Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Christian Vogt

Entwicklung einer Anwendung zur Analyse des Lastverhaltens von Webdiensten mit WebSocket-API

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Christian Vogt

Entwicklung einer Anwendung zur Analyse des Lastverhaltens von Webdiensten mit WebSocket-API

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Dipl.-Ing. Schneider
Zweitgutachter : Prof. Dr. rer. nat. Dipl.-Ing. Lehmann

Abgegeben am 10. August 2012

Christian Vogt

Thema der Bachelorthesis

Entwicklung einer Anwendung zur Analyse des Lastverhaltens von Webdiensten, die eine WebSocket-API bereitstellen

Stichworte

Webdienst, HTML5, WebSocket, HTTP, Lasttest, Modellbasiertes Testen

Kurzzusammenfassung

Diese Ausarbeitung gibt einen Einblick in die momentane Entwicklungslage von Webdiensten. Ziel war es, mit der zunehmenden Komplexität dieser mitzuhalten und eine Überlastsituation zu vermeiden. Mithin wurde eine Anwendung entwickelt, die Webdienste mit WebSocket-API auf ihr Lastverhalten hin untersucht. Außerdem werden Möglichkeiten aufgezeigt, mit denen die Ergebnisse interpretiert und Testabläufe unter Zuhilfenahme von UML-Anwendungsfalldiagrammen erstellt werden können.

Christian Vogt

Title of the paper

Development of an Application to analyse load specific behavior of Web services that provide a WebSocket API

Keywords

Web service, HTML5, WebSocket, HTTP, Load testing, Model-based testing

Abstract

This paper provides an insight into the present stage of the development of web services. The main intention was to keep up with their increasing complexity and to avoid a situation of overload. The result of this aim was the the development of an application that analyses web services with a WebSocket API concerning their load performance. In this context, possibilities will be pointed out to interprete the results and to create test processes by means of UML use case diagrams.

Danksagung

An dieser Stelle möchte ich mich bei meiner Familie für die Unterstützung während des gesamten Studiums, meiner Freundin besonders für ihre Geduld und Nachsicht und natürlich meinen Kommilitonen danken, die im Laufe meines Studiums zu sehr guten Freunden geworden sind und deren Hilfsbereitschaft und Unterstützung ich sehr zu schätzen weiß.

Meinen Betreuern Prof. Dr. Jochen Schneider und Prof. Dr. Thomas Lehmann gilt besonderer Dank für ihre Zeit, Geduld und konstruktive Anregungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

503: Service Unavailable. Mit dieser Meldung werden Besucher von Webseiten immer wieder konfrontiert. Sei es das Online-Fernsehprogramm, das kurz vor 20 Uhr abgerufen werden soll, oder ein Blick auf den Fußball-Ticker zum Top-Spiel der Woche kurz vor Spielende.

Mit fortschreitender Entwicklung des Internets werden immer komplexere Anwendungen über den Web-Browser ausgeliefert und in der „Cloud“, beispielsweise in einem entfernten Rechenzentrum, ausgeführt. Die Anwender sind auf einen reibungslosen Betrieb angewiesen und ein Ausfall des Systems wirkt sich auf eine viel größere Anzahl von Benutzern aus. Aus diesem Grund ist es erforderlich, die Anwendung effektiv testen zu können, bevor und während sie produktiv eingesetzt wird. Einen wichtigen Teil hierzu tragen Unit-Tests bei, die dem Entwickler automatisiert aufzeigen, wo sich Programmierfehler verbergen könnten. Ein zunehmend wichtiger Faktor ist allerdings das Verhalten des Systems unter Last, welches sich mit spezialisierter Software herbeiführen lässt.

Neue dynamische Techniken verändern das Internet, beginnend mit der Entwicklung von AJAX vor 7 Jahren hin zu neuen Konzepten wie den WebSockets. Ein klassischer Test-Aufbau mit der üblichen Lasttest-Software lässt sich im Fall der WebSockets nicht mehr umsetzen. Diese Lücke soll das im Rahmen dieser Bachelor-Thesis beschriebene *wsload* füllen und dem Entwickler von modernen Webdiensten mit WebSocket-API eine Möglichkeit geben, die Skalierbarkeit seiner Anwendung zu überprüfen und die Auswirkung von vielen parallelen Benutzern zu simulieren.

2 Grundlagen

Dieses Kapitel gibt eine detaillierte Einleitung in das Thema und definiert eine Problemstellung, aus der eine Zielsetzung und zuletzt eine Abgrenzung resultieren.

2.1 Geschichtlicher Abriss der Webentwicklung

Seit Tim Berners-Lee et al. 1992 erstmals die Idee des World Wide Web (kurz Web, WWW) vorstellten [1], entwickelt es sich in einer rasanten Geschwindigkeit. Sein Traum war es, dass der Mensch sein Wissen über eine Plattform teilen und erweitern kann, auf die jeder Zugriff hat, und somit eine kollektive, vernetzte Wissensdatenbank entsteht. Dieser Traum ist heute Wirklichkeit und Dienste wie das freie Online-Lexikon Wikipedia sind zur Normalität geworden. Laut IWS [6] waren im Jahr 2011 bereits 32,7% der Weltbevölkerung mit dem Internet verbunden und somit in der Lage, auf die riesigen Datenmengen zuzugreifen.

Die Idee von vernetzten Dokumenten in Form von HTML wurde in der Zwischenzeit weiterentwickelt und der eben beschriebene, reine dokumentbasierende Austausch zwischen System und Benutzer um dynamische Elemente erweitert. Mit der Veröffentlichung von JavaScript konnte das statische Dokument nun mit diesen Elementen ergänzt werden und näherte sich immer weiter einem „Programm [an], das es dem Anwender ermöglicht, am Computer spezielle Aufgaben (...) durchzuführen“[1].

Ein großer Teil des heutigen Webs ist geprägt von multimedialen, sozialen und spielerischen Aspekten. Da die Möglichkeiten von HTML und JavaScript für diese Bereiche oft nicht ausreichten, wurden mit Adobe Flash, Microsoft Silverlight etc. proprietäre, kommerzielle Alternativen geschaffen, die allerdings weder auf offenen Standards beruhen, noch auf allen internetfähigen Geräten verfügbar sind. HTML5, dass sich derzeit bei der WHATWG in der (schon weit fortgeschrittenen) Entwicklung befindet[2], gilt als Nachfolger und soll Plugins wie Flash oder Silverlight ersetzen, woran Adobe selbst beteiligt ist [11].

[1] Duden online zum Suchbegriff „Anwenderprogramm“
[2] http://whatwg.org/html

Für die Kommunikation zwischen Server und Client wurde das Protokoll HTTP[3] eingeführt. Es dient ursprünglich der Übertragung von „Hypertext“ aus dem WWW zu einem Browser, der diesen als Dokument anzeigt. Durch die Erweiterung des Protokolls ist HTTP heute nicht mehr nur auf Hypertext beschränkt, sondern wird zunehmend zum Austausch beliebiger Daten verwendet. Das Protokoll ist zustandslos, jede Anfrage an einen Server ist also eine von den vorangegangenen unabhängige, stellt also eine komplett eigenständige Transaktion dar, die nur vom Client aus initiiert werden kann.

Mit der steigenden Dynamik des Webs wurde das Konzept von HTTP zum Problem, da jede Anfrage vom Client dazu führte, dass aufgrund der Zustandslosigkeit das gesamte Dokument neu geladen werden musste (ein neuer Request-Response-Cycle wird vom Browser angestoßen). Daraufhin wurde 2005 mit AJAX ein Konzept entwickelt, das bis heute im Einsatz ist [2]. Hierbei erfolgt die Kommunikation weiterhin über HTTP, allerdings ist kein Neuladen der Webseite nötig.

Da AJAX auf HTTP[4] beruht, ist die Kommunikation nur vom Client aus initiierbar. Soll dieser trotzdem auf Änderungen am Server reagieren können, muss ein Polling-Verfahren eingesetzt werden, was wiederum Overhead und Latenz bedeutet. Mit der fortschreitenden Entwicklung des Webs, bei der immer mehr Daten zwischen Client und Server ausgetauscht werden [8], musste ein effizientes, bidirektionales Verfahren eingesetzt werden, das nicht auf HTTP aufbaut.

Eine Lösung mit offenen Standards, die im Rahmen von HTML5 eingeführt wurde und als sehr effizient gilt, ist das WebSocket-Protokoll[5]. Vor- und Nachteile zum AJAX-Konzept sind hier stark anwendungsabhängig. Oft lohnt sich ein Einsatz von WebSockets, wenn die Anwendung intensiven Nachrichtenaustausch zwischen Server und Client voraussetzt, beispielsweise bei der möglichst latenzarmen Darstellung von Börsenkursen.

[3] http://tools.ietf.org/html/rfc2616
[4] genauer dem XMLHttpRequest, siehe http://tools.ietf.org/html/rfc2616
[5] http://tools.ietf.org/html/rfc6455

2.2 Problemstellung

Seit Jahren ist eine Transition von der lokalen Anwendung hin zu einem über das Internet vernetzten System zu erkennen. Im Gegensatz zur klassischen Anwendung, die auf dem Heim-Computer ausgeführt wird, können über das Web viele Benutzer parallel arbeiten [5].

Mit dieser Entwicklung entstehen auch neue Anforderungen an die Testumgebung. Klassische Unit-Tests können zwar Auskunft über die Funktionalität eines Vorgangs innerhalb des Systems geben, sie reichen jedoch häufig nicht aus, um Aussagen über die Auswirkungen von parallelen Anfragen zu treffen. Für eine effiziente Entwicklung mit dem Fokus auf das Web muss so früh wie möglich im Entwicklungsprozess die Möglichkeit gegeben sein, das Lastverhalten des Systems zu simulieren.

Viele Systeme im Web bieten APIs, über die andere Systeme Daten austauschen oder abfragen können. Ist das eigene Systeme auf fremde APIs angewiesen, müssen diese in die Tests eingeschlossen werden und der Testaufwand steigt weiter. Um die komplexen Zusammenhänge testen zu können, die mit der Vernetzung der Systeme auftreten, kann gegen die eigenen Schnittstellen getestet werden, die auch der Anwender während des Betriebs verwendet. So ist sichergestellt, dass alle involvierten Systeme einbezogen werden und die Tests auch bei Veränderung des Systems gültig bleiben.

Dieser Ansatz ist mit einem *Black-Box-Test* vergleichbar, dient allerdings nicht dem Ziel, das System auf Übereinstimmung mit der Spezifikation zu überprüfen, sondern bereits während der Entwicklung mögliche Schwachstellen aufzudecken. In der Literatur findet man für dieses Verfahren den Begriff *Grey-Box-Test* [4, 7], bei dem der Testersteller im Gegensatz zum *Black-Box-Test* Kenntnisse über den Aufbau des Systems hinter der API hat.

Zwei Fragen, die es für den Entwickler von Webdiensten im Bezug auf Lastverhalten zu beantworten gilt, sind:

- Welche Anzahl an Benutzern muss meine Anwendung parallel verarbeiten?
- Wie skaliert meine Anwendung, wenn neue Benutzer hinzukommen?

Diese Fragestellungen sind weder neu, noch wurden sie durch die Entwicklung des WebSockets-Protokolls aufgebracht, sondern sie stellen sich früher oder später jedem Entwickler von Webanwendungen bzw.-diensten.

Die in Kapitel 3.3 vorgestellten verfügbaren Lösungen, um diese Fragestellung zu beantworten, reichen jedoch für die an die Anwendung gestellten Anforderungen, die in Kapitel 3 spezifiziert werden, nicht aus. Der Grund hierfür ist, dass die Großzahl der Webanwendungen und -dienste heute auf HTTP basieren, weshalb auch die Testumgebungen auf dieses Protokoll ausgerichtet sind.

2.3 Zielsetzung

Ziel dieser Arbeit ist eine Software, die es einem Entwickler möglichst früh im Entwicklungsprozess ermöglicht, eine **Komponente mit WebSocket-API** von der Client-Seite aus auf ihr Lastverhalten zu testen. Komponenten sind dabei typischerweise Webdienste, Webanwendungen oder aber jedes andere System, dass das WebSocket-Protokoll[6] implementiert.

Der entscheidende Wert für die **Analyse des Lastverhaltens** ist hierbei das **Zeitverhalten**, genauer die Zeit vom Versenden einer Anforderung bis zum Erhalt des Ergebnisses am Client. Die gesammelten Werte sollen über eine Schnittstelle anderen Anwendungen verfügbar gemacht werden können.

Neben der Anwendungsentwicklung ist die **theoretische Interpretation der Ergebnisse** ein wichtiger Aspekt dieser Arbeit. Es sollen Möglichkeiten aufgezeigt werden, wie die Resultate in einen sinnvollen Zusammenhang gebracht werden können und mit welchen Mitteln die Testerstellung vereinfacht werden kann.

Die Software soll ein **agiles Testen** insofern fördern, als dass über eine hohe Automatisierung ein schnelles Feedback ermöglicht und dabei eine möglichst geringe Komplexität in der Testerstellung erhalten wird.

2.4 Abgrenzung

Es sind bereits viele Anwendungen auf dem Markt verfügbar, mit denen man Webdienste und -anwendungen auf Lastverhalten testen kann. Dabei werden verschiedene Ansätze verfolgt. Sehr konträre Ansätze werden dabei durch die Anwendungen *HP Load Runner* und *ApacheBench (ab)* umgesetzt und hier kurz vorgestellt, um die im Rahmen dieser Ausarbeitung zu entwickelnde Anwendung positionieren zu können.

ApacheBench von der Apache Software Foundation kann eine große Anzahl von HTTP-Anfragen an einen Server senden. Abb. 2.1 zeigt, wie 1000 HTTP GET-Anfragen an einen Webserver versendet werden und verdeutlicht die Simplizität der Anwendung.

```
1 ab -n 1000 http://christianvogt.de
```

Abbildung 2.1: Beispiel eines ApacheBench Aufrufs

[6] Nach der RFC6455, vgl. http://tools.ietf.org/html/rfc6455

Bei dieser Testart können allerdings keine Abläufe getestet, sondern lediglich einfache Seitenaufrufe simuliert werden. Muss beispielsweise vor dem Test eine Authentifizierung erfolgen, so kann diese mit *ApacheBench* nicht in den Testablauf integriert werden.

Einen anderen Ansatz verfolgt das Tool *HP LoadRunner*. Hier können komplexe Abläufe programmiert oder durch spezielle Browser-Plugins aufgezeichnet werden, die dann parallel ausgeführt werden, um reale User zu simulieren. Um einen Test ausführen zu können, sind im Gegensatz zu *ApacheBench* viele Voraussetzungen zu erfüllen. Neben den Lizenzkosten ist die Installation nicht trivial, erfordert mehrere physikalische Server und Installationen von Analyse-Agents auf den beteiligten (zu testenden) Systemen.[7]

Die zu entwickelnde Anwendung soll zwar im Gegensatz zu *ApacheBench* komplexe Abläufe simulieren können, dabei allerdings nicht die Komplexität aufweisen, die mit der Einführung einer Anwendung wie dem *HP LoadRunner* einhergeht. Der Fokus liegt auf der Ergebnisermittlung von lastspezifischen Merkmalen einer WebSocket-API von Webdiensten, die vom Client aus gemessen werden können. Weitere Messungen hinter der API (z. B. der Datenbank-Performance) sind nicht vorgesehen.

Des Weiteren soll die zu entwickelnde Anwendung im Sinne des *Separation Of Concerns* nur für die Erzeugung von Testdaten zuständig sein. Etwaige Auswertungen werden im Rahmen dieser Thesis theoretisch besprochen, müssen jedoch anwendungsfallabhängig vom Tester erstellt werden.

Die Begriffe Performance-, Last- und Stresstest werden häufig unterschiedlich interpretiert. Im Rahmen dieser Thesis werden die Begriffe wie folgt verwendet. Ein Performancetest wiederholt einen ausgewählten Testfall unter einer Grundlast. Lasttests verknüpfen Testfälle und testen die resultierende Testfall-Kette auf Performanz, während Stresstests die Belastung des Systems bewusst über eine Lastgrenze hinaus steigern.

[7] Eine grafische Übersicht über die Serverkonstellation von HP LoadRunner findet sich beispielsweise unter http://reflow.scribd.com/frd3qfeiobghgn/images/image-3.jpg

2.5 Aufbau der Ausarbeitung

Im Rahmen dieser Bachelorthesis wird eine Anwendung entwickelt, mit der Lastanalysen von Webdiensten mit WebSocket-API erstellt werden können, und theoretisch erarbeitet, wie die Ergebnisse interpretiert werden können. Die Ausarbeitung gliedert sich in sieben Kapitel, die hier kurz vorgestellt werden.

Aufbau der Kapitel

Kapitel 1 - Einleitung

In dem ersten Kapitel wird das Thema der Ausarbeitung kurz vorgestellt.

Kapitel 2 - Grundlagen

Neben der Vermittlung von Kenntnissen der Grundlagen moderner Web-Entwicklung werden die Problemstellung ermittelt und die Ziele definiert.

Kapitel 3 - Anforderungen an die Applikation

Dieses Kapitel dient zur Analyse der Zielsetzung und Ableitung von Anforderungen. Diese Anforderungen werden außerdem mit bestehenden Lösungen verglichen.

Kapitel 4 - Konzeption

In diesem Kapitel werden das Anwendungskonzept erläutert und die Schnittstellen beschrieben.

Kapitel 5 - Realisierung

Diskussion der umsetzungsspezifischen Aufgaben zur Realisierung im Hinblick auf die verwendete Programmierumgebung.

