Swift On Linux/Einleitung/

Aus SwiftWiki

Einleitung

Swift

Einleitend sollte erklärt werden was Swift ist. Swift[1] ist eine eigens von Apple entwickelte Programmiersprache speziell für Apple Produkte. Bei der Einführung der Sprache wurde besonderer Wert auf Schnelligkeit und Sicherheit der Sprache gelegt[2]. Dabei soll eine große Palette von Anwendungsgebieten erreicht werden, von systemnaher Programmierung zu mobilen Apps und Cloud-Anwendungen. Besonders die Entwicklungsumgebung Xcode und die Möglichkeit Swift-Code mit bestehenden Objective-C-Code über Cocoa zu verbinden, sollte Entwickler anlocken. Swift selbst ist dabei eine streng statisch typisierte Programmiersprache im Gegensatz zu Objective-C. Typinferenz des Compilers erleichten dabei den Schreibaufwand massiv. Daher ist es keine Überraschung das sich Swift immer wieder unter den beliebtesten Programmiersprachen befindet. Seit Dezember 2015 steht nun Swift unter der Apache-2.0-Lizenz zur Verfügung und ist damit Open-Source und deshalb die Verwendung auf Linux-Maschinen möglich.

Warum Swift?

Sollte die Möglichkeit Swift auf Linux zu verwenden noch nicht genug Ansporn sein, sollte doch noch über mögliche positive und negative Aspekte gesprochen werden. Neben beliebten Frameworks wie ASP, Ruby on Rails, Spring, Express, Grails und vielen mehr, sind Swift Frameworks absolute Newcomer. Die beiden Frameworks Perfect und Vapor klettern aber bereits die Liste der beliebtesten Frameworks nach oben, obwohl Vapor erst seit Juli 2016 und Perfect seit November 2016 existieren. Noch befinden sich viele Features in den Anfangsstadien, aber gerade das sollte junge Entwickler anlocken, besonders iOS Entwickler die bereits erfahrung mit der Spache haben.

Stand der Technik

Die letzte release-fertige Swift Version für Linux ist 3.1.1 (Stand July, 2017) während Swift selbst bereits auf Version 4 ist. Zu den offiziellen Web Frameworks auf Swift Basis finden sich drei große:

- *Perfect* : Welches als Toolkit für Anwendungen und REST-Services in Swift entwickelt wurde[3]
- *Vapor* : Ein flexibles und sehr schnelles Web Framework.[4]
- *Kitura* : Ein neues Web Framework von IBM[5]

Zurück zur Startseite des Buches

1. Apple Inc.: Swift 4, [online] https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ [27.07.2017].
2. Apple Inc (2017a): About Swift, [online] https://swift.org/about/ [27.07.2017].
3. Perfect Soft (2016): Perfect, [online] https://perfect.org [27.07.2017].

4. Vapor (2016): Vapor, [online] https://vapor.codes [27.07.2017].
5. Kitura (2016): Kitura, [online] http://www.kitura.io [27.07.2017].

Swift On Linux/Haupteil/

Aus SwiftWiki

Hauptteil

Der Hauptteil der Arbeit bearbeitet folgende Punkte:

- **Server in der Box** : Das Aufsetzen eines Linux Servers für die Ausführung von Swift Programmen
- **Installation von Swift** : Die Installation von Swift auf dem Linux Server
- **Installation des Beispielservers** : Die Installation des Beispielsservers
- **Vergleich mit anderen Sprachen** : Der Vergleich mit der Sprache JavaScript

Swift On Linux/Haupteil/Server in der Box/

Aus SwiftWiki

Einleitung

Dieser Teil des *Wikibooks* zeigt wie ein Linux-Server (Ubuntu Server 16.04 LTS), im konkreten Fall in einer virtuellen Maschine (VirtualBox), aufgesetzt wird. Dieser Teil richtet sich an unerfahrene Benutzer bzw. um eine einheitliche Ausgangslage mit dem *Wikibook* zu schaffen. Sollte ein Server bereits vorhanden sein, kann dieser Teil auch übersprungen werden.

Vorbereitung

VirtualBox

VirtualBox wird von *Oracle* zur Verfügung gestellt und erlaubt das Virtualisieren eines Betriebssystems. Dieses dient unter anderem als Sandbox und bietet eine notwendige Sicherheit gegen Schaden am eigentlichen Betriebssystem.

VirtualBox kann von *virtualbox.org* für Windows, Mac und Linux herunter geladen werden.

