsicht erfordern ein für diese Arbeit nicht vorgesehenes Budget. Demzufolge wird dieser Aspekt der Aufgabe hier nicht weiter behandelt. Weiterhin werden die im späteren Verlauf dieser Arbeit beschriebenen Testsysteme nicht detailliert erläutert. Die Funktionsweise der Testserver sind grundsätzlich für das Ausüben von Lasten irrelevant.

2 Projektvorbereitung

Dieser Abschnitt beschreibt die die wichtigsten Tools zur technischen Umsetzung von PLT mit CSJ. Im zweiten Teil wird ein Ansatz für die Lösung der Problemstellung unter der Benutzung dieser Tools vorgestellt. Ein Testkonzept wird aufgestellt, um einen strukturierten Ablauf der Untersuchung zu garantieren und um einen klaren Vergleich zwischen der aktuellen XLT Lösung und der neuen CJS Umsetzung zu ermöglichen.

2.1 Lösungsrelevante Werkzeuge

Um einen genaueren Überblick über die gewünschten Funktionalitäten des Testkonzepts zu erhalten, werden zunächst die Möglichkeiten der aktuellen Lösung betrachtet. Bei XLT handelt es sich um ein mehrzweckmäßiges PLT Tool mit Möglichkeiten zur Testautomatisierung[3]. Die Testautomatisierung wird über eine erweiterte Form von WebDriver realisiert. WebDriver ist eine Schnittstelle zur Fernsteuerung von Benutzeragenten. Es bietet über ein Programmiersprachenunabhängiges Protokoll die Möglichkeit, Webbrowser aus der Ferne zu steuern.[4] Lasttests und Testautomatisierung erlauben es clientseitige Performancetests durchzuführen. Dies ermöglicht es, Tests über einen kompletten Browser auszuführen, anstatt alle Eingaben manuell vorzunehmen. XLT basiert auf Java und ist somit auf den meisten gängigen Computer lauffähig. Zusätzlich erlaubt Java nicht nur die Konfiguration von, sondern auch die freie Anpassung sämtlicher Testskripte. Darüber hinaus existieren zahlreiche Java-Bibliotheken, deren Funktionalitäten sich in jegliche Testdateien einbinden lassen.

[3] [Xce22b]
[4] [Wor22b]

Folgende Metriken lassen sich von einem Computer oder einem verteilten System von Rechenmaschinen mit Hilfe von XLT messen:

Wert	**Beschreibung**
Systemauslastung (Testsystem)	Wie sehr wird die CPU beansprucht? Wie viel RAM wird genutzt?
Fehler	Wo treten Fehler auf? Wann treten Fehler auf?
Nutzerzahlen	Wie viele Nutzer verwenden den Shop aktuell? Wann tauchen Besucherspitzen auf?
Lade- und Antwortzeiten	Wie entwickeln sie sich bei welchen Nutzerzahlen?
Entwicklungen über Zeiträume	Gehen die Werte nach einiger Zeit zurück? Wann gibt es Lastspitzen?

Tab. 2: Messbare Leistungsmetriken eines Systems mit Hilfe von XLT

Alle in dieser Tabelle dargestellten Funktionalitäten müssen für das Ziel dieser Arbeit mit einem einfacheren Werkzeug umgesetzt werden. Einige Projekte der DS verwenden bereits das Tool CJS, um automatisierte browsergesteuerte Tests zu realisieren. CSJ ist ein modernes Ende-zu-Ende Testing Framework mit einer besonderen verhaltensorientierten Syntax.[5] Die Tests werden als einfache Abfolge über die Tätigkeiten eines Nutzers auf einer Webseite beschrieben. Jeder Test wird in einer Szenario-Funktion geschrieben, welche Befehle für einen Browser enthalten. CSJ übergibt diese Befehle an Hilfsprogramme, welche eine Browsersitzung starten. Dieses Framework basiert auf NodeJS.

NodeJS ist eine plattformübergreifende JavaScript Laufzeitumgebung. Sie eignet sich hauptsächlich für skalierbare Webapplikationen. Zusätzlich können weitere Funktionalitäten schnell und einfach über zahlreiche open-source-Pakete zu einem Projekt hinzugefügt werden. Für diese Arbeit spielt zum Beispiel das Vega-Paket eine bedeutende Rolle. Da die Analyse von Messwerten ein wichtiger Teil von PLT sind, ist es lohnenswert diese Daten in verschiedenen Formen darzustellen. Dazu gehört das Erstellen von Graphen, um visuell Verhaltensmuster der Testserver erkennen zu können.

[5] [Cod22]

Vega ist eine Visualisierungsgrammatik zur Erzeugung von Diagrammen. Die Spezifikation der zu erstellenden Abbildung wird von der JavaScript Laufzeitumgebung analysiert. Vega bietet eine bequeme Darstellung für die rechnergestützte Generierung von Visualisierungen und kann als Grundlage für neue APIs und visuelle Analysewerkzeuge dienen.[6]

Der Zusammenhang der Module und ihrer Aufgaben wird in diesem Diagramm verdeutlicht:

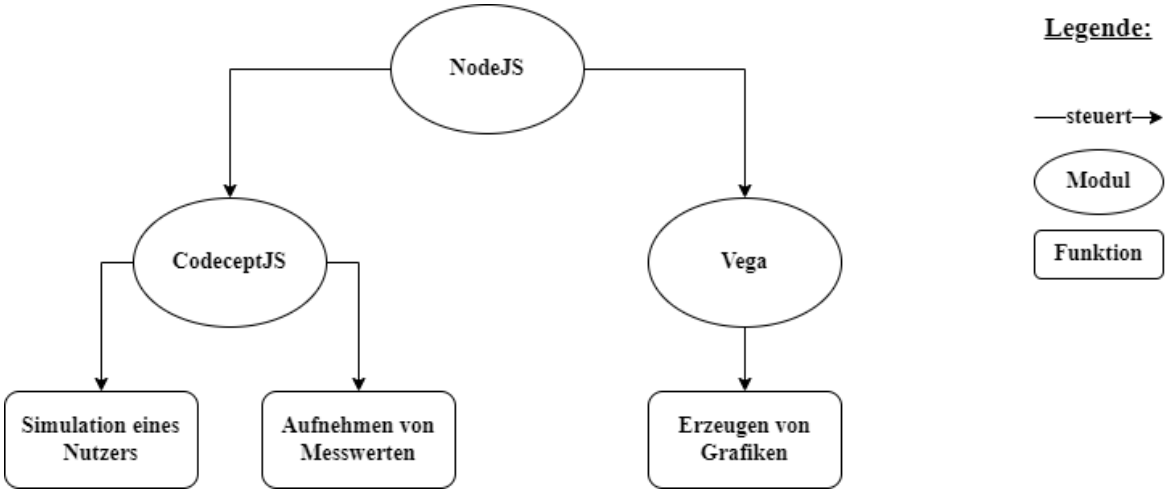

Abb. 2: Komponenten für diesen Versuch

Das Kernstück der Applikation ist der Lasterzeuger. Dabei dienen die Testskripte für CJS dazu, das Verhalten der virtuellen Nutzer (VN) nachzuahmen. Während die Nutzer den gewünschten Server belasten, werden innerhalb der Testskripte Ladezeiten gemessen und aufgezeichnet. Aus diesen Messwerten werden dann mit Hilfe von Vega Graphen erzeugt. NodeJS bietet dabei die Laufzeitumgebung für beide Tools. Jeder steuernde Pfeil in dieser Abbildung muss durch ein eigenes Skript realisiert werden, welche einige Funktionalitäten von NodeJS übernehmen. Der genaue Ablauf und Inhalt dieser Skripte werden im nachfolgenden Kapitel näher beschrieben.

[6] [Veg22]

2.2 Messungen von Lade- und Antwortzeiten

In webbasierten Umgebungen werden bei der Messung von Ladezeiten konkret Ende-zu-Ende Antwortzeiten gemessen. Genauer beschreibt das die Zeit zwischen dem Aus- und Eingang von gewünschten Datenpaketen zwischen zwei Geräten. In diesem Fall bezieht es sich auf den Computer des Nutzers, worauf der Webbrowser läuft, und dem Server, auf dem die Webapplikation zur Verfügung gestellt wird. Die Messungen spiegeln die wahrgenommenen Wartezeiten des Nutzers bis zur vollständigen Ladezeit einer Webseite oder Antwort einer Suchanfrage wider. Bei der Definition von Ende-zu-Ende Antwortzeiten muss unterschieden werden zwischen der Zeit, bis eine HTML-Datei heruntergeladen wird, und der Zeit bis andere Komponenten der Seite wie beispielsweise Bilder und Skripte geladen wurden.[7]

Bei der Suche nach Systemengpässen wird normalerweise das System unter hoher Last untersucht. Dies wird gelegentlich auch als Stresstest bezeichnet. Eine wichtige Kennzahl, die es zu verfolgen gilt, ist die sogenannte Roundtrip-Antwortzeit. Sie beschreibt die Zeit zwischen der Anfrage eines Programms an einen Server und dem Empfang der Antwort. Dies spiegelt die genaue Zeit wider, die ein echter Benutzer bis zum vollständigen Laden einer Webseite warten müsste. Außerdem wird die Zeit zwischen dem Erhalt einer Anfrage und der Zeitpunkt bis zum Senden einer Antwort überwacht. Andere ausschlaggebende Maße sind zum Beispiel die Prozessorauslastung und Arbeitsspeichernutzung des Servers.[8] Der übliche Ansatz eines Lasttests ist die Verwendung eines inkrementellen Arbeitslastmodells worin so lange VN hinzugefügt werden, bis die Ladezeiten das System unbrauchbar machen. Im Detail bedeutet das die Zeit, in welche ein durchschnittlicher Nutzer die Webseite verlassen würde. Mit den Messdaten des Browsers kann die Ladezeit von verschiedenen Nutzeraktionen analysiert werden. Die Anzahl der Benutzer, bei denen die Antwortzeit eine bestimmte Grenze überschreitet, unterscheidet sich normalerweise zwischen den ausgeführten Aktionen. Je nachdem wie oft Webseiten aufgerufen werden und von welchen technischen Ressourcen sie angetrieben werden, können Ladezeiten unterschiedlich ausfallen.

---

[7] [Men02]
[8] [Dra06], S. 6

Aktionen, bei denen die Grenze frühzeitig überschritten wird, stellen einen Engpass des Systems dar und lassen in der Regel Rückschlüsse auf das zugrunde liegende Subsystem schließen.[9] PLT werden üblicherweise mit parametrisierten Arbeitslasten durchgeführt. Ein Grundbaustein für Lasttests ist die Definition eines VN. Ein VN simuliert das Verhalten eines echten Besuchers einer Webseite. Ein Lasttest ist nur tauglich, wenn die VN sich möglichst realitätsnah verhalten. Dies beinhaltet übliche Tätigkeiten auf einer Webseite, Lese- und Denkzeiten und auch das Verlassen aufgrund von langen Wartezeiten oder Fehlerausgaben. Wenn ein Lasttest nicht ausreichend genau wahre Nutzer simulieren kann, können die Testergebnisse verfälscht werden. Beispielsweise erzeugen VN welche ihre Sitzung vorzeitig abbrechen, eine geringere Last als die, die alle ihre gewünschten Aktionen vollständig durchführen können. Durch solche Verfälschungen wird es für den Seitenbetreiber schwerer, die potenziellen Umsatzverluste einzuschätzen. Dies sind wichtige Metriken auf der Geschäftsstufe.[10]

Vor dem Beginn der Testplanung müssen bestimmte Richtwerte bekanntgegeben werden, um die Tests ordentlich zu skalieren. Jede Sitzung und jede Nutzeranfrage beanspruchen einen Teil der verfügbaren Rechenressourcen des angesprochenen Systems. Je mehr Clients der Server bearbeiten muss, desto mehr Speicher und Rechenleistung werden benötigt. Darüber hinaus muss noch beachtet werden, dass der Rechenaufwand für bestimmte Aufgaben höher sein können als andere. Beispielsweise wäre die Eingabe einer Produktsuche in einem Onlineshop (OS) viel aufwändiger als das Aufrufen eines Impressums. Grund hierfür wäre die zusätzliche erforderliche Rechenleistung für eine Suche, Sortierung oder ggf. Abfrage einer Datenbank, um ein Ergebnis zurückzusenden. Ein Impressum hingegen würde den Webserver vergleichsweise weniger belasten, da dem Nutzer hauptsächlich nur statischer Text geliefert werden muss.