Kapitel 6 - Verwendung

Einsatz der Anwendung und theoretische Methoden der Ergebnisinterpretation.

Kapitel 7 - Zusammenfassung und Ausblick

Im letzten Kapitel werden die Ergebnisse zusammengefasst und ein Ausblick auf die weitere Entwicklung gegeben.

Anmerkungen

In dieser Arbeit wird Gebrauch von Terminologie gemacht, die noch keiner festen Fachdefinitionen unterliegt oder dessen wörtliche Übersetzung im deutschen Sprachgebrauch nicht gängig ist. An diesen Stellen wird auf den englischen Begriff zurückgegriffen.

Des Weiteren sind sowohl der Quellcode und die Dokumentation der Anwendung, als auch alle UML-Diagramme im Hinblick auf die Weiterverwendung in englischer Sprache verfasst. Klassen- und Methodennamen werden, wenn im Textfluss verwendet, wie folgt dargestellt:

- Beispiel eines Klassennamens: `MyClass()`
- Beispiel eines Methodennamens: `myMethod()`.

Begriffsdefinition

- **Zu testendes System** bezeichnet das System, welches sich dem Test unterzieht
- **Testsystem** bezeichnet das System, von dem der Test ausgeht.

3 Anforderungen an die Applikation

In diesem Kapitel werden die verschiedenen Aspekte des letzten Kapitels weiterentwickelt und überprüft, welchen Nutzen die zu entwickelnde Anwendung haben soll. Daraus werden einzelne Anforderungen abgeleitet, um diese dann zu strukturieren. Die abgeleiteten Anforderungen sind im Anhang B zusammengefasst aufgeführt.

3.1 Anforderungsanalyse

Um die Anforderungen an die Software zu formulieren, werden die Kernpunkte der Zielsetzung (Kapitel 2.3) und Abgrenzung (Kapitel 2.4) aufgenommen und weiterführend untersucht, indem sie auf folgende Kernfragestellungen reduziert werden:

- Für welche Einsatzzwecke ist die Anwendung bestimmt?
- Was kennzeichnet einen Test?
- Was wird gemessen?
- Was wird ausgewertet?
- Wie wird agil getestet?
- Ist ein verteilter Testaufbau notwendig?
- Wie wird die Anwendung veröffentlicht?

Diese Kernfragen gilt es in diesem Kapitel zu beantworten.

Für welche Einsatzzwecke ist die Anwendung bestimmt?

Die typischen Komponenten, die von der zu entwickelnden Anwendung getestet werden sollen, sind Webdienste und indirekt Webanwendungen. Da diese wichtigen Begriffe keiner klaren Definition unterliegen, werden sie kurz erläutert.

Webanwendung

> Eine Webanwendung ist der Überbegriff für eine Software, die über das Internet ausgeliefert wird und in einem Browser lauffähig ist. Sie kann sowohl client- als auch serverseitige Komponenten verwenden und lässt einen Benutzer mit einem System über eine grafische Oberfläche interagieren.
> Der Unterschied zu einer Webseite sind die gesteigerte Dynamik und die Hintergrundkommunikation zwischen Client und Server ohne manuelle Aktualisierung des Browserfensters.

Webdienst

> Im Gegensatz zur Webanwendung ist ein Webdienst nicht für die direkte Kommunikation zwischen Benutzer und System vorgesehen, sondern wird von einem anderen System angesprochen. Heutige Webanwendungen verwenden häufig Webdienste, die die Funktionalität der Anwendung kapseln oder erweitern und mittels einer API verfügbar machen.

Ein effizientes Testen ist nur möglich, wenn eine API zur Systemkommunikation bereitgestellt wird, wie es bei einem Webdienst der Fall ist. Im Gegensatz zu dem Test eines Webdienstes ist bei dem Test der Anwendung oft eine größere Anzahl an Systemen betroffen (z. B. ein Event-Handling der GUI-Elemente oder clientseitige Validierung) und die Interpretation der Testergebnisse in Bezug auf die Client-Server-Kommunikation wird schwieriger. Um also eine aussagekräftige Analyse des Lastverhaltens zu ermöglichen, sind Testfälle nötig, die jeweils einen begrenzten Funktionsumfang abdecken und dieselben Schnittstellen verwenden, die bei der Systemkommunikation verwendet werden.

Was kennzeichnet einen Test?

Ein Testablauf besteht aus mehreren Testfällen, die zu einer Testsuite zusammengefasst werden können. Dabei ergeben sich unterschiedliche Möglichkeiten, wie die Testfälle in der Testsuite ausgeführt werden:

- parallel
- sequenziell
- in beliebiger Reihenfolge.

Die Software soll die Testfälle sequenziell ausführen, so dass der Testentwickler die Reihenfolge bestimmen kann. Testsuiten werden immer parallel ausgeführt. Für die Kommunikation zwischen den Testfällen soll eine Schnittstelle bereitgestellt werden, über die Daten von einem Testfall zum nächsten weitergegeben werden können. Zwischen den Testfällen soll eine Möglichkeit bestehen, eigene Inhalte/Code einzufügen, deren Ausführungszeit nicht in die Zeitmessung eingehen. An dieser Stelle soll auch eine Zeitverzögerung oder eine Synchronisation möglich sein, mit der sich die parallelen Testsuiten kontrolliert ausführen lassen.

Was wird gemessen?

Die Analyse des Lastverhaltens geschieht mittels einer Zeitmessung. Die zu messende Zeit lässt sich, wie in Gleichung 3.1 dargestellt, in drei Zeiten aufteilen. Hierbei beschreiben die Transportzeiten die Latenz der Verbindung (WAN je nach Entfernung/Verbindungsart meist <100ms, im LAN <10ms) und die Verarbeitungszeit die Zeit, die der Server benötigt, um die eingehende Nachricht zu verarbeiten und ein Ergebnis zurückzuliefern. Die Transportzeiten für Hin- und Rückweg sind meist ähnlich groß, weshalb man sie näherungsweise zusammenfassen kann, wie in Gleichung 3.2 dargestellt. Der von der Testanwendung zu messende Wert ist also $t_{Testdauer}$ nach Gleichung 3.2.

$$t_{Testdauer} = t_{Transport(Hinweg)} + t_{Verarbeitung} + t_{Transport(Rückweg)} \tag{3.1}$$

$$t_{Testdauer} = t_{Verarbeitung} + t_{Transport} \tag{3.2}$$

Die Transportzeit ($t_{Transport}$) eines Clients lässt sich typischerweise über ICMP (Ping) feststellen und kann bei Bedarf später von den ermittelten Testzeiten subtrahiert werden. Dabei ist zu beachten, dass sich die Transportzeit durch die im Internet verwendete Paketvermittlung bei jedem Paket ändern kann. Um eine möglichst exakte Messung der Verarbeitungszeit zu erhalten, müsste ein Lasttest in einem LAN ausgeführt werden, in dem die Transportzeiten vorher gemessen werden können und sich über den Testzeitraum kaum ändern.

Gerade bei inländischen Verbindungen sind die Transportzeiten aber oft stabil und schwanken nur um wenige Millisekunden. Wenn ein typischer Testfall deutlich länger dauert als die Transportzeitschwankungen des verwendeten Netzes groß sind, lässt sich ein aussagekräftiges Ergebnis ermitteln.

Des Weiteren kann die Analyse des Lastverhaltens nicht mit der Ausführung eines einzelnen, sondern muss mit einer Vielzahl an parallelen Tests durchgeführt werden, um eine entsprechende Client-Anzahl zu simulieren. Durch die Bildung eines Mittelwerts über die gemessenen Zeiten könnte ein Fehler durch einzelne hohe Transportzeiten reduziert werden. Es muss bedacht werden, dass sich die Transportzeiten der Clients deutlich voneinander unterscheiden können. Eine Satellitenverbindung weist im Vergleich zu einer kabelgebundenen typischerweise eine deutlich höhere Latenz auf.

Was wird ausgewertet?

Bei einer beispielhaften Testsuite mit mehreren enthaltenen Testfällen fallen folgende messbare Werte an:

- Ablaufzeit der gesamten Testsuite
- Ablaufzeit der einzelnen Testfälle

Um einen Testablauf auswerten zu können, müssen neben den Messwerten weitere Informationen gespeichert werden, um die Ergebnisse zuordnen zu können. Folgende Daten müssen nach einem Testablauf zur Verfügung stehen, um weiter ausgewertet zu werden.

- Bezeichnung der Testsuite
- Datum von Start und Ende der Suite
- Anzahl der parallelen Testsuite-Durchläufe
- Bezeichnung der enthaltenen Testfälle
- Ein/Ausgabeparameter der Testfälle
- Zeitmessungsergebnis der einzelnen Testfälle
- Startzeitpunkt des jeweiligen Testfalls

Im Sinne des Separation Of Concerns obliegt die Verantwortung der Resultatinterpretation bei dem Testersteller. Die erforderlichen Daten sollen nach dem Testablauf auch für andere Anwendungen verfügbar sein, zum Beispiel über eine Datenbank.

Wie wird agil getestet?

Die zu entwickelnde Software soll sich einfach und schnell konfigurieren lassen und auch den Prozess der Testerstellung einfach halten, damit schnelle Ergebnisse erzielt werden können. Es sollte nicht länger als einige Stunden dauern, sich mit der Software vertraut zu machen und erste Testabläufe durchzuführen.

Die Ergebnisse sollen persistent gespeichert werden können, sodass zu einem späteren Zeitpunkt auf sie zugegriffen werden kann.

Ist ein verteilter Testaufbau notwendig?

Clients, die einen Dienst über das Web verwenden, bilden ein stark verteiltes System. Benutzt man ein einzelnes Testsystem, um diese Clients zu simulieren, kann sich das auf die Messdaten auswirken. Außerdem darf das System, das die Tests ausführt, zu keinem Zeitpunkt so stark belastet sein, dass sich hier Verzögerungen ergeben, da das Resultat dann nicht mehr interpretierbar ist. Die Anwendung soll die Möglichkeit bieten, mehrere Testsysteme gleichzeitig zu kontrollieren und die Last zu verteilen, um die Auswirkungen eines verteilten Aufbaus zu ermitteln.

Wie wird die Anwendung veröffentlicht?

Um die weitere Entwicklung zu fördern, soll der Quellcode der Anwendung zugänglich sein und unter einer freien Lizenz veröffentlicht werden.

3.2 Ableitung von Requirements

Für den weiteren Entwicklungsgang lassen sich die Antworten zu den Fragestellungen aus Kapitel 3 auf ihre Grundinformation reduzieren und mit eindeutigen Bezeichnern versehen. Darüber hinaus werden die Anforderungen gruppiert und zugeordnet. Das entstandene Dokument findet sich im Anhang (siehe Kapitel 7) und zeigt die Struktur auf, die aus der Anforderungsanalyse (Kapitel 3.1) entstanden ist. Um die Anforderungen zu strukturieren, wurden die folgende Kategorien eingeführt:

- Schnittstellen
- Systembedingungen
- Testerstellung
- Testauswertung
- Zeitmessung

3.3 Vergleich mit bestehenden Lösungen

Die aufgestellten Anforderungen ermöglichen es, bereits in der frühen Phase der Entwicklung einen Vergleich mit den auf den Markt befindlichen Lösungen vorzunehmen. Das Konzept muss dahin gehend überprüft werden, ob es bereits umgesetzt ist oder eine Erweiterung eines bestehenden Projekts ausreichen würde. Dafür wird eine Marktübersicht erstellt und die entsprechende Anwendung kurz vorgestellt.

Später erfolgt ein Feature-Vergleich, mit dem aufgezeigt wird, dass derzeit keine der Anwendungen die Anforderungen, wie sie in diesem Kapitel aufgestellt worden sind, erfüllen kann.

Marktübersicht

Eine Recherche ergab die folgenden Projekte, die hier kurz vorgestellt und deren Ähnlichkeiten herausgearbeitet werden.

- The Grinder[1]
- HP Load Runner[2]
- wsbench[3]
- Selenium[4]
- diverse Unit-Test Frameworks[5]

The Grinder

The Grinder ist ein auf Java basierendes, frei verfügbares Framework mit dem Fokus auf verteilte Tests von Webservern. Die Testfälle werden in den Skriptsprachen Jython oder Clojure definiert. Auch andere Protokolle (wie SOAP) werden unterstützt, allerdings nur aufbauend auf dem Request-Response-Modell von HTTP. Durch die Skriptsprachen und den modularen Aufbau lassen sich Tests schnell erstellen.

HP Load Runner

HP LoadRunner ist in der Lage, mit verschiedenen Protokollen umzugehen, hauptsächlich aber wie The Grinder auf HTTP ausgerichtet. Seit 2011 ist das Tool in der Lage, einfache AJAX-Requests aufzunehmen und in die Tests einfließen zu lassen. Es werden nicht nur clientseitige Daten gemessen, sondern auch mittels serverseitigen Agents zusätzliche Informationen gesammelt. HP Load Runner ist ein kommerzielles Produkt, dessen Einführung in den Entwicklungsprozess mit erheblichem Aufwand verbunden ist.

[1] http://grinder.sourceforge.net/
[2] http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451
[3] https://github.com/pgriess/wsbench/
[4] http://seleniumhq.org/
[5] z. B. JUnit

wsbench

Wsbench ist ein freies Tool, das viele parallele Anfragen an eine WebSocket-API stellt und somit Last erzeugt. Es ist vergleichbar mit dem in den Grundlagen (Kapitel 2) vorgestellten *ApacheBench* und lässt sich sehr simpel bedienen. Allerdings lassen sich weder komplexe Testabläufe generieren, noch der Nachrichteninhalt der Anfragen beeinflussen.

Selenium

Mit Selenium lassen sich Browser steuern, sodass automatisierte Tests erstellt werden können, die Abläufe auf einer Webseite ausführen. Dabei wird die Benutzer-System-Kommunikation simuliert, was eine GUI, die im Browser dargestellt werden kann, voraussetzt.

Unit-Test Frameworks

Unit-Tests werden verwendet, um einzelne Module einer Software auf Funktionalität zu überprüfen. Sie zählen zu den sogenannten White-Box-Tests bei denen der Quellcode und die Auswirkung auf das Testsystem bekannt sind. In modernen agilen Softwareentwicklungsmethoden sind automatisierte Unit-Tests eine wichtige Komponente. Allerdings sind sie gerade bei verteilten Anwendungen nicht ausreichend für Aussagen über die Lastsituation im realen Betrieb, da häufig weder gegen die externe API getestet wird, noch parallele Tests stattfinden.

Fazit zur Marktübersicht

Die Marktübersicht zeigt anhand einiger populärer Projekte, dass die Notwendigkeit von Lasttests gerade im Bereich des Webs erkannt wurde und bereits viele Lösungen vorhanden sind. Alle umfangreicheren Lasttest-Lösungen basieren allerdings auf der Annahme, dass das HTT-Protokoll für die Kommunikation verwendet wird. Mit der Entwicklung des WebSocket-Protokolls, das nicht auf dem zustandslosen HTTP basiert, reichen die vorhandenen Lösungen nicht mehr aus.

Projekte wie das vorgestellte *wsbench* können keine komplexeren Abläufe simulieren und bieten somit nur einen geringen Informationsgehalt. Selenium hingegen lässt sich für das funktionale Testen gut einsetzen, jedoch nicht für einen Lasttest von Webdiensten, da jede Testinstanz in einem Browserfenster läuft. Dies bedeutet großen Overhead. Zudem wird nur

eine Kommunikation zwischen Benutzer und System, nicht aber die erforderliche System-System-Kommunikation simuliert.

Unit- und Lasttests bieten unterschiedliche Vorteile und schließen sich nicht aus, sondern sollten in der modernen Softwareentwicklung von Webdiensten in Kombination verwendet werden.

4 Konzeption

Dieser Abschnitt befasst sich mit der Konzeption der an die Anwendung gestellten Vorgaben. Dabei kommen hauptsächlich UML-Diagramme zum Einsatz, die den Ablauf der Anwendung beschreiben sollen und zum Verständnis beitragen. Da die eigentliche Umsetzung iterativ erfolgt und immer wieder Auswirkungen auf das Konzept hat, bezieht sich dieses Kapitel auf die Version wsload 0.1.0[1], die sich auch im Anhang dieser Arbeit befindet.

Nach Abschluss der Konzeption wird eine Programmierumgebung gewählt, mit der das Konzept umgesetzt wird (vgl. Kap. 5). Die Wahl dieser Umgebung führte dazu, dass die Diagramme nochmals überarbeitet werden mussten. Der Grund hierfür ist die Verwendung von asynchronen Callbacks im Programmablauf.

Eine Callback-Funktion wird als Parameter übergeben und zu einem späteren Zeitpunkt wiederum mit dem Ergebnis als Parameter aufgerufen. Abb. 4.1 zeigt den Zusammenhang zwischen Quellcode und dem dazugehörigen UML-Klassendiagramm.

Um den Ablauf darstellen zu können, werden Aktivitätsdiagramme eingesetzt. Die Verwendung von asynchronen Callback-Funktionen erfolgt nach der aktuellen UML-Spezifikation[2], wird aber hier auf Grund der geringen Verbreitung kurz erläutert.

```
1  function MyClass () {
2    //constructor
3  };
4  MyClass.prototype.calculateTime = function(callback){
5    var result = 5 + 5;
6
7    //execute Callback after 100ms
8    setTimeout(function () {
9      callback(null, result);
10   },100);
11 };
```

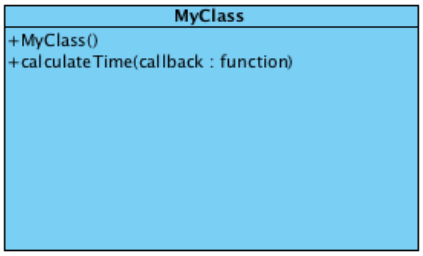

Abbildung 4.1: Beispiel einer Klasse mit einer asynchronen Methode in JavaScript

Eine Aktivität, die die Beispielklasse nutzt, ist unter Abb. 4.2 und der entsprechende Quellcode unter Abb. 4.3 dargestellt. Es wird ein Objekt der Klasse `MyClass()` instanziiert und

[1] https://github.com/chris–/wsload/commit/d26f59f5eed4ae86f1e32b293d030e05f17dfb1b
[2] vgl. UML Spezifikation 11-08-06/ 11.3.10 - CallOperationAction/ 11.3.8 - CallAction

an eine *Call Operation Action* übergeben. Diese führt auf dem *target*, also auf der übergebenen Objektinstanz die Methode `calculateTime()` aus.

Die *Call Operation Action* erbt laut Spezifikation von der *Call Action*, die durch das boolsche Attribut `isSynchronous` gekennzeichnet werden kann. Ist dieses Attribut `false`, wird die zugeordnete Methode nur aufgerufen und nicht auf das Ergebnis gewartet. Sobald das Ergebnis zur Verfügung steht, liegt es an dem Output-Pin (`result`) der *Call Operation Action* an und wird von der entsprechenden *Action* übernommen.

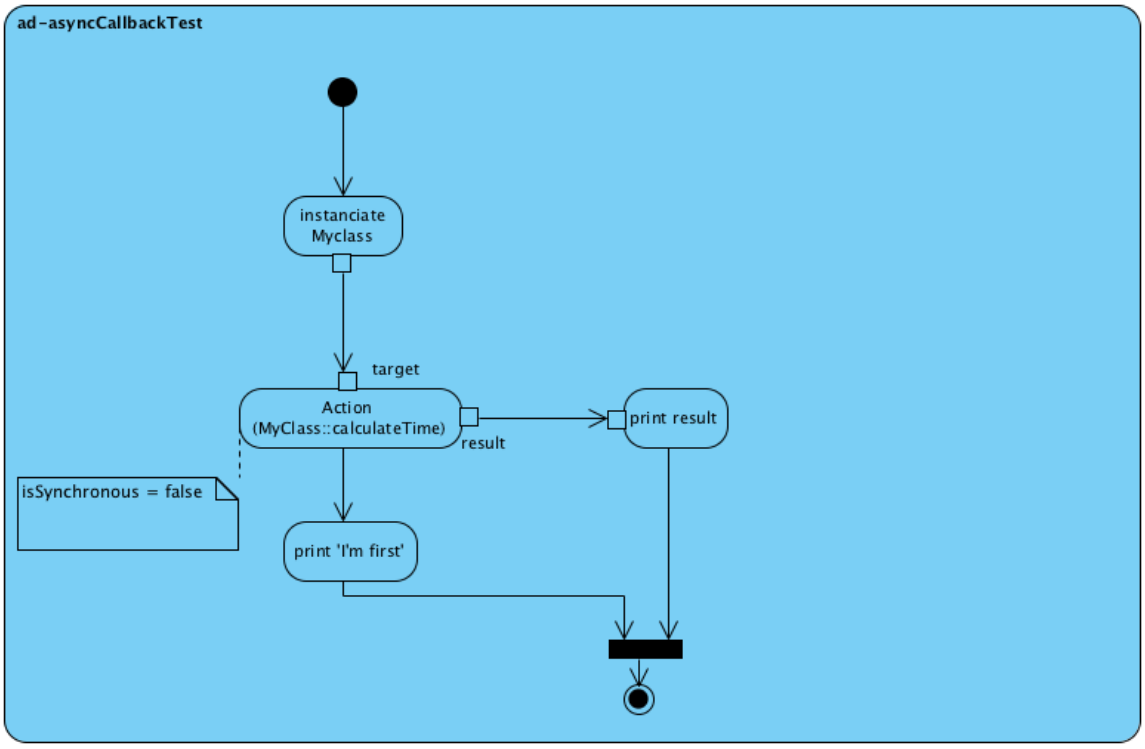

Abbildung 4.2: Beispiel eines UML Aktivitätsdiagramms mit asynchronen Callbacks

Abb. 4.3 stellt den Quelltext des Aktivitätsdiagramms aus Abb. 4.2 dar. Zeile 2 instanziiert ein Objekt der beschriebenen Klasse, auf dem in Zeile 11 eine Methode ausgeführt wird. Zeile 5-8 definieren eine Callback-Funktion, die aufgerufen wird, sobald die aufgerufene Methode die Bearbeitung abgeschlossen hat. Nach dem Aufruf der asynchronen Methode `MyClass.calculateTime()` wird auf der Konsole ein Text ausgegeben. Diese Ausgabe erfolgt zeitlich vor dem Aufruf des Callbacks, was die Asynchronität des Methodenaufrufs verdeutlicht.

```
1  //instanciate a MyClass Object
2  var myClassInstance = new MyClass();
3
4  //create a callback function that prints the result
5  var callbackFunction = function (err,result) {
6    if (err) console.log('error occured: ' + err );
7    else console.log(result);
8  };
9
10 //call asynch method
11 myClassInstance.calculateTime(callbackFunction);
12
13 //print string to console
14 console.log('I am first');
15
16 // console output:
17 // I am first
18 // 10
```

Abbildung 4.3: Ausführung eines asynchronen Callbacks in JavaScript

4.1 Anwendungskonzept

In diesem Kapitel wird zuerst das Prinzip eines Testablaufs vorgestellt, bevor auf die Schnittstellen und den persistenten Speicher eingegangen wird, in dem die Resultate abgelegt werden.

Das Anwendungskonzept sieht **Testfälle** (engl. test cases) vor, die ausführbaren Code enthalten. Diese werden von der Anwendung in eine **Testsuite** gekapselt. Zuletzt wird ein **Testablauf** erstellt, der die parallele Ausführung mehrerer Testsuiten ermöglicht. Diese Hierarchie wird hier detailliert besprochen.

Für einen Überblick über die Anwendung bietet sich ein Klassendiagramm an, wie es in Abb. 4.4 dargestellt ist.

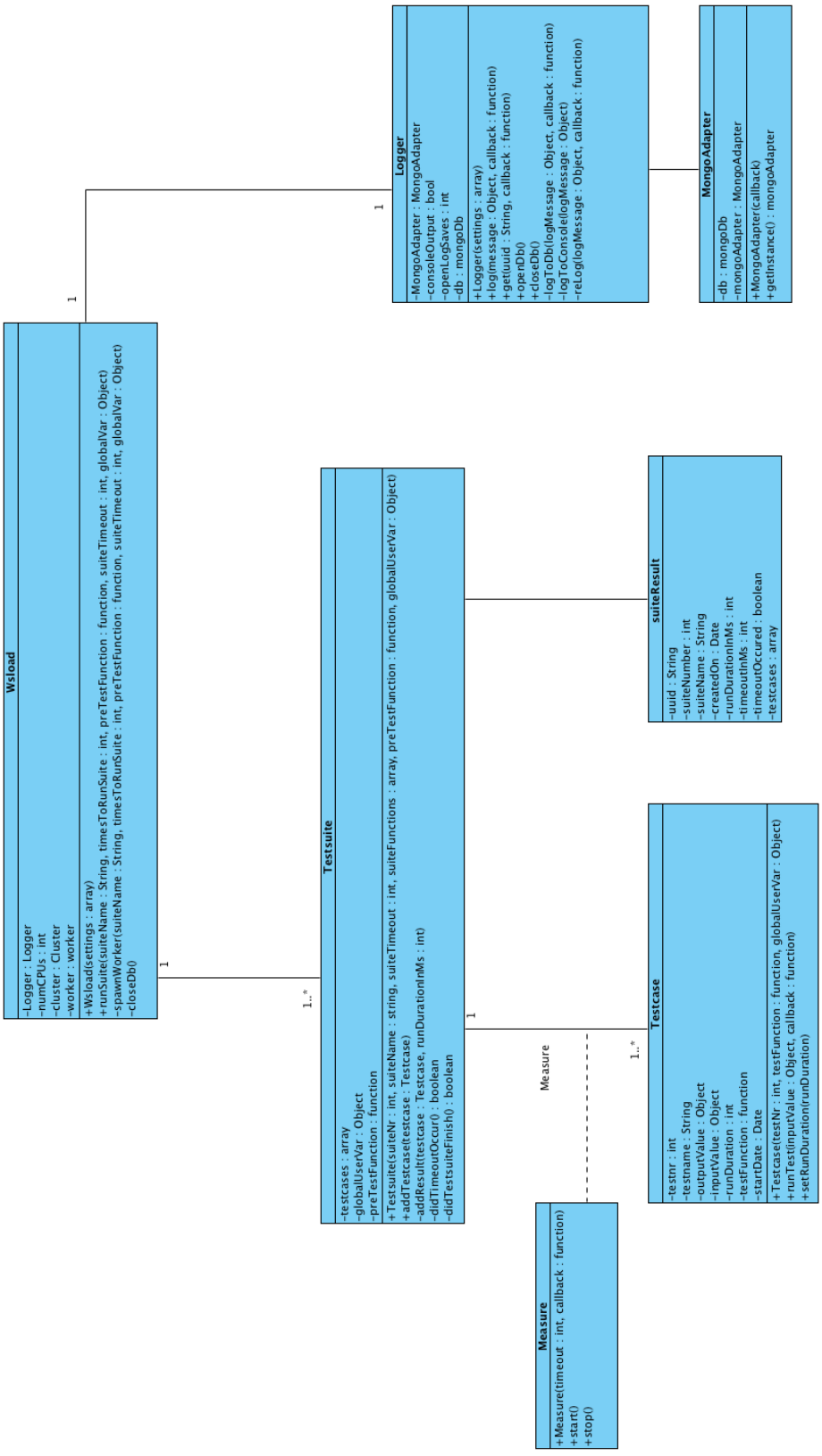

Abbildung 4.4: UML Klassendiagramm für Überblick über die Anwendung wsload

Testfall

Ein Testfall bezeichnet einen möglichst kleinen Anwendungsfall, dessen Ausführungsdauer gemessen werden kann. Der Testfall wird in Codeform im System hinterlegt und automatisch ausgeführt. Dabei ist es möglich, Parameter zu übergeben, die dem getesteten Code zur Verfügung stehen und Parameter zurückzugeben, welche dann wiederum dem nächsten Testfall zur Verfügung stehen. Abbildung 4.5 zeigt das entsprechende UML Aktivitätsdiagramm.

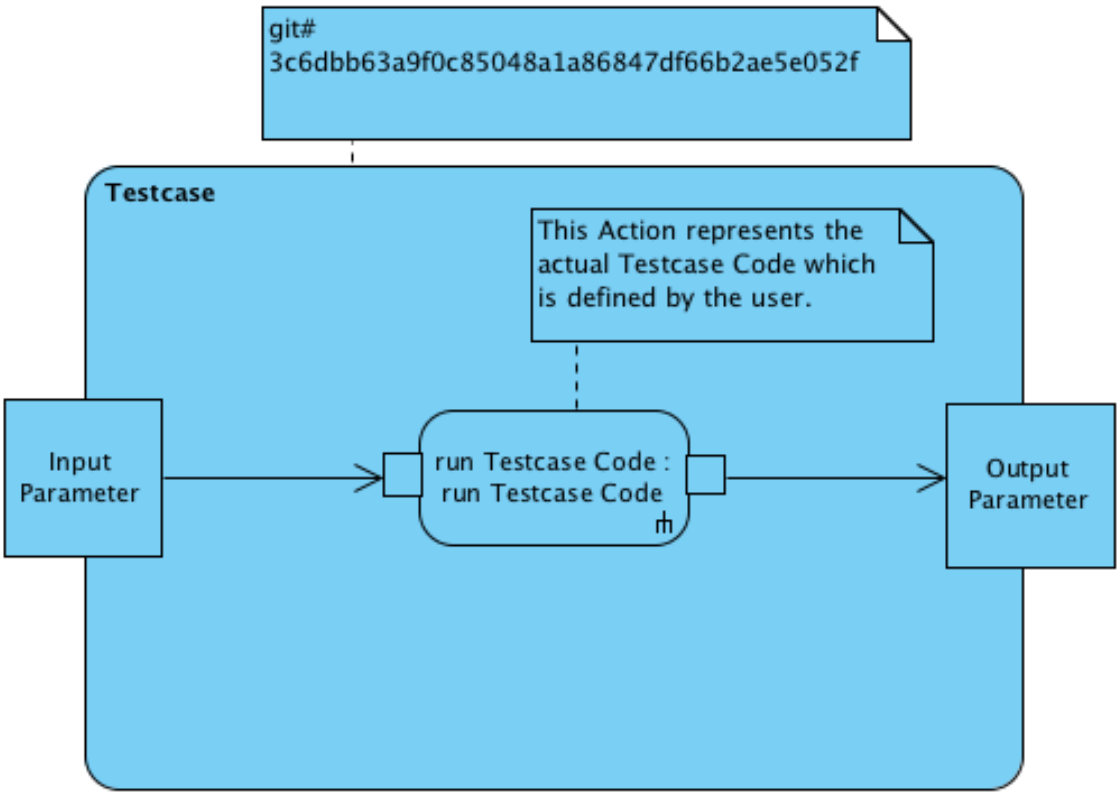

Abbildung 4.5: Testcase - Activity Diagram

Um präzise Ergebnisse zu erhalten, sollten die Testfälle möglichst atomar gehalten werden. Bei der Zeitmessung ist der Testfall die kleinste Auflösung, es lässt sich also bei komplexen Testfällen keine Aussage darüber treffen, wie sich die für die Ausführung des Testfalls gemessene Zeit aufteilt.

Testfall ausführen

Um einen Testfall auszuführen, müssen bestimmte Voraussetzungen geschaffen werden. Dafür wird die Methode `Testcase.runTest()` verwendet, die in Abb. 4.6 dargestellt ist. Die Action *executeTestfunction* stellt den eigentlichen Aufruf des Testfallcodes dar, wie er in Abb. 4.5 vorgestellt wurde.

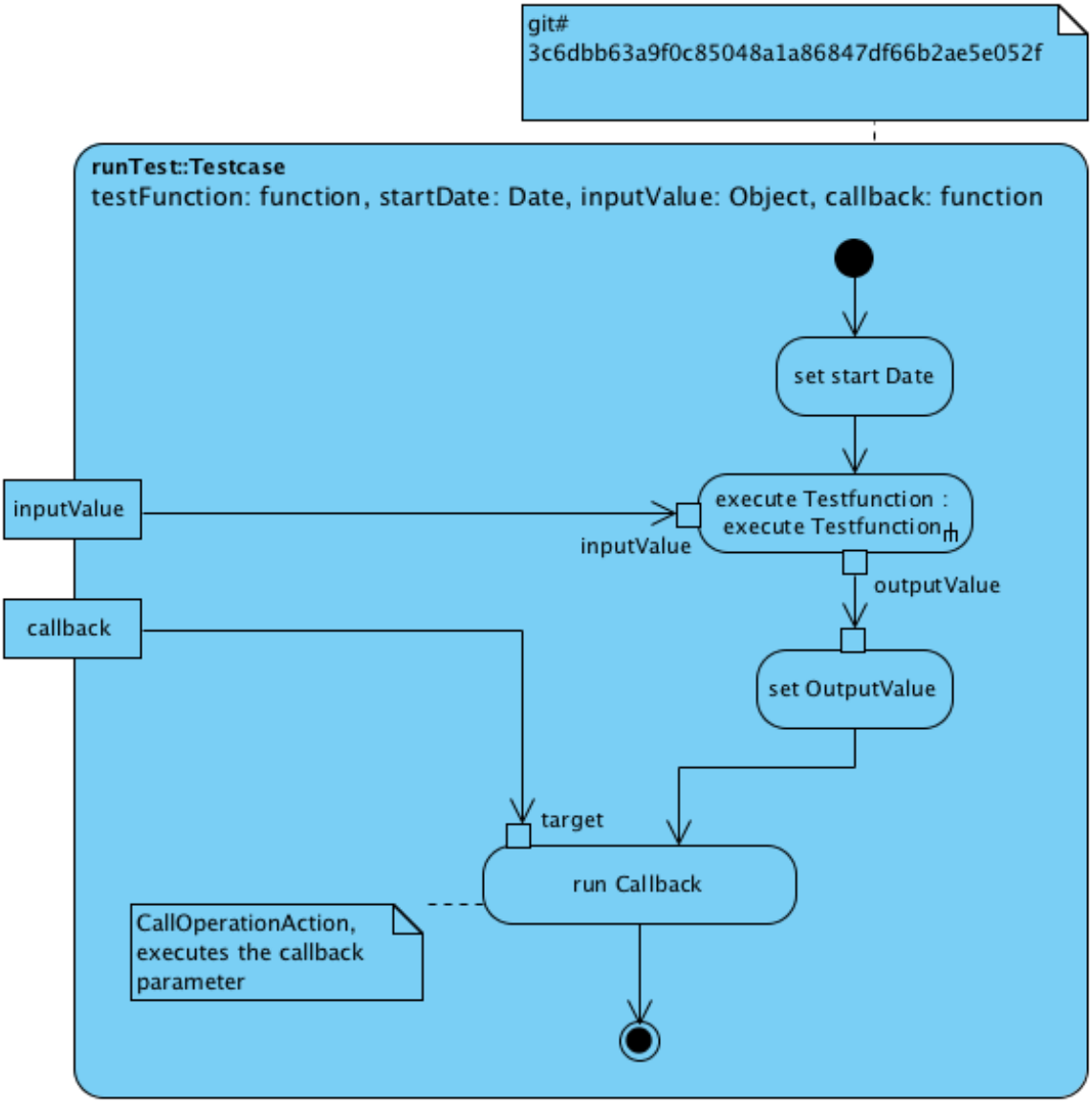

Abbildung 4.6: Run Testcase - Actitity Diagram

Testsuite

Die Testsuite kapselt mehrere Testfälle, die zeitlich nacheinander ablaufen sollen, und ist für die Messung der Ablaufzeit der einzelnen Testfälle verantwortlich. Mess- und Rückgabewerte laufen hier zusammen und können nach Abschluss der Suite weiterverarbeitet werden. Außerdem kann evtl. vorhandener *preTest-Code* vor jedem Testfall ausgeführt werden. Falls ein Testfall einen Timeout auslöst, wird die Testsuite beendet, da der nachfolgende Testfall nicht mehr ausgeführt werden kann (die Rückgabewerte des vorherigen Testfalls stehen nicht zur Verfügung).