Ubuntu Server

Ubuntu Server ist ein leichtgewichtiges Betriebssystem speziell für den Serverbetrieb und Ubuntu selbst ist eine einsteigerreundliche Linux Distribution. Als Beispiel wurde hier die Version 16.04 LTS gewählt, welche zum derzeitigem Stand (2017) die Aktuellste ist.

Ubuntu Server 16.04 LTS kann von *ubuntu.com* herunter geladen werden.

Durchführung

Nach der Installation von VirtualBox und dem Herunterladen der iso-Datei wird VirtualBox gestartet. Klicke den *NEW* Button. Wähle als *Type* Linux und als *Version* Ubuntu. Ein passender Name für den Server sollte gewählt werden, z.B. "Swift-Server". Folge der Installation, wähle "Memory" im grünen Bereich, für das *Wikibook* waren es 1800MB und ca. 8GB festem Speicher. In der linken Box erscheint nun unser Server, welcher durch Doppelklick oder "START" Button gestartet wird. Es erscheint ein Dialog bei dem das zuvor geladene Image (iso-File) ausgewählt wird. Wähle "Install Ubuntu Server", wähle die Sprache und das Keyboard-Layout. Bei der Wahl des Servernames bietet sich der Name an, der für die VirtualBox gewählt wurde. Nach der Wahl des Benutzernamens und eines Passwortes muss eine Partitionierungsmethode gewählt werden, wie zum Beispiel "Guided - use entire disk and set up LVM". Bei den Werkzeugen sollten Standard System Werkzeuge gewählt werden, wobei auf alle anderen Werkzeuge, wie zum Beispiel die graphische Benutzeroberfläche, verzichtet werden kann. Nachdem die Installation abgeschlossen wurde, sollte das System gestartet und alle nötigen Updates durchgeführt werden. Dafür können die folgenden Befehle verwendet werden:

```
sudo apt-get update
```

```
sudo apt-get -y upgrade
```

Fazit

Es steht nun ein leichtgewichtiger Linux Server zu Verfügung, welcher sich ideal für das Arbeiten mit Swift, bzw. der Entwicklung eines Swift basierten Webservers eignet. Weiter zur Installation von Swift

---

Zurück zur Startseite des Buches

Swift On Linux/Haupteil/Installation von Swift/

Aus SwiftWiki

Inhaltsverzeichnis

Installation von Swift

Da Swift eine Sprache ist, die vorrängig für MacOS erstellt wurde, ist es erst mit ein wenig Aufwand auf Linux Distributionen lauffähig. Dazu muss die Toolchain von Swift.org in ein Verzeichnis geladen werden und je nach Wunsch in der Systemumgebung definiert werden. Eine genaue Installationsanleitung kann ebenfalls unter Swift.org nachgelesen werden. Für die Code Beispiele ist der "user" gegen den tatsächlichen Username auszutauschen. Um dies zu vereinfachen ist der Username unseres Beispielusers "user". Lade die aktuelle stabile Toolchain auf der Download Seite von Swift.org herunter und speichere es in einem beliebigen Verzeichnis z.B. im Downloads-Verzeichnis des aktuellen Users. Danach muss clang installiert werden, um das Compilieren von Swift Programmen zu ermöglichen. Dies ist mit dem Befehl auf einer Linux Bash möglich[1]:

```
sudo apt-get install clang
```

Danach muss die heruntergeladenen Swift Toolchain in ein geeignetes Verzeichnis extrahiert werden und der PATH Variable hinzugefügt werden. Dazu erstellen wir im ersten Schritt das Verzeichnis "install" im user Home-Verzeichnis

```
sudo mkdir /home/user/install
```

Entpacken der Toolchain in das zuvor erstellte "install" Verzeichnis:

```
sudo tar -xf /home/user/Downloads/swift-3.1.1-RELEASE-ubuntu16.10.tar.gz -C /home/user
/install
```

Um das Ausführen von Swift leichter zu gestalten, kann Swift der PATH-Variable temporär hinzugefügt werden:

```
export PATH=${PATH}:/home/user/install/swift-3.1.1-RELEASE-ubuntu16.10/usr/bin
```

Um dauerhaft, z.B. nach einem Neustart, Swift im Terminal zur Verfügung zu haben, kann der Pfad zur Toolchain in die Datei /etc/environment eingetragen werden. Dazu wird die Datei environment mit einem Texteditor, in unserem Fall "VIM" geöffnet und erweitert:

```
sudo vim /etc/environment
```

und folgender Text angefüget:

:/home/user/install/swift-3.1.1-RELEASE-ubuntu16.10/usr/bin

Danach sollte die Installation mit der Ausgabe der Version von Swift überprüft werden:

```
swift --version
```

welches die Ausgabe:

```
Swift version 3.1.1 (swift-3.1.1-RELEASE)
```

zur Folge hat. Eventuell müssen die Besitzrechte des Swift-Installationsordner geändert werden, um Zugriffrechte zu gewähren. Dazu wird folgender Befehl auf den Ordner angewandt:
`sudo chown -R user:user ~/install/swift-3.1.1-RELEASE-ubuntu16.10/`

Damit ist die Installation abgeschlossen.