[9] Ebenda, S. 6 f.
[10] [Men02]

Die wichtigsten Paramater bei der Erstellung von Lasttests sind:

Parameter	**Beschreibung**
Lastintensität	Wird normalerweise in Sitzungen pro Stunde gemessen
Lastverteilung	Die Art von Testszenarien und deren Anteile am gesamten Nutzerverhalten
Nutzerverhalten	Typische Verhaltensweisen von Nutzern wie beispielsweise Bedenkzeiten oder Wartezeittoleranz

Tab. 3: Die wichtigsten Parameter für Lasttests

Typische Ergebniswerte von Lasttests beinhalten:

- Die Anzahl von vollständig durchgeführten Testszenarien
- Die Anzahl abgebrochener Testszenarien mit der jeweiligen Fehlerbegründung
- Die Ladezeiten von Webseiten und Transaktionen

Vor der Durchführung bietet es sich an, die gewünschte Performance einer Anwendung festzuhalten. Anschließend müssen Ziele und Ablauf der Tests in einem Lasttesttool festgehalten werden. Mögliche Punkte können beispielsweise sein:

- Die Anzahl der VN über festgelegte Zeiträume
- Akzeptable Antwortzeiten bzw. Verzögerungen
- Zu welchen Zeiten sind mehr oder weniger Verzögerung akzeptabel?
- In Welchem Umfang sind Ausreißerwerte akzeptabel?
- Wie hoch ist das maximal erforderliche Datenvolumen?
- Festlegung der Fehlertoleranz
- Die durchschnittliche und obere Auslastung des OS (vom Betreiber zu erfragen)

Für das Ausüben von Lasten und Messen deren Auswirkungen müssen drei Sachverhalte geklärt werden: Als erstes muss die verwendete Hardware betrachtet werden. Wichtige Leistungskriterien der Server sind die Menge und Art an verfügbarem Arbeitsspeicher sowie die Art des Prozessors und wie viele Kerne er besitzt.

Zweitens muss die Anzahl an Sitzungen bestimmt werden. Dazu werden die Menge an VN betrachtet, die über einen bestimmten Zeitraum den Server beanspruchen. Der Dritte Sachverhalt ist der Verteilungsgrad der Sitzungen. Die Tests sollten an reale Verläufe des zu testenden Systems angepasst sein. Bei einem OS zum Beispiel durchstöbert ein Großteil der aktuell verbundenen Nutzer nur die Produkte während nur ein geringer Anteil einen Kaufprozess abschließt.

Nachdem die grundlegenden Voraussetzungen geklärt sind, können Leistungskriterien für die Webapplikation aufgenommen werden. Wichtige Leistungskriterien können folgende Werte sein:

- Seitenaufrufe je Tag/Stunde
- Suchanfragen je Tag/Stunde
- Bestellungsvorgänge je Tag/Stunde
- Nutzer gleichzeitig auf dem System
- Durchschnittliche Dauer je Sitzung
- Seitenaufrufe je Sitzung

Darüber hinaus müssen Zielwerte für Nutzeraktionen festgelegt werden. Diese Werte müssen unterschritten werden, um die Ladezeiten als akzeptabel einzustufen. Sie können beliebig vom Webseitenbetreiber ausgesucht werden, sollten sich aber idealerweise an die Werte aus Abb. 1 halten. Für diesen Versuch werden folgende Grenzwerte definiert:

Antwortzeit	**Zielwert (in MS)**	**Grenzwert (in MS)**
Seitenaufruf	500	2000
Suche starten	2000	5000
Produkt in Warenkorb legen	1000	3000
Bestellung aufgeben	2000	5000

Tab. 4: Zielwerte für Ladezeiten bei verschiedenen Nutzeraktionen

Die in dieser Tabelle dargestellten Werte wurden für diese Arbeit willkürlich gewählt. Aufwändigere Aktionen haben dabei höhere Zielwerte. Mit diesen Grenzwerten lässt sich ein Testkonzept aufsetzen, mit dem sich die Auswirkungen von Serverlasten überprüfen lassen.

2.3 Konzept zur Umsetzung der Tests

Da die erforderlichen Messwerte bestimmt wurden, müssen nun Szenarien definiert werden, an denen sie getestet und aufgenommen werden können. Diese Arbeit konzentriert sich hauptsächlich auf Ladezeiten von Webshop Oberflächen. Aus diesem Grund werden alle Benutzeraktionen aus der Sicht eines Kunden durchgeführt. Häufig allgemein auftretende Ereignisse bei OS können folgende sein:

Testfall	**Beschreibung**
Besuch	Der Nutzer besucht die Startseite und verlässt sie wieder sofort.
Browsing	Der Nutzer wählt zufällig ein Feld auf der Navigationsleiste aus und klickt auf ein zufälliges Produkt zur Betrachtung.
Suchen	Der Nutzer gibt einen Suchbefehl in der Suchleiste ein und wählt das erstgefundene Produkt aus.
Warenkorb	Beginnt mit dem Testfall Browsing: Der Nutzer fügt ein Produkt zum Warenkorb hinzu.
Checkout	Beginnt mit dem Testfall Warenkorb: Der Nutzer gibt Bestelldaten ein und bestätigt die Bestellung.
Registrierung	Der Nutzer besucht die Startseite und erstellt einen neuen Benutzeraccount.
Login	Der Nutzer besucht die Startseite und logt sich mit seinen Benutzerdaten ein.

Tab. 5: Beschreibungen von Testfällen

Lasttests benötigen sowohl ein Testsystem als auch einen Lasterzeuger. Für das Testkonzept werden daher mindestens zwei Geräte benötigt. Wenn der Lasterzeuger selbst Performanceeinbußen registriert, würden die Testresultate verfälscht werden. Aus diesem Grund wird Hierbei ein Testsystem über eine virtuelle Maschine (VM) simuliert. Auf der VM wird ein HTTP Server aufgesetzt, welcher eine einfache Benutzeroberfläche zur Navigation verfügt, mit der gezielt Funktionen mit unterschiedlichen Rechenaufwänden für den Server ausgeführt werden können. Um eine Last schneller und leichter auszuüben, werden der VM möglichst geringe Rechenressourcen zugeteilt.

Beim Lasterzeuger muss sichergestellt werden, dass dieser selbst nur einfache Anfragen versendet, welche ihn selbst nicht überlasten können. Für diesen Versuch wird der Lasterzeuger gleichzeitig die Hostmaschine für die VM des Testsystems sein. Dies ermöglicht eine leichte Anpassung von Testdateien. Somit kann die Leistungsüberprüfung beider Systeme reibungslos verlaufen.

Zu Beginn wird das System ohne sonderliche Belastung auf Antwortzeiten getestet. Dabei wird mehrmals nur eine Anfrage zur selben Zeit an den Server geschickt. Aus den Werten wird dann ein Durchschnitt ermittelt. Das gleiche wird mit verschiedenen Arten von Anfragen wiederholt; Einmal für rechenschwache Anfragen und je einmal für rechenaufwändigere Anfragen. Hinterher werden diese Tests erneut ausgeführt, aber mit mehreren Nutzern parallel. Dabei werden die Ressourcenwerte der VM mit aufgezeichnet, um nachzuweisen, dass diese auch tatsächlich unter der Rechenlast leidet. Wenn diese Vorbereitungstests erfolgreich sind und sinnvolle Ergebnisse liefern, wird ein vollständiger Lasttest auf einem echten OS Testsystem ausgeführt. Dieser versucht dann verschiedene Szenarien von Nutzeraktionen in ordnungsgemäßen Verhältnissen zu simulieren. Dieser Test wird dann auf verschiedene parallele Nutzerzahlen skaliert.

Der Hauptnutzen der Lasttests liegt in den aufgenommenen Testwerten. Sie müssen so gewählt werden, dass ein Zusammenhang zwischen Nutzerzahlen, Nutzeraktionen und Antwortzeiten sichtbar ist. Es ist daher sinnvoll verschiedene Testszenarien zu definieren welche in einem bestimmten Verhältnis Kundenaktionen in einer realen Shopumgebung simulieren. Die dabei aufgenommenen Messwerte müssen in einem sinnvollen Format gespeichert werden, um sie mit möglichst geringem Aufwand auswerten zu können. Es würde die Arbeit erheblich erleichtern, wenn die Werte über externe Tools ausgewertet werden können. Insbesondere wäre eine automatisierte Erzeugung von Graphen über ein vorgefertigtes Framework hilfreich

Abschließend müssen die Tests so eingerichtet werden, dass sie leicht zu verstehen, anpassbar und portabel sind. Dabei soll jedes Testszenario in ein eigenes Skript niedergeschrieben werden. Diese Skripte werden dann über ein NodeJS Hilfsprogramm gesteuert und ausgeführt. Dazu muss eine Konfigurationsdatei angelegt werden, die genauere Einstellungen der Testausführung ermöglicht.

Das Ziel dabei sollte sein, ein möglichst einfaches Testprogramm zu erhalten, welches über einen Befehl einen vorkonfigurierten clientseitigen Lasttest auf einer beliebigen Webplattform ausführen kann.

2.4 Technische Voraussetzungen

Wie bereits im vorherigen Abschnitt erwähnt, braucht dieser Versuch ein Gerät zur Erzeugung einer Last und einen dazugehörigen Lastempfänger. Als Hardware muss für den ersten Teil des Versuchs nur ein Rechner zur Verfügung gestellt werden. Dieser sollte leistungsfähig genug sein, eine dezente Menge an Browsersitzungen ohne Leistungseinbußen offen halten zu können. Das Testsystem hat keine spezifischen Anforderungen. Es muss sich dabei nur um einen Rechner handeln, auf dem ein Server läuft, welcher über eine API-Anfragen beantworten kann. Natürlicherweise muss diese für den Lasterzeuger über ein Netzwerk erreichbar sein.

Der Lasterzeuger benötigt ein Betriebssystem, welches mit NodeJS kompatibel ist. Zusätzlich muss eine Software zum Aufsetzen und Verwalten einer VM auf dem Gerät installiert werden. Die dann darauf laufende VM muss so konfiguriert werden, dass eine Netzwerkverbindung zum Host besteht. Außerdem müssen der VM signifikant weniger Rechenressourcen zur Verfügung gestellt werden, als das Hostsystem eigentlich bieten kann.

NodeJS ist eine JavaScript Laufzeitumgebung. Der Entwickler muss daher grundlegendes Wissen über JavaScript Programmierung haben. Kenntnisse über relevante NodeJS Module sind dabei vom Vorteil. Insbesondere dazu gehört Erfahrung im Umgang mit CJS für diesen Versuch. Des Weiteren sind Kenntnisse über die verwendeten Hilfsprogramme wie beispielsweise die Verwaltungssoftware der Test-VM hilfreich. Je nachdem wie das Testsystem skaliert ist, muss der Entwickler in der Lage sein, weitere Maschinen entweder lokal, über Cloud-Computer oder mit Hilfe von VM für die Testumgebung bereitstellen zu können. Kenntnisse über die Datenformate JSON und CSV sowie zu denen relevanten Analysewerkzeuge erleichtert die Auswertung der Testergebnisse. Eine solche Aufgabe eignet sich am besten für einen Qualitätssicherheitsmanager.

