

Swift on Linux

Project Work

submitted in conformity with the requirements for the grade of
lecture "Projektarbeit"
in the Bachelor degree programme **Software Design**

FH JOANNEUM (University of Applied Sciences), Kapfenberg

Supervisor: DI Johannes Feiner, FH JOANNEUM Kapfenberg

submitted by: Stefan Moder, Christian Hofer
personnel identifier: 1500000000, 1510418043

July 2017

**Assignment for the project work of
Stefan Moder, Christian Hofer
Matr. no. 15000000000, 1510418043**

**Subject:
“Swift on Linux”**

Abstract

Write your abstract here.

Kapfenberg, 24.07.2017

Academic adviser:

DI Johannes Feiner

Stefan Moder, Christian Hofer

Formal declaration

We hereby declare that we have produced the present work by ourself and without any aids other than those mentioned herein. Any ideas taken directly or indirectly from third party sources are indicated as such. This paper has not been published or submitted to any other examination board in the same or a similar form.

Kapfenberg, 24.07.2017

Stefan Moder, Christian Hofer

Acknowledgement

Thanks to ...

Contents

1	Introduction	1
1.1	Some L ^A T _E X Basics	1
2	Umsetzung	4
2.1	Die Umgebung	4
2.2	Der Reverenz-Server	5
2.3	Der Swift-Server	8
2.4	Vergleich der Implementierungen	13
2.5	Leistungsvergleich	26
2.6	Fazit	29
2.7	Probleme und Lösungen	29
3	Conclusion and Outlook	30
	References	33

List of Tables

1.1	Olive green heading	2
1.2	A grey table	3
2.1	Module der Server	14
2.2	Test: 5.000 Request on "Login Page"	28
2.3	Test: 50.000 Request on "Login Page"	28

List of Figures

1.1	Train engine in Kapfenberg	2
2.1	Request-Abarbeitung von Express.js und Perfect	27

Chapter 1

Introduction

This template shall provide some considerations and text examples for your Master's thesis.

Background. Describe the background, the prerequisites for your work ...

Objective. The aim of this master's thesis is ...

Terms and definitions. Technical terms ... abbreviations are summarised at the end (in "Acronyms"), e.g. **ABI!** (**ABI!**) or **MITM!** (**MITM!**). If **ABI!** is referenced again, only the acronym is printed (as hyperlink though).

For literature research use e.g. **acm:diglibrary** (**acm:diglibrary**) or **ieee:xplore** (**ieee:xplore**) as available from the FH JOANNEUM Library web page.

Harvard citation style is implemented in this template: **Batina:2011:HER:2188496.2188508**
Fernandez-Mir:2011:SRA:2188496.2188506 **Li:2008:TRP:2043628.2043631**

1.1 Some L^AT_EX Basics

This section is a *really very short* summary of L^AT_EX features. Do not forget to remove it after finishing your thesis.

Here you have an included graphic (figure 1.1).



Figure 1.1: Train engine in Kapfenberg

Code listings require the *listings* package which, in turn, requires some settings¹; see command `\lstset{}` in preamble of this template. Additionally the package *courier* should be used because the defaults do not provide for proper syntax highlighting.

```
1 void main(int argc, char *argv[])
2 {
3     printf("Hello world!");
4 }
```

Listing 1.1: Main programme

In order to see what's possible – here are two fancy tables: 1.1 and 1.2.

Version	Description	Author(s)	Date
1.0	Initial	Ohrt	July 15, 2014
1.1	Filled section “Open Issues”	Ohrt	July 16, 2014
1.2	Added section “Restrictions”	Ohrt	September 15, 2014

Table 1.1: Olive green heading

View also the preamble of this file for explanations.

Here is a reference to listing 1.1.

¹... because the defaults do not fit all purposes

Error	Solution
Java.lang.OutOfMemoryError: PermGen space (32-/64-bit issue)	-XX:MaxPermSize=1024M
Error occurred during initialization of VM <i>or</i> Could not reserve enough space for object heap	increase or remove -Xms value e.g. -Xms128m -Xmx512m (Eclipse default: -Xms40m -Xmx512m)

Table 1.2: A grey table

Umsetzung

Folgend wird die Umsetzung beschrieben, die in mehrere Bereiche aufgeteilt wurde. Diese Teilbereiche gliedern sich wie folgt:

- Vorbereitung der Entwicklungsumgebung
- Erstellen des Reverenzservers in Javascript mit Node.js
- Erstellen des Testservers in Swift mit Perfect
- Vergleich der beiden Server
- Fazit

2.1 Die Umgebung

Für die Entwicklung des Swift-Servers musste eine Linux Distribution gefunden werden, die den Ansprüchen einer Entwicklungsumgebung entspricht. Mehrere Faktoren spielten dabei eine Rolle. Die wichtigsten Punkte dabei waren die Stabilität der Distribution, die Möglichkeit Entwicklungsumgebungen beziehungsweise Programme zu installieren ohne großen Aufwand mit dem auflösen von Abhängigkeiten zu haben, und geringer Wahrscheinlichkeit möglicher Inkompatibilität. Weiters sollte es möglich sein diese als Virtuelle Maschine laufen zu lassen.

Stabilität Die Stabilität ist wichtig da Swift die Sprache "C" verwendet und es die Möglichkeit gibt, die Virtuelle Maschine, zum Beispiel durch das Abstürzen des OS zu beschädigen. Außerdem sollte es möglich sein, Programme aller Art auf dem OS

zu installieren, sodass es keine Überraschungen im Laufe des Projektes gibt. Weiter wurde auch seitens der Swift-Entwickler Ubuntu empfohlen. Daher fiel die Wahl auf die zu diesem Zeitpunkt aktuelle Ubuntu Version 16.04 LTS, da diese auch möglichst leichtgewichtig ist.

Entwicklungsumgebung Da Swift eine noch nicht sehr weit verbreitete Sprache außerhalb der Mac-Welt ist, war es eine Herausforderung einen IDE, die auf Linux lauffähig ist, zu finden, in der eine Unterstützung in Form von zum Beispiel Syntax-highlighting oder Autovervollständigung für Swift zur Verfügung steht. Zur Lösung des Problems wurde *CLION* von *JetBrains* (JetBrain s.r.o., 2017) in der Version 2017.2 verwendet, das ein Plugin für Swift bereithält.

Virtuelle Maschine Für das Betriebssystem musste die Möglichkeit bestehen, es als eine Virtuelle Maschine auf einem PC lauffähig zu bekommen. Dazu wurde als Umgebung für die VM *VirtualBox* von *Oracle* installiert und darauf die vorhin erwähnte Ubuntu 16.04 LTS Distribution aufgesetzt.

2.2 Der Reverenz-Server

Um den Swift-Server vergleichen zu können, wurde zuvor ein Server benötigt, der verschieden serverseitige Anforderungen erfüllt. Anforderungen, die nicht direkt mit Node.js und Express.js in Verbindung stehen, zum Beispiel Responsive Web, welches mit CSS erreicht wurde, werden hier nicht berücksichtigt. Dieser Server wurde bereits im Zuge der Lehrveranstaltung "Rich Internet Application" von Michael Rotinger, Stefan Moder und Christian Hofer implementiert und erfüllt folgende serverseitige Anforderungen:

- Verwendung eines aktuellen Frameworks
- Dynamische Website Erstellung
- Registrierung, Speichern von Userdaten und verschlüsseln des Passwortes
- Login und Sessionverwaltung
- Caching
- Socketverbindung
- Model View Controller Pattern

- Logging
- Kompression

Framework Der Server ist in JavaScript geschrieben, wobei hier Node.js mit dem Framework Express verwendet wurde. Node.js ist eine I/O-Umgebung, die als einziger Thread auf dem Server läuft, der niemals geblockt wird. Stattdessen arbeitet es auf Event-Basis mit Callbacks. Jeder Request ist als Event zu verstehen und wird zum Beispiel als Datenbankabfrage erkannt. Der Mainthread wartet dabei nicht darauf, dass die Datenbankabfrage fertig ist und blockiert inzwischen, sondern kann in der Zwischenzeit weitere Requests behandeln. Bei Beendigung des I/O-Prozesses wird durch den Callback die eventuelle Response gesendet. Neben der Asynchronität von Node.js ist die Geschwindigkeit von dieser JavaScript-Maschine enorm. Entwickelt wurde Node.js auf Basis der "Google Chrome V8 engine" die JavaScript enorm schnell bearbeiten kann. Unterstützt durch das Framework Express können weitere Vorteile wie Middleware und modulbasierte Plugins verwendet werden. Weitere Vorteile von Express sind, dass es ein offenes, schnelles und unkompliziertes Web-Framework ist, das auch mobile Anwendungen unterstützt. Grundsätzlich werden 4 Hauptaufgaben von Express.js zur Verfügung gestellt:

- Middleware: diese bildet ein Array mit Funktionen, mit denen Request standardmäßig vorbearbeitet werden können, zum Beispiel setzen einer Variabel
- Routing: ist ein fester Bestandteil des Frameworks und ist mit dem Unterschied, das es aufgerufen werden muss mit einer Middleware vergleichbar
- Erweiterungen für Request und Response Objekte
- Views: Dynamisches rendern von HTML

Diese vier Bestandteile ermöglichen ein schnelles Erstellen eines WebServices (Hahn, Evan M., 2016). Express wird von der Stiftung Node.js projektiert und verwaltet und wird auch ständig weiterentwickelt (StrongLoop, Inc./IBM, o.J.a). Mittlerweile steht auch auf Basis von Express.js ein neues Framework LoopBack zur Verfügung, welches die Entwicklung von dynamischen end-to-end REST APIs für verschiedene Geräte und Browser zulässt, inklusive der Entwicklung von Client Application auf Android, iOS und AngularJS SDKs (StrongLoop, Inc./IBM, o.J.b).

Dynamische Websites Die Websites können mit der Eingabe von Usern verändert werden, zum Beispiel durch die Anzeige des Usernamens auf der Website. Dies soll

serverseitig ermöglicht werden bevor das HTML an den Client gesendet wird. Im Reverenzserver übernimmt dies das Modul "Handlebars". Diese Module ermöglichen es HTML-Skelete zu erstellen und mit Eingaben zu befüllen.

Registrieren, speichern, verschlüsseln Der Server ermöglicht es dem Client sich in einen gesicherten Bereich zu bewegen, dazu erfolgt zuvor eine Registrierung. Dazu werden die Daten des Users in Form eines JSON-Strings im Filesystem gespeichert. Für das Filehandling wurde das standardmäßig in Node.js enthaltene Modul "fs" verwendet. Dieses Modul übernimmt jegliche I/O-Aufgaben betreffend dem Filehandling, wie auch das Einlesen von Dateien. Eine Variation wäre das Verwenden einer Datenbank zum Beispiel Redis. Bevor das Passwort des Users gespeichert wird, soll dieses natürlich verschlüsselt werden, welches mit dem Modul "bcrypt-nodejs" erreicht wurde. Dieser verwendet eine "salted" SHA256 Verschlüsselung, welches asynchron oder synchron ausgeführt werden kann.

Login und Session Nach dem Einloggen des Users wird auf dem Server eine explizite Session gestartet. Diese wird automatisch durch Express und dem Modul "express-session" verwaltet und benötigt nur die Initialisierung und die Löschung der Session. Dadurch ist eine minimale Sessionverwaltung erreicht.

Caching Das Caching wurde durch das Setzen von Caching Headern erreicht. Das Setzen dieser Header wurde durch Aufruf von selbst implementierten Methoden erreicht, welche wiederum im Laufe der Requestbehandlung aufgerufen werden. Dazu wurde kein externes Modul verwendet beziehungsweise benötigt.

Socketverbindung Für die Socketverbindung im Server, um einen Chat zu betreiben, wurde das Modul "socket.io" verwendet. Dieses Modul bildet einen Adapter, um eine unkomplizierte Errichtung einer WebSocket-Verbindung zu implementieren.

Model View Controller Pattern Der Reverenzserver wurde im MVC-Pattern geschrieben, da diese zu den wichtigsten Design Patterns in der Webentwicklung zählt. Das MVC ist eines der offensten Patterns und behält sich einen großen Spielraum bereit. Das Pattern kann mehr als allgemeine Empfehlung gesehen werden als eine Anleitung, da es dem Entwickler selbst obliegt, zu entscheiden, welche Klassen, Objekte, Methoden oder Funktionen zu den einzelnen Fraktionen des MVC gehören. Beim Implementieren des Servers wurde eine grobe Einteilung getroffen. Controller nehmen Request

entgegen, steuern den Kontrollfluss und senden die Response, Models führen Operationen auf Daten aus und Views sind für die Generierung des HTML-Codes zuständig.

Logging Für eine Erleichterung der Entwicklung und im späteren Betrieb wurde auch eine Logging miteingerichtet, welches mit verschiedenen Logging-Level arbeitet. Die Informationen werden nicht nur auf der Konsole ausgegeben, sondern auch in ein Log-File geschrieben und wird für eventuelle Debugging auch im File System gespeichert.

Kompression Die HTML Daten werden vor der Übermittlung an den Client per gzip komprimiert. Dafür wurde das Modul *compression* verwendet.

2.3 Der Swift-Server

Um einen Server in Swift zu programmieren, muss die in 2.1 erwähnte Ubuntu Distribution mehrere Voraussetzungen erfüllen. Verschiedene C-Bibliotheken und "Clang" müssen installiert werden. Diese einzelnen Bedingungen und deren Installationsbedingungen und -vorgänge werden nun näher erläutert.

2.3.1 Swift auf Linux installieren

Für die Installation von Swift sind vier wichtige Bibliotheken bzw. Programme notwendig, die vorinstalliert werden müssen, damit die danach installierte Swift-Toolchain installiert und ausgeführt werden kann.

- clang
- libc-dev
- libcurlpp
- git (optional)

Clang Clang ist ein Front-End Compiler für C-basierte Sprachen wie C, C++, Objective C/C++, OpenCL C und andere Sprachen für den LLVM Compiler. Einige Ziele des Clang sind ein schnelles compilieren mit wenig Speicherverwendung zu erreichen, ausführliche Diagnose- und Fehlerberichte zu erstellen und auszugeben und

GCC kompatibel zu sein. Dadurch wird eine bessere Kompatibilität mit verschiedenen IDEs erreicht und Sprachen wie Swift können auf Betriebssystemen kompiliert werden. Clang ist unter der BSD Lizenz verfügbar und darf daher auch als integraler Bestandteil von kommerziellen Projekten verwendet werden (LLVM Foundation, o.J.). Clang ist bei der Kompilierung von Code bis zu drei mal so schnell als zum Beispiel der GCC. Zusätzlich hält der Clang Compiler auch den Clang Static Analyzer bereit, der automatisch Bugs im Code (LLVM Foundation, 2017). Im Projekt ist Clang nur bei der Compilierung beteiligt und wird im Code nur einmal direkt verwendet.

```
1 // Save guard against ios or windows usage
2 #if !os(Linux)
3 import Glibc
4 print("We are sorry this is only meant to be run on Linux")
5 exit(1)
6 #endif
```

Listing 2.1: OS Prüfung

Bei diesem Codeteil handelt es sich um eine Prüfung des Betriebssystems, wobei darauf geprüft wird, dass das System eine Linux Distribution ist und ansonsten abgebrochen wird. Die Syntax dieser OS Prüfung entspricht Clang und wird daher auch direkt von dieser kompiliert. Das compilieren von Swift wird ebenfalls von Clang übernommen, welches zu Maschinencode für die LLVM überführt wird und auf der Objective-C runtime läuft.

Der LLVM Compiler Ein Fehler wäre zu denken, LLVM wäre ein Acronym, tatsächlich ist es der vollständige Name des Compilers. Das LLVM begann als Wissenschaftsprojekt auf der Universität von Illinois und hat das Ziel, eine moderne SSA-basierte Kompilierungsstrategie zur Verfügung zu stellen. Dabei soll es möglich sein, statische wie auch dynamische Kompilierung von verschiedenen Programmiersprachen durchzuführen. LLVM ist derzeit zu einer großen Sammlung von modularen und wiederverwendbaren Compilern und Toolchains geworden, die unter der BSD Lizenz verfügbar sind (LLVM Foundation, 2017). Tatsächlich arbeitet der LLVM Compiler mit einer Vielzahl von compiler Techniken. Darüber hinaus verfügt er über eine Codeoptimierung und Generierung, den `llvm-gcc` als Backend und den Clang Front-end und unterstützt die Microsoft Intermediate Language und die .Net Virtual Machine. Der LLVM wurde entwickelt, weil ältere bereits existierende C Compiler nicht mehr weiterentwickelt wurden. Die größte Stärke des LLVM liegt darin, dass er aus mehreren Compiler-Modulen besteht, die sich gegenseitig ergänzen. Daraus wurden wiederum

Compiler gebaut, die für verschiedene Aufgaben geeignet sind z.B. als erweiterter C-Compiler oder als spezialisierte Runtime-Maschine. Ein weiterer Vorteil ist das der `llvm-gcc 4.2` mit den gcc command line Optionen und Sprachen, sowie dem `makefiles` kompatibel ist (**llvmpulbic**).

libc-dev und libcurlpp Diese Pakete sind Bibliotheken, die benötigt werden, um die Ausführung aller Swift-Bibliotheksfunktionen auszuführen. Nähere Informationen zu diese Paketen sind für das Projekt nicht notwendig.

Git Für die Versionierung und der gemeinsamen Bearbeitung wurde Git verwendet, das jedoch nicht direkt für den Server notwendig ist und soll daher nur erwähnt bleiben.

Installation Zur Installation wird der aktuelle Release heruntergeladen. Diese liegt als `.tar.gz` vor und kann in ein beliebiges Verzeichnis entpackt werden. Als Erleichterung wurde der Pfad in die Umgebungsvariable von Ubuntu hinzugefügt, um ein leichteres compilieren möglich zu machen bzw das Ausführen der REPL zu erleichtern. Der Server wurde mit dem Release 3.1.1 für Ubuntu 16.10 entwickelt.

2.3.2 Das Swift Package

Die Swift Sprache ist als Sammlung verschiedener Projekte gegliedert:

- Swift Compiler
- Standard Bibliothek
- Core Bibliothek
- LLDB Debugger
- Swift Package Manager
- Xcode Playground support

Swift Compiler Der Swift Compiler übersetzt den Source Code in effiziente und ausführbare Maschinensprache. Zuerst erfolgt das Parsen des Codes und das Erstellen des Abstract Syntax Trees, der für die darauffolgende semantische Analyse benötigt wird. Aus dem AST wird somit ein "well-formed, fully-type-checked" AST. Die semantische Analyse stellt sicher, dass es möglich ist zu compilieren. Danach wird der

Clang Importer benötigt, der verschiedene Clang Module importiert, die benötigt werden, um den AST auf C oder Objective-C APIs umzusetzen. Als nächster Schritt wird der AST mithilfe der Swift Intermediate Language optimiert und in eine sogenannte "raw" SIL umgeformt. Aus der SIL wird wiederum die "canonical" SIL erstellt. Dies wird erreicht, indem eine weitere sogenannte garantierte Transformation durchgeführt wird, in der zusätzliche Datenflussdiagnostik durchgeführt wird, zum Beispiel die Verwendung von nicht initialisierten Variablen. Danach wird nochmals eine Optimierung über die SIL durchgeführt wie das automatische Referenzieren, zähl-Optimierung, Devirtualisierung und allgemeine Spezialisierung. Ab diesem Punkt wird der Code weitergegeben an die LLVM IR Code Generierung, sodass aus der SIL eine LLVM IR wird und diese durch die LLVM zu Maschinencode übersetzt werden kann (Swift.org, 2017a).

Standard Bibliothek Die Standard Bibliothek von Swift hält verschiedene Beschreibungen von Daten-Typen, Protokollen und Funktionen bereit. Auch primitive Typen werden hier beschrieben. Die Standard Bibliothek wird in drei Teile gegliedert:

- **core:** Darin befinden sich alle Definitionen für Daten-Typen, Protokolle und Funktionen
- **runtime:** Dies wird als Zwischenschicht zwischen Compiler und der Core Standard Bibliothek geführt. Diese ist für dynamische Operationen zuständig wie z.B. das Casten oder Memory Management
- **SDK Overlays:** Hält verschiedene Pakete und Modifikationen bereit, um existierende Objective-C Frameworks in Swift einzubinden

Die Standard Bibliothek ist zwar in Swift geschrieben, jedoch weicht sie vom üblichen Swift Code ab. Diese Unterschiede sind auf den Nutzen der Bibliothek zurückzuführen, da diese z.B. benötigt wird, um die SIL zu erstellen oder gesamt als "public" zur Verfügung stehen soll. Zusätzliche Tools wie "gyb" sind notwendig, um sized Integer Typen zu deklarieren oder das Testen ist eng mit dem Compiler verbunden beziehungsweise ohne diesen nicht möglich (Swift.org, 2017d).

Core Bibliothek Diese hält verschiedene higher-level Funktionalität bereit als die Standard Bibliothek. Der Hauptteil der Bibliothek wird mit *import Foundation* im Code importiert und damit in einer spezifischen .swift-Datei zur Verfügung gestellt. Erst mit diesem Import steht sie beim compilieren zur Verfügung. Die Bibliothek soll verschiedene stabile und nützliche Features in folgenden Bereichen bereithalten:

- generell verwendete Typen z.B. URLs, Character Sets, Collections usw.
- durchführen von Unit Tests
- Networking Grundlagen
- Persistence, Listen, Archive, JSON parsing, XML parsing
- Datum-, Zeit- und Kalenderfunktionalität
- OS-Spezifisches Verhalten
- Interaktionen mit dem File System

Diese Bibliothek ist laut Swift.org noch nicht vollkommen und ausreichend entwickelt, sondern ist derzeit im Anfangsstadium. Diese wurde veröffentlicht, um es der Community zu ermöglichen von Anfang an bei der Entwicklung mit zu gestalten und mit zu wirken. Derzeit sind drei Teile der Bibliothek vorhanden. Diese sind *Foundation*, welche die Basis für alle Swift Projekte darstellt, *libdispatch*, die das Multithreading und Multitasking ermöglicht, sowie *XCTest*, welches ein Framework für das Erstellen von Unit Test bildet (Swift.org, 2017e).