Swift REPL

Swift bietet bei dieser Installation ein Terminal Tool namens Swift-REPL (Swift-Read-eval-print-loop), welches es ermöglicht, Swift direkt auf der Konsole zu testen und auszuführen. Da dies für das Ausführen des Servers nicht notwendig ist, wird hier darauf nicht näher eingegangen, dazu aber jedenfalls mehr auf Swift.org .[2]

Swift Package Manager

Der Package Manager ist eine Möglichkeit ein Skeleton für ein Projekt zu erstellen. Der Package Mangaer erstellt einen Verzeichnisbaum, der eine gewisse Grundstruktur ermöglicht. Diese Struktur wird auf Swift.org noch in der Version 3.0 beschrieben, jedoch änderte sich dies mit dem RELEASE von Version 3.1. Zuerst wird ein Verzeichnis erstellt, dass das neue Projekt repräsentiert und in weiterer Folge als Projektordner bezeichnet wird.

```
mkdir Hello
cd Hello
swift package init
```

Mit dem Ausführen des Init-Befehls wird unter anderem eine Package.swift Datei erstellt, in dem alle

Abhängigkeiten des Projektes gespeichert werden. Weiter wird ein Verzeichnis Sources erstellt, in dem sich nur .swift-Dateien befinden dürfen. Im Verzeichnis Tests befinden sich die Testklassen[3].

```
├──Package.swift
├──Sources
│   └──Hello.swift
└──Tests
    ├──HelloTests
    │   └──HelloTests.swift
    └──LinuxMain.swift
```

Sollte eine Strukturierung z.B. mit Hilfe von Ordner in Sources erfolgen, ist direkt unter dem Sources Verzeichnis ein weiteres Verzeichnis z.B. "exe" anzulegen, in dem die gewünschte Ordnerstruktur z.B. für ein MVC-Pattern erstellt werden kann. Es ist aufgrund des Compilers und der Modularität nicht möglich, mehrere Ordner und .swift-Dateien gemischt im Sources-Verzeichnis zu compilieren. Dies liegt daran, dass der Compiler Verzeichnisse als Module versteht und jeweils nur ein Modul erstellt werden kann. Daher compiliert er nur den ersten gefundenen Ordner.

Falsche Struktur

```
├──Package.swift
├──Sources
│   ├──model
│   │   └──fileHandler.swift
│   ├──view
│   │   └──renderView.swift
│   ├──controller
│   │   └──router.swift
│   └──main.swift
└──Tests
    ├──FileTests
    │   └──FileHandlerTests.swift
    └──-LinuxMain.swift
```

Richtige Struktur

```
├──Package.swift
├──Sources
│   └──exe
│       ├──model
│       │   └──fileHandler.swift
│       ├──view
│       │   └──renderView.swift
│       ├──controller
│       │   └──router.swift
│       └──main.swift
└──Tests
    ├──FileTests
    │   └──FileHandlerTests.swift
    └──LinuxMain.swift
```

Swift compiling

[4] Der Swift Compiler kann mit dem Befehl

```
swift build
```

gestartet werden, dazu muss man sich im Projektordner befinden. Durch das Aufrufen des Compilers werden die im Package.swift eingetragenen Abhängigkeiten z.B. von Github heruntergeladen und compiliert. Dabei wird das Verzeichnis ".build" erstellt, in dem sich der ausführbare Code befindet. Erst danach wird der eigene produzierte Code compiliert und ebenfalls ins Verzeichnis "/.build/debug" gespeichert. Das Compilieren von Abhängigkeiten erfolgt nur einmal nach dem Download oder dem Download von Aktualisierungen. Die Packages werden im Ordnerbaum abgespeichert, damit sie nicht bei jedem Compilieren heruntergeladen werden müssen. In der alten Version 3.0 wurde dazu ein eigenes Verzeichnis "Packages" angelegt und die aus Github ausgecheckten Module in Form von Source Code gespeichert. Seit dem Update auf die Version 3.1 wird dieses Verzeichnis "Packages" im Überverzeichnis ".build" mit abgespeichert. Auch hier ist der Source Code der Module zu finden. Der Compiler ingnoriert beim Compilieren grundsätzlich alle Verzeichnisse ab dem Sources-Ordner, was zur Folge hat, dass Swift-Dateien bzw. Klassen nicht wie in bekannten anderen Sprachen z.B. mit einem "import"-Befehl zur Verfügung gestellt werden müssen, sondern automatisch als eine große Swift-Datei angesehen werden.