Das später in dieser Arbeit aufgeführte Wrapperskript sollte auch verschiedene Lastverläufe managen können. Zum einen muss ein Modus oder Testfall festgelegt werden, worin Nutzerzahlen allmählig ansteigen. Gegenüberstehend braucht es auch einen Testfall, wo viele Nutzer zum selben Zeitpunkt auf eine Seite auftreffen. Bei einem Stresstest wird ein geringer Startwert festgelegt der linear erhöht wird, bis das System Fehler aufweist und der Last nicht mehr standhält. Es wird versucht, mögliche Fehler aufzudecken, die nur unter hoher Auslastung des Systems auftreten. Um das System an seine Grenzen zu bringen, wird eine hohe Nutzerlast simuliert. Diese Nutzlast wird mit einer hohen Anzahl von VN und sich schnell wiederholenden Aktionen erzeugt. Dabei wird nur eine kleine bzw. keine Bedenkzeit des Nutzers verwendet. Bei einem Dauerlasttest wird das System einer kontinuierlichen Last über einen längeren Zeitraum ausgesetzt, um die Langzeitstabilität und den damit verbundenen Ressourcenverbrauch zu überwachen.

Ergebnisse dieser Tests werden in sogenannte Reports geschrieben. Es handelt sich dabei um Dateien, welche die gestarteten Testszenarien analysieren und die Messwerte in einem leicht auszuwertenden Format beinhalten.

Die Reports werden hierarchisch nach ihrer Art eingeteilt:

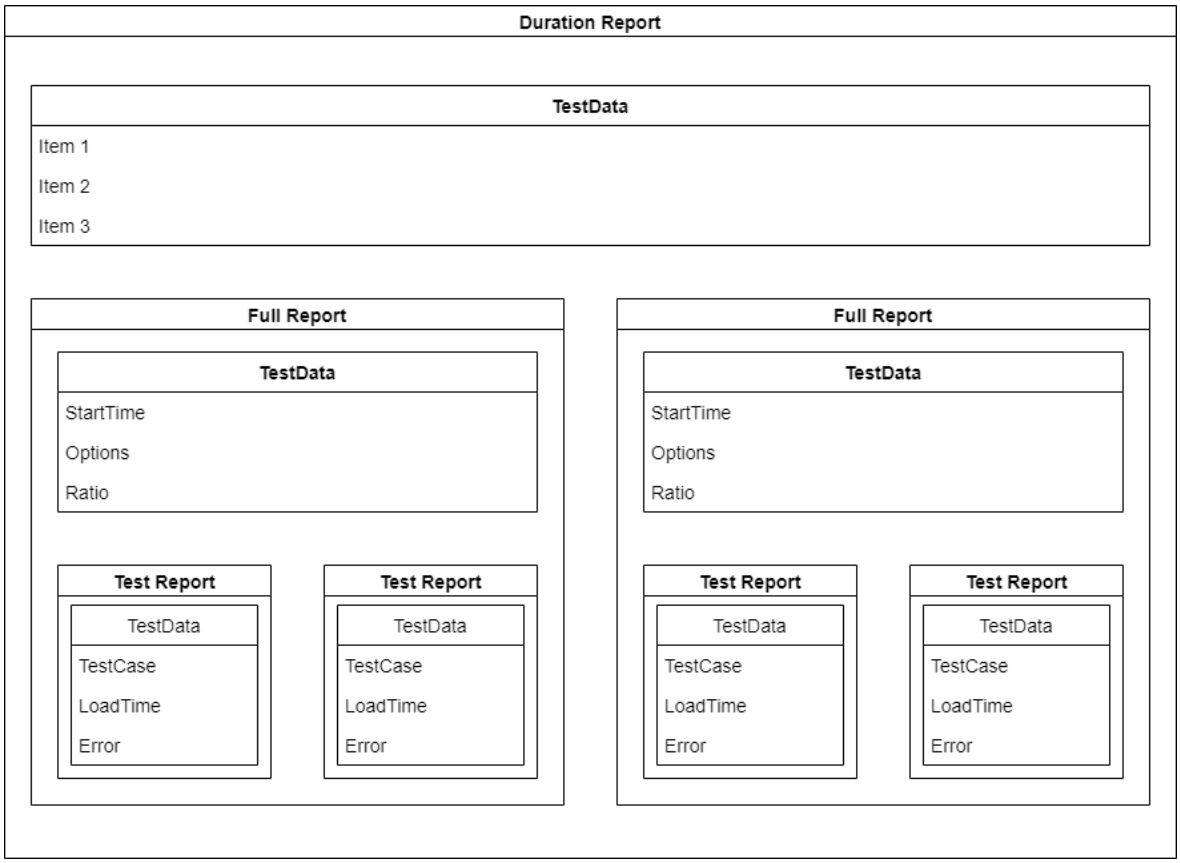

Abb. 3: Struktur der Testreports

Diese Abbildung zeigt die Hierarchie der erzeugten Testreports. Auf der obersten Ebene befindet sich der vollständige Container für alle generierten Reports. Dieser besitzt als Kerndaten alle Informationen über die aktuelle Testkonfiguration. Inhalt der Konfiguration ist die Anzahl an VN, der Zeitraum über denen die Nutzer das System beanspruchen und welche Arten von Szenarien in welchen Verhältnissen durchgeführt werden. Innerhalb des Containers werden die Teilreports verpackt. Sie enthalten Informationen über das jeweils durchgeführte Szenario. Dazu gehört die Start- und Endzeit des jeweiligen Testskripts sowie dessen Status. Diese Teilreports besitzen jeweils einen weiteren Container für die Messungen einzelner Nutzeraktionen innerhalb eines Testskripts. In der Regel sind dies Lade- und Antwortzeiten von Webseiten oder deren Komponenten. Bei einem Fehlerfall enthalten diese Reports noch zusätzlich genauere Informationen über was für eine Art Fehler vorgetreten ist.

Bei der Konzeption der Fehleranalyse-Funktionalitäten wird sich an die bisher eingesetzte XLT Lösung gehalten. XLT zeichnet stets auf, zu welcher Zeit was für ein Fehler während des Lasttests aufgetreten sind.

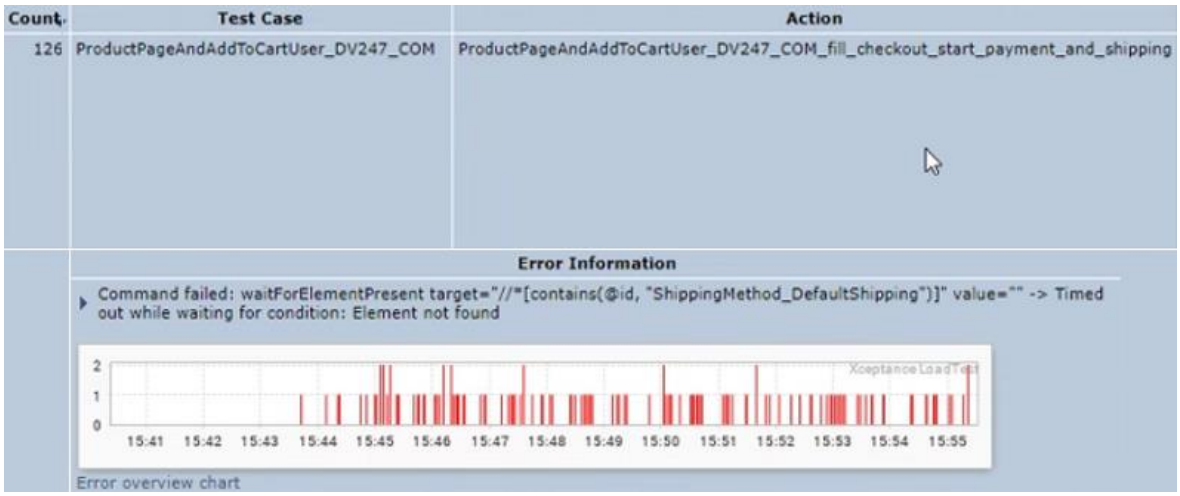

Abb. 4: Screenshot zur Fehlerausgabe bei XLT Tests

Auf dieser Abbildung wird ein Auszug aus einer XLT Fehleranalyse dargestellt. Sie beinhaltet Informationen über welche Aktion nicht ordnungsgemäß durchgeführt werden konnte. Die Spalte „Test Case“ dient hierbei zur Nachverfolgung in welcher Programmdatei der Fehler aufgetreten ist. Zusätzlich wird ein Graph über das Vorkommen der Fehler ausgegeben. Dieser hilft Trends oder Regelmäßigkeiten bei Fehlerauftreten zu erkennen. Diese Funktionalität in der eigenen PLT Lösung einzubauen, wird als Anforderung übernommen.

Bei der Aufnahme von Zeitmesswerten muss eine Methode gewählt werden, deren Ergebnisse über mehrere verschiedene Computer vergleichbar sind. Zur Leistungsüberprüfung von Webseiten kann dabei auf die Performance Schnittstelle von modernen Browsern zurückgegriffen werden.[11] Die darin vorgenommenen Zeitmessungen erfolgen über die High Resolution Time API. Diese Spezifikation definiert einige unterschiedliche Möglichkeiten zur Erstellung von Zeitstempeln welche auf eine kontextunabhängige, stabile, monotone auf Millisekunden genaue Uhr basieren.

[11] [Moz22]

Die Notwendigkeit einer stabilen monotonen Uhr stammt von dem Fakt ab, dass unabhängige Systemuhren Messungen verfälschen können und somit unbrauchbar werden. Beispielsweise wird bei der Navigation auf ein Dokument, Laden von Ressourcen oder Ausführung eines Skripts eine Uhr gewünscht, deren Messwerte jede Millisekunde monoton ansteigt. Der Vergleich von Zeitstempeln über verschiedene Kontexte ist essenziell, um beispielsweise Threads zu synchronisieren oder um einen bestimmten Zeitablauf genau darstellen zu können.[12]

Mit dem in diesem Kapitel aufgebauten Testkonzept können nun die Testskripte umgesetzt werden.

[12] [Wor22a]

3 Entwicklungsprozess der Tests

3.1 Durchführung an einer lokalen Maschine

Die erste Phase dieser Untersuchung beschäftigt sich damit herauszufinden, ob CJS sich überhaupt für das Erzeugen von Lasten eignet. Es wird zuerst mit der Einrichtung der Testumgebung begonnen. Um den Entwurfsprozess so aufwands- und kostengering wie möglich zu halten, wird hierbei eine VM aufgesetzt. Als Programm für die Verwaltung von VMs wird Oracle VirtualBox herbeigezogen.[13] Grund dafür ist, dass die meisten Softwareentwickler der DS bereits mit VirtualBox arbeiten und somit weniger Installationsaufwand für die zukünftige Verwendung des Testprogramms besteht. Der VM werden wenig Rechenressourcen zugeteilt, um eine Last leichter auslösen zu können, ohne das Hostsystem selbst zu überfordern. Das Hostsystem für diesen Versuch verfügt über acht Prozessorkerne und 32 Gigabyte Arbeitsspeicher. Es wird beschlossen, dem Testsystem zwei Prozessorkerne des Hosts zur Verfügung zu stellen, damit einer davon sich hauptsächlich auf die Serveranwendung konzentrieren kann, während der andere die Systemprozesse übernimmt. Die Größe des Arbeitsspeichers wird auf zwei Gigabyte beschränkt. Als nächstes wird ein auf Linux basierendes Betriebssystem heruntergeladen und auf dem Testsystem installiert. Grund dafür ist, dass es sich kostenfrei und schnell installieren lässt. Hierbei wird bestmöglich auf zusätzlich mitbereitgestellte Software verzichtet, um Verfälschungen von Testergebnissen später vorzubeugen. Nachdem die VM erfolgreich aufgesetzt und das Betriebssystem darauf installiert ist, wird der Server für die Lasttests eingerichtet. Um den Aufwand für zukünftige Vorhaben weiter gering zu halten, wird dabei erneut auf NodeJS zurückgegriffen. Es wird daher NodeJS auf dem Testsystem installiert und anschließend darüber eine API aufgesetzt, welche auf verschiedene HTTP Anfragen hört. Die API enthält je nach Anfrage Methoden mit unterschiedlichen Rechenbedarf. Zusätzlich zum Server sollte das System noch eine Möglichkeit zur Leistungsüberwachung bieten, um eine Last wirklich nachweisen zu können. Es wird hierbei der einfache auf den meisten Linux vorinstallierten Programm „top“ verwendet.

[13] [Ora22]

Zwar ist damit nun das Testsystem lauffähig, jedoch ist der darin enthaltene Server von außen nicht erreichbar. Um dieses Problem zu beseitigen, müssen die Ports des Hostsystems mit den Ports des Gastsystems verknüpft werden. Dies wird mit Hilfe der VM-Einstellungen in VirtualBox umgesetzt.

Um sicherzustellen, dass alle Voraussetzungen für das Schreiben von CJS Tests für das Testsystem erfüllt sind, wird versucht die darin enthaltene API über den Webbrowser des Hostsystems zu erreichen. Nachdem die Verbindung sichergestellt wird, kann mit dem Erzeugen der Testskripte begonnen werden. Sie werden zunächst möglichst einfach gehalten, indem sie ein Browserfenster öffnen und sich nur mit dem Testserver verbinden. Dabei wird eine Anfrage an die API gesendet, welche je nach Inhalt einen unterschiedlichen Rechenaufwand für eine Antwort fordert. Um die Serverlast zu messen, wird ein externes Programm verwendet, welches die CPU-Belastung auf dem Testsystem folgendermaßen darstellt:

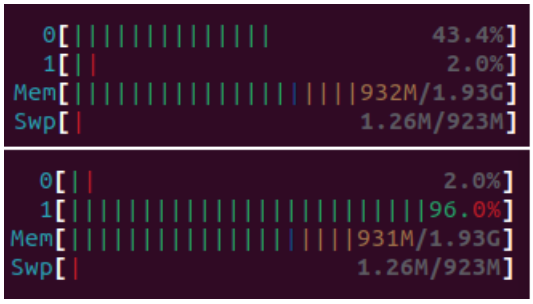

Abb. 5: Auslastung des Testsystems bei unterschiedlicher Anzahl von virtuellen Nutzern

Das Leistungsüberwachungsprogramm zeigt die Auslastung jedes CPU-Kerns der Linux Maschine prozentual an. Darüber hinaus wird auch die verwendete Menge des Arbeitsspeichers und der Auslagerungsdatei dargestellt. Wie bereits in diesem Kapitel erwähnt, besitzt die Maschine zwei Kerne, welche hier mit 0 und 1 repräsentiert werden. Der jeweils höhere belastete Kern bearbeitet dabei die Funktionen der API, während der andere hauptsächlich andere Systemprozesse verwaltet. Die Abbildung zeigt, dass die Serverlast bei einer höheren Anzahl von Anfragen ansteigt. Beim Senden von sehr rechenaufwändigen Anfragen kann es sogar vorkommen, dass der Browser keine Antwort rechtzeitig erhält und daher einen Fehler ausgibt. Da Fehler von Servern und Browsern zum Alltag gehören, kann dies genutzt werden, um eine saubere Fehlerbehandlung bei den Skripten einzurichten.

Das Messen von Antwortzeiten wird über die Nutzung der Performance Schnittstelle der Browser realisiert. [14] Aus diesem Versuch zeigt es sich deutlich, dass CJS alle Voraussetzungen erfüllt, einen echten Lasttest ausführen zu können.

Damit ist die Grundfunktionalität für das Ausüben von Lasten mit Hilfe von CJS sichergestellt. Da das Testsystem nur eine einfache API enthält, spiegelt sie kein realitätsgetreues Webumfeld wider. Die nächste Phase dieses Versuchs ist daher die Tests so zu erweitern, dass sie auf eine echte externe Shopumgebung ausgeführt werden können.

3.2 Durchführung auf einem externen System

In der DS laufen für Testzwecke Kopien von manchen Kundenshops auf betriebsinternen Servern. Diese eignen sich hervorragend für das Durchführen der in dieser Arbeit behandelten PLT, da sie weitgehend alle Funktionalitäten eines echten OS beinhalten. Zusätzlich werden bereits für diesen Versuch hilfreiche Systeme zur Überwachung der Netzwerk- und Systemleistung der Server eingesetzt. Es sind daher neben der Ausführung der Testskripte keine weiteren Anpassungen der neuen Testumgebung notwendig.

In den meisten Fällen werden Lasttests durch Unterprogramme gestartet, welche Skripte ausführen, die das Verhalten eines Webnutzers simulieren. Aktionen eines Nutzers auf einem OS können vielseitig sein und unterschiedliche Rechenaufwände benötigen. Es wird daher für eine klare Abgrenzung ein Testskript je mögliches Nutzervorhaben erstellt. Die jeweiligen Aktionen für jedes Testskript lassen sich dabei aus Tab. 5 entnehmen. Für jeden VN wird ein eigenes Unterprogramm ausgeführt. Datensätze wie beispielsweise Login-Daten oder Suchbegriffe können entweder vordefiniert oder zufällig generiert werden. [15]

[14] [Moz22]
[15] [Dra06], S. 10

Viele Testfälle bauen in einer Shopumgebung aufeinander auf. Zum Beispiel kann eine Bestellung nicht erfolgen, ohne dass ein Produkt zuvor in den Warenkorb gelegt wurde. Es wird daher ein Verhaltensmodell aufgesetzt, bei denen aufeinanderfolgende Nutzeraktionen zufällig über eine bestimmte Wahrscheinlichkeit ausgeführt werden. Eine Variation der Werte hierbei entspricht eher einem realen Umfeld. Es wird folgendes Benutzermodell für diesen Versuch verwendet:

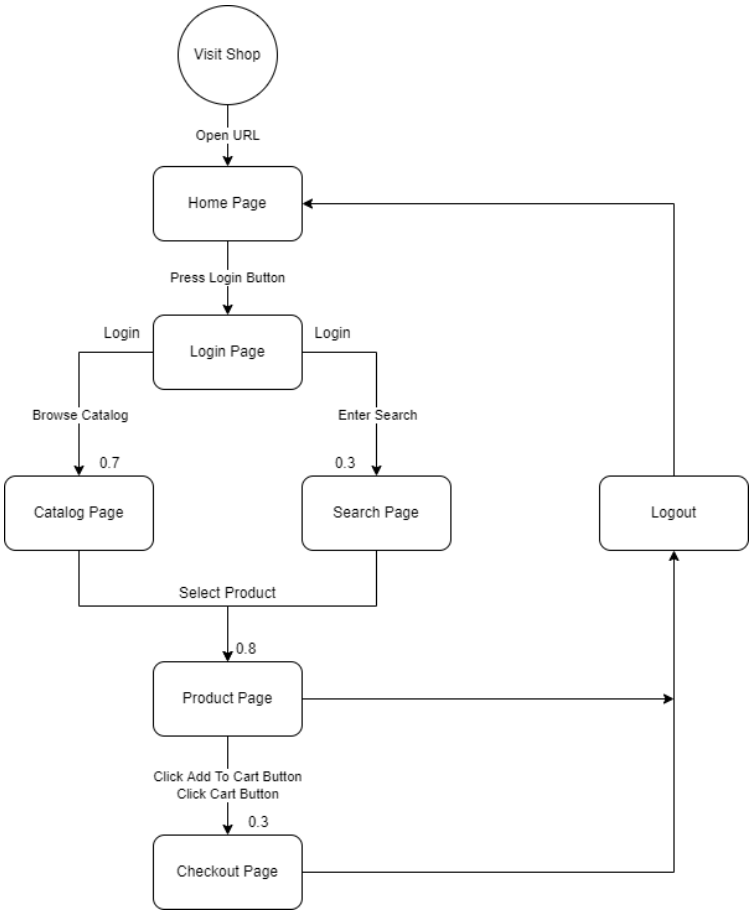

Abb. 6: Lasttestmodell für diesen Versuch

Das Modell spiegelt einen simplen Ablauf eines OS-Besuchers wider. Der VN öffnet zunächst die Homepage des OS und logt sich danach in sein Nutzerprofil ein. Hinterher wird zufällig entschieden, ob der Nutzer dann entweder eine Suchanfrage stellt oder nur durch den Katalog stöbert. Mit einer bestimmten Wahrscheinlichkeit wird dann ein zufälliges Produkt ausgewählt und dessen Detailseite aufgerufen. Sofern dann entschieden wird, ob das Produkt in den Warenkorb gelegt werden soll oder nicht, verlässt der Nutzer dann entweder die Seite oder fährt mit dem Einkauf fort. Zwischen den einzelnen Schritten können zufällig weitere Aktionen stattfinden wie beispielsweise Rücksprünge, Fehleingaben und weitere Funktionen der Webseite.

Anders als bei den Tests auf der VM stehen dem neuen Testserver nun deutlich höhere Rechenressourcen zur Verfügung. Außerdem sind auf einem realen Shop normalerweise mehr Nutzer zur selben Zeit aktiv, als in den Tests bisher simuliert. Es muss daher eine deutlich höhere Last erzeugt werden können. Da für einen Lasttest mehrere Browsersitzungen gleichzeitig laufen müssen, muss eine parallelisierte Ausführung der Testfälle gewährleistet werden. NodeJS ist zwar hauptsächlich eine Single-Threaded-Applikation[16], jedoch bietet es eine Möglichzeit zur Auslagerung von Funktionen auf sogenannte Worker Threads, welche eine Parallelisierung ermöglichen. Daraus folgt, dass jede Testdatei auf einem separaten Worker Thread ausgeführt werden muss, um eine Browsersitzung von einem unabhängigen Nutzer zu simulieren. CJS nutzt und erweitert die Funktionalitäten von Worker Threads. Es erlaubt sämtliche Testdateien in einer selbstbestimmten Anzahl von Threads auszuführen.

Bei der Ausführung von parallelen Tests fällt jedoch auf, dass bereits nach wenigen Browserinstanzen sich die Zeit für das Starten eines Browsers auf der lokalen Maschine stark erhöht. Das Problem dabei ist, dass CJS so konfiguriert ist, dass nach 30 Sekunden ein Test fehlschlägt wenn keine Instanz gestartet werden konnte. Es wird vermutet, dass das Öffnen eines Browserfensters einen höheren Rechenbedarf erfordert als das Durchführen von Testschritten. Nach genaueren Untersuchungen hat es sich herausgestellt, dass CJS alle Instanzen zur gleichen Zeit startet. Es ist unklar, ob es sich hierbei um ein Problem seitens des lokalen Computers oder NodeJS handelt, jedoch treten bei einem zeitverzögerten Start der Browserinstanzen diese Fehler nicht hervor. Aus diesem Grund muss der Quellcode von CJS so angepasst werden, dass die Funktion zum Starten einer Sitzung für alle Worker Threads überschrieben wird, damit sie nicht gleichzeitig sondern zeitverzögert gestartet werden. Wie im Abschnitt 2.3 erwähnt, dürfen die Nutzer nicht auf einem Schlag den Server erreichen, sondern müssen allmählig über einen bestimmten Zeitraum nach und nach eintreffen.

[16] [Ope22]

Es steht daher nahe, eine Konfigurationsdatei zu erstellen, welches eine Zeitspanne vorgeben kann, über die VN auf den Server gesendet werden. Programmdateien von CJS selbst werden dabei so angepasst, dass die Worker Threads die Instanzen nur über diese vordefinierte Zeitspanne starten. Die Konfigurationsdatei enthält weiterhin Informationen über welche Tests in welchen Anteilen durchgeführt werden sollen. Zusätzlich dazu wird eine Option implementiert, welche die maximale Wartezeit einer Aktion einstellt. Wird diese überschritten, dann zählt der Testversuch als fehlgeschlagen.