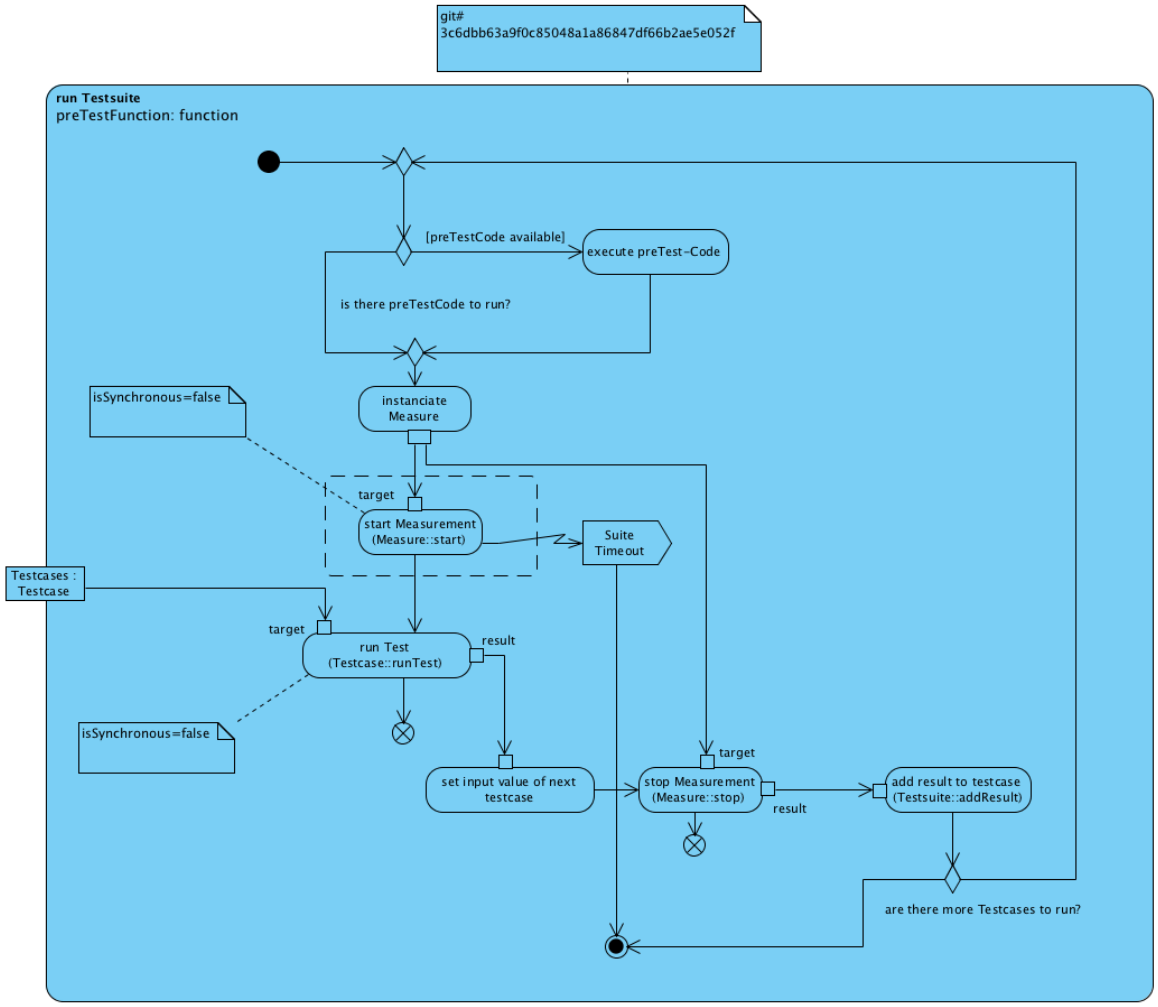

Abbildung 4.7: Testsuite - Actitity Diagram

Testablauf

Ein Testablauf kapselt die Ausführung mehrerer Testsuiten. Eine Testsuite simuliert dabei jeweils einen Benutzer, ein Testablauf mit 100 parallelen Testsuiten simuliert also 100 Benutzer. Die Klasse `Wsload()` ist für diesen Ablauf zuständig (siehe 4.4) und lässt sich beliebig oft instanziieren, um verschiedene Testsuiten gleichzeitig ausführen zu können. Abb. 4.8 zeigt den Programmablauf, um eine Testsuite zu starten und das Ergebnis der gesamten Suite einer Instanz der Klasse `Logger()` zu übergeben (vgl. Klassendiagramm Abb. 4.4).

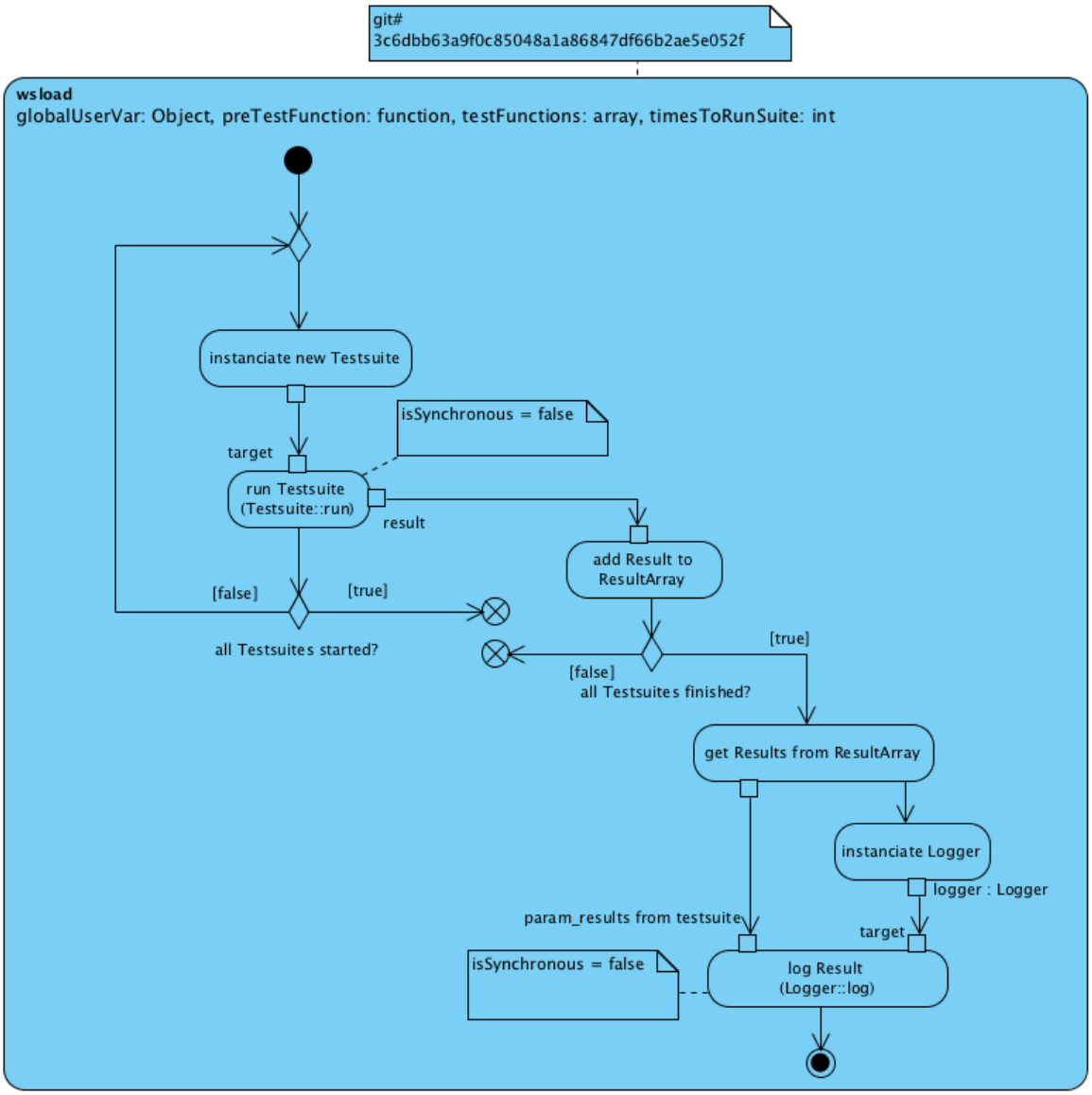

Abbildung 4.8: Testablauf - Actitity Diagram

4.2 Ermittlung der Ergebnisse

Die Ergebnisermittlung beinhaltet zwei Komponenten, nämlich die Messung der Ablaufzeiten und deren Auswertungsanalyse.

Zeitmessung Testfall

Eine Hauptaufgabe der Anwendung ist es, die gemessene Zeit für die Ausführung eines Testfalls zu ermitteln. Dabei wird ein einfaches Verfahren benutzt, das die Differenz der Zeit vor und nach dem Testfall nutzt, um das Ergebnis zu erhalten. Außerdem muss im Falle eines Timeouts, also wenn der Testfall einen zu langen Zeitraum zur Beendigung benötigt, die Messung beendet werden.

Die Klasse `Measure()` (vgl. Klassendiagramm Abb. 4.4) bietet die Möglichkeit, diese asynchrone Zeitmessung durchzuführen. Die dem Testfall übergeordnete Testsuite startet und beendet die Messung. Deren Verwendung ist in Abb. 4.7 dargestellt.

Auswertungsanalyse

Im Sinne des Seperation Of Concerns obliegt die Verantwortung für die Analyse der gewonnenen Daten nicht der Anwendung. Um auf die Ergebnisse zugreifen zu können, kann die Klasse `Logger()` (vgl. Klassendiagramm Abb. 4.4), genauer die Methode `Logger.get()` verwendet werden, die den eindeutigen Bezeichner eines Testablaufs als Parameter übernimmt und alle aufgenommenen Daten für die etwaige weitere Verarbeitung zurückliefert.

Soll der Zugriff auf die Auswertungsdaten aus einer anderen Anwendung erfolgen, kann über die externen Schnittstellen auf die Daten zugegriffen werden, was in Kap. 4.3 beschrieben ist.

4.3 Schnittstellen

Das Kapitel Schnittstellen erklärt, wie die Testergebnisse veröffentlicht werden und die Kommunikation zwischen dem Testsystem und dem zu testenden System stattfindet.

Persistenter Speicher

Abb. 4.8 zeigt die Ausführung quasi-paralleler Testsuiten. Sind alle Testsuiten abgeschlossen oder durch einen Timeout beendet, wird die Aktivität *log Result* gestartet, um die Ergebnisse persistent zu speichern. Abb. 4.9 zeigt den relevanten Ausschnitt aus dem Klassendiagramm (Abb. 4.4).

Im weiteren Verlauf der Entwicklung wurde für die persistente Speicherung eine Datenbank gewählt. Die Datenbankverbindung wird zu Beginn des Testablaufs geöffnet. Bei einer sehr kurzen Testdauer kann es passieren, dass diese noch nicht verfügbar ist, wenn der Testablauf bereits abgeschlossen ist. Tritt dieser Fall auf, wird die Log-Nachricht verzögert und erneut versendet. Schlägt auch der zweite Versuch fehl, wird die Konsole der Anwendung für die Ausgabe der Testdaten verwendet. Des Weiteren kann über die Methode `get()` auf alle in der Datenbank vorhandenen Testergebnisse zugegriffen werden.

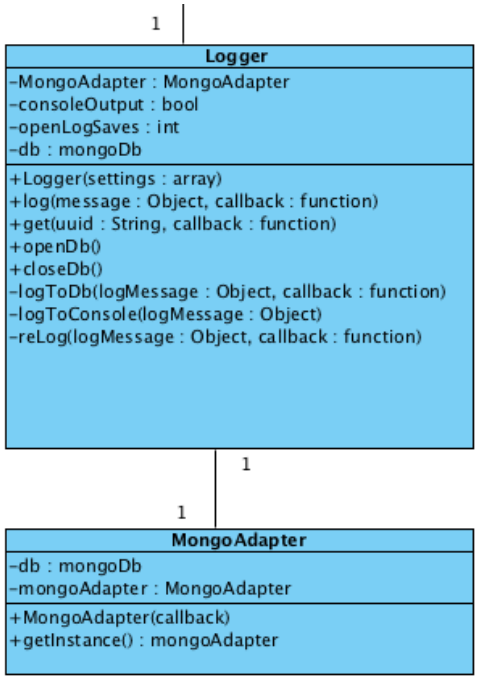

Abbildung 4.9: Logger - Klassendiagramm

WebSocket

Der Hauptnutzen der Software soll im Test von Webservices mit WebSocket-API liegen. Das WebSocket-Protokoll ist vom IETF definiert[3], während die JavaScript-API im Rahmen der HTML5-Entwicklung von der W3C definiert wurde[4][9].

Für die Verbindung vom Client zum Server über das WebSocket-Protokoll mit Hilfe der JavaScript-API ist ein eindeutiger Bezeichner (URI) notwendig. Dieser hat z. B. folgende Form: `ws://echo.christianvogt.de/`. Nun kann über den Konstruktor `WebSocket()` eine Verbindung aufgebaut werden. Danach lassen sich Event Handler zuweisen, mit denen sich die Socket-Verbindung nutzen lässt. Die Verbindung zu einem Beispielhaften „Echo“-Dienst ist in Abb. 4.10 dargestellt.

```
1 var uri = "ws://echo.christianvogt.de"; //define endpoint
2 var websocket = new Websocket(uri); //open a websocket connection to the endpoint
3 websocket.onopen = function (evt) { //add handler for open event
4   console.log('connected to: ' + uri); //print a success message on the client
5 };
6 websocket.onmessage = function (evt) { //add handler for incoming messages event
7   websocket.send(evt.data); //return whatever message reaches the client
8 };
```

Abbildung 4.10: Aufbau einer WebSocket-Verbindung

Entscheidend bei der parallelen Verwendung von mehreren WebSocket-Verbindungen ist, dass für jede Testsuite tatsächlich ein neuer Socket erstellt wird. Ein Caching (also die Wiederverwendung der Sockets im Programmablauf) außerhalb einer Testsuite würde zu einer Ergebnisverfälschung führen, da ein realer Benutzer auch eine dedizierte Socketverbindung verwendet.

[3] in der RFC6455
[4] nach http://dev.w3.org/html5/websockets/

5 Realisierung

In diesem Kapitel werden wichtige Aspekte der Realisierung untersucht. Kapitel 5.1 beschäftigt sich mit der Wahl des Frameworks, das für die Realisierung verwendet werden soll, während Kapitel 5.2 genauer auf Realisierungsfragen eingeht, die sich mit der Wahl des Frameworks stellen.

5.1 Wahl der Programmierumgebung

Die Hauptaufgabe der Zeitmessung eines vorher programmierten Testablaufs lassen sich in vielen modernen Programmiersprachen bzw. Frameworks umsetzen. Letztlich fiel die Wahl auf Node.js[1] vom Hersteller Joyent, dessen Framework in den letzten Jahren immer wieder als „Buzzword“ in der Presse zu finden war und dem eine hohe Leistung in der parallelen Verarbeitung von vielen Verbindungen nachgesagt wird.

Die IETF hat erst im Dezember 2011[2] die finale Version des WebSocket-Protokolls vorgelegt. Da die Implementierung in anderen Programmiersprachen zwar durchaus möglich wäre, aber den Rahmen dieser Thesis sprengen würde, beschränkt sich der Vergleich der Programmierumgebungen auf diejenigen, die derzeit eine vollständige Implementation anbieten. Eine Recherche ergibt einige Frameworks bzw. Programmierumgebungen (siehe Tabelle 5.1).

Framework	WebSocket-Implementierung	Programmiersprache
-	jWebSocket	Java
-	pywebsocket	Python
node.js	socket.io, native	JavaScript

Tabelle 5.1: Vergleich der WebSocket-Implementierungen

Die meisten Entwickler, die einen Webservice bzw. eine WebSocket-API testen, haben bereits Erfahrungen mit JavaScript gemacht. Aus der Web-Entwicklung ist die Skriptsprache

[1] http://nodejs.org
[2] http://www.rfc-editor.org/rfc/rfc6455.txt

mit dem Sprachkern ECMAScript nicht mehr wegzudenken. Selbst Microsoft bietet in seiner neuesten Version von Windows bzw. deren Oberfläche Metro die Möglichkeit, Anwendungen mit HTML5 und JavaScript zu erstellen[3], was die weite Verbreitung der Skriptsprache belegt.