Der Swift Compiler unterstützt die Systemsprachen von C und Objective-C, was durch den Clang Importer erreicht wird. Der Clang-Importer importiert Clang-Module, die die C und Objective-C APIs auf die Swift APIs mappen, wodurch das Compilieren auf Linux bewerkstelligt wird.

Server installieren

Der Server wird auf Github ausgecheckt und compiliert. Davor können zwei Einstellungen in der Config.swift eingestellt werden. Der Host vorab auf "localhost" eingestellt und der Port der vorab auf "3000" eingestellt ist. Das Projekt ist unter Github/Projekt mit Git auszuchecken oder als zip herunterladen. In diesem Repository befindet sich der Ordner NodeServer, in dem der Reverenz-Server in JavaScript mit Node und Express.js geschrieben ist. Im Ordner SwiftServer befindet sich der in Swift geschriebene Server, der für diese Installation relevant ist. Damit der Server läuft, ist der Node-Server irrelevant, sodass dieser gelöscht werden oder ausgeführt werden kann wie im Kapitel "Installation Node Server" beschrieben ist.

Installation mit Git

```
mkdir SwiftServer
cd SwiftServer
git clone https://github.com/hofchris15/Projektarbeit.git
swift build
```

oder

```
swift build --configuration release
```

Mit dem ersten Befehl wird in einem beliebigen Verzeichnis der Ordner SwiftServer erstellt, der als Projektordner dient. Danach wechselt man in das Verzeichnis und checkt den Server von Github aus. Danach liegt der Server in Form von Source Code im Verzeichnis "SwiftServer" auf und kann mit dem letzten Befehl compiliert werden. Mit diesem Kommando wird eine debug Compilierung durchgeführt, was für die Entwicklung benötigt wird, da mehr Informationen während der Laufzeit an den Entwickler ausgegeben werden. Um den Server jemanden außerhalb des Entwicklungsbereichs zu übergeben, wird empfohlen den Compiler auf den release-Modus zu stellen. Dazu wird der Befehl `swift build --configration release` ausgeführt. Je nach Compiler-Konfiguration wird das Produkt im Verzeichnis .build/debug/exe oder .build/release/exe abgelegt. Damit ist die Installation des Servers abgeschlossen und

der Server kann gestartet werden.

Installation ohne Git

Der Ablauf ohne Git ist geringfügig anders als bei der Installation mit Git. Dazu wird der Server als .zip-Archiv von Github heruntergeladen und in das zuvor erstellte Verzeichnis "SwiftServer" entpackt.

```
mkdir SwiftServer
cd SwiftServer
sudo tar -xcf /path/to/SwiftServer.zip ./
swift build
```

oder

```
swift build --configuration release
```

Danach ist die Installation auch ohne Git abgeschlossen und der Server kann gestartet werden.

Server starten

Der Server kann über ein Terminal gestartet werden, nachdem dieser compiliert wurde siehe auch: Installation von Swift#Server installieren
Danach kann der Server mit nachfolgenden Kommandos gestartet werden. Einzige Voraussetzung ist, das man sich bereits mit dem Terminal im Projektordner befindet.

```
./.build/debug/exe <code>

oder mit

<code>./.build/release/exe
```

Danach sollte folgende Ausgabe auf dem Terminal angezeigt werden:

```
[DBG] init HTTPServer()
[DBG] makeRoutes()
[INFO] set config
[INFO] Starting HTTP server on 0.0.0.0:3000
```

Mit der letzten Zeile wird darüber informiert, dass der Server gestartet wurde und auf dem Port 3000 hört. Somit kann dieser mit einem beliebigen Browser angesteuert werden. Dazu wird in die Adressleiste localhost:3000 eingegeben und es sollte die Login-Seite des Services angezeigt werden.

Weiter mit einigen Details zum Server und den Packages, die verwendet werden.

Details zum Server

Der Server entstand in Zusammenhