FH | JOANNEUM
University of Applied Sciences

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: 1510418049, 1510418043

July 2017

Assignment for the project work of
Stefan Moder, Christian Hofer
Matr. no. 1510418049, 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

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Kurzfassung

Kurzfassung In dieser Arbeit wurde mit Hilfe des Perfect Frameworks ein Webserver auf einem Ubuntu Server 16.04 LTS erstellt. Zur Erstellung wurde die Programmiersprache Swift in der Version 3.1.1 verwendet, welches open-source für die Verwendung mit Linux zur Verfügung steht. Der implementierte Server wurde mit einem Webserver auf Basis von Express.js Frameworks verglichen. Zur Dokumentation des Projektes wurde ein WikiBook erstellt, welches beschreibt wie man allgemein einen Webserver mit Swift auf Linux erstellt.

[Stand Juli 2017]tiobe liegt Swift auf Platz 12 hinter Go (Platz 10) und Perl (Platz 11), aber vor Ruby (Platz 13). Auch ist dem Index zu entnehmen, dass sich Swift um zwei Plätze verbessert hat. In der *Developer Survey* von stackoverflow schafft es Swift auf Platz 11 der meist benutzen Programmiersprachen und im Ranking der Beliebtesten sogar auf Platz 4 vor Go und Python(**survey2017**). Im Jahr zuvor war Swift bei der Beliebtheit sogar auf Platz 2(**survey2016**). Besonders die Verwendbarkeit von Swift für alle Arten von Projekten, von iOS Development, über systemnahe Programmierung, mobile Apps, bis hin zu Cloud Technologien, sorgen für eine hohe Beliebtheit. Genauso wie die Sicherheit von Swift, wie durch automatische Speicherverwaltung, Array Indexes werden auf out-of-bound Zugriffe geprüft, Optionals erzwingen die Behandlung von möglichen nil-Werten und Variablen sind immer initialisiert. Andererseits scheinen bestimmte Teile der Programmiersprache wie Closures, Error-Handling und Optional gerade Einsteigern das Erlernen von Swift zu erschweren(**studyonswift**).

Basics of Swift Um die Sprache zu verstehen, beziehungsweiße um die Erlernbarkeit zu beurteilen folgt nun eine kleine Einführung basierend auf *The Swift Programming Language*(**swiftbook**). Swift kennt Konstanten und Variablen, wobei durch eine Doppelpunkt getrennt der Typ annotiert wird. Wenn möglich kann der Typ auch inferiert werden.

```
1        var : String =    Swift     // Ein String
2        var =  FH -Joanneum  // Auch ein String durch
    Typinferenz
3        let =    Linux     // Eine Konstante
4        var x = 0, y = 1, z = 3  // Mehrere Variablen in einer Zeile
```

Listing 1.1: Variablen und Konstanten

Als nächstes Kollektionen. Swift kennt Array, Set, Dictionary. Diese sind standardmäßig „mutable“, das heißt änderbar, wird aber die Kollektion als `let` sprich Konstante initialisiert, wird sie unveränderbar.

```
1 var intarray = [Int]()   // Ein Array aus Integern
2 var stringarray = [   Bier   ,    Wein   ,
3           Wasser   ] // Array bereits mit Werten gef llt
4 let anotherone = [Int]()        // Immutable Array
5 var intset = Set<Int>() // Ein Set, der Typ muss hashbar sein
6 var dictionary = [Int: String]()  // Ein Dictionary mit einem Key (
    Int) Value(String) Paar
```

Listing 1.2: Array

Der Kontrollfluss wird in Swift ähnlich zu anderen Sprachen geregelt. Swift kennt `for`, `while`, `if` und `switch`.

```
1 let demo = [   Eins   ,    Zwei   ,    Drei   ,    Vier   ]
2 // Ein for-Loop
3 for number in demo {
4        print( Nummer   = \(number)!   ) // print() schreibt in die
    Konsole
5 }
6 // Ausgabe
7 // Nummer = Eins!
8 // Nummer = Zwei!
9 // Nummer = Drei!
10 // Nummer = Vier!
11 var  counter = 0
12 var max = 2
13 // Ein while-Loop
14 while counter < max {
```

```
15         print(  Der  Z hler ist \(counter)  )
16         counter = counter + 1
17 }
18 // Ausgabe
19 // Der Z hler ist 0
20 // Der Z hler ist 1
21
22 // repeat while
23 repeat {
24         print(  Der  Z hler ist \(counter)  )
25 } while counter > 0
26 // Ausgabe
27 // Der Z hler ist 1
28 // Der Z hler ist 0
29
30 var age = 19
31 // If Abfragen
32 if age >= 18 { // Eine Altersabfrage
33    print( Hier  ist ein  B i e r )
34    //  ber  18 bekommt man Bier
35 } else {
36   print ( Hier  ist ein Glass mit  W a s s e r )
37   // Unter 18 ein Glass Wasser
38 }
39
40 let char: Character =    F
41 // Switch Case
42 switch char {
43     case    A   :
44         // Hier ist der Code  f  r  A
45     case    B   ,    C   :
46         // Hier ist der Code  f  r  B und C
47     case    F   :
48         print( Der  Code von  F  )
49         fallthrough
50         // Der nachfolgende Case wird auch ausgef hrt
51     case    D   :
52         print( Der  Code von  D  )
53     default:
54         // Wenn nichts passt
55 }
56 // Ausgabe
57 // Der Code von F
58 // Der Code von D
59
```

```
60 // Label und break
61 outer: while true { // while-Loop mit Label
62         for i in 0..<100 { // W rde von 0 bis 99 z hlen
63                 print( Index \(i) )
64                 break outer
65                 // Bricht beide Loops und springt zum Label
66         }
67 }
68 // Ausgabe
69 // Index 0
```

Listing 1.3: Array

Funktionen in Swift bestehen aus Funktionsname, Parameter (eventuell mit Label) und Rückgabewert. Funktionen haben nur einen Rückgabewert, dieser kann aber ein Optional sein, bzw. können Funktionen auch Fehler werfen.

```
1  // Eine Funktion die einen String als Parameter bekommt
2  // und einen String zur ck gibt
3  //    sag    ist das Label des Parameters
4  //    name    ist der Name der Variable
5  func einefunktion(sag name: String)     String {
6          print( Name = \(name) )
7          return Ein String als R ckgabewert
8  }
9  // Aufruf
10 einefunktion(sag: Bert )
11 // Ausgabe (der R ckgabewert wird hier ignoriert)
12 // Name = Bert
13
14
15 // Mit _ kann das Label beim Aufruf unterdr ckt werden
16 // Void als R ckgabewert bedeutet kein R ckgabewert
17 func keinlabel(_ name: String)     Void {
18         print( Name = \(name) )
19 }
20 // Aufruf
21 keinlabel( Hugo ) // Name = Hugo
22 einefunktion(sag: Hugo ) // Name = Hugo
23
24 // In-Out Parameters erlaubt Parameter in der
25 // Funktion zu ndern , sofern
26 //diese keine Konstanten sind
27 func wechseldich(wechsel name: inout String) {
28         name = Bob
29 }
```

```
30 // Aufruf
31 var name =    Billy
32 print(name) // Billy
33 wechseldich(wechsel: name)
34 print(name) // Bob
```

Listing 1.4: Funktionen

Der nächste Punkt ist Closures, diese entsprechen in etwa Lambda-Ausdrücken. Diese erlauben Callback-Funktionen direkt inline zu definieren.

```
1 // Diese Funktion ben tigt eine Callback Funktion
2 func function(callback: (String)  Void  ) {
3         callback(   Gru    von der  F u n k t i o n )
4 }
5 // Und hier der Aufruf mittels Closure
6 function(callback: { s in print(s) }) // G r u   von der Funktion
7 // Oder in der Trailing-Variante
8 function(callback: (String)      Void ) {
9         s in print(  Trailing   \(s)    )
10 }
```