Node.js ist ein Framework für netzwerkfähige Applikationen, die besondere Anforderungen an Skalierbarkeit und Performance haben. Es basiert auf der JavaScript Runtime des Chrome-Browsers von Google und verwendet ein Event-getriebenes, nicht-blockierendes I/O-Modell. Hierbei wird bei Ein/Ausgabeoperationen (z. B. Datenbankzugriffen) der Programmablauf nicht blockiert, sondern eine Rückruffunktion (Callback) übermittellt, die ausgeführt wird, sobald das Ergebnis zur Verfügung steht. Dieses Verfahren hat besonders bei einer hohen Anzahl von parallelen Vorgängen (Anfragen) Vorteile, da Verbindungen nicht aufrechterhalten werden müssen, z. B. mit parallelen Threads. Node.js bzw. die Chrome Runtime übersetzt die in JavaScript programmierten Anwendungen vor der Ausführung in Maschinencode, was zu einer weiteren Erhöhung der Ausführungsgeschwindigkeit führt.

Node.js ist derzeit in der Version 0.8.1 verfügbar, wird ständig weiterentwickelt und zum Beispiel bei LinkedIn, Yahoo oder eBay produktiv eingesetzt. Es lässt sich mittels des Paketmanagers *npm* mit Modulen erweitern. Die Anwendung, die im Rahmen dieser Thesis entsteht, soll ebenfalls über den Paketmanager *npm* verteilt werden.

5.2 Umsetzung des Designs

Die Umsetzung der Anwendung erfolgt nach den UML-Diagrammen, die im Kapitel 4 diskutiert wurden. Auf Grundlage dieser Diagramme lässt sich die Anwendung fast vollständig umsetzten. Der Fokus in diesem Kapitel liegt daher auf umgebungsabhängigen Designentscheidungen, die sich durch die Wahl des Frameworks ergeben. Dazu gehören die Datenbankanbindung für die persistente Speicherung und der Clusterbetrieb, der die Anwendungslast auf mehrere Prozessorkerne verteilt.

Datenbankanbindung

Es bestehen keine komplexen Relationen zwischen den aufzunehmenden Daten (außer, dass eine Testsuite mehrere Testfälle enthält). Außerdem ist die anfallende Datenmenge gering, was anhand eines Beispiels aufgezeigt wird (siehe Tabelle 5.2). Bei einem typischen Testablauf fallen ca. 3 MB an Daten an. Diese Größenordnung lässt sich gut im Arbeitsspeicher des Testsystems ablegen, der im Vergleich zum Festspeicher eine deutlich höhere Lese/Schreibleistung ermöglicht.

[3] http://msdn.microsoft.com/de-de/library/windows/apps/br211385.aspx

Speicherobjekt	Datentyp	typische Größe
Ergebnis eines Testfalls	String	100 B
Einzelne Testsuite	Array[Testfall]	$30 * 100\,\text{kB} = 3\,\text{kB}$
Gesamter Testablauf	Array[Testsuite]	$1000 * 3\,\text{kB} = 3\,\text{MB}$

Tabelle 5.2: Anfallende Datenmenge bei einem Testablauf mit 1000 parallelen Suiten und jeweils 30 Testfällen

Die NoSQL Datenbank MongoDB erfüllt alle Kriterien, um als persistenter Speicher für die aufzunehmenden Daten zu fungieren (vgl. Kap. 3) und ist darüber hinaus sowohl unter Linux, Windows und Mac OS X lauffähig. Die gute Unterstützung für Node.js mit dem über npm installierbaren Paket *mongodb* und die fast konfigurationslose Installation sind weitere Faktoren, die für diese Lösung sprechen. Des Weiteren stehen für diverse Programmiersprachen offizielle Treiber zur Verfügung[4]. Für den Zugriff auf die Datenbank wird die Klasse `Logger`, die in Kapitel 4.3 vorgestellt wurde, um einen Adapter erweitert, der den Datenbankzugriff kapselt (siehe Abb. 4.9, `MongoAdapter`).

Die Ergebnisse werden erst in die Datenbank kopiert, wenn die jeweilige Testsuite abgeschlossen ist (also genau 1x pro Testsuite). Dadurch wird verhindert, dass eine zu hohe Anzahl an Schreibzugriffen das Testsystem ausbremst. Die eigentliche Datenbankverbindung wird während des Programmablaufs nur einmal geöffnet, um den Overhead durch mehrere Verbindungen einzusparen (vgl. Abb. 4.8 Action *Log Result*), was allerdings zur Folge hat, dass ein etwaiger Absturz der Anwendung während des Betriebs den Verlust der jeweiligen Testdaten bedeutet.

```
1 var mongo = require('mongodb');
2 var db = new mongo.Db('test', new mongo.Server('localhost',mongo.Connection.DEFAULT_PORT, {}), {});
3 db.open(function(err, db) {
4   if (err) {
5       throw Error("Error - cannot open Database");
6     }
7 });
```

Abbildung 5.1: Verbindung zu einer MongoDB-Instanz aus Node.js

Abb. 5.1 zeigt einen Verbindungsaufbau aus einem Node.js-Projekt, der über den nativen Treiber *mongodb* gestartet wird. Nach der Verbindungsöffnung können Abfragen formuliert werden, wie in Abb. 5.2 dargestellt. Diese Abfragen bilden die Grundlage für die Auswertungsanalyse, auf die später im Kapitel 6.2 eingegangen wird.

[4] http://www.mongodb.org/downloads#Drivers

```
db.collection('suites', function(err, collection){
  if(err) {
    throw err;
  }
  var cursor = collection.find({uuid:param_uuid});
  var results = []; //holds the data
  cursor.each(function (err,doc){
    if(err) {
      throw err;
    }
    results.push(doc);
  });
```

Abbildung 5.2: Ausführung einer einfachen MongoDB-Abfrage

Node.js Cluster

Die JavaScript Runtime des Chrome-Browsers, die auch in *Node.js* zum Einsatz kommt, ist „single-threaded“, das heißt, sie verwendet für die Ausführung der Anwendung also nur einen Thread und somit einen Prozessorkern des Hosts, auf dem die Anwendung ausgeführt wird. Um die Leistungsfähigkeit von modernen Mehrkernprozessoren zu nutzen, muss das Modul *Cluster* verwendet werden, mit dem mehrere Prozessinstanzen der Anwendung abgespalten werden können.

Durch den modularen Aufbau der Anwendung lässt sich ein Testablauf bei der Verwendung des Cluster-Moduls aufteilen, und jeweils ein Teil der auszuführenden Testsuiten an einen Cluster-Prozess weitergeben. Die Anzahl der Worker-Prozesse entspricht immer der Anzahl der Prozessorkerne des Systems. Die entsprechende Funktionalität findet sich in der Methode `Wsload._spawnWorker()`. Der Bereich, für den die einzelnen Worker zuständig sind, lässt sich über eine Berechnung bestimmen (siehe Abb. 5.3), sodass unabhängig von der Anzahl der Worker bzw. Prozessorkerne genau die Anzahl an Suiten ausgeführt wird, die vorgegeben ist.

```
//calculate Testsuite start# for this worker, check github issue#2
var start = (workerId * timesToRunSuite)/workerCount;
start = Math.round(start);

//calculate Testsuite stop# for this worker, check github issue#2
var stop = (((workerId+1) * timesToRunSuite)/workerCount) - 1;
stop = Math.round(stop);
```

Abbildung 5.3: Aufteilung eines Testablaufs auf mehrere Worker-Prozesse

6 Verwendung

Das Ergebnis der Realisierung aus Kapitel 5 ist die Anwendung *wsload*. Der Einsatz und die Möglichkeiten, die sich aus der Verwendung der Anwendung ergeben, werden in diesem Kapitel vorgestellt.

Dazu wird in Kapitel 6.1 die Installation beschrieben, ein einfacher Test durchgeführt und die Testergebnisse dargestellt. Danach wird in Kapitel 6.2 theoretisch besprochen, wie diese Ergebnisse interpretiert werden können. Im Kapitel 6.3 werden die theoretischen Erkenntnisse praktisch angewendet und gezeigt, wie Testfälle modelliert werden können.

6.1 Verwendung von wsload

Dieses Kapitel zeigt, wie die Anwendung eingesetzt werden kann und welche Voraussetzungen dafür erfüllt sein müssen.

Voraussetzungen

Die Anforderungen für die Anwendungsinstallation (siehe Kapitel 3) besagen, dass der Zeitraum von Installation bis zur Verwendung der Anwendung möglichst gering sein soll.

Die Voraussetzungen, die das Testsystem erfüllen muss, beinhalten die Installation von *Node.js* in einer Version >=0.8 mit dem dazugehörigen Paketmanager *npm* und der NoSQL-Datenbank MongoDB, wie Tabelle 6.1 darstellt. Die Anwendung sowie alle Komponenten sind unter jedem aktuellen Linux oder Mac OS X lauffähig. Theoretisch ist auch eine

Komponente	Name	Version
JavaScript Framework	Node.js	>=0.8.0
Node.js Paketmanager	npm	mit Node.js mitgeliefert
NoSQL Datenbank	MongoDB	>=2.0.6

Tabelle 6.1: Voraussetzungen für wsload

Windows-Unterstützung gegeben, diese ist unter *Node.js* allerdings noch nicht vollständig ausgereift.

Installation

Die Installation der Anwendung erfolgt über den Paketmanager *npm*, was in Abb. 6.1 dargestellt ist. Um die Umgebung zu testen, kann *npm* auch die Unit-Tests der Anwendung ausführen.

```
npm install wsload
npm test wsload
```

Abbildung 6.1: Installation von wsload über npm

Erstellung eines Testablaufs

Abb. 6.2 zeigt einen kompletten Testablauf, der einige sehr simple Operationen durchführt.

```
var Wsload = require('wsload');

var testcase1 = function add (input, callback, global) {
  var k = 1+2;
  callback(null,k); //set k as output value
}
testcase1.timeout = 15; //set testcase timeout to 15ms


var testcase2 = function subtract (input, callback, global) {
  var i = input;  //matches k from testcase1 (=3)
  var j = i-1;
  setTimeout(function(){ //execute asynchronously after 100ms
    callback(null,j); //output value is 2
  },100);

}
testcase2.timeout = 110; //set testcase timeout to 110ms


var wsload = new Wsload({logTarget:'console'});
//Parameter: suiteName, suiteRuns, testcases, preTestFunc, suiteTimeout, global object
wsload.runSuite('myFirstSuite', 2, [testcase1,testcase2] ,null,     100,    null);
```

Abbildung 6.2: Kompletter Testablauf mit 2 Testfällen und 2 parallelen Durchläufen

Das Ergebnis des Testlaufs, das in der Datenbank persistent gespeichert wird, ist in Abb. 6.3 abgebildet. Den Suiten wurde eine identische UUID zugewiesen, mit der zu einem späteren

Zeitpunkt auf diese Daten zugegriffen werden kann. Es ist zu erkennen, dass ein **Suite-Timeout** ausgelöst wurde, da das boolsche Attribut `timeoutOccured` den Wert `true` trägt, während die Testfälle eine geringere `runDurationInMs` aufweisen, als der jeweilige `timeout`-Wert groß ist.

```
[ { uuid: '77dd7e4d-0654-4f93-b925-d4e4a454ba8b',
    suiteNumber: 1,
    suiteName: 'myFirstSuite',
    createdOn: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
    runDurationInMs: 105,
    timeoutInMs: 100,
    timeoutOccured: true,
    testcases:
     [ { testnr: 0,
         inputValue: 'null',
         outputValue: '3',
         runDurationInMs: 1,
         startDate: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
         timeout: 15 },
       { testnr: 1,
         inputValue: '3',
         outputValue: '2',
         runDurationInMs: 104,
         startDate: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
         timeout: 110 } ] } ]
[ { uuid: '77dd7e4d-0654-4f93-b925-d4e4a454ba8b',
    suiteNumber: 0,
    suiteName: 'myFirstSuite',
    createdOn: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
    runDurationInMs: 122,
    timeoutInMs: 100,
    timeoutOccured: true,
    testcases:
     [ { testnr: 0,
         inputValue: 'null',
         outputValue: '3',
         runDurationInMs: 1,
         startDate: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
         timeout: 15 },
       { testnr: 1,
         inputValue: '3',
         outputValue: '2',
         runDurationInMs: 121,
         startDate: Wed Aug 08 2012 10:56:41 GMT+0200 (CEST),
         timeout: 110 } ] } ]
```

Abbildung 6.3: JSON-kodiertes Ergebnis eines Testablaufs in der Datenbank

6.2 Analysemöglichkeiten

Nach der Erstellung eines Testablaufs und der Möglichkeit, auf die Testdaten zuzugreifen, können verschiedenste Analysen erstellt werden, die die Daten interpretieren. Einige dieser Analysen werden in diesem Kapitel vorgestellt, wobei der Fokus der Auswertung auf der Beantwortung folgender Fragen liegt, die sich auf das zu testende System beziehen [10]:

- Wie ändert sich das Antwortzeitverhalten in Abhängigkeit von der Last?
- Kann mit dem System auch unter hoher Last noch akzeptabel gearbeitet werden?
- Zeigt das System undefiniertes Verhalten (z. B. Absturz)?
- Kommt es zu Dateninkonsistenz?
- Geht das System nach Rückgang der Überlast wieder in den normalen Bereich zurück?

Für die Analyse werden Daten aufbereitet, die aus verschiedenen Testabläufen gewonnen und in der Datenbank gespeichert wurden. Die für die Analysen notwendigen Datenbankabfragen wurden auf Grund ihrer Kompaktheit und der besseren Zuordnungsfähigkeit am Ende jeder Abfrage eingefügt.

Vergleich der Ablaufzeit von Testsuiten

Die Gesamtablaufdauer der Suiten (Abb. 6.4) lassen sich gut vergleichen und bieten einen ersten Überblick über das Zeitverhalten der Anwendung. Dabei ist nicht unbedingt die Gesamtlaufdauer der jeweiligen Testsuite von Interesse, da keine Aussage über die enthaltenen Testfälle gemacht wird. Entscheidend sind der Vergleich der Ablaufzeiten und etwaige Differenzen zueinander.

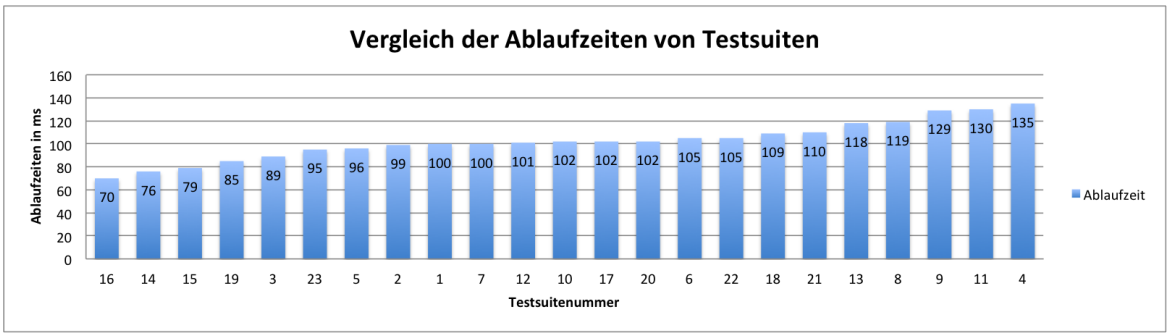

Abbildung 6.4: Vergleich der Ablaufzeiten von Testsuites

Bei einer größeren Anzahl an Testsuiten bietet es sich an, die auftretenden Häufigkeiten der Ablaufzeit zu vergleichen. Abb. 6.5 zeigt diese Analyse. Die ermittelten Ablaufzeiten wurden in Klassen von je 10 ms eingeteilt, und können sowohl relativ (vgl. Abb. 6.5) als auch absolut (vgl. Abb. 6.6) skaliert dargestellt werden.

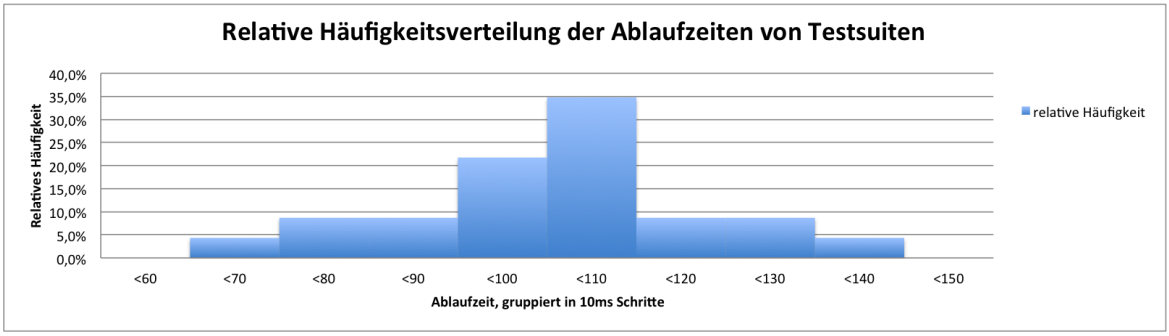

Abbildung 6.5: Relative Häufigkeitsverteilung, Gruppierung in 10 ms

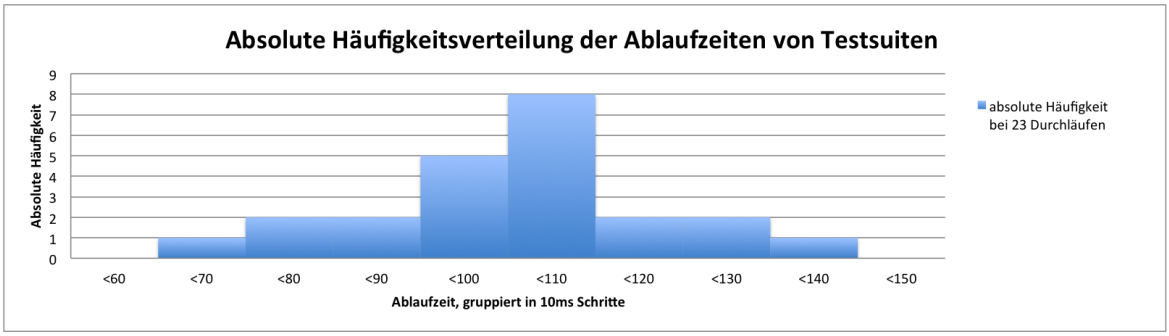

Abbildung 6.6: Absolute Häufigkeitsverteilung, Gruppierung in 10 ms

```
db.suites.find({timeoutOccured:false, uuid:'d5f31827-0205-44e9-93f6-59b93c491354'}, {uuid:'*',
    runDurationInMs:'*', _id:0})
```

Abbildung 6.7: MongoDB-Query - Ablaufzeiten eines Testablaufs

Vergleich der Ablaufzeit von Testfällen

Während die Untersuchung der Testsuite-Ergebnisse einen guten Überblick über das Zeitverhalten des Testsystems verschafft, kann eine Testfallanalyse Ergebnisse über die beinhalteten Testfälle liefern (vgl. Abb. 6.8).