Nachweisen lässt sich die Last über Überwachungstools des Servers. Während eines Testlaufes müsste die CPU-Ausnutzung zuerst bis zu einem bestimmten Punkt ansteigen und ab diesen dann das Niveau beibehalten. Nach dem Ausprobieren einiger Konfigurationen konnte folgende Auslastung auf dem Testsystem erzielt werden:

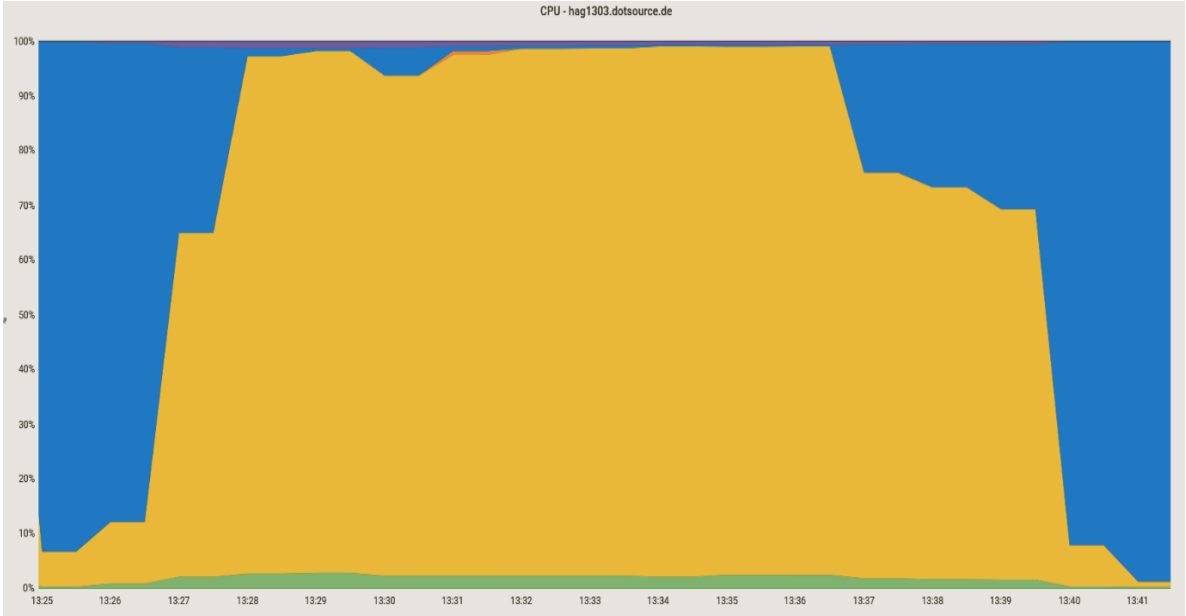

Abb. 7: Auslastung des Testsystems während eines Lasttest

Der Graph zeigt den gewünschten Verlauf eines PLT. Es wird über einen bestimmten Zeitraum eine Last ausgeübt und für eine gewisse Zeit beibehalten. Das Ausüben von Lasten mit Hilfe von CJS auf einer realitätsnahen Shop Umgebung ist somit nachweislich vollständig funktionsfähig. Der nächste Schritt ist nun Messungen durchzuführen und auszuwerten.

3.3 Implementierung von Reports

Bisher konnte die Funktionalität zur Ausübung einer Last sichergestellt werden. Jedoch sind Lasten ohne auswertbare Messungen nutzlos. Aus diesem Grund wird an dieser Stelle das Einrichten von Messungen behandelt. Alle Messwerte müssen sowohl Ladezeiten als auch Fehlerquoten behandeln. Im vorherigen Punkt wurde ein Skript erstellt, welches Browserinstanzen zeitverzögert parallel starten kann. Dieses muss nun erweitert werden, indem der Zustand der VN beobachtet und analysiert werden. Damit eine Kommunikation zwischen dem Skript und der Worker Threads bestehen kann, wird ein Hörer eingerichtet. Dieser erfasst gesendete Performancedaten der VN aus den Tests. Für jeden Test wird ein Report erstellt, welches den Startzeitpunkt und Daten zur Identifizierung festlegt. Wenn ein Test abgeschlossen ist, wird der Report um die Endzeit erweitert und erhält zusätzlich detaillierte Informationen über die gewünschte Ladezeit eines Abschnittes der Tests. Zunächst werden die Reports im JSON Format aufgezeichnet. Dies bietet die Möglichkeit, unterschiedliche Testabschnitte unabhängig voneinander zu testen. Zum Beispiel können nicht nur Ladezeiten von gesamten Seiten erfasst werden, sondern auch von Teilabschnitten. Der Verlauf lässt sich über folgendes Diagramm darstellen:

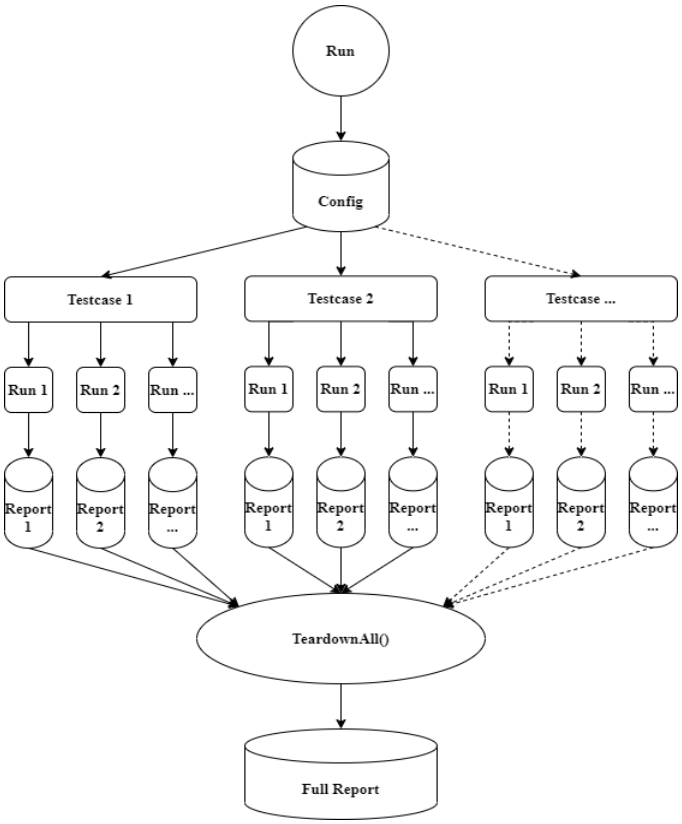

Abb. 8: Ablauf der Reportgeneration

Die VN werden zunächst über ein Skript gestartet. Dieses Skript entnimmt die Nutzeranzahl und deren Verhalten aus der Konfigurationsdatei. Danach werden die Nutzersitzungen gestartet und Messwerte aufgenommen. Jeder VN zeichnet seine Ladezeiten in Reports auf und sendet sie an das Startskript. Nachdem alle Sitzungen geschlossen sind und der Test abgeschlossen wird, fasst das Skript alle Messwerte zusammen und erstellt einen vollständigen Report nach Abb. 3.

XLT bietet eine Vielzahl an Möglichkeiten zur Darstellung der aufgenommenen Messwerte eines PLT. Um den Funktionsbedarf für die neue Lösung möglichst weit abzudecken, muss ein gleichwertig umfangreiches Tool herbeigenommen werden. Das in Abschnitt 2.1 erwähnte NodeJS Modul Vega zeichnet sich auf Grund der diversen Möglichkeiten zur Erstellung von Graphen als dafür sehr geeignet aus. Insbesondere werden die Graphen so eingerichtet, dass sie die in den Tests aufgenommenen Antwortzeiten der Webseiten über einen gewissen Zeitraum aufzeigen können.

Eine weitere Schwierigkeit liegt in der Auswertung von Serverfehlern. CJS beendet die Testausführung wenn ein Schritt fehlschlägt. Dies wirkt dem Lasttest entgegen, da zunehmend Clients geschlossen werden was wiederum die Gesamtlast reduziert. Aus diesem Grund wird ein CJS Plugin aktiviert, welches die Testausführung wiederholt, wenn ein Schritt fehlschlägt. Darüber hinaus wird das Startskript so erweitert, dass es auch fehlgeschlagene Tests abfangen und daraus einen Report erzeugen kann. Nach der sichergestellten Aufzeichnung von Messwerten kann mit der Auswertung dieser begonnen werden. Wichtig bei der Analyse ist das Erkennen von Abnormalitäten und negative Trends. Es muss deutlich identifiziert werden, ob die Tests reibungslos verlaufen oder ob Unregelmäßigkeiten in den Ladezeiten auftreten. Ebenfalls ist eine Zusammenfassung der Messdaten notwendig. Am besten eignen sich Graphen zur Auswertung der Testverläufe. Aus diesem Grund wird ein Skript angelegt, welches ein Diagramm aus den Messungen generieren kann. Als Zusätzliche Funktion werden Informationen zu Fehleraufkommen zusammengefasst und ausgegeben. Dies erfolgt über zwei Wege. Zum einen werden die Arten der Fehler zusammengefasst und eine Übersicht auf der Konsole des Anwenders angezeigt. Zum anderen wird ein Graph erstellt, welcher auf einer Zeitachse darstellt, wann ein Fehler aufgetreten ist. Dieser wird nach dem XLT Vorbild aus Abb. 4 nachgebildet.

Zum Schluss sollte sichergestellt werden, dass dieses Programm für ähnliche zukünftige Testversuche herbeigezogen werden kann. Dafür wird der Quellcode für die Mitarbeiter der DS öffentlich bereitgestellt. Zusätzlich wird eine ausführliche Dokumentation zur Installation und Funktionsweise dieses Tools angelegt. Neben einem Teil der in dieser Arbeit vorhandenen Inhalte werden folgende Aspekte aufgeführt:

- Ein Link für den Download der Projektdateien
- Installationsanleitung
- Bekannte Bugs
- Verwendete Technologien
- Beschreibung der Projektdateien
- Beispiele zum Aufsetzen eigener Tests

Darüber hinaus werden Beispiele zu Resultaten aufgezeigt. Die Ergebnisse aus dieser Arbeit werden aus dem nächsten Kapitel entnommen.

4 Evaluation der Testumgebung

4.1 Ergebnisse

In der ersten Phase dieses Versuchs wurde untersucht, ob CJS sich überhaupt für das Durchführen von PLT eignet. Da CJS ein komplettes Browserfenster steuern kann, ist es vollständig in der Lage sich durch gewöhnliche Webseiten durchzunavigieren. Realistische Testszenarien in gewöhnlichen Verhältnissen lassen sich daher frei ohne Probleme erstellen. Demzufolge lässt sich bestätigen, dass CJS auch das Verhalten von Webseiten bei hoher Auslastung evaluieren kann.

Im Zweiten Teil wurden die in Tab. 5 aufgelisteten Testskripte erstellt und auf einem Test Shopsystem durchgeführt. Des Weiteren wurde dabei die Tauglichkeit für die Aufnahme von Messwerten überprüft. Bei der Durchführung von PLT konnten auf dem Testsystem folgende Metriken aufgezeichnet werden:

Nutzerzahl	**Antwortzeit Ø in MS**	**Auslastung in %**	**Fehlerrate in %**
1	235	10	0.00
25	241	35	0.00
50	302	85	0.00
75	539	100	1.79
100	1491	125*	9.42
125	3286	150*	17.91

Tab. 6: Antwortzeiten bei unterschiedlicher Nutzeranzahl auf dem Testsystem

Die Messungen hierbei wurden lediglich mit Hilfe der Performance Timing API aufgenommen. Anhand dieser Zahlen lässt sich erkennen, dass CJS genug Möglichkeiten bietet, unterschiedliche Messungen hinsichtlich Ladezeiten aufzuzeichnen. Die Werte in dieser Tabelle entsprechen den Erwartungen, dass die Antwortzeiten und Fehlerraten ab 100% Auslastung signifikant ansteigen. Auffällig ist hierbei, dass die Serverlast nicht linear mit der Nutzerzahl ansteigt. Es wird vermutet, dass Optimierungsvorgänge wie beispielsweise Caching die Testergebnisse beeinflussen. Aus den in Tab. 4 entnommenen Grenzwerten lässt sich in diesem Beispiel sagen, dass das System unter maximal 75 parallelen Nutzern akzeptabel läuft.

Das Reporting Tool kann die genommen Messwerte als Graph darstellen. Es wird dabei für jeden unterschiedlichen Testfall jeweils eine eigene Grafik angelegt. Das folgende Diagramm zeigt beispielhaft die aufgenommenen Messwerte für den Test eines Checkout-Prozesses.

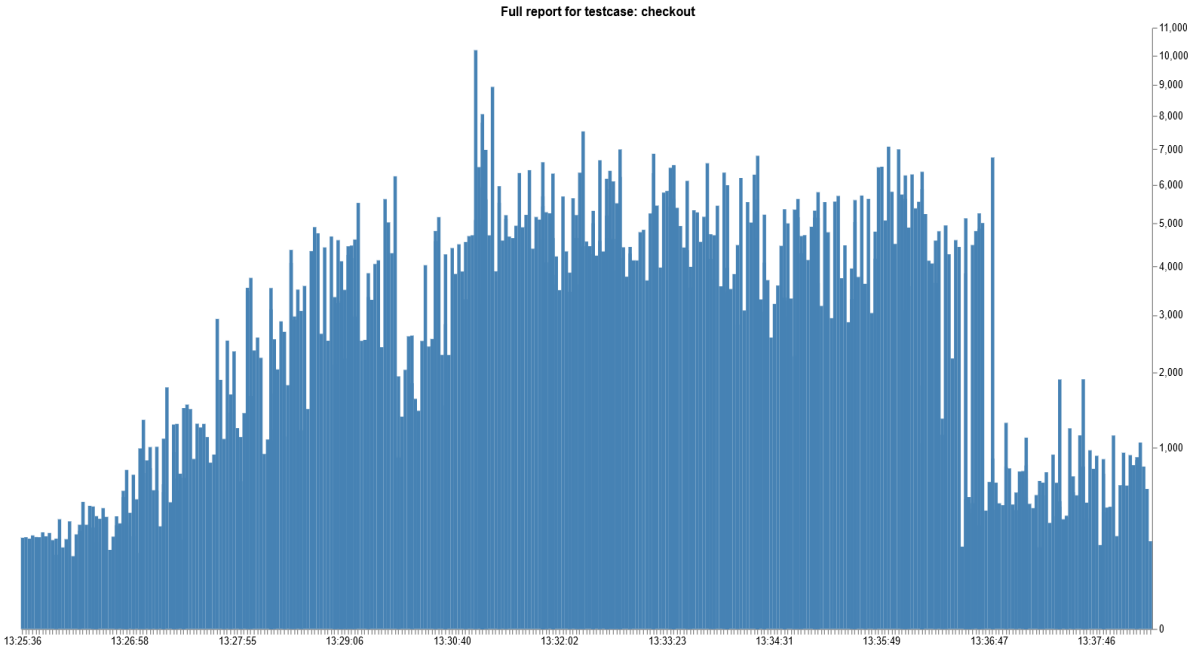

Abb. 9: Automatisch erzeugter Graph über die Ladezeiten eines Checkout-Prozesses

Auf der X-Achse wird der Startzeitpunkt der Messungen dargestellt. Die Y-Achse gibt den jeweiligen Messwert in Millisekunden an. Die Messwerte entsprechen dabei völlig den Erwartungen. Zu Beginn befinden sich wenige Nutzer auf dem System und die Ladezeiten sind gering. Im späteren Verlauf steigen mit zunehmenden Nutzerzahlen die Ladezeiten stark an. Die hohen Ladezeiten bestehen so lange, bis die Clients ihre Tests abschließen indem entweder alle Versuche erfolgreich durchlaufen sind oder sie durch Fehler beendet wurden.

Da Nichtbearbeitung oder Zeitüberschreitungen von Anfragen zu Fehlern führen können, müssen diese mitaufgezeichnet werden. Die Anwendung wurde so entwickelt, dass sie die von NodeJS erkannten Fehler direkt in die Reports mitschreiben. Dies ermöglicht eine detaillierte Auskunft über Fehlervorkommen zu erhalten.

4.2 Auswertung der Anwendung

Ein wichtiger Aspekt für diese Arbeit ist zu überprüfen, inwiefern die gewünschten PLT Anforderungen über CJS umgesetzt werden kann. Als Ziel wurde die Umsetzung von möglichst vielen Funktionalitäten aus dem bisher genutzten XLT Programm festgelegt.

Funktion	**Xceptance Load Test**	**CodeceptJS**
Ausmessen von Antwortzeiten	Ja	Ja
Ausmessen von Ressourcenauslastung des Servers	Ja	Nein
Erkennen von Fehlern	Ja	Ja
Erzeugen von Reports	Ja	Möglich
Erzeugen von Graphen	Ja	Möglich

Tab. 7: Funktionalitätsvergleich zwischen Xceptance Load Test und CodeceptJS

Der Funktionsumfang ist bei der CJS Umsetzung bislang geringer als in der aktuell verwendeten Lösung. Trotz noch mangelnder Funktionalitäten bietet die CJS Lösung jedoch den Vorteil sehr leicht aufgesetzt zu werden. Der Entwickler kann über wenige Konsolenbefehle das gesamte System installieren und sofort anfangen, eigene Tests für sein gewünschtes System zu erstellen. Auch Schulungen von neuen Entwicklern verläuft deutlich einfacher, da CJS eine leicht verständliche verhaltungsgesteuerte Syntax verwendet. Zusätzlich lassen sich Dokumentationen über NodeJS und dessen Module leicht online nachschlagen.

Die Fehleranalyse ist essenziell für die Durchführung eines PLT. Eine Hilfestellung ist es dabei zu sehen, was dem Nutzer beim Auftreten eines Fehlers angezeigt wird. CJS beinhaltet ein Plugin, mit dem ein Abbild der aktuellen Browsersitzung aufgenommen werden kann. Dieses Plugin ermöglicht den genauen Ablauf eines Testlaufs bis zu einem Fehlerauftreten als Video aufzuzeichnen. Im zuletzt durchgeführten Testlauf wurden folgende zwei Bildschirmaufnahmen bei Fehlern erzeugt:

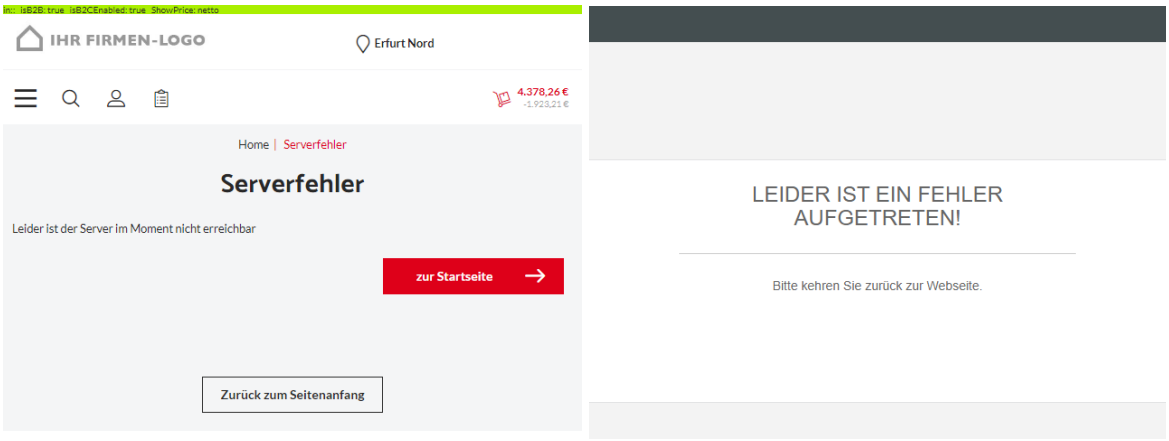

Abb. 10: Fehlerausgaben des Testsystems

Auf dieser Abbildung sind zwei Screenshots zu sehen, welche von CJS bei einem unerwarteten Fehler aufgenommen wurden. Sie zeigen zwei verschiedene Fehlermeldungen des Servers an. Die Meldungen sind je nach Fehlertyp unterschiedlich. Bei der ersten handelt es sich um ein Problem bei der Datenverarbeitung der Nutzeranfragen. Diese kann aus vielen Gründen vorkommen. Die zweite ist jedoch die Unfähigkeit des Servers die jeweilige Anfrage ordnungsgemäß zu beantworten. Dieser Fehler taucht ausschließlich nur bei Abweisung von Anfragen aufgrund von hohen Belastungen auf. Es zeigt sich somit, dass CJS Fehler sowohl erkennen als auch analysieren kann, was für einen Lasttest äußerst wichtig ist. Der Entwickler kann somit genauer nachvollziehen an welcher Stelle oder Funktion genau das Testskript versagt hat. Anhand der nebenbei eigenständig erstellten Reports lassen sich zusätzliche Informationen wie die Art des Fehlers oder der genaue Zeitpunkt des Auftretens aufzeichnen.

Darüber hinaus bietet die Anwendung Möglichkeiten zur Generation von Graphen. Die Anzahl an Fehlervorkommen lässt sich zum Beispiel über ein Histogramm ausgeben:

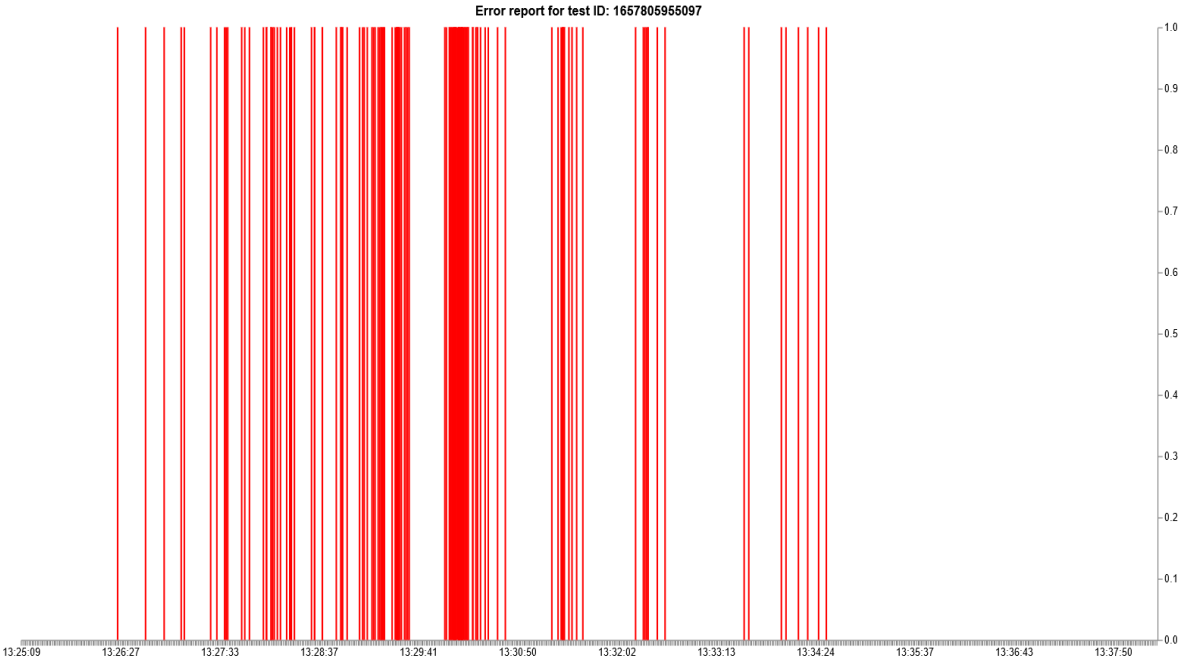

Abb. 11: Verteilung von Fehleraufkommen bei einem Lasttest

Es handelt sich hierbei um ein Balkendiagramm, welches auf einer Zeitachse eine Markierung setzt, wenn zu dieser Uhrzeit ein Fehler während des Tests aufkam. Verglichen mit Abb. 4 ähnelt der Aufbau und die Gestaltung des Graphen sehr dem von XLT. In diesem Beispiel wurden über 15 Minuten 175 VN auf dem Server simuliert. Jede rote Linie hierbei bedeutet, dass ein Fehler bei der Durchführung von Tests aufgetreten ist und die Sitzung nicht fortgesetzt werden konnte. Es ist deutlich zu sehen, dass sich die Linien über die ersten fünf Minuten verdichten. Ab diesen Zeitpunkt ist die Anzahl an Fehlervorkommen am höchsten, bis sie dann im späteren Verlauf nach und nach abschwächt. Dies Entspricht den Erwartungen, da höhere Lasten auch zu mehr Fehler führen. Daher steigen die Raten bis zur maximalen Anzahl an Clients an und fallen hinterher ab, nachdem die ersten Sessions ihre Tests durchlaufen oder auf Grund von zu häufigen Fehler abbrechen. Diese Werte korrelieren mit den Messungen der Antwortzeiten aus Abb. 7. Ab ca. 13.30 Uhr wurden die höchsten Antwortzeiten gemessen und zur selben Zeit war die Fehlerrate außerordentlich hoch, was sich an dem dicken roten Streifen in Abb. 11 wiedererkennen lässt.

Es existieren verschiedene Kategorien von Fehlern. Sie treten an unterschiedlichen Stellen des jeweiligen Systems auf. Die für die PLT relevanten Fehler sind hierbei zum einen von CJS ermittelte Fehler. Sie zeigen Probleme bei der Darstellung von Inhalten einer Seite. Zum anderen können Fehler auf der Seite von NodeJS entstehen. Sie werden in dieser Untersuchung hauptsächlich durch die nicht vorhergesehene Parallelisierung hervorgerufen. Die dritte Kategorie stammt hierbei von dem Server. Dieser kann entweder absichtlich Anfragen ablehnen oder diese durch unbehandelte Fehler nicht ordnungsgemäß bearbeiten. In allen Fällen können die Fehler analysiert und ausgewertet werden, indem die jeweilige Ausgabe in der Konsole betrachtet wird. Beim soeben beschriebenen Testlauf wurden folgende Fehler erkannt:

Art	**Beschreibung**	**Vorkommen**
Error	Dies sind von CJS erkannte Fehler bei der Testdurchführung. Sie treten auf, wenn auf der Webseite eine Aktion nicht durchgeführt werden konnte, da die Voraussetzungen nicht erfüllt wurden. Beispielsweise könnte dies durch das gewünschte Anklicken einer Schaltfläche passieren, wenn diese noch nicht geladen wurde.	70
elementHandle.type	Wenn eine Aktion durchgeführt wird, ohne eine Reaktion zu erhalten, taucht dieser Fehler auf. Dies kann zum Beispiel durch Verzögerungen bei der Skriptausführung des Servers entstehen	1
ConnectionError	Die Verbindung wurde vom Server abgelehnt. Dies kann geschehen, wenn dieser zu viele Anfragen auf einen Schlag erhält und sie nicht mehr bearbeiten kann.	1

TypeError	Bislang ist dieser Fehler unbekannt. Aus verschiedenen Logdateien wird angenommen, dass es sich hierbei um Fehler durch die Parallelisierung von Sitzungen handelt.	46
browserType.launch	Bei diesem Fehler konnte keine Browsersitzung gestartet werden. Hierbei handelt es sich um einen Fehler auf der Seite des Lasterzeugers. Im Normalfall bedeutet diese Meldung dass für den Rechner nicht mehr genug Ressourcen zur Verfügung stehen, um weitere Browserfenster zu öffnen. Diese Fehler haben keinen Bezug zum Testsystem und sind daher für die PLT irrelevant.	27

Tab. 8: Aufkommen und Beschreibung von Fehlern bei der Durchführung von Tests

In Kombination mit den Messwerten können diese Fehlermeldungen dem Entwickler einen genauen Einblick über die Schwachstellen des Systems geben.

Dieses Kapitel konnte somit aufzeigen, dass fast alle Funktionalitäten eines PLT mit Hilfe von CJS umgesetzt werden können, und dass dies einige Vorteile gegenüber XLT bietet.

5 Rück- und Ausblick des Versuchsverlaufs

5.1 Bewertung der Untersuchung

Anfangs waren die Erwartungen gering, da die Dokumentation von CJS nie PLT-Tauglichkeit erwähnt hat. Es wurde vermutet, dass die Kombination aus NodeJS und zahlreichen Chromium Browsern zu viel Ballast der Systemressourcen und eine hohe Fehleranfälligkeit mit sich bringen würde. Dies würde für eine zu hohe Anforderung von Rechenleistung führen und damit bei einer höheren Skalierung zu unwirtschaftlich werden. Die größte Schwierigkeit bestand Anfangs in der Parallelisierung der Browsersitzungen. Zunächst wurde nur die Fehlermeldung, dass die Browser nach 30 Sekunden nicht starten konnten, angezeigt. Die Vorgabe für diese Zeitüberschreitung zu erhöhen hat zwar mehr Sitzungen ermöglicht, jedoch nicht das zugrunde liegende Problem behoben. Erst nach der Implementierung des zeitverzögernden Starts der VN wurde eine Lösung gefunden.

Darüber hinaus wurden die Ergebnisse durch das bestehende Shop Testsystem bedingt verfälscht. Zum Beispiel wurde die Last anfangs auf zwei Systeme auf Grund eines Lastverteilers ausgeübt. Statt einer erwarteten 100% Auslastung war das Testsystem nur geringfügig beeinflusst:

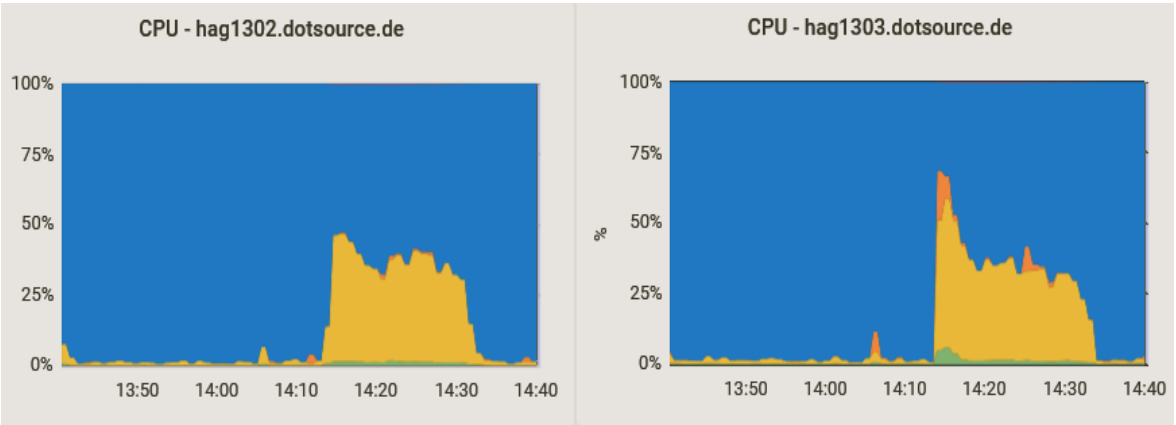

Abb. 12: Nachweis eines Lastverteilers auf dem Testsystem

Obwohl dies zunächst problematisch erscheint, stellt es sich im Nachgang als nützlich heraus. Echte Shopsysteme verwenden ebenfalls Lastverteiler. Somit zeigt der Versuch mehr, dass sich die Anwendung wirklich auf reale Systeme Ausführen lässt.

Das Entwickeln der Testskripte verlief weitgehend reibungslos. CJS ist sehr leicht verständlich und die Verwendung ist so intuitiv wie die zu testenden Webseiten selbst. Für anfängliche Tests war das Gerät zur Lasterzeugung nicht Rechenstark genug. Ab einer bestimmten Last des Erzeugers selbst wird dieser so stark verlangsamt, dass es wenigere Nutzeraktionen auf dem Testsystem ausführen und somit die Last nicht weiter erhöhen kann. Eine weitere Hürde bestand in der Fehlerbehandlung von CJS. Wenn während eines Tests ein Fehler auftrat, wie beispielsweise das nicht rechtzeitige Laden einer zu betätigenden Schaltfläche, bricht CJS den Testlauf ab. Dies führt nach und nach zum Verlust von Browsersitzungen und somit auch zu einer Verringerung der gewünschten Last.

5.2 Zukünftiger Verlauf und Verbesserungsmöglichkeiten

Inwieweit die Anwendung in Zukunft verwendet wird, ist im Moment noch unklar. Obwohl es eindeutig wesentlich einfacher in der Bedienung ist als XLT, müssen viele Funktionalitäten noch manuell entwickelt und verbessert werden. Es ist definitiv möglich ein einfacher zu bedienendes Programm zu erstellen, jedoch wäre der Aufwand der initialen Entwicklung sehr hoch. Jedoch hat es das Potential in Zukunft viel Einschulungs- and Nutzungszeit für die Softwareentwickler einzusparen. Bei einer höheren Investition kann daraus auch ein kommerzielles Produkt entstehen.

Das Starten und Aufrechterhalten der Browserinstanzen kostet dem System einige Rechenressourcen. Die Menge an offenen Sitzungen ist daher von den Systemressourcen her begrenzt. Je nach dem über welche Kapazitäten der zu testende Server verfügt, kann es unmöglich sein, mit nur einer Maschine diesen zu belasten, insbesondere wenn Lastverteiler auf dem Testserver zum Einsatz kommen. Für die Anwendung auf größere Systeme müssen daher mehrere rechenstärkere Maschinen herbeigezogen werden. Es würde sich daher empfehlen, ein Skript zu erstellen, welches nach einer Konfiguration beliebig viele Maschinen beispielsweise über einen Cloud-Computing Anbieter mietet und die gewünschten Testskripte auf diesen ausführt. Im Gegensatz zu aktuellen Möglichkeiten mit CJS ließe sich so ein Stresstest ausführen, indem viele VN zum selben Zeitpunkt anstatt allmählich über einen längeren Zeitraum am Testsystem eintreffen. Das Reporting Tool ist so ausgelegt, dass es alle zu einem Test dazugehörigen Reports zusammenfassen und auswerten kann. Die Verteilten Maschinen brauchen somit nur ihre Reports an den Hosts senden.

Die Reports selbst sind noch sehr verbesserungsbedürftig. Zum einen tauchen auf jedem Test starke Ausreißerwerte auf. Insbesondere die ersten Messwerte erscheinen unverhältnismäßig hoch und schießen weit über den Durchschnitt heraus. Eine Erklärung dafür konnte bisher noch nicht gefunden werden. Es könnte sich dabei um einen Nebeneffekt der nicht vorgesehenen Nutzung von parallelen Browsersessions handeln. Eine mögliche Lösung wäre das Entfernen aller Werte im obersten Perzentil. Zusätzlich wäre es sinnvoll durchschnittliche Ladezeiten per Testfall wie unter den in Tab. 6 aufgelisteten Rahmenbedingungen aufzuzeichnen. Dies Hilft dabei zu erkennen, welche Vorgänge im OS von einer Last am meisten betroffen werden. Somit können Schwachstellen im System leichter ermittelt werden. Das Format, in dem die Messwerte festgehalten werden, ist aktuell JSON. Es wurde sich zunächst für dieses Format entschlossen, da es den Vorteil mit sich bringt, reibungslos ungewöhnliche Werte wie beispielsweise Fehler einzufügen. Nachteil davon jedoch ist, dass auf alle Werte auch deren Beschriftung inbegriffen sind. Dies kann bei langen Tests mit hohen Nutzerzahlen zu hohen Dateigrößen der Reports führen. Eine Alternative könnte dazu das CSV Format sein, da es aufgrund fehlender Wertebezeichnungen und minimalen Syntax vergleichsweise weniger Speicher benötigt und ebenso leicht ausgewertet werden kann.

Auch die Graphen sind verbesserungswürdig. Aktuell werden sie als Balkendiagramme dargestellt und jeder Messwert erzeugt eine dünne Linie. Für auf Zeit basierende Diagramme sind jedoch Liniendiagramme besser geeignet. Darüber hinaus mangeln den Grafiken noch detailliertere Beschriftungen und Legenden. Zum Beispiel zeigt der Graph mit den Fehlerraten aus Abb. 11 auf der Y-Achse Dezimalstellen zwischen 0 und 1 an. Da Fehler entweder nur vollständig Auftreten können oder nicht, ist die Achsenbeschriftung ungeeignet. Dies lässt sich noch erweitern, indem Schaltflächen eingebaut werden, welche die Achsen in bestimmte Werte eingrenzen. Die Analyse von Fehlerauftreten wurde bislang nur sehr rudimentär implementiert. Softwareentwickler können Bugs schneller beheben, wenn sie detaillierte Informationen über die Fehlervorkommen erhalten. Aktuell nimmt die Anwendung zwar auf, zu welchem Zeitpunkt welcher Fehler aufgetreten ist, jedoch werden keine Zusammenhänge oder Trends ermittelt. Darüber hinaus müssen für PLT irrelevante Fehler beseitigt werden (siehe Tab. 8).