Listing 1.5: Closures

Vor den Klassen und Strukturen noch eine schnelle Einführung in Enumerations.

```
1 // Hier eine Enumeration mit komplexerm Typ
2 enum FHEntity {
3     case student(nr: Int, name: String)
4     case lehrender(name: String)
5 }
6
7 var std = FHEntity.student(nr: 13, name: "Geheim")
8 // Lege einen neuen Studenten an
9
10 switch std {
11     case .student(let nr, let name):
12         print("\(name)") // Student
13     case .lehrender(let name):
14         print("\(name)") // Lehrender
15 } // Ausgabe Geheim
```

Listing 1.6: Enumerations

Und nun zu Strukturen und Klassen in Swift. Klassen können mehr als Strukturen und zwar Vererbung, Type-Casting, Deinitialisierung und ARC. Werden Klassen einer

neuen Variable zu geordnet, wird nur die Referenz übertragen, bei Strukturen wird eine Kopie erstellt.

```
// Eine einfache Klasse
class A {
    var name: String
    init(_ name: String) {
        self.name = name
    }
}
// Eine Einfache Struktur
struct B {
    var name: String
    init(_ name: String) {
        self.name = name
    }
}
var a = A("Montag") // neue Instanz der Klasse A
var b = B("J nner") // neue Struktur B
var newera = a // nur Referenz
var newerb = b // neue Struktur mit kopierten Werten
newera.name = "Dienstag" // neuer Name f r die Klasse
newerb.name = "Februar" // neuer Name f r die Struktur
print("Die alte Klasse ist \(a.name) " +
      "die neue ist \(newera.name)")
print("Die alte Struktur ist \(b.name) " +
        "die neue ist \(newerb.name)")
// Die alte Klasse war Dienstag die neue ist Dienstag
// Die alte Struktur war J nner die neue ist Februar
```

Listing 1.7: Klassen und Strukturen

Die Funktionen einer Klasse sind vom Syntax her leicht anders als in anderen Programmiersprachen, aber dafür sehr intuitiv.

```
// Eine Klasse mit zur Veranschaulichung
class SomeClass {
        lazy data = Data() // lazy initialisiert erst bei der ersten
    Nutzung
        var name: String
        var value: Int {
                get { // Der Getter
                        return 42 * 12 // Kalkuliert diesen Wert bei
    der R ckgabe.
                }
                set(newValue) { // Der Setter, w rde auch ohne
    newValue gehen
```

```
10                          self.value = newValue
11                  }
12          }
13
14          //    Mutating    sorgt daf r , dass die Funktion
15          // die Membervariablen direkt setzten darf.
16          mutating func newSomeclass(data: Data, name: String) {
17                  self.name = name
18                  self.data  = data
19          }
20          class func klassenfunktion() {
21                  // Klassen Funktionen sind wie statische Funktionen
    in
22                  // anderen Programmiersprachen und kann vererbt
    werden,
23                  // bei Strukturen wird static als Schl sselwort
    verwendet
24          }
25
26          deinit {
27                  // Die Instanz wird zerst rt
28          }
29  }
30
31  // Vererbung wir mit    :    erm glicht
32  class SomeSubClass: SomeClass {
33          // Sub class of SomeClass
34          // Mit dem Schl sselwort override werden
35          // Funktionen der Eltern  berschrieben
36  }
```