Diese Auswertung ist dafür geeignet, einzelne überproportional schnelle/langsame Ablaufzeiten zu identifizieren. Des Weiteren kann über das Verhältnis von Mittelwert (avg) zu Extrema (min/max) eine Erkenntnis darüber gewonnen werden, ob es „Ausreißer“ gab, also Testfälle mit besonders kurzer bzw. langer Ablaufzeit im Vergleich zu den anderen.

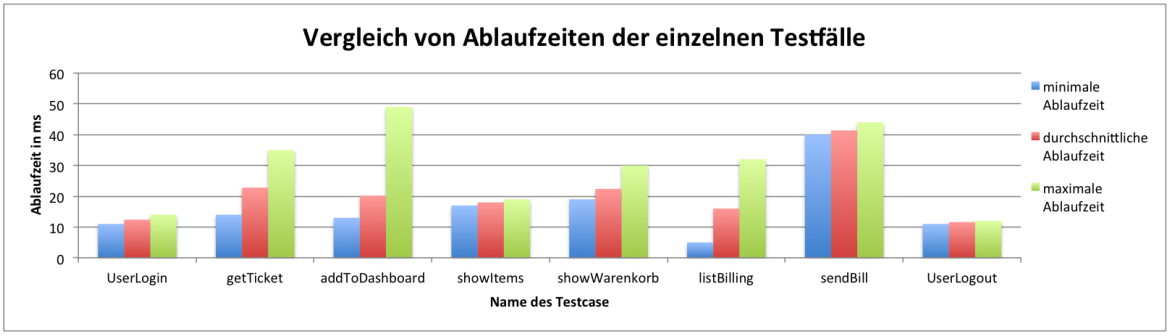

Abbildung 6.8: Minimale/Durchschnittliche/Maximale Ablaufzeit von Testfällen

```
var testname = 'testcasebezeichnung';
db.suites.group({
  cond: {'testcases.testname':testname, $or: [{uuid:'615fcfd1-71a5-440d-b963-293312d00bd1'}]}, //
      array of uuids to use
  keyf: function (doc) {
    var testname = doc.testcases[0].testname;
    return {'testname':testname};
  },
  reduce: function (obj,out) {
    var testcaseDuration = obj.testcases[0].runDurationInMs;
    out.tcount++;
    out.tsum += testcaseDuration;
    if ((out.tmin === 0) || (out.tmin > testcaseDuration)) {
      out.tmin = testcaseDuration;
    };
    if ((out.tmax === 0) || (out.tmax < testcaseDuration)) {
      out.tmax = testcaseDuration;
    };
  },
  initial: {tsum:0, tcount:0, tmin:0, tmax:0},
  finalize: function (out) {
    out.avg = out.tsum/out.tcount;
  }
});
```

Abbildung 6.9: MongoDB-Query - Ablaufzeiten der Testfälle im Vergleich

Darstellung des absoluten zeitlichen Ablaufs von Testfällen

Soll das parallele Verhalten von Testfällen weiter untersucht werden, kann eine Darstellung wie in Abb. 6.10 gewählt werden. Hier wird der absolute Startpunkt des Testablaufs als Referenz verwendet, um die Ausführungszeiten der Testsuiten bzw. der beinhalteten Testfälle darzustellen.

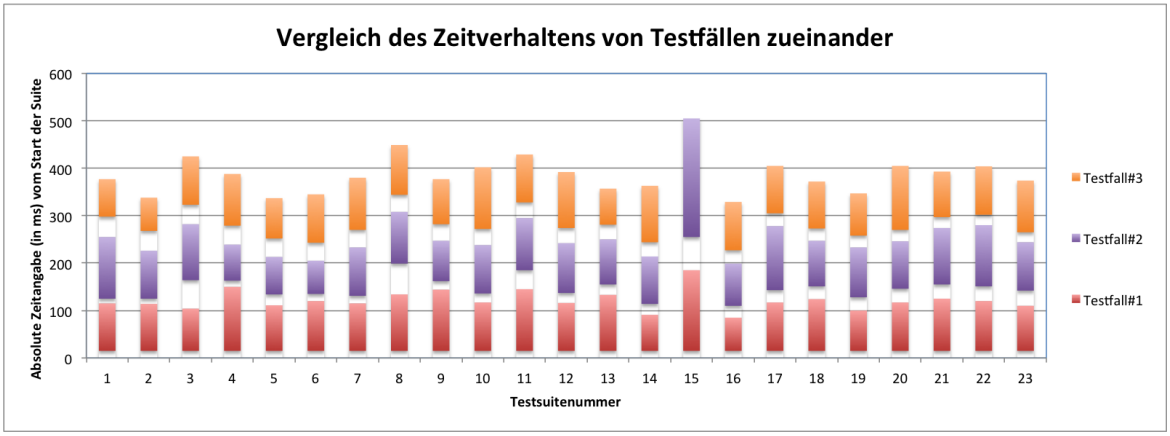

Abbildung 6.10: Parallele Ausführungszeiten der Testfälle

Die Abstände zwischen den „Balken“, die die Testfälle darstellen, stellen die Dauer des preTest-Ablauf dar gedauert hat. In diesem Fall hat Testsuite Nr. 15 für den ersten Testfall im Vergleich zu den anderen Suiten am längsten gebraucht und den dritten Testfall auf Grund eines Timeouts nicht ausgeführt. Die Beispieldaten lassen vermuten, dass es zu einem Problem kommt, wenn der erste Testfall eine überdurchschnittlich lange Ablaufzeit benötigt oder Testfall 2 zeitlich parallel zu Testfall 3 gestartet wird.

Um die parallele Ausführung von verschiedenen Testfällen zu untersuchen, bietet es sich an, in dem ersten „preTest“-Ablauf eine definierte Zeitverzögerung zu verwenden. Dadurch kann jeder Testfall parallel zu anderen Testfällen ausgeführt werden und damit dessen Wirkung aufeinander getestet werden. Das Ergebnis einer solchen Auswertung ist in 6.11 zu sehen.

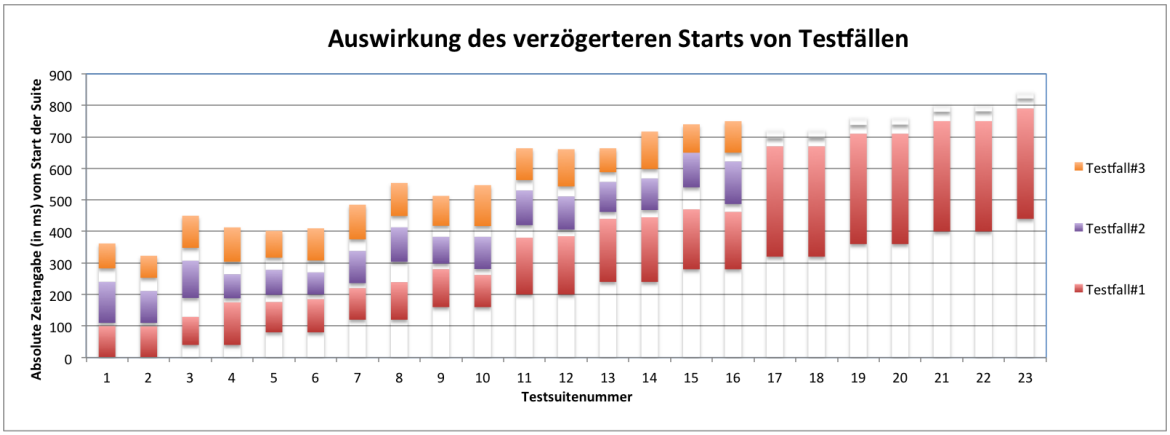

Abbildung 6.11: Auswirkung von verzögertem Start auf die Testsuite

Scheinbar wirkt sich der verzögerte Start stark auf das Verhalten des Testfalls 1 aus, genauer scheint die parallele Ausführung von Testfall 1 und 3 zu einer deutlich erhöhten Laufzeit bzw. zu einem Timeout zu führen.

```
db.suites.find({uuid:'d5f31827-0205-44e9-93f6-59b93c491354'}, {_id:0, 'testcases.testname':'*', '
    testcases.runDurationInMs':'*' , 'testcases.startDate':'*'})
```

Abbildung 6.12: MongoDB-Query - Ablaufzeiten eines Testablaufs

Ermittlung von Benutzergrenzen im Bezug auf die maximal akzeptierte Testlaufzeit

Eine weitere denkbare Auswertung mit den durch die Testabläufe sammelbaren Daten ist die Ermittlung einer User-Anzahl, die einen bestimmten Vorgang parallel ausführen kann, bevor eine maximal akzeptierte Ablaufzeit erreicht wird.

Abb. 6.13 zeigt ein mögliches Ergebnis. Mit linear ansteigender Anzahl der ausgeführten parallelen Testsuiten steigt die Laufzeit exponentiell an (blau markiert). Die rote Gerade zeigt eine fiktive Grenze, die die maximale Laufzeit repräsentiert. Diese obere Grenze sollte bereits aus der Anforderungsanalyse der jeweiligen Anwendung bekannt sein, hängt allerdings auch stark von den enthaltenen Testfällen ab.

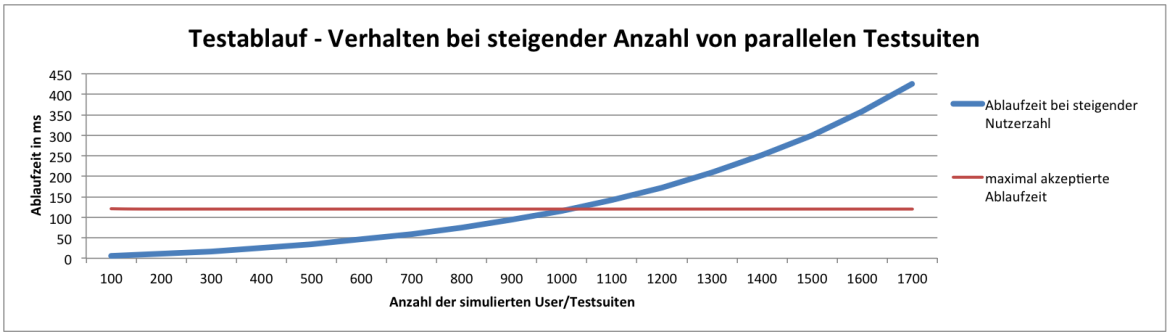

Abbildung 6.13: Ermittlung der maximalen Benutzerzahl

```
db.suites.group({
  cond: {$or: [{uuid:'d5f31827-0205-44e9-93f6-59b93c491354'}, {uuid:'32c2f134-8fd2-47f6-b902-120
      e46a2d998'}]}, //array of uuids to use
  key: {uuid:true},
  reduce: function (obj,out) {
    out.tcount++;
    out.tsum += obj.runDurationInMs;
  },
  initial: {tsum:0, tcount:0},
  finalize: function (out) {
    out.avg = out.tsum/out.tcount;
  }
});
```

Abbildung 6.14: MongoDB-Query - Ablaufzeiten mehrerer Testabläufe

6.3 Lastanalyse einer Anwendung

Um die in Kapitel 6.2 aufgestellten theoretischen Analysen anzuwenden, bedarf es einer testfähigen Anwendung. Diese wird hier kurz vorgestellt und danach auf ihr Lastverhalten untersucht. Daraufhin wird der Testaufbau geändert und der Testablauf von mehreren unabhängigen Hosts gestartet und die Ergebnisse verglichen (Kapitel 6.4), um die Auswirkungen eines verteilten Aufbaus zu erkennen.

An diesem praktischen Beispiel wird aufgezeigt, wo sich die Software einsetzten lässt und wie sinnvolle Aussagen zum Lastverhalten getroffen werden können.

Anwendungsvorstellung

Basierend auf einer socket.io[1] API bietet die sich in der Entwicklung befindende Anwendung einem User die Möglichkeit, ein Fußball-Bundesligateam zu managen. Die Hauptmodell-Elemente sind hier kurz aufgeführt.

- *Player* ist die Repräsentation eines Fußball-Spielers, zum Beispiel „Marcel Jansen“
- Ein *Team* ist die Analogie zu einem Bundesliga-Verein, zum Beispiel dem „Hamburger SV“
- *User* sind menschliche Spieler, die den Dienst nutzen
- *UserTeam* ist eine Menge von *Player* und das vom *User* gemanagte Team.

Die Spielmechanik wird hier kurz erklärt, um die Testanalyse zu planen:

- Ein *Player* ist einem *Team* fest zugeordnet (z. B. „Marcel Jansen“→„Hamburger SV“).
- Ein *UserTeam* ist einem *User* fest zugeordnet.
- Ein *User* kann einen *Player* kaufen, der dann seinem *UserTeam* zugeordnet wird.
- Dabei hat ein *Player* einen Wert und ein *User* ein Budget. Wird ein *Player* von einem *User* gekauft, wird sein Wert vom Budget des *Users* abgezogen.
- Der *User* kann nur *Player* kaufen, wenn der Wert dieses *Players* das Budget des *Users* nicht übersteigt.

[1] socket.io ist eine WebSocket-Implementierung mit Fallback auf andere Protokolle, wenn der Browser keine WebSocket-Unterstüzung bietet.

Will ein Benutzer den Webservice verwenden, muss er sich zuerst authentifizieren. Dafür stehen verschiedene Varianten zur Verfügung, die über ein Framework[2] serverseitig verarbeitet werden. Ist der Authentifizierungsvorgang erfolgreich, wird dem Client eine verschlüsselte SessionID überlassen, mit der bestimmte API-Funktionen aufgerufen werden können. Abb. 6.15 zeigt eine Auswahl dieser öffentlichen API.

```
1 findUser : function (param_user, param_auth, param_cb)
2 findPlayer : function (param_player, param_auth, param_cb)
3 findTeamByPlayer : function (param_player, param_auth, param_cb)
4 findPlayersByNameReturnsArray : function (param_name, param_auth, param_cb)
5 findTeamByName : function (param_name, param_auth, param_cb)
6 buyPlayer : function (param_player, param_auth, param_cb)
7 sellPlayer : function (param_player, param_auth, param_cb)
```

Abbildung 6.15: Öffentliche fLeague API

Analyse einer Anwendung

Um eine Analyse der Anwendung zu ermöglichen, müssen Testfälle erstellt werden. Für die Erstellung dieser Testfälle und die später erfolgende Zusammenfassung von Testfällen zu Testsuiten, sind verschiedene Faktoren zu beachten, da nur so ein aussagekräftiges Ergebnis erzielt werden kann.

Die Testerstellung lässt sich mit der eines Unit-Tests vergleichen. Wie bei den Unit-Tests entscheiden viele Faktoren darüber, wie die Testfälle und Testsuiten modelliert werden sollten, um einen Nutzen aus den durch die Anwendung generierten Daten ziehen zu können.

Für eine Lastanalyse muss im Vorfeld untersucht werden, wie sich diese Last im Produktivbetrieb verteilt. Bei der vorgestellten Anwendung (vgl. Kap. 6.3) kann beispielsweise festgestellt werden, dass die API-Funktion bei einem typischen Anwendungsfall `fLeague.buyPlayer()` deutlich seltener genutzt wird, als `fLeague.findPlayer()`, da diese mehrmals aufgerufen wird, wenn ein Kaufvorgang ausgeführt wird. Solche Informationen lassen sich z. B. aus Anwendungsfalldiagrammen entnehmen. Weitere Faktoren sind vielleicht in Bereichen der Anforderungsanalyse aufzufinden, z. B. die Anzahl der User, die das System aufnehmen soll bzw. ob oder wie es mit einer steigenden Anzahl an Benutzern skalieren soll.

Des Weiteren muss sich der Testersteller darüber im Klaren sein, dass eine Lastanalyse unter Umständen nur auf einen Fehler hinweist, allerdings nicht aufzeigen kann, an welcher Stelle dieser im System auftritt. Das liegt daran, dass nur die API angesprochen wird und nicht das hinter der API stehende System. Beispielsweise kann der API ein Load-Balancer

[2] passport.js - OpenID/OAuth basiertes Single Sign-On Framework

vorgeschaltet sein. Das führt dazu, dass zwar trotzdem etwaige lastspezifische Fehler gefunden werden können, diese aber nicht direkt der Fehlerquelle zuordenbar sind. Dieses Verhalten wird ausgeprägter, je mehr Systeme zusammenarbeiten. Mit größerer Verteilung des Systems gestaltet sich eine Fehlersuche also zunehmend schwieriger.

Um diese Schwierigkeiten zu begrenzen, sollte bei stark verteilten Systemen im Entwicklungsprozess so früh wie möglich ein Lasttest eingebunden werden. Dieser könnte automatisiert erfolgen und den Entwickler bei Fehlern benachrichtigen. Tritt ein solcher Fehler auf, kann der Entwickler die Fehlerquelle auf den Abschnitt beschränken, den er seit der letzten Ausführung bearbeitet hat.

Im Rahmen dieses Kapitels soll ein beispielhafter Testablauf für die Anwendung *fLeague* im Hinblick auf die oben genannten Faktoren erstellt werden.

Testmodellierung

Die Testmodellierung teilt sich in zwei Vorgänge, die Testfallmodellierung und die Testsuitemodellierung, die hier an einem praktischen Beispiel vorgenommen werden. Ähnlich wie bei funktionalen Tests, ist jede Modellierung abhängig von der zu testenden Anwendung. Damit aussagekräftige Ergebnisse erhoben werden können, muss ein Konzept erstellt werden.

Unterschiede gibt es vor allem in der verwendeten API, der voraussichtlichen User-Anzahl und der Struktur der Testfälle und -suiten. Dieses Kapitel zeigt eine beispielhafte Konzeption eines Testablaufs. Sind bei der Konzeption der zu testenden Anwendung UML Anwendungsfalldiagramme (engl. use case diagram) zum Einsatz gekommen, lassen sich diese sehr gut als Vorlage für Tests verwenden, da sie eine Sicht von außen auf das System darstellen. Ein Anwendungsfalldiagramm der Anwendung *fLeague*, von dem der Testablauf abgeleitet wird, ist in Abb. 6.16 abgebildet.

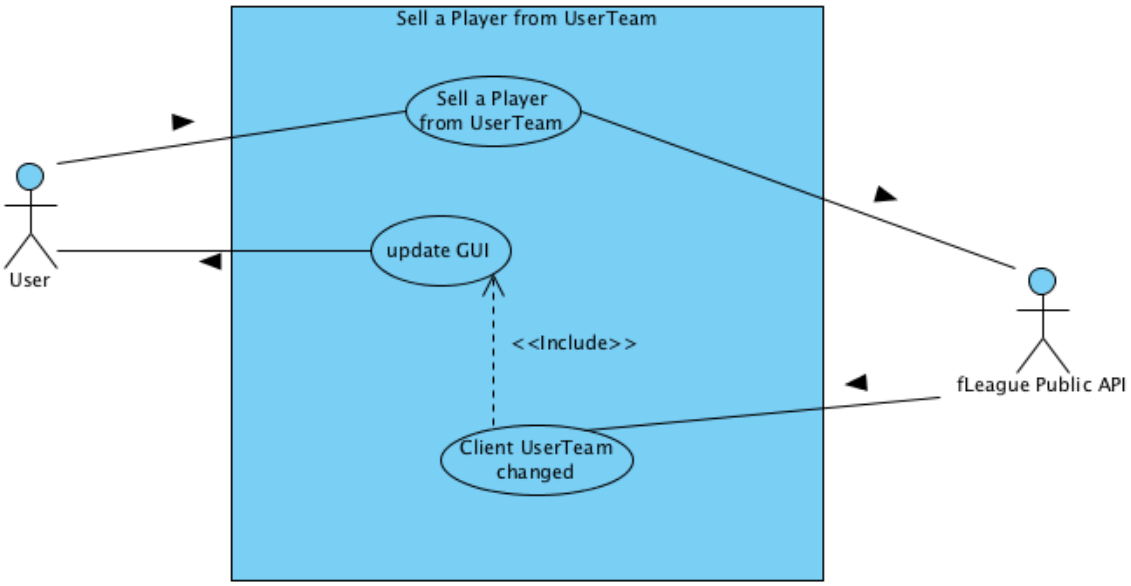

Abbildung 6.16: UML Anwendungsfalldiagramm eines fLeague Vorgangs

Testfallmodellierung

Bei der Testfallmodellierung könnte im Sinne der Atomizität jede API-Funktion mit einem Testfall abgedeckt werden. Der Umgang mit diesen Testfällen, also die Zusammenstellung in Testsuiten ist dann allerdings sehr aufwendig. Eine geschicktere Variante ist, verschiedene typische Verhaltensmuster des Users nachzubilden. Als Grundlage dafür kann ein Anwendungsfall (engl. use case) verwendet werden, wie er in Abb. 6.16 dargestellt ist.

Ausgehend von dieser Vorlage wird untersucht, welche API-Funktionen an dem Anwendungsfall beteiligt sind. Danach kann dieser Ablauf in Testfälle aufgespalten und wieder in einer Testsuite gruppiert werden.

Die folgende Liste an Aktionen beschreibt die System-System-Kommunikation, die für den Ablauf aus Sicht des Webdienstes notwendig ist.

1. Anmelden des Users am System (Authentifizierung)
2. Anzeige des UserTeam
3. Anzeige eines Players
4. Verkauf eines Players
5. Anzeige des UserTeam

6. Abmelden des Users vom System

Für jeden Einzelschritt kann ein Testfall erstellt werden. Der 2. Schritt „Anzeige des User-Team“ ist unter Abb. 6.17 abgebildet.

```
/**
* Testfallbeispiel. Zu einem bekannten usernamen wird das User-Object ermittelt. Dann koennen die
    enthaltenen Player im UserTeam gefunden werden und in einem Array zurueckgegeben werden.
*
*   @param  input   outputvalue from last testcase
* @param  callback  Testcase Callback-function, starts next test with given arguments (error, output)
* @param  global    Object for global accessible variables throughout all tests
*/
var testcase1 = function listUserTeam(input, callback, global) {
  //get the logged in user from input or global
  var username = global.username;
  //get user object
  global.fLeagueAPI.findUser(username, global.auth, function (err,user) {
    if(err) {
      callback(err);
    } else {
      //array for the players that are in the users team
      var userPlayers = [];
      //we now got the userTeam in user.userTeam
      user.userTeam.forEach(function (player) {
        //get player information for each player in userTeam
        userPlayers.push(player);
        //forEach is asynchronous, so check if all players are returned
        if (user.userTeam.length === userPlayers.length) {
          //set output value to "userPlayers"
          callback(null,userPlayers);
        };
      });
    };
  });
};
```

Abbildung 6.17: Beispiel für einen fLeague Testcase

Testsuitemodellierung

Eine Testfallerstellung nach User-Verhalten bietet die Möglichkeit, später verschiedene Testsuiten zu erstellen, die unterschiedliche komplexere Abläufe simulieren. Sobald einige Testfälle vorliegen, können diese in Suiten zusammengefasst werden. Eine Suite beschreibt möglichst genau einen Ablauf, der im Betrieb der Anwendung vorkommt. Bei der Zusammenstellung der Testsuiten zahlt sich eine gute Struktur der Testfälle aus, sodass viele Testsuiten mit den gleichen Testfällen erstellt werden können. Abb. 6.18 zeigt die fLeague Testsuite, die das Anwendungsfalldiagramm in Abb. 6.16 darstellt.

```
1 var buyPlayerTestsuite = [];
2
3 buyPlayerTestsuite.push(testcase_login);
4 buyPlayerTestsuite.push(testcase_listUserTeam);
5 buyPlayerTestsuite.push(testcase_showPlayerDetail);
6 buyPlayerTestsuite.push(testcase_userBuysPlayer);
7 buyPlayerTestsuite.push(testcase_listUserTeam);
8 buyPlayerTestsuite.push(testcase_logout);
```

Abbildung 6.18: Beispiel für eine fLeague Testsuite

Testergebnisse

Mit den gewonnen Daten aus der Testsuite können nun Ergebnisse gewonnen werden, die mit Hilfe der Auswertungen im Kapitel 6.2 interpretiert werden können.

Die Testerstellung und -ausführung ist simpel, wenn der Entwickler bereits Erfahrungen mit JavaScript gesammelt hat. Über Anwendungsfalldiagramme lassen sich Testsuiten definieren, da diese bereits eine Sicht von außen auf das System bereitstellen. So lassen sich Lasttests schon sehr früh im Entwicklungsprozess definieren und können bei der Entwicklung behilflich sein, indem sie lastspezifische Schwachstellen aufdecken und die Funktionalität verifizieren.

Theoretisch ist auch die automatische Code-Generierung von Testfällen möglich, wie sie in [3] vorgestellt wurde, wobei die Anwendungsfalldiagramme dann eine detaillierte Sicht auf die Systemkommunikation bieten müssen, als das in dem hier vorgestellten Beispiel der Fall ist (vgl. 6.16).

6.4 Notwendigkeit eines verteilten Aufbaus

Die Anwendung wsload ist derzeit auf einem einzelnen Testsystem lauffähig. Durch einen verteilten Aufbau könnten allerdings mehrere Testsysteme zur gleichen Zeit Testabläufe ausführen. Die Realisierung eines verteilten Aufbaus geschieht mit Hilfe der Anwendungen *wsload-worker* und *wsload-master*, die hier kurz vorgestellt werden, allerdings nicht Teil dieser Ausarbeitung sind.

Es stellt sich die Frage, wann eine solche horizontale Skalierung des Testsystems Sinn ergibt. Erstens kann ein Lasttest nur Ergebnisse liefern, wenn das Testsystem nicht durch die Testabläufe gebremst wird. Ist dies der Fall, sind die Ergebnisse einer Zeitmessung hinfällig, da abhängig von der Lastsituation des Testsystems. Des Weiteren kann es vorkommen, dass der eigentliche Kommunikationskanal zwischen Testsystem und zu testendem System limitiert. Dazu gehören die Verbindung (Bandbreite, Latenz) zwischen den Systemen und das eigentliche Protokoll, das verwendet wird, bzw. dessen Implementierung.

Die Anwendung *wsload* verwendet für die eigentliche Kommunikation mit dem zu testenden System unterschiedliche Protokolle bzw. Module, über die der Testanwender selbst entscheiden kann. Aus diesem Grund müssen die Auswirkungen eines verteilten Aufbaus Protokoll-spezifisch betrachtet werden.

Verteilter Aufbau

Es kommen mehrere *wsload-worker* zum Einsatz, die jeweils auf einem Testsystem ausgeführt werden. Diese sind in der Lage, Testabläufe auszuführen, die von einem *wsload-master* verteilt werden. Der Kommunikationsablauf ist in Abb. 6.19 dargestellt und geschieht wie folgt:

1. *Wsload-master* sendet einen Testablauf zu einem *wsload-worker*
2. Ein *wsload-worker* übermittelt den Testablauf an *wsload*
3. Die *wsload*-Instanz führt den Testablauf durch und übergibt die Ergebnisse zurück an den *wsload-worker*
4. Der *wsload-worker* übermittelt die Ergebnisse an den *wsload-master*
5. Sobald alle *wsload-worker* ihre Ergebnisse übermittelt haben, speichert der *wsload-master* die Ergebnisse in eine Datenbank.

Fazit zum verteilten Aufbau

Die horizontale Skalierung des Systems wird notwendig, sobald das Testsystem durch zu hohe Auslastung verfälschte Ergebnisse liefert. Die Verteilung brachte im Fall der Beispielanwendung (vgl. Kap. 6.3) keinen Vorteil, da der Webdienst selbst die limitierende Komponente darstellt und sich die Messwerte kaum änderten. Abhängig von der Leistungsfähigkeit des Webdienstes kann eine horizontale Skalierung notwendig werden, gerade wenn eine serverseitige Lastverteilung zum Einsatz kommt.

Eine generelle Aussage gestaltet sich schwierig, da das Verhalten des verwendeten Protokolls eine Rolle spielt. Soll neben der WebSocket- beispielsweise eine HTTP-Kommunikation stattfinden, so bremst diese auf Grund des Overheads den Testablauf enorm und eine horizontale Skalierung des Testsystems wird gegebenenfalls früher notwendig. Auch die Verwendung von Sub-Protokollen wie SOAP führt zu einer Steigerung der Belastung des Testsystems, da zusätzliche Logik (z. B. XML-Parsing) zum Einsatz kommen muss.

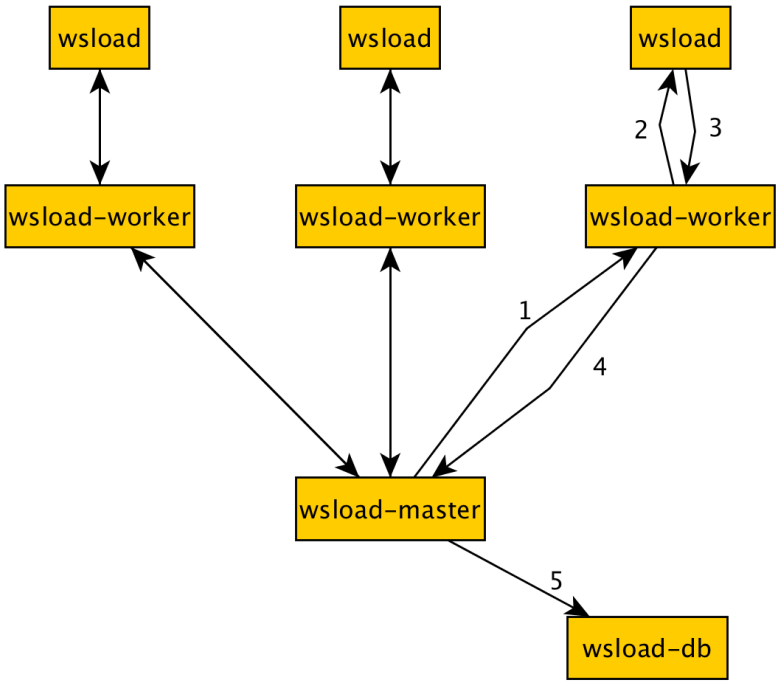

Abbildung 6.19: Verteilter Aufbau von wsload

Sind die Ressourcen des Testsystems knapp, kann durch die horizontale Skalierung die Last verteilt werden. Ab 75% CPU-Auslastung des Testsystems muss die Notwendigkeit eines verteilten Aufbaus in Erwägung gezogen werden.

7 Zusammenfassung und Ausblick

Das letzte Kapitel dieser Arbeit fasst die Erkenntnisse zusammen, die diese Arbeit erbracht hat, und gibt einen Ausblick über die Möglichkeiten, die sich ergeben haben.

Zusammenfassung

Diese Arbeit und die im Rahmen dieser Arbeit entwickelte Anwendung *wsload* geben einem Entwickler von Webdiensten die Möglichkeit, lastspezifische Daten eines zu testenden Systems zu sammeln und diese auszuwerten. Dadurch, dass die externe Schnittstelle für den Testablauf verwendet wird, sind die Tests unabhängig von der Systemkomplexität hinter der API. Um die Resultate einordnen zu können, müssen Lasttests besonders im Bereich der Webdienste möglichst früh im Entwicklungsprozess auf ihr Lastverhalten hin untersucht werden.

Des Weiteren wurden die Auswertungsergebnisse besprochen und infolge dessen Analysemöglichkeiten aufgezeigt, mit denen sich diese Ergebnisse interpretieren lassen.

An einem konkreten Beispiel ließ sich zeigen, dass unter Verwendung von UML-Anwendungsfalldiagrammen eine besonders effektive Testerstellung möglich ist. Abschließend wurde diskutiert, wann ein verteilter Testaufbau notwendig ist.

Die Anwendung wurde mit Hilfe des Frameworks *node.js* realisiert, während sich der Großteil der Entwicklungsarbeit auf die im Vorfeld erstellten UML-Diagramme beschränken ließ. Es stellte sich heraus, dass mit JavaScript und *node.js* eine effiziente und objektorientierte Software-Entwicklung mit Hilfe von UML möglich ist.

Möglichkeiten

Während der Arbeit an *wsload* und der Auswertung der gewonnenen Ergebnisse sind diverse Ideen entstanden hinsichtlich der Frage, in welche Richtung zusätzliche Entwicklungsarbeit vorstellbar wäre.

Horizontale Skalierung

Die horizontale Skalierung auf mehrere Systeme bietet die Möglichkeit, ein Netz von Testsystemen zu schaffen, welches wiederum von mehreren Testerstellern genutzt werden könnte und somit selbst einen Webdienst darstellt. Ergänzt durch ein Abrechnungsmodell und die Fähigkeit, unterschiedliche Testabläufe parallel auszuführen, könnten die Ressourcen des Testnetzes an potenzielle Kunden weitergeben werden.

Synchronisation während der Testabläufe

Bei der parallelen Ausführung von Testsuiten kann es vorkommen, dass sich bestimmte Testfälle synchronisieren sollen, also solange gewartet wird, bis das System in jeder Testsuite bei einem bestimmten Testfall angekommen ist. Das müsste außerhalb der Zeitmessung passieren, um die Testergebnisse nicht zu verfälschen, dürfte allerdings die Komplexität der Testerstellung nicht unnötig erhöhen.

Testprotokollerweiterung

Das WebSocket-Protokoll bietet die Möglichkeit, Sub-Protokolle einzusetzen. Beispielsweise ließe sich eine *SOAP*-Kommunikation über WebSockets betreiben. Von dieser Möglichkeit wird derzeit nicht direkt Gebrauch gemacht. Die Auswahl der aufsetzenden Kommunikationsmöglichkeiten ist dafür zu groß (*SOAP*, *REST*, *socket.io*, andere proprietäre Lösungen etc.) und würde zusätzliche Komplexität fordern. Es kann darüber nachgedacht werden, ob andere (Sub-)Protokolle mit einem Plugin-System nachgerüstet werden können.

Literaturverzeichnis

[1] BERNERS-LEE, Tim u. a.: World-Wide Web: The Information Universe. (1992). – URL http://www.emeraldinsight.com/products/backfiles/pdf/backfiles_sample_5.pdf

[2] GARRETT, Jesse J.: *Ajax: A New Approach to Web Applications*. 2005. – URL http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[3] HARTMANN, Jean ; VIEIRA, Marlon ; FOSTER, Herb ; RUDER, Axel: *UML-based Test Generation and Execution*. 2005. – URL http://www.nilachakra.org/documents/material/T%20-%20testing.pdf

[4] KICILLOF, Nicolas ; GRIESKAMP, Wolfgang ; TILLMANN, Nikolai ; BRABERMAN, Victor: Achieving Both Model and Code Coverage with Automated Gray-Box Testing. (2007). – URL http://research.microsoft.com/pubs/81199/p1-kicillof.pdf

[5] MELZER, Ingo u. a.: *Service-orientierte Architektur*. Spektrum Akademischer Verlag, 2010. – ISBN 978-3-8274-2550-8

[6] MINIWATTS MARKETING GROUP: *Internet Users in the World*. 2012. – URL http://www.internetworldstats.com/stats.htm

[7] PATTON, Ron ; BINDER, Robert: *Grey Box Testing*. 2008. – URL http://www.site.uottawa.ca/~ssome/Cours/SEG3203/gboxtesting.pdf

[8] RAMACHANDRAN, Sreeram: *Web metrics: Size and number of resources*. 2010. – URL https://developers.google.com/speed/articles/web-metrics

[9] WESSENDORF, Matthias: WebSocket: Annäherung an Echtzeit im Web. (2011). – URL http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html

[10] WIKIPEDIA: *Lasttest (Computer)*. – URL http://de.wikipedia.org/wiki/Lasttest_(Computer)

[11] WINOKUR, Danny: *Flash to Focus on PC Browsing and Mobile Apps; Adobe to More Aggressively Contribute to HTML5*. 2011. – URL `http://blogs.adobe.com/conversations/2011/11/flash-focus.html`

Anhang

Dieser Ausarbeitung sind zwei gedruckte und ein elektronischer Anhang (CD-ROM) angefügt. Die gedruckten Anhänge finden sich erneut in elektronischer Form auf der CD-ROM.

Anhang A

Quellcode-Dokumentation ab Seite 64

Anhang B

Ergebnis der Anforderungsanalyse ab Seite 73

Verzeichnisstruktur der CD-ROM

- **Bachelorthesis**
  (./Thesis/Thesis.pdf)
  Das vorliegende Dokument im PDF-Format.
- **Wsload: Quellcode**
  (./wsload/)
  Wurzelverzeichnis der Anwendung *wsload*.
- **Wsload: Dokumentation**
  (./wsload/doc/outline/index.html)
  API-Dokumentation der Anwendung *wsload* für die Ansicht in einem Browser.
- **Konzeption: Anforderungen**
  (./Konzeption/Anforderungen.pdf)
  Anforderungen an die Anwendung *wsload* im PDF-Format.

- **Konzeption: UML-Diagramme**
  (./Konzeption/UML/Thesis.vpp)
  Projektarchiv mit allen UML-Diagrammen für Visual Paradigm

- **Online-Quellen des Literaturverzeichnisses**
  (./Literatur/)
  Alle im Literaturverzeichnis angegebenen Online-Quellen sind hier nochmals im PDF-Format zu finden.

A Quellcode-Dokumentation

Es folgt die Quellcode-Dokumentation der Anwendung *wsload*.

Class Index | File Index

Classes

*_global_*
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

File Index

/Users/christian/Documents/Uni/wsload/lib/Logger.js

/Users/christian/Documents/Uni/wsload/lib/Measure.js

/Users/christian/Documents/Uni/wsload/lib/MongoAdapter.js

/Users/christian/Documents/Uni/wsload/lib/Testcase.js

/Users/christian/Documents/Uni/wsload/lib/Testsuite.js

/Users/christian/Documents/Uni/wsload/Wsload.js

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

Class Index

Class Index | File Index

Classes

global

Logger

Measure

MongoAdapter

Testcase

Testsuite

Wsload

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

Class Logger

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Defined in: Logger.js.

Class Summary	
	Logger(param_settings) Initialize a new `Logger` The Logger logs results from the testruns to a db or, if not reachable, to the console

Method Summary	
	closeDb() closes the db with help of the adapter
	get(param_uuid, param_cb) Gets results from database with given uuid
	log(param_logMessage, param_cb) Logs a message
	openDb() open the db from the adapter

Class Detail

Logger(param_settings)

Initialize a new `Logger` The Logger logs results from the testruns to a db or, if not reachable, to the console

Parameters:

{Object} **param_settings**

Method Detail

closeDb()

closes the db with help of the adapter

---

get(param_uuid, param_cb)

Gets results from database with given uuid

Parameters:

{Integer} **param_uuid**
{Function} **param_cb**

---

log(param_logMessage, param_cb)

Logs a message

Parameters:

{Object} **param_logMessage**
{Function} **param_cb**

---

openDb()

open the db from the adapter

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Class Measure

Defined in: Measure.js.

Class Summary	
	Measure(param_timeout, param_cb) Initialize a new `Measure` Object.

Method Summary	
	start() starts a measurement
	stop() stops a measurement

Class Detail

Measure(param_timeout, param_cb)

Initialize a new `Measure` Object. Measure time (in milliseconds) from a defined start to stop. When a timeout is specified, the callback will be executed when the timeout is reached.

Parameters:

{Integer} **param_timeout**
{Function} **param_cb**

Method Detail

start()

starts a measurement

stop()

stops a measurement

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Class MongoAdapter

Defined in: MongoAdapter.js.

Class Summary	
	MongoAdapter(param_cb) Initialize a new `MongoAdapter`.

Method Summary	
	close(force, param_cb) Closes the connection to the Database

Class Detail

MongoAdapter(param_cb)

Initialize a new `MongoAdapter`. Connects to a MongoDB on localhost TODO:Let the user decide, on which host the DB runs.

Parameters:

{Function} **param_cb**

Method Detail

close(force, param_cb)

Closes the connection to the Database

Parameters:

{Boolean} **force**
Force disconnection
{Function} **param_cb**

Class Testcase

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Defined in: Testcase.js.

Class Summary

	Testcase(param_testNr, param_testFunction, param_globarlUserVar) Initialize a new `Testcase` A "Testcase" encapsulates the actual test function

Method Summary

	getResult() Returns the test case result
	runTest(param_inputValue, param_cb) Run the test function with a given input parameter
	setRunDuration(param_runDuration) Sets the result from a time measurement for this test case (unit:milliseconds)

Class Detail

Testcase(param_testNr, param_testFunction, param_globarlUserVar)

Initialize a new `Testcase` A "Testcase" encapsulates the actual test function

Parameters:

{Integer} **param_testNr**
{Function} **param_testFunction**
{Object} **param_globarlUserVar**

Method Detail

{Object} **getResult**()

Returns the test case result

Returns:

{Object} result

---

runTest(param_inputValue, param_cb)

Run the test function with a given input parameter

Parameters:

{Object} **param_inputValue**
{Function} **param_cb**

---

setRunDuration(param_runDuration)

Sets the result from a time measurement for this test case (unit:milliseconds)

Parameters:

{Integer} **param_runDuration**

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

Class Testsuite

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Defined in: Testsuite.js.

Class Summary

	Testsuite(param_testRunGuid, param_suiteNumber, param_suiteName, param_suiteTimeout, param_suiteFunctions, param_preTestFunction, param_globalUserVar) Initialize a new `Testsuite`.

Method Summary

	addResult(param_testcase, param_timeInMs) Adds a runtime result to a test case
	addTestcase(param_testcase) Adds a new Testcase to the Testsuite
	preTestFunction(cb) preTestFunction is run before every testcase.
	run() Runs this Testsuite

Class Detail

Testsuite(param_testRunGuid, param_suiteNumber, param_suiteName, param_suiteTimeout, param_suiteFunctions, param_preTestFunction, param_globalUserVar)

Initialize a new `Testsuite`. A Testsuite encapsulates a number of `Testcase` that need to be run in order.

Parameters:

{String} **param_testRunGuid**
{Integer} **param_suiteNumber**
{String} **param_suiteName**
{Integer} **param_suiteTimeout**
{Array} **param_suiteFunctions**
{Function} **param_preTestFunction**
{Object} **param_globalUserVar**

Method Detail

addResult(param_testcase, param_timeInMs)

Adds a runtime result to a test case

Parameters:

{Testcase} **param_testcase**
{Integer} **param_timeInMs**

---

addTestcase(param_testcase)

Adds a new Testcase to the Testsuite

Parameters:

{Testcase} **param_testcase**

---

preTestFunction(cb)

preTestFunction is run before every testcase. can be overridden by user function

Parameters:

{Function} **cb**

---

run()

Runs this Testsuite

Class Wsload

Class Index | File Index

Classes

global
Logger
Measure
MongoAdapter
Testcase
Testsuite
Wsload

Defined in: Wsload.js.

Class Summary	
	Wsload(param_settings) Initialize a new Wsload object

Method Summary	
	runSuite(param_suiteName, param_timesToRunSuite, param_testFunction, param_suiteTimeout, param_globalUserVar, param_globalVar) runs all suites

Class Detail

Wsload(param_settings)

Initialize a new Wsload object

Parameters:

{Object} **param_settings**

Method Detail

runSuite(param_suiteName, param_timesToRunSuite, param_testFunction, param_suiteTimeout, param_globalUserVar, param_globalVar)

runs all suites

Parameters:

{String} **param_suiteName**
{Integer} **param_timesToRunSuite**
{Array} **param_testFunction**
{Integer} **param_suiteTimeout**
{Object} **param_globalUserVar**
param_globalVar

Documentation generated by JsDoc Toolkit 2.4.0 on Thu Aug 09 2012 11:13:48 GMT+0200 (MESZ)

B Ergebnis der Anforderungsanalyse

Es folgt die zusammengefasste Anforderungsanalyse der Anwendung *wsload*.

Anforderungen an die Lasttest-Applikation

Schnittstellen

#SC-10 API nach RFC6455 wird unterstützt
Um einen Lasttest durchführen zu können, muss das zu testende System die WebSocket API nach RFC6455 implementieren.

#SC-20 Die Auswertungen können in eine Datenbank gespeichert werden
Auf dem Testssytem befindet sich eine Datenbankinstanz, aus der externe Anwendungen auf die Testergebnisse zugreifen können.

Testerstellung

#TE-10 Einzelne Testfälle lassen sich erstellen
Das System ermöglicht es, Testfälle in Code-Form zu erstellen, die eine bestimmte Funktionalität über die WebSocket-API testen.

#TE-11 Die Testfälle lassen sich einfach erstellen
Das System lässt sich einfach bedienen und ein Testfall kann ohne lange Einarbeitung erstellt werden.

#TE-12 Die Testfälle sind durch eine Bezeichnung gekennzeichnet
Ein eindeutiger Bezeichner identifiziert einen Testfall.

#TE-20 Die Testfälle lassen sich in Suiten zusammenfassen
Die erstellten Testfälle lassen sich in Suiten gruppieren. Mehrere Testfälle können in unterschiedlichen Suiten enthalten sein.

#TE-21 Testsuiten lassen sich parallel ausführen
Eine Testsuite kann beliebig häufig parallel ausgeführt werden. Eine Testsuite soll das Verhalten unter Last simulieren, wobei die Testfälle einzelne Schritte im Testablauf darstellen. Parallele Testfälle simulieren somit parallele Userzugriffe.

#TE-22 Der Testersteller kann wählen, wie viele parallele Suiten ausgeführt werden sollen

#TE-23 Die Testfälle in den Suiten laufen zeitlich nacheinander ab
„Wasserfall"-Modell: Der erste Testfall in einer Suite muss abgeschlossen sein, bevor der zweite Testfall gestartet wird.

#TE-24 Parameter/Rückgabewerte lassen sich an den nächsten Test in der Suite weitergeben

#TE-25 Der Testersteller kann eine Zeit vorgeben, nach der ein Testfall bzw. eine Suite als fehlgeschlagen gilt

#TE-26 Der Testersteller kann zwischen den Tests eigenen Code einfügen
Zwischen den Testfällen in einer Suite kann der Testersteller eigenen Code einfügen, z.B. um eine Verzögerung der nächsten Testausführung zu erreichen.

#TE-27 Der Testersteller kann eine Zeit vorgeben, nach der eine Testsuite als fehlgeschlagen gilt

#TE-30 Die Testsuiten sind durch eine Bezeichnung gekennzeichnet
Ein eindeutiger Bezeichner identifiziert eine Testsuite.

Zeitmessung

#ZM-10 Zeitmessung der Ausführungszeit eines Testfalls
Die Ausführungszeit jedes Testfall in seiner Testsuite wird einzeln gemessen. Dabei wird sowohl die Zeitdauer des Testfalls als auch die Startzeit gespeichert, um eine Referenz auf andere parallele Testfälle/Testsuiten zu erhalten.

#ZM-20 Zeitmessung der Ausführungszeit einer Testsuite
Die Ausführungszeit einer gesamten Testsuite wird gemessen.

Testauswertung

#TA-10 Es können zusammengefasste und detaillierte Auswertungen von Testsuiten erstellt werden

#TA-20 Die zusammenfassende Auswertung einer Testsuite enthält weitere Informationen
- Bezeichnung der Testsuite
- Datum von Start und Ende der Suite
- Anzahl der (parallelen) Testsuite-Durchläufe
- Mittelwerte eines Testsuite-Durchlaufs
- Minimal- und Maximalzeit eines Testsuite-Durchlaufs

#TA-30 Die detaillierte Auswertung einer Testsuite enthält zusätzlich zu #TA-20 weitere Informationen
- Zeitmessungsergebnis der einzelnen Testsuiten
- Zeitmessungsergebnis der einzelnen Testfälle
- Bezeichner der Testfälle zur Zuordnung

#TA-40 Testdaten für Beispielsuite mit zwei Testfällen
Beispiel für erhobene Daten bei einer Testsuite mit 2 Testfällen, die gespeichert werden sollen:

Testsuite-Bezeichnung
Ausführungszähler Suite
Ausführungszeit Suite
Timeout Testsuite
Startzeit Testsuite
Testfall1-Bezeichnung
    Rückgabewert Testfall1
    Ausführungszeit Testfall1
    Timeout Testfall1
    Startzeit Testfall1
Testfall2-Bezeichnung
    Rückgabewert Testfall2
    Ausführungszeit Testfall2
    Timeout Testfall2
    Startzeit Testfall2

(zusätzliche) Systembedingungen

#SB-10 Das zu testende System ist vom Testsystem über ein Netzwerk erreichbar

#SB-20 Das zu testende System bietet eine WebSocket-API an.

#SB-30 Das Testsystem lässt sich innerhalb von einer halben Stunde einrichten.

Glossar

AJAX Asynchronous JavaScript and XML bezeichnet ein Konzept der asynchronen Datenübertragung, die es einem Browser erlaubt, neue Daten von einem Webserver zu laden, ohne die Seite manuell zu aktualisieren.

API Application programming interface ist ein Programmteil, der eine Schnittstelle beschreibt, über die andere Systeme angebunden werden können.

GUI Graphical user interface bezeichnet eine Software-Komponente, über die ein Benutzer mit Hilfe von grafischen Oberflächenelementen interagieren kann.

HTML Hypertext Markup Language bezeichnet eine Auszeichnungssprache zur Beschreibung von Inhalten wie Texte oder Tabellen. HTML-Dokumente stellen die Grundlage des WWW dar und werden in einem Browser angezeigt.

HTTP Hypertext Transfer Protocol ist ein Übertragungsprotokoll, das hauptsächlich zur Übertragung von Webseiten zu einem Webbrowser verwendet wird.

IETF Internet Engineering Taskforce ist eine Organisation, die sich mit der technischen Weiterentwicklung der für das Internet relevanten Techniken befasst.

LAN Local area network ist ein Netzwerk von Rechnern, das sich über einen kleinen bzw. lokalen geographischen Bereich erstreckt.

REST Representational State Transfer ist ein Programmierparadigma für Webdienste und Webanwendungen, dass jede Funktion des Systems über eindeutige Bezeichner (URI) erreichbar ist.

RFC Request for Comments bezeichnet einen Standardisierungsvorschlag, der im Rahmen von verschiedenen Schritten von einer Idee zum Standard evolviert.

SOAP Simple Object Access Protocol bezeichnet ein auf XML-basiertes Netzwerkprotokoll, mit dessen Hilfe sich Daten zwischen Systemen austauschen lassen.

UML Unified Modeling Language ist eine grafische Modellierungssprache, mit der Strukturen und Abläufe von Systemen dargestellt werden können.

URI Uniform Resource Locator definieren ein Schema für eine Identifikationsbeschreibung einer Ressource.

W3C World Wide Web Consortium bezeichnet ein Gremium, das zur Aufgabe hat, relevante Techniken des WWW zu standardisieren.

WAN Wide area network ist ein Netzwerk von Rechnern, das sich über einen großen geographischen Bereich erstreckt.

WHATWG Web Hypertext Application Technology Working Group bezeichnet eine Arbeitsgruppe, die wie die W3C an neuen Standards im Bereich des Internets arbeitet.

WWW World Wide Web ist die Bezeichnung für ein verteiltes System, das mit einem Browser über das Internet abrufbar ist und dem Anwender Informationen anzeigt.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 10. August 2012	
Ort, Datum	Unterschrift