LLDB Debugger Der LLDB Debugger ist ein Teil des Clang-Projektes. Besonderheit hier ist, dass der Debugger als Grundlage für den Swift REPL dient. Laut Swift.org hat dies folgende Vorteile.

- a) Der Hauptvorteil ist, dass die REPL einen vollwertigen Debugger zur Verfügung hat, der sogar Breakpoint zulässt.
- b) Weiter kann sich die REPL von einem kritischen Error erholen, sodass eine Fehlermeldung angezeigt wird und die REPL nicht neu gestartet werden muss
- c) Die REPL erhält Zugang zu jeglichen Sprachfeatures
- d) konsistentes Format des Ergebnisses

Diese Vorteile wurden nicht im Zuge der Umsetzung geprüft und sind daher nur ein Übertrag der von Swift.org genannten Vorteile (Swift.org, 2017c).

Swift Package Manager Swift gliedert sich wie Node.js und Express.js in Modulen. Diese werden wie bei Node.js in einer Package Datei verwaltet. Diese Package.swift Datei enthält alle Abhängigkeiten, die für ein Projekt notwendig sind. Der Eintrag der Abhängigkeit wird manuell durchgeführt, indem diese in die Package.swift eingetragen wird. Diese wird als Array beschrieben und enthält die Source URL und die Version, die heruntergeladen werden soll. Diese Packages könnten danach mit dem *import*

... im Projekt zur Verfügung gestellt werden. Der Manager erstellt vorab eine Struktur für das Projekt, die aus der *Package.swift* Datei besteht, dem Verzeichnis *Sources* inklusive der Datei *Hello.swift* und einem Testverzeichnis indem bereits die Datei *HelloTests.swift* und *LinuxMain.swift* enthalten sind. Die *LinuxMain.swift* Datei importiert die notwendige Bibliothek *XCTest*, welche bereits im Absatz 2.3.2 beschrieben wurde (Swift.org, 2017b).

Xcode Playground support Xcode ist nur in geringen Teilen für Linux von Interesse, da die Hauptfunktionen nur mit der IDE Xcode verwendet werden können. Xcode ist wiederum nur für Mac konzipiert, sodass hierauf nicht näher eingegangen wird.

2.4 Vergleich der Implementierungen

In diesem Kapitel geht es um die Umsetzung des in `refsec:derreverenzserver` beschriebenen Express.js Servers als Swift-Server. Dieser soll die gleichen Anforderungen erfüllen, welche waren: 1. Verwendung eines aktuellen Frameworks 2. Dynamische Website Erstellung 3. Registrierung, Speichern von Userdaten und verschlüsseln des Passwortes 4. Login und Sessionverwaltung 5. Caching 6. Socketverbindung 7. Model View Controller Pattern 8. Logging 9. Kompression.

2.4.1 Framework

Als Framework für den Swift Server standen zwei verschiedene Frameworks zur Auswahl:

- Perfect: als Toolkit für Anwendungen und REST-Services
- Kitura: ein neues WebFramework von IBM

An dieser Stelle ist zu erwähnen, dass es ein weiteres Framework namens *Vapor* gibt, welches erst nach der Entwicklung des Servers mit Perfect zu spät entdeckt wurde.

Perfect Perfect wird auf deren Homepage als WebServer und Toolkit für Swift-Entwickler beschrieben, um Applikationen und andere REST-Services zu entwickeln. Es kann verwendet werden um clientseitig wie auch serverseitig zu programmieren und wird als ideales Backbone für Cloud und mobile Technologien bezeichnet. Entwickler können damit produktiver arbeiten und benötigen dazu nur eine Sprache. Bereits auf der Homepage wird Perfect mit Node.js selbst verglichen und bietet eine Vielzahl

Anforderung	Node.js	Perfect
Dynamische Website Erstellung	handlebars	Workaround
Registrierung	-	-
Filehandling (speichern)	fs	Foundation
Verschlüsselung	bcrypt	SwiftBeaver
Login	-	-
Sessionverwaltung	express-session	PerfectSession
Caching	-	-
Socketverbindung	socket.io	PerfectWebSockets
Logging	winston	PerfectLogger und PerfectRequestLogger
Kompression	compression	GzipSwift
Routing (parsen von Requests und Responses)	express	PerfectHTTP

Table 2.1: Module der Server

an Bibliotheken an. Diese sollten mit den Node.js und Express.js Modulen, sofern diese benötigt wurden, im Zweck vergleichbar sein, welches in der folgenden Tabelle veranschaulicht wird.

2.4.2 Dynamische Website Erstellung

Wie der Tabelle zu entnehmen ist, wurde für die dynamische Website-Gestaltung ein Workaround implementiert. Ausschlaggebend dafür war das Fehlen von Kontrollstrukturen beim von Perfect entwickelten Package **Mustache**. Dieses Package hält wie auch Handlebars die Möglichkeit bereit, Bereiche eines HTML-Layout bzw. Templates, die mit "`{{}}`" gekennzeichnet wurden, mit Variablen zu füllen. Unterschied liegt darin, dass Mustache keine Kontrollfluss-Befehle unterstützt wie z.B. Handlebars mit einer If-Abfrage.

```

1 const hbs = require('handlebars');
2 const layout = require('./layout');
3 const fs = require('fs');
4 const logger = require('../controllers/logging.controller').logger;
5
6 hbs.registerPartial('chat', fs.readFileSync('./views/partials/Chat.html', 'utf-8'));
7
8 function renderChatView(user) {
9     logger.debug("renderChatView for: " + user);
10    const viewModel = {bodyPartial: 'chat', user: user};

```

```

11     return layout(viewModel);
12 }
13
14 module.exports = {
15     renderChat: renderChatView
16 };

```

Listing 2.2: Handlebars

Wie im Listing 2.2 zu sehen ist, wird Handlebars import und in Zeile 10 das zuvor registriert HTML Partial abgerufen und in der Variable `viewModel` gespeichert. Für das Partial interessant ist, dass es eine Kontrollstruktur hat, welche darauf prüft ob die Variable "user" auch tatsächlich einen Wert hat. Sollte dies nicht der Fall sein, wird der Platzhalter im Partial nicht angezeigt. Dass der "user" nicht *null* ist, wird dadurch verhindert, dass der User als Session-Variable verwendet wird und es sich beim Chat um einen gesicherten Bereich handelt. Der bearbeitete Partial wird danach dem Layout übergeben, sodass eine vollständige Seite erstellt ist.

```

1 import Foundation
2 import PerfectLib
3 import PerfectLogger
4 ...
5 /// pre read layout
6 var layout: String! = nil
7 var msg: String = ""
8 func setupLayout() -> Void {
9     if (layout == nil) {
10         layout = getFileView(file: "layout.html")
11     }
12 }
13
14 var navbar: String! = nil
15 func setupNavBar() -> Void {
16     if (navbar == nil) {
17         navbar = getFileView(file: "navbar.html") ?? ""
18     }
19 }
20 ...
21 /// Build page from layout and navbar withview
22 func buildView(_ view: String?, _ user: String, _ msg: String) ->
    String? {
23     var result = layout?.replacingOccurrences(of: "{{ main }}", with:
        view ?? "Partial not found!!")
24     result = result?.replacingOccurrences(of: "{{ navbar }}", with:
        navbar ?? "<a href=\"\"/>No navigation</a>")

```

```

25     result = result?.replacingOccurrences(of: "{{ user }}", with:
        user)
26     result = result?.replacingOccurrences(of: "{{ host }}", with:
        Config.host)
27     result = result?.replacingOccurrences(of: "{{ port }}", with: "
        \ (Config.port) ")
28     result = result?.replacingOccurrences(of: "{{ message }}", with:
        msg)
29     return result
30 }
31 ...
32 func renderChatView(user: String) -> String? {
33     LogFile.debug("Rendering Chat")
34     setupLayout()
35     setupNavBar()
36     return buildView(getFileView(file: "chat.html"), user, "")
37 }

```

Listing 2.3: Swift Workaround for Handlebars

Im Gegensatz ist für Swift, Lst.: 2.3 ein Workaround für den Zusammenbau der HTML implementiert. Verglichen mit einem Hausbau ist es auch in Swift notwendig, bei der Basis zu beginnen und die Seite nach und nach, nach oben zusammenzubauen. Die dafür beauftragte Methode **renderChatView** bekommt dafür den Usernamen übergeben und versucht die Webseite zusammenzubauen. Dazu wird mit der Methode "setupLayout" in einer globalen Variable das Layout gespeichert, danach die Navigationleiste ebenfalls wie das Layout vorbereitet und global gespeichert. Danach wird die Funktion "buildView" aufgerufen. Diese Funktion ersetzt zum Großteil die Funktionen von Handlebars. Die If-Abfrage wird hier jeweils mit den drei übergebenen Variablen durchgeführt. Ist eine der Variablen *nil*, so wird diese gegen einen leeren String ersetzt, sodass dieser in der fertigen Website nicht zu sehen ist.

Dabei ist zu beachten, dass diese Abfragen in Swift einzeilig sein können, sowie diese hier umgesetzt werden. So wird mit dem "?" erklärt, dass das Layout in Zeile 16 kein String sein muss, sondern auch *nil* sein kann. Danach wird im Layout der Substring `{{ main }}` gesucht und dieser durch das zuvor eingelesene Partial oder wenn nicht vorhanden mit "Partial not found" ersetzt. Da das Layout ein statisches File ist und das einlesen synchron erfolgt, sollte diese zumindest nie *nil* sein. Danach wird die Navigationsleiste eingefügt, der User in die Navigationsleiste gesetzt und ebenfalls im HTML enthaltenen Script für die Socketverbindung der Host und der Port gesetzt. Danach wird noch die Message, die vom User eingegeben worden ist, eingetragen und der gesamte HTML-Code vom Controller an den User gesendet.

Während es Node.js möglich ist, während dem Zusammenbau der Seite weitere Requests zu behandeln, muss der Swift-Server währenddessen blockieren und weitere Anfragen vorübergehend abweisen. Da das Zusammenbauen der Seite im Millisekunden Bereich liegt, ist dies noch kein Problem.

2.4.3 Registrieren, Speichern, verschlüsseln

Node.js und Express.js unterstützen von Haus aus das Serialisieren von Objekt in JSON-Strings, während für Swift ein größerer Aufwand notwendig ist, um ein Objekt in ein JSON-String zu konvertieren.

```
1 import Foundation
2 import PerfectLib
3 import PerfectLogger
4 import AES256CBC
5
6 public class Profile: JSONConvertibleObject {
7     static let registerName = "profile"
8     var username = ""
9     var password = ""
10    var firstname = ""
11    var lastname = ""
12    var email = ""
13    var key = ""
14
15    override public init() {
16    }
17
18    override public func setJSONValues(_ values: [String: Any]) {
19        self.username = getJSONValue(named: "username", from: values
20        , defaultValue: "")
21        self.password = getJSONValue(named: "password", from: values
22        , defaultValue: "")
23        self.firstname = getJSONValue(named: "firstname", from:
24        values, defaultValue: "")
25        self.lastname = getJSONValue(named: "lastname", from: values
26        , defaultValue: "")
27        self.email = getJSONValue(named: "email", from: values,
28        defaultValue: "")
29        self.key = getJSONValue(named: "key", from: values,
30        defaultValue: "")
```

```

25     }
26
27     override public func getJSONValues() -> [String: Any] {
28         return [
29             JSONDecoding.objectIdentifierKey: Profile.
registerName,
30             "username": username,
31             "password": password,
32             "firstname": firstname,
33             "lastName": lastname,
34             "email": email,
35             "key": key
36         ]
37     }
38 }

```

Listing 2.4: JSON stringify in Swift

Die gezeigte Definition eines Objektes, in diesem Fall das Profil eines Users, muss von der Klasse *JSONConvertibleObject* ableiten und die beiden Methoden *setJSONValues* und *getJSONValues* überschreiben. Zusätzlich muss das Objekt einen Registrierungsschlüssel erhalten wie in Zeile 8 in Lst: 2.4. Dieser Registrierungsschlüssel ist für die Identifizierung des JSON Objektes in der Registrierungstabelle für JSON Convertible Object notwendig. Dieser Schlüssel muss jedoch nicht persistiert werden, sondern wird beim Start des Servers angelegt und bleibt erhalten, solange der Server läuft. Das Registrieren des Objektes muss manuell mindestens einmal erfolgen, da ansonsten die Befehle für das Serialisieren fehlschlagen. Die Registrierung erfolgt mit:

```

1  JSONDecoding.registerJSONDecodable(name: Profile.registerName,
    creator: {return Profile()})

```

Listing 2.5: JSON Object registration

Dieser Befehl aus 2.5 wird im Server direkt beim Start des Servers ausgeführt und das Profil-Objekt ist somit automatisch registriert, sodass die Befehle *profile.jsonEncodedString()* zum Serialisieren oder *try profile?.jsonDecode()* zum Deserialisieren und gleichzeitigen initialisieren des Objektes ausgeführt werden können.

In Node.js reichen die Befehle: *JSON.stringify(profile)* und/oder *JSON.parse(profile)*, wobei das *JSON.parse()* kein fertig initialisiertes Objekt zurückgibt, sondern nur eine Collection im key:value Format.

Filehandling Für den Login ist es notwendig, die Daten zu speichern. Dieses ist natürlich auch mit verschiedenen Datenbanken möglich. Perfect bietet dafür Packages für SQLite, MySQL, MariaDB, PostgreSQL, FileMaker, MongoDB und Apache CouchDB. Bei diesem Projekt ist aber die Speicherung in ein JSON File verwendet worden, um das FileHandling der Standard Bibliothek zu testen. Die Swift Bibliothek hält zwei Funktionen bereit, welche für das FileHandling verwendet werden können.