Listing 1.8: Wie funktionieren Klassen

Nun zum ARC, der Speicherverwaltung von Swift. Bei ARC werden die Referenzen gezählt die auf ein Objekt zeigen und erst wenn es keine Referenzen mehr gibt wird die Instanz gelöscht. Memoryleaks können in Swift dann entstehen wenn zwei Instanzen gegenseitig Referenzen auf einenander haben. Um dieses Problem zu lösen wird das Schlüsselwort **weak** verwendet.

```
1  class Auto {
2      // Das Schl sselwort weak, sorgt daf r ,
3      // dass der Motor deinitialisiert, sobald
4      // nur noch schwache (weak) Referenzen auf
5      // ihn zeigen
6      weak var motor: Motor?
```

```
7       // Jedes Auto hat einen Motor
8
9       init(motor: Motor) {
10          self.motor = motor
11          self.motor!.auto = self
12      }
13      deinit { print("Das Auto wurde zerlegt")}
14  }
15  class Motor {
16      var auto: Auto?
17      // Jeder Motor geh rt zu einem Auto
18
19      init() {}
20      deinit { print("Der Motor wurde zerlegt")}
21  }
22
23  // neue Instanzen initialisieren
24  var v6: Motor? = Motor()
25  var porsche: Auto? = Auto(motor: v6!)
26
27  porsche = nil // Noch h lt Motor eine starke Referenz
28
29  v6 = nil // Nun h lt nur noch Auto eine schwache Referenz
30  // v6 wird deinitialisiert
31  // damit gibt es keine Referenz auf porsche
32  // und porsche wird deinitialisiert
33
34  // Ausgabe
35  // Der Motor wurde zerlegt
36  // Das Auto wurde zerlegt
```

Listing 1.9: Speicherverwaltung mit 'weak'

Das zweite Feature von Swift mit dem Einsteiger Probleme haben, sind Optionale und die Verkettung von Optionalen. Dabei handelt es lediglich um Variablen oder Konstanten die auch `nil` sein können, dass heißt sie können auch ein Rückgabewert sein. Um an den wahren Wert der Variable zu kommen, muss diese ausgewickelt (unwrapped) werden. Dies geschieht durch „?“ oder „!“.

```
1  var a: Int?     // Optional welches Int oder nil sein kann
2  var b: Int!     // Auch ein Optional, jedoch wickelt sich dieses
       automatisch aus
3
4  // Aufruf mit Optional Verkettung
5  if let var = a?.max { // a muss mit ? ausgewickelt werden, b nicht
```

```
6         // Kann .max aufgerufen werden
7 } else {
8         // a ist nil
9 }
10
11 // Optionale erlauben einen Defaultwert anzugeben
12 var c: Int?
13 c = a ?? 42 //  c hat den Wert von a oder wenn a nil ist 42
14
15 // Wird mit ! ausgewickelt, macht der Entwickler ein Versprechen,
16 // dass das Optional nicht nil ist
17 var d = c! // Dank dem Defaultwert von oben, ist c nie nil
```

Listing 1.10: Optionale und Verkettung

Zum Schluss zum basis Wissen über Swift noch Fehlerbehandlung (error handling). Fehler in Swift sind einfach Enumerationen. Funktionen, welche Fehler werfen können werden mit `throws` deklariert. Um die Funktion auszuführen kann ein do-try-catch-Block, try? oder try! verwendet werden.

```
1 // Ein Fehler in Swift
2 enum TutorialError : Error { case KnownError }
3 func schlecht() throws -> String {
4 // throws erlaubt der Funktion einen Fehler zu werfen
5     throw TutorialError.KnownError // throw wirft den Fehler
6 }
7 // do-try-catch-Block um Fehler zu fangen
8 defer {
9     // Der Code im defer-Block wird am Ende ausgef hrt
10    // egal ob try funktioniert oder ein Fehler auftritt
11 }
12 do {
13    try schlecht()
14 } catch TutorialError.KnownError {
15    // Der Fehler wird gefangen und der Code hier ausgef hrt.
16 }
17 // Mit try? wird Code ausgef hrt, tritt
18 // jedoch ein Fehler auf wird nil zur ckgegeben
19 var text: String? = try? schlecht() // Durch den Fehler wird text =
     nil
20
21 // try! l sst sich Code wie try? Ohne do-try-catch-Block ausf hren
22 // bei einem Fehler kommt es, aber zum Absturz
```

Listing 1.11: Fehlerbehandlung

Damit währen die Grundzüge der Programmiersprache erklärt. Jedoch gibt's einen wichtigen Aspekt von Swift der die Sprache für diese Arbeit erst interessant macht.

Swift ist Open-Source Swift ist seit Dezember 2015 unter der Apache-2.0-Lizenz verfügbar(**swiftorg**). Damit ist Entwicklung von Programmen auf Linux Rechnern mit Swift möglich. In dieser Arbeit beschäftigen wir uns im Speziellen mit Implementierung eines Webservers. Dabei stehen einige Frameworks zur Verfügen, auf die im Teil Stand der Technik genauer eingegangen wird.

1.2 State of the Art

Das Ziel der Erstellung eines Webservers auf einer Linux-Maschine mittels der Programmiersprache Swift erfordert eine Erhebung des Stands der Technik. Als erstes wurde erhoben welche Frameworks für die Implementierung eines Swift Webserver zur Verfügung stehen. Durch Nachforschung konnten die folgenden Webframeworks für Swift auf Linux gefunden werden: Perfect(**perfect**), Vapor(**vapor**), Kitura(**kitura**), Zewo(**zewo**), Swifter(**swifter**). Die Frameworks Zewo und Vapor laufen dabei unter der MIT Lizenz, während Perfect unter der apache, beziehungsweise Kitura unter der apache2 Lizenz zur Verfügung stehen. Alle Frameworks haben eine aktive und angerschierte Gemeinschaft von Entwicklern, welche meist über die Kommunikationsdienste Slack und Gitter, beziehungsweise Email erreicht werden können. Es folgen einige Kurzfassungen der Frameworks.

Frameworks

Zewo Zewo ist ein leichtgewichtiges Framework. Die Entwickler wurden durch die Programmiersprache Go inspiriert um ähnliche Nebenläufigkeit in mittels `libmil` zu erreichen. Durch die Einführung von Go-artigen *"communicating sequential processes"* versuchen die Entwickler von Zewo die Menge der oft schwere lesbaren Ketten von Callback-Funktionen zu reduzieren. Interessant ist auch, dass das Projekt sehr Modular aufgebaut ist. Auch haben die Entwickler von Zewo ihr Routing als Baumstruktur implementiert, die das Routing noch schneller machen soll. Als Nachteil muss gesagt werden, dass der Code oft nicht Kommentiert wurde und so sehr umständlich nach den richtigen Funktionen gesucht werden muss. Am Ende sollte zu Zewo noch

gesagt werden, dass die Entwickler sich mit dem Vapor-Framework 1.2 auf einen Http-Standard für Open-Source Swift geeinigt haben, der über https://github.com/open-swift/S4 abgerufen werden kann(**openswiftthttp**). Dies macht den Code zwischen Vapor und Zewo austauschbar.

Swifter Swifter ist ein absolut minimaler Http-Server. Mit der Entwicklung von Swifter wurde begonnen, bevor Swift Open-Source wurde. Das minimalistische Framework kann Dateien aus Ordnern teilen, statische HTML-Seiten ausliefern und Web-Sockets erstellen. Einerseits fehlen Dokumentation und API, andererseits sind durch die Beispiele auf der Projektseite auf GitHub alle Anwendungsmöglichkeiten abgedeckt.

Swifton Als kleine Anmerkung hier ist Swifton (Kofferwort aus: Swift on Rails). Hierbei handelt es sich um ein auf Zewo (1.2) basierendes Webframework welches stark von Ruby on Rails inspiriert wurde. Das Projekt wird aber zur Zeit nicht aktiv weiterentwickelt, da die Entwickler durch die stark statische Art von Swift das Projekt zu sehr eingeschränkt sehen(**swifton**).

Kitura Kitura ist ein Webframework von IBM und ein damit verbundenes starkes Entwicklerteam. Kitura unterstützt unteranderm Middleware, SSL/TLS und FastCGI. Dieses Framework ist in der Verwendung ähnlich zu Express.js und verwendet wieder Callback-Funktonen. Kitura funktioniert mit Cloud-Plattformen wie Bluemix und ist dabei autoskalierbar. IBM investiert stark in Open-Source Swift mit Workshops auf WWDC(**WWDC**), Präsentationen, Video-Einführungen(**ibmdemo**) und online Kursen auf Plattformen wie Udemy(**udemyswift**). Populären Blogs wie *raywenderlich.com* und *stormpath.com* zufolge ist das Ziel von IBM Java als Enterprise-Sprache durch Swift abzulösen und Kitura ist Teil dieser Unternehmung.(**rayenterprise**)(**stormswiftserver**) Zur API ist zusagen, dass die wichtigsten Referenzen vorhanden und dokumentiert sind.

Vapor Vapor ist mit Entstehungsdatum July 2016 eines der Ältesten der hier genannten Frameworks und wahrscheinlich das Älteste mit der Absicht Linux zu verwenden. Es unterstützt WebSockets. TLS, Trusted Encryption und BCrypt Hashes werden standardmäßig verwendet. Die API ist sehr ausführlich, wenn nicht sogar komplett, jedoch wie der Code selbst nicht vollständig dokumentiert, im Gegensatz zu Kitura1.2 wo die API nur das Nötigste enthält, aber dafür gut Dokumentiert ist. Wie bei Zewo 1.2 erwähnt gibt es einen Http-Standard auf den sich die Autoren geeinigt haben

(https://github.com/open-swift/S4). Dies macht den Code von Vapor mit Zewo austauschbar. Vapor scheint das Einsteiger freundlichste der hier genannten Frameworks zu sein.

Perfect Perfect verfolgt die Philosophie, Swift als fundamentale Programmiersprache zu verwenden, also Apps und Server in Swift programmieren. Dazu gibt Perfect volle Xcode Unterstützung für Entwicklung und Debugging. Perfect stellt Werkzeuge für mehr als nur für einen Webserver zur Verfügung, wie Datenbankenunterstützung, JSON-Bearbeitung, Streaming (Kafka Module), Message Queue (Mosquitto), Secure Sockets Encryption, SMTP, XML, Logging und viele mehr. Diese Menge an zusätzlichen Werkzeugen und das schnell robuste Framework spielen wahrscheinlich eine Rolle, weshalb Perfect auf GitHub das beliebteste Framework für Server-Side Swift ist (stand Juli 2017).

Chapter 2

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-highliting oder Autovervollständigung für Swift zur Verfügung steht. Zur Lösung des Problems wurde *CLION* von *JetBrains* (**jetbrain**) 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. Stattdesen 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 JavaScipt 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 (**expressinaction**). Express wird von der Stiftung Node.js projektiert und verwaltet und wird auch ständig weiterentwickelt (**expressjs**). 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 (**loopback**).

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 "Handlbars". 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 Vorraussetzungen 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
- libicu-dev
- libcurllpp
- 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 kompatibl 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 (**clang**). 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**). 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 Objctive-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 Universiät von Illinoi 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**). 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 Vitual 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 gegenseitg 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**).

libicu-dev und libcurllpp 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 diese 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 Maschinen Code übersetzt werden kann (**swiftcompiler**).

Standard Bibliothek Die Standard Bibliothek von Swift hält verschiedene Beschreibung 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 Definition 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 (**swiftbibliothek**).

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 (**swiftcore**).

LLDB Debugger Der LLDB Debugger ist ein Teil des Clang-Projektes. Besonderheit hier ist, das 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 (**swiftdebugger**).

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* enhalten sind. Die LinuxMain.swift Datei importiert die notwendige Bibliothek XCTest, welche bereits im Absatz 2.3.2 beschrieben wurde (**swiftpm**).