Der größte Vorteil den CJS gegenüber XLT bietet, ist die deutlich einfachere Bedienung. Es besteht die Möglichkeit einen solchen Test täglich oder nach sämtlicher Codeänderung durchzuführen. So ließen sich Trends erkennen. Zusätzlich könnte eine tägliche Ausführung Probleme rechtzeitig erkennen, indem ein Alarm ausgegeben wird, wenn die Messwerte einen bestimmten Grenzwert überschreiben. Dies wäre besonders empfehlenswert bei Softwareprojekten, in denen Quellcode kontinuierlich bearbeitet wird.

Ein wichtiger Aspekt von PLT ist das Aufnehmen vieler verschiedener Messwerte, um ein möglichst genaues Bild von den Auswirkungen von Lasten zu erhalten. Bisher wurden dabei über die Browser Performance Timing API nur Antwortzeiten bis zum Erhalt einer vollständigen Webseite aufgenommen. Webseiten selbst können auch Daten während ihrer Verwendung aus anderen Quellen abfragen. Antwortzeiten zu solchen Ereignissen werden zurzeit nicht aufgenommen und berücksichtigt. Beispielsweise können dynamische Seiteninhalte ebenso durch eine Last des Webservers verlangsamt werden.

Das Erzeugen von Reports erfolgt bisher manuell. Es existieren Zahlreiche Frameworks und Tools für das Erstellen von Test Reports. Im Laufe dieser Arbeit wurden verschiedene von CJS empfohlene Testframeworks ausprobiert. Jedoch beziehen sich alle nur auf das Bestehen oder Fehlschlagen der Tests. Es wird meist nur erfasst, um welche Uhrzeit welche Aktion durchgeführt wird oder ob diese nicht möglich ist. Ladezeiten selbst werden zwar teilweise erfasst, jedoch in keinem für PLT sinnvollen Format aufgezeichnet und ausgegeben. Aus diesem Kontext empfiehlt es sich eine weitere Untersuchung durchzuführen, welche sich mit der Integration von den Messwerten der Ladezeiten in zumindest einem oder mehrere gängige Testframeworks auseinandersetzt. Der Vorteil bei der Verwendung eines externen Tools zur Auswertung von Reports wäre ein Ersparnis der Entwicklungszeit. Zusätzlich müsse man sich nicht mit Bugfixes und Testing dieser Programme beschäftigen. Außerdem wäre ein externes Werkzeug wahrscheinlich deutlich reicher an Funktionalitäten als eine selbst geschriebene Lösung.

Diese Arbeit hat sich hauptsächlich auf die Tools CJS und XLT konzentriert. Es existieren jedoch zahlreiche andere Frameworks für das Erstellen von PLT. Daher könnte eine weitere Untersuchung an diese Arbeit anknüpfen, welche sich mit der gleichen Umsetzung über ein anderes Tool beschäftigt. Besser wäre noch ein Vergleich verschiedener Testframeworks mit der aktuell erstellten Lösung

6 Fazit

Studien von Google haben belegt, dass steigende Ladezeiten von Webseiten die Absprungraten von Nutzern negativ beeinflussen. Für Unternehmen, welche ihr Hauptgeschäft online betreiben, ist diese Information sehr relevant. Denn höhere Absprungraten können geringere Kundenzufriedenheit und auch sinkende Umsatzraten zur Folge haben. Einer der zahlreichen Gründe, warum die Ladezeit einer Webseite höher sein kann, ist eine Überlastung des Webservers. Wenn dessen Prozessor oder Arbeitsspeicher übermäßig beansprucht werden, dann müssen Prozesse warten, bis Rechenressourcen wieder freigelegt werden. Jeder Nutzer beansprucht Rechenressourcen des Systems. Daraus folgt, dass höhere Nutzerzahlen zu stärkerer Auslastung des Webservers führen. Zusätzlich können eine hohe Last und längere Ladezeiten Bugs auslösen, welche verschiedene ungewünschte Effekte mit sich bringen. Im schlimmsten Fall könnte eine Serverapplikation komplett abstürzen und unerreichbar werden. Insbesondere bei OS könnte dies schwere finanzielle Folgen haben. Um so einen Ernstfall vorzubeugen, werden Tests entwickelt, welche die Systemperformance unter Last prüfen. Bei PLT werden eine hohe Anzahl von Nutzern auf einem System simuliert, welche verschiedene realitätsnahe Besucheraktionen durchführen. Dabei wird die Menge an VN stetig erhöht, bis ein bestimmter Grenzwert erreicht wird. Während dieser Tests werden die Ladezeiten der besuchten Webseiten aufgezeichnet.

Bisher verwendet die DS das Tool XLT, um die Performance der OS ihrer Klienten zu prüfen. Obwohl XLT ein umfassendes Spektrum an Möglichkeiten zur Messung von Webseiten- und Serverperformance bietet, ist es sehr aufwändig aufzusetzen. Dies verursacht höhere Zeitaufwände und Kosten beim Einrichten von XLT sowie bei der Schulung von Mitarbeitern in das System. Aus diesem Grund wurde beschlossen, alternative PLT Werkzeuge einzuführen. Dabei wurde das Tool CJS in Erwägung gezogen, da es bereits erfolgreich für andere Tests innerhalb verschiedener DS Projekte verwendet wird. CJS ist jedoch nicht für Lasttests vorgesehen und eine solche zweckentfremdende Verwendung ist bislang undokumentiert. Das Ziel dieser Arbeit war es daher zu untersuchen, ob und inwiefern sich PLT mit Hilfe von CJS umsetzen lassen.

Um dieses Ziel zu erreichen wurde ein Versuch in zwei Stufen unterteilt. Zuerst wurde geprüft, ob sich CJS überhaupt für diese Aufgabe eignet. Dazu musste ermittelt werden, ob sich ausreichend viele parallele Nutzersitzungen realisieren lassen. Dafür wurde auf einer VM ein Testserver aufgesetzt, auf denen über unterschiedliche Anfragen bestimmte Lasten ausgeübt werden können. Anhand des Servers wurden die Testskripte für CJS so aufgesetzt, dass viele parallele Nutzer angelegt und somit eine gewisse Last auf dem Testserver erzeugt werden können. Nachdem CJS sich für diese Aufgabe als tauglich erwiesen hat, wurde die zweite Phase dieser Arbeit eingeleitet. In dieser wurde dann Anhand eines realitätsnahen Testsystems eines OS ein vollständiger Lasttest durchgeführt. Dazu mussten zunächst verschiedene Testszenarien definiert werden. Zusätzlich wurde bestimmt, an welchem Testschritt genau welcher Messwert aufgenommen wird. Diese Szenarien werden anhand von Testskripten durchgeführt. Diese Skripte erstellen eine bestimmte Anzahl von VN und simulieren das Verhalten von echten Besuchern einer Webseite. Es wurde für die VN das Lasttestmodell aus Abb. 6 verwendet und die Skripte an die jeweiligen Aktionen ausgerichtet. Zwar konnten die Skripte ordnungsgemäß durchgeführt werden, jedoch kamen bei der Parallelisierung von Sitzungen Probleme auf. CJS überlastet beim gleichzeitigen Start von Browserfenstern den Lasterzeuger und die Durchführung schlug fehl. Dies konnte mit Anpassungen des CJS Quellcodes behoben werden.

Nach der Problembehandlung konnte ein PLT auf dem Testsystem vollständig durchgeführt werden. Es konnte Anhand von Systemüberwachungswerkzeugen gezeigt werden, dass über die Skripte eine gewünschte Überlast erzeugt werden konnte. Die Auswirkungen der Lasten lassen sich problemlos über einen gängigen modernen Webbrowser messen. Die Messungen lassen sich ebenfalls durch die klare Struktur der einzelnen Reports reibungslos vornehmen. Insbesondere wurde der Fokus auf die Qualität der Reports gelegt. Jede Messung lässt sich eindeutig auf eine bestimmte Aktion zu einem bestimmten Zeitpunkt zurückführen. Des Weiteren werden verschiedene Arten von Ladezeiten gemessen und Durchschnitte berechnet. Aus diesen Werten können Graphen zu jedem Testszenario generiert werden. Zusätzlich fallen die Auswertungen zu den Fehlern detailliert aus. Die Reports zeichnen auf, an welcher Stelle eines Testskripts der VN nicht mehr ordnungsgemäß seine Tätigkeiten vollrichten konnte. Außerdem wird die genaue NodeJS Fehlerausgabe mit in die Reports aufgenommen. Anhand dieser Daten können die Fehleraufkommen gruppiert und deren Verlauf in Graphen dargestellt werden.

Das schließende Ergebnis ist, dass es durchaus möglich ist, PLT mit CJS zu schreiben. Der Vorteil wäre dabei in der sehr einfachen Bedienung im Vergleich zur bisher verwendeten XLT Lösung. Da diese Arbeit belegen konnte, dass PLT mit CJS umsetzbar sind, muss nun von Seiten der DS entschieden werden, ob sich eine Weiterentwicklung lohnt. Zwar ist das Werkzeug bereits funktionsfähig, mangelt aber noch an vielen Möglichkeiten hinsichtlich der Auswertung von Messergebnissen. Wenn die eigene PLT Lösung sich als profitabel herausstellt, wird das aktuell verwendete XLT Tool durch die neue Anwendung abgelöst. Aufgrund der einfachen Bedienung würden zukünftige Qualitätskontrollen schneller und teilweise automatisiert erfolgen. Insgesamt lässt sich damit das Ergebnis der Arbeit als erfolgreich betrachten und konnte einen tieferen Einblick in die Umsetzung eines Lasttest mit Hilfe von browsersteuernden Tools gewähren.

Literaturverzeichnis

[An18] An, D.: „Find out how you stack up to new industry benchmarks for mobile page speed“, o.O., 2018.
https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/
Abruf: 10.05.2022

[Cod22] Codecept.io: „Getting Started | CodeceptJS“, o.O., 2022.
https://codecept.io/basics/#architecture
Abruf: 18.05.2022

[Dra06] Draheim Dirk: „Realistic load testing of Web applications“, Freie Universität Berlin, Institut für Informatik, Berlin, 2006

[Moz22] MDN contributors: „Performance - Web API Referenz | MDN“, o.O., 2020.
https://developer.mozilla.org/de/docs/Web/API/Performance
Abruf: 09.06.2022

[Men02] Menasce, D.: „Load testing, benchmarking, and application performance management for the web“, Computer Measurement Group, Reno, 2002

[Ope22] OpenJS Foundation: „About | Node.js“, o.O., 2022.
https://nodejs.org/en/about/
Abruf: 19.05.2022

[Ora22] Oracle: „Oracle VM VirtualBox“, o.O., 2022.
https://www.virtualbox.org/
Abruf: 24.05.2022

[Veg22] Vega: „A Visualization Grammar“, o.O., 2022.
https://vega.github.io/vega/
Abruf: 12.07.2022

[Wor22a] World Wide Web Consortium: „High Resolution Time“, o.O., 2022.
https://www.w3.org/TR/hr-time/
Abruf: 09.06.2022

[Wor22b] World Wide Web Consortium: „WebDriver“, o.O., 2022.
https://www.w3.org/TR/webdriver/
Abruf: 17.05.2022

[Xce22a] Xceptance: „XLT Testautomatisierung und Lasttests“, Jena, 2022.
https://www.xceptance.com/de/xlt/#myCarousel
Abruf: 10.05.2022

[Xce22b] Xceptance: „About XLT“, Jena, 2022.
https://docs.xceptance.com/xlt/about-xlt/
Abruf: 17.05.2022

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Bachelorarbeit mit dem Thema:

Evaluation und Umsetzung von Performance-Tests mit Hilfe von CodeceptJS

ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Projektarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum Unterschrift