- **DIR:** diese Library hält Funktionen bereit, die es ermöglichen, das **Working Directory** zu verändern. Damit wird der Prozess des Servers in das gewünschte Verzeichnis gelegt.
- **FILE:** diese Library ermöglicht es Files zu erstellen, zu befüllen, zu verändern und zu löschen. Dabei muss der Pfad und das File angegeben werden.

```

1 /*
2  * change working directory into exe
3  */
4 func setupDir(_: Void) -> Void {
5     let workingDir = Dir("./Sources/exe")
6     if workingDir.exists {
7         do {
8             try workingDir.setAsWorkingDir()
9             LogFile.debug("Working directory set to \(workingDir.
10                name)")
11         } catch {
12             LogFile.debug("error in getFile() setting WorkingDir: \(
13                error)")
14         }
15     } else {
16         LogFile.error("Directory \(workingDir.path) does not exist.
17             Main executable not started from root of MVC cannot find
18             resources?")
19         exit(2)
20     }
21     issetup = true
22 }
23 var issetup = false

```

Listing 2.6: Setzen des Working Directory

Das Working Directory wird beim Server mit der Methode *setupDir()* gesetzt, siehe Lst: 2.6. Diese Methode wird in der Main File beim Start des Servers aufgerufen, um diese in das Working Directory "exe" zu verlegen. Damit wird der Prozess in diesem Verzeichnis ausgeführt. Das Working Directory tiefer in die Baumstruktur des Servers

hineinzulegen, empfiehlt sich nicht, da es nicht mehr möglich ist, aus diesem Verzeichnis heraus zu kommen. Jegliche File Operation muss nun von diesem Verzeichnis aus angegeben werden, um Files zu erstellen oder zu lesen.

```

1 /**
2  * gets content of the specified file and returns it as String
3  */
4 func getFile(file: String) -> String? {
5     LogFile.debug("Trying to access \(file)")
6     ...
7     let thisFile = File(file)
8     LogFile.debug("file set to \(thisFile.path)")
9     do {
10         try thisFile.open(.readWrite)
11     } catch {
12         LogFile.error("Could not open file, error: \(error)") //
Error
13     }
14     var content: String? = nil
15
16     defer {
17         thisFile.close()
18     }
19     do {
20         content = try thisFile.readString()
21     } catch {
22         LogFile.error("not able to read File: \(error)") // Error
23     }
24     return content
25 }

```

Listing 2.7: Methode zum lesen einer Datei in Swift

Das FileHandling wird in Node.js durch das Modul "fs", das standardmäßig bei der Initialisierung installiert wird, ermöglicht:

```

1 profileExists: function (username, email) {
2     username = val.blacklist(username, new RegExp('\\W') );
3     var file = dataDir + username + "/" + username + '.json';
4     try{
5         var data = fs.readFileSync(file, 'utf-8');
6         var fields = JSON.parse(data);
7         if(fields.email === email) {
8             return true;
9         } else {
10             logger.info("User " + username + "(" + email + ") does not

```

```
        exist");
11         return false;
12     }
13     } catch (err) {
14         return false;
15     }
16 }
```

Listing 2.8: Methode zum lesen einer Datei in Node.js

Auf beiden Seiten, Swift und Node.js, wird das File synchron gelesen, da es für die Anmeldung wichtig ist, dass der User auch tatsächlich berechtigt und vorhanden ist. Ansonsten würde es auch möglich sein, per Callbacks das Filehandling durchzuführen. Bei Node.js ist dies leichter durchzuführen als bei Perfect, da Swift selbst zwar Callbacks unterstützt, Perfect jedoch noch nicht.

Verschlüsselung Für die Sicherheit der User Daten ist auch das Verschlüsseln der Passwörter wichtig. Dazu wird in beiden Fällen, Perfect und Express, eine AES256 Verschlüsselung vorgenommen, die "gesalzen" wurde. Unter "gesalzen" versteht man das Verschlüsseln von Daten zu einen Hash, der mit einem sogenannten "Salt" erstellt wird. Der Salt ist eine beliebig lange Folge an Zeichen, die im JSON File des Users mit abgespeichert werden muss, damit eine Authentifizierung in Folge des Nachberechnens des Hashes oder entschlüsseln des Hashes erfolgen kann. In Express wurde dazu das Modul "Bcrypt" verwendet, auf Seiten Perfect das Package "SwiftyBeaver". Beide Packages erfüllen den gleichen Zweck und sind sehr leicht in der Handhabung, außer dass bei SwiftyBeaver zur Authentifizierung des Users das gespeicherte Passwort mit dem Salt entschlüsselt werden muss, während bei Bcrypt der Hash mit dem Schlüssel (=Salt) nachgerechnet wird. Bei bestandener Überprüfung wird eine Session gestartet und der User auf den internen Bereich geleitet.

2.4.4 Session

Die Sessionverwaltung ist in beiden Fällen mit einem zusätzlichen Modul gelöst. Express bietet hierfür die "express-session" an, während Perfect die "PerfectSession" bereit hält. Die Beiden benötigen das Erstellen einer Session pro User und beim Logout die Löschung dieser. Das Initialisieren der beiden Session ist jeweils ein Einzeiler, der leicht verständlich ist. Interessanter ist die Zerstörung der Session, da diese in express-session durch eine Methode "destroy" bereitsteht, während in PerfectSession diese selbst zu implementieren ist:

```

1 const destroySession = function (req, res) {
2   logger.debug("destroySession");
3   req.session.destroy();
4   res.redirect(303, "/login")
5 };

```

Listing 2.9: Löschen der Session in Node.js

```

1 func logoutHandler(request: HTTPRequest, _ response: HTTPResponse) {
2   LogFile.debug("logoutHandler()")
3   if let _ = request.session?.token {
4     request.session = PerfectSession()
5     response.request.session = PerfectSession()
6   }
7   sendSeeOther(response, value: "/")
8 }

```

Listing 2.10: Löschen der Session in Swift

Bei der in 2.10 verwendeten Methode wird die alte Session mit einer neu initialisierten Session überschrieben.

2.4.5 Caching

Für das Caching wurde in Node.js und in Swift die Caching Header gesetzt. Beide Sprachen sind hier laut ihrer Dokumentation zu setzen.

2.4.6 Socketverbindung

Die Socketverbindung verlangt jeweils eine Anfrage des Clients, um die Verbindung von einer statuslosen HTTP Verbindung zu einer WebSocketverbindung aufzuwerten. Dies ist in Node.js mit dem Modul "socket.io" mit einer Anfrage und der Entgegennahme der Anfrage in wenigen Schritten erledigt:

```

1 $(function () {
2   var socket = io.connect('http://localhost:3000');
3   $('form').submit(function () {
4     var username = document.getElementsByTagName("span");
5     for (var j = 0; j < username.length; ++j) {
6       user = username[j].outerHTML;
7     }
8     user = user.match(/>([a-z0-9A-Z]*)</);
9     var message = user[1] + ": " + $('m').val();

```

```

10         socket.emit('chat message', message);
11
12         $(' #m').val('');
13
14         return false;
15     });

```

Listing 2.11: Clientseitiger Code für die Socketverbindung

Die Anfrage des Clients wird mit der Zeile 2 in 2.11 erledigt.

Der Server wird gesamt in Node.js zu einem WebSocket-Server aufgewertet:

```

1  const http = require('http').Server(app);
2  ...
3  const server = app.listen(config.port);
4  const io = require('socket.io').listen(server);
5  ...
6  io.on('connection', function(socket) {
7      socket.on('chat message', function(msg) {
8          io.emit('chat message', msg);
9      });
10 });

```

Listing 2.12: Serverseitiger Code für die Socketverbindung

Dabei ist anzumerken, dass die Initialisierung des Servers, wie in 2.12 Zeile 1-4 von der Dokumentation von Socket.io (Socket.IO, o.J.) abzuändern ist, damit die Socketverbindung aufgebaut werden kann.