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	SwiftyBeaver
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 Handlbars. 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
       , defaultValue: "")
20         self.password = getJSONValue(named: "password", from: values
       , defaultValue: "")
21         self.firstname = getJSONValue(named: "firstname", from:
       values, defaultValue: "")
22         self.lastname = getJSONValue(named: "lastname", from: values
       , defaultValue: "")
23         self.email = getJSONValue(named: "email", from: values,
       defaultValue: "")
24         self.key = getJSONValue(named: "key", from: values,
       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 Serialzisieren 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 initalisieren 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.
    name)")
10         } catch {
11             LogFile.debug("error in getFile() setting WorkingDir: \(
    error)")
12         }
13     } else {
14         LogFile.error("Directory \(workingDir.path) does not exist.
    Main executable not started from root of MVC cannot find
    resources?")
15         exit(2)
16     }
17     issetup = true
18 }
19 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.

```
/**
 * gets content of the specified file and returns it as String
 */
func getFile(file: String) -> String? {
    LogFile.debug("Trying to access \(file)")
    ...
    let thisFile = File(file)
    LogFile.debug("file set to \(thisFile.path)")
    do {
        try thisFile.open(.readWrite)
    } catch {
        LogFile.error("Could not open file, error: \(error)") //
    Error
    }
    var content: String? = nil

    defer {
        thisFile.close()
    }
    do {
        content = try thisFile.readString()
    } catch {
        LogFile.error("not able to read File: \(error)") // Error
    }
    return content
}
```

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:

```
    profileExists: function (username, email) {
    username = val.blacklist(username, new RegExp('\\W') );
        var file = dataDir + username + "/" + username + '.json';
        try{
            var data = fs.readFileSync(file, 'utf-8');
            var fields = JSON.parse(data);
            if(fields.email === email) {
                return true;
            } else {
        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:

```
const destroySession = function (req, res) {
    logger.debug("destroySession");
    req.session.destroy();
    res.redirect(303, "/login")
};
```

Listing 2.9: Löschen der Session in Node.js

```
func logoutHandler(request: HTTPRequest, _ response: HTTPResponse) {
    LogFile.debug("logoutHandler()")
    if let _ = request.session?.token {
        request.session = PerfectSession()
        response.request.session = PerfectSession()
    }
    sendSeeOther(response, value: "/")
}
```

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 Dokumention 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:

```
$(function () {
        var socket = io.connect('http://localhost:3000');
        $('form').submit(function () {
            var username = document.getElementsByTagName("span");
            for (var j = 0; j < username.length; ++j) {
                user = username[j].outerHTML;
            }
            user = user.match(/>([a-z0-9A-Z]*)</);
            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 (**socketio**) 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.sendStringMessage(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.sendStringMessage(string: string, final: true)
    {
33                      LogFile.debug("Sending to \(user)")
34                      self.handleSession(request: request, socket:
    socket)
35                  }
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 Informtion 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 (**winston**).

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.contentEncoding,
    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.gzipped()
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.

(d) 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:

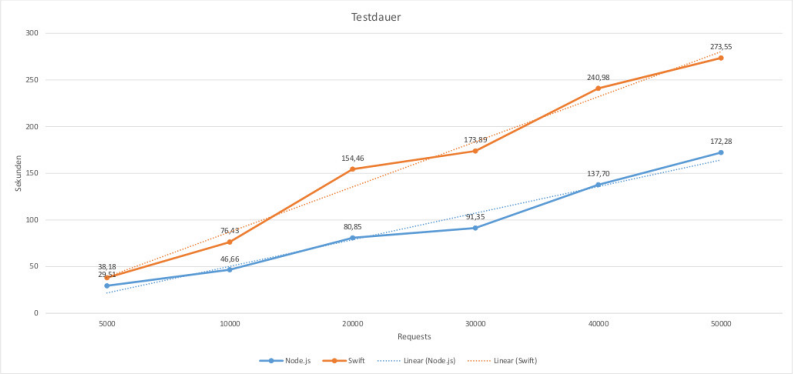

Figure 2.1: Request-Abarbeitung von Express.js und Perfect

	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 abgehandlet werden mussten. Dadurch ergab es sich, dass die Menge der übertragenen Bytes linear anstgestiegen sind, 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 engegen, 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. Dass 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, das 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	central processing unit
REST	Representational State Transfer
WWDC	World Wide Developer Conference
ARC	Automatic Reference Counting