Beim Swift Server wird die Socketverbindung wie folgt erreicht:

```

1  const socket = new WebSocket('ws://{ host }:{ port }/socket', '
    chat');

```

Listing 2.13: Clientseitiger Code für die Socketverbindung

```

1  import PerfectWebSockets
2
3  /**
4   * Handler for chat
5   */
6  class ChatHandler: WebSocketSessionHandler {
7      // Protocol
8      let socketProtocol: String? = "chat"
9      var user: String? = nil
10     func handleSession(request: HTTPRequest, socket: WebSocket) {
11         socket.readStringMessage {

```

```

12         string, op, fin in
13         guard let string = string else {
14             if let u = self.user {
15                 left(by: u)
16                 for (_, conn) in chat {
17                     conn.sendMessage(string: "User \(u)
logged out", final: true) {
18                         self.handleSession(request: request,
socket: socket)
19                     }
20                 }
21                 LogFile.info("User \(u) left Session")
22             }
23             socket.close()
24             return
25         }
26         if let u = request.session?.token {
27             LogFile.info("User \(u) joined Session")
28             joined(by: u, on: socket)
29         }
30         LogFile.debug("Socket:: read:\(string) op:\(op) fin:\(
fin)")
31         for (user, conn) in chat {
32             conn.sendMessage(string: string, final: true)
33             {
34                 LogFile.debug("Sending to \(user)")
35                 self.handleSession(request: request, socket:
socket)
36             }
37         }
38     }
39 }

```

Listing 2.14: Serverseitige Socketverbindung

Bei PefectWebSocket ist es notwendig 2.14, dass das Erzeugen und auch das Beenden der Socketverbindung ausprogrammiert wird.

2.4.7 Logging

Um das Entwickeln leichter zu machen, wurden in Express der Winston Logger verwendet. Dieser Logger unterstützt die Ausgabe von Informationen auf der Konsole

und das Persistieren der Information in einem Log-File. Außerdem kann das Logging-Level des Loggers eingestellt werden, sodass während der Implementierung wichtige Informationen für die Entwicklung ausgegeben werden. Danach kann der Logging-Level erhöht werden, sodass nur Informationen die z.B. für einen Kunden wichtig sind, ausgegeben werden. Was ausgegeben wird, wird vom Entwickler während der Implementierung festgelegt (Robinson, Charlie / Bergman, Matthew / Squires, Marak, 2017-05-02).

2.4.8 Kompression

im Gegensatz zu Node.js, dass die Middleware *compression* für das komprimieren der Daten bereit hält, muss in Swift und dem Modul *GzipSwift* eine Funktion implementiert werden. Weiter muss auch der encoding header manuell gesetzt werden, damit der Browser die erhaltenen Daten verarbeiten kann.

```

1
2 ...
3 func loginGetHandler(_ request: HTTPRequest, _ response:
    HTTPResponse) {
4     LogFile.debug("loginGetHandler()")
5     if let view = renderLoginView("") { // renderLoginView gives
        either String or nil -> renderLoginView
6         response.appendBody(bytes: compress(view)) // String
7         response.addHeader(HTTPResponseHeader.Name.contentType,
            value: "Gzip")
8     ...
9 func compress(_ data: String) -> [UInt8] {
10     LogFile.debug("Trying to compress data")
11     let enc = data.data(using: .utf8)!
12     let zipped = try? enc.gzip()
13     return [UInt8](zipped!)
14 }
```

Listing 2.15: Kompression und Header

In Zeile 5 des Codes 2.15 wird die Kompressionsfunktion aufgerufen und in Zeile 6 der Header gesetzt. Für die Kompression müssen die Daten in Unicode zerlegt werden, danach gezippt werden und zum Schluss noch in ein Bytearray zerlegt werden. Dieses Bytearray wird danach in der Response als Body gesendet. Wenn auch Swift einen Mehraufwand gegenüber Node.js in der Programmierung verursacht, ist die Komprimierung der Daten in Swift stärker als bei Node.js.

2.5 Leistungsvergleich

Der Leistungsvergleich der Server wurde mit Apache Bench durchgeführt, nachdem mit YSlow die Server auf den gleichen Stand gebracht worden sind. Für den Benchmark wurde folgende VM verwendet:

- Ubuntu 16.04 LTS
- RAM: 6.360 MB
- CPU: 1
- Netzwerk: Intel(R) Ethernet Connection (2) I218-V

Da ein Rechner mit nur einer CPU sehr unwahrscheinlich ist, wurde der Swift und der Node.js auch auf einer Maschine mit 4 Kernen getestet. Leider wurde der Swift-Server aufgrund einem *mallo(): corrupted memory action* abgebrochen. Der Grund dafür wird im Kapitel Probleme und Lösungen behandelt. Für das Testen jedoch müsste der Befehl Benchmark insofern geändert werden, dass der Swift-Server keine gleichzeitigen Requests bekommt. Wird jeder Request nacheinander abgesendet funktioniert auch das abarbeiten der Request auf mehreren CPUs.

2.5.1 Yslow

Mit Yslow wurden die beiden Server angeglichen, um die Tests mit Apache Bench aussagekräftig gestalten zu können. Trotz das bei der Kompression des Swift-Server die Note **F** steht ist im Header und auch in der Netzwerkanalyse des Chrome und Firefox Browsers angezeigt.

A Make fewer HTTP requests	A Make fewer HTTP requests
E Use a Content Delivery Network (CDN)	E Use a Content Delivery Network (CDN)
A Avoid empty src or href	A Avoid empty src or href
B Add Expires headers	A Add Expires headers
F Compress components with gzip	F Compress components with gzip
A Put CSS at top	A Put CSS at top
A Put JavaScript at bottom	A Put JavaScript at bottom
A Avoid CSS expressions	A Avoid CSS expressions
n/a Make JavaScript and CSS external	n/a Make JavaScript and CSS external
A Reduce DNS lookups	A Reduce DNS lookups
B Minify JavaScript and CSS	C Minify JavaScript and CSS
A Avoid URL redirects	A Avoid URL redirects
A Remove duplicate JavaScript and CSS	A Remove duplicate JavaScript and CSS
A Configure entity tags (ETags)	A Configure entity tags (ETags)
A Make AJAX cacheable	A Make AJAX cacheable
A Use GET for AJAX requests	A Use GET for AJAX requests
A Reduce the number of DOM elements	A Reduce the number of DOM elements
A Avoid HTTP 404 (Not Found) error	A Avoid HTTP 404 (Not Found) error
A Reduce cookie size	A Reduce cookie size
C Use cookie-free domains	C Use cookie-free domains
A Avoid AlphaImageLoader filter	A Avoid AlphaImageLoader filter
A Do not scale images in HTML	A Do not scale images in HTML
A Make favicon small and cacheable	A Make favicon small and cacheable

(a) Yslow des Node-Servers (b) Yslow des Swift-Servers

2.5.2 Apache Bench

Im ersten Schritt wird der Server mit 500 gleichzeitig durchgeführten Requests beauftrag bis insgesamt 5000 Request abgehandelt wurden. Zur weiteren Überprüfung wird der Test mit 10.000, 20.000, 30.000, 40.000 und 50.000 zu je 500 parallelen Requests durchgeführt. Daraus wurde folgendes Ergebnis erstellt:

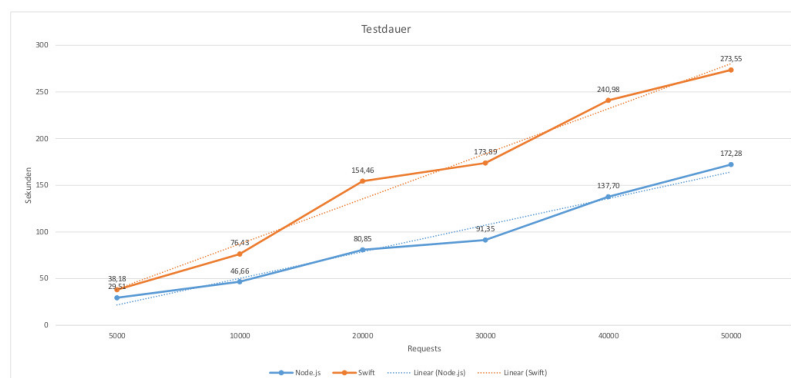


Figure 2.1: Request-Abarbeitung von Express.js und Perfect

	Express.js	Perfect
Document	LoginPage	LoginPage
Document Length	1.988 bytes	932 bytes
Concurrency Level	500	500
Time taken for tests [sec]	29,507	38,178
Complete Requests	5.000	5.000
Failed requests	0	0
Total transferred [bytes]	11.070.000	6.950.000
HTML transferred [bytes]	9.940.000	4.660.000
Requests per second [#/sec]	169,45	130,97
Time per request [ms] (mean)	2.950,702	3.817,783
Time per request [ms] (across concurrent requ.)	5,901	7,636

Table 2.2: Test: 5.000 Request on "Login Page"

	Express.js	Perfect
Document	LoginPage	LoginPage
Document Length	1.988 bytes	932 bytes
Concurrency Level	500	500
Time taken for tests [sec]	172,275	273,546
Complete Requests	50.000	50.000
Failed requests	0	0
Total transferred [bytes]	110.700.000	695.000.000
HTML transferred [bytes]	99.400.000	466.000.000
Requests per second [#/sec]	290,23	182,78
Time per request [ms] (mean)	1.722,747	2735,462
Time per request [ms] (across concurrent requ.)	3,445	5,471

Table 2.3: Test: 50.000 Request on "Login Page"

Bei dem Test wurde die Login-Page angefordert. Dabei änderte sich nur die Menge der Requests die abgehandelt werden mussten. Dadurch ergab es sich, dass die Menge der übertragenen Bytes linear anstieg, die Zeit pro Request bei mehrmaligen durchlaufen jedoch stark schwankte. Die Differenz der Dokumentenlänge ergibt sich aus der unterschiedlich starken Kompression der Server. Somit blieb für die Abb. 2.1 nur ein Wert der sich auf beiden Seiten ändert, die *Dauer des Tests*.

2.6 Fazit

Aufgrund der Werte in der Abb.: 2.1 kann der Rückschluss gezogen werden, dass jeder zusätzliche Request einen linearen Anstieg der Durchführungsdauer zur Folge hat. Diese Aussage zeigt, dass der Swift Server sowie auch der Node.js Server die Requests synchron abarbeiten, wobei Node.js die Zeit jedoch mit Callbacks effizienter nutzt. Der Node.js Server verwendet die Zeit für I/O-Operationen, auch wenn diese synchron implementiert wurden, um nachfolgende Requests weiter zu bearbeiten. Ein großes Defizit hat der Perfect Server aufgrund des Workarounds um die Funktion von Handlebars nachzustellen. Dieses Workaround blockiert bei jedem einlesen der Partials, welche als eigene Files gespeichert sind.

Zusätzlich nimmt der Swift Server keinen folgenden Request entgegen, bevor nicht der zurzeit bearbeitete Request abgeschlossen und gesendet wurde. Swift hat einen Vorteil durch die stärkere Komprimierung die zwar mehr Zeit beim bearbeiten von Requests benötigt, jedoch eine geringere Menge an Daten übermittelt werden müssen.

2.7 Probleme und Lösungen

Der Swift Server wurde auf einer VM getestet, die nur einer CPU verfügt, wie bereits in 2.5 erwähnt. Nachdem die VM um drei weitere Kerne auf 4 erweitert wurde, konnte der Server die Requests nicht mehr Ordnungsgemäß ausführen. Das lag daran, dass der Server nicht für Multitasking ausgelegt wurde, und der Server beim Bearbeiten von Requests versucht, gleichzeitig das benötigte HTML für mehrere Requests zusammenzubauen. Auf einem Rechner mit mehreren Kernen muss der Apache Bench insofern abgeändert werden, dass der *Concurrency Level* auf 1 gestellt ist. Damit sind die Test jedoch wieder nicht aussagekräftig, und der Server müsste in diese Richtung überarbeitet werden. Schlussfolgernd, würde der Server auch ein viertel der Zeit benötigen, verglichen mit der derzeitigen Entwicklung.

Chapter 3

Conclusion and Outlook

Your text here ...

Acronyms

OS	operating system
IDE	integrated development environment
BSD	Berkeley Software Distribution
SSA	Static Single Assignment
AST	Abstract Syntax Tree
SIL	Swift Intermediate Language
LLVM IR	LLVM Intermediate Representation
SDK	Software Development Kit
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
XML	Extensible Markup Language
REPL	read-eval-print-loop
CPU	
REST	

Bibliography

- Hahn, Evan M. (2016). *Express in Action, Writing, building, and testing Node.js applications*. Shelter Island, NY, USA: Manning Publications. ISBN: 9781617292422.
- JetBrain s.r.o. (2017). *CLion, A cross-platform IDE for C and C++*. Available from: <https://www.jetbrains.com/clion/> [July 28, 2017].
- LLVM Foundation (o.J.). *clang: a C language family frontend for LLVM*. Available from: <https://clang.llvm.org/index.html> [July 29, 2017].
- (2017). *The LLVM Compiler Infrastructure*. Available from: <https://llvm.org/> [July 29, 2017].
- Robinson, Charlie / Bergman, Matthew / Squires, Marak (2017-05-02). *winston*. Available from: <https://github.com/winstonjs/winston#logging-levels>.
- Socket.IO (o.J.). *How to user - Using with Express 3/4*. Available from: <https://socket.io/docs/#>.
- StrongLoop, Inc./IBM (o.J.a). *Express - Schnelles, offenes, unkompliziertes Web-Framework für Node.js*. Available from: <https://expressjs.com/de/> [July 29, 2017].
- (o.J.b). *LoopBack - The Node.js API Framework*. Available from: <http://loopback.io/> [July 29, 2017].
- Swift.org (2017a). *Compiler Architecture*. Available from: <https://swift.org/compiler-stdlib/#compiler-architecture>.
- (2017b). *Package Manager*. Available from: <https://swift.org/package-manager/>.
 - (2017c). *REPL and Debugger*. Available from: <https://swift.org/lldb/#why-combine-the-repl-and-debugger>.
 - (2017d). *Standard Library Design*. Available from: <https://swift.org/compiler-stdlib/#standard-library-design>.

Swift.org (2017e). *Swift Core Libraries*. Available from: <<https://swift.org/core-libraries/#foundation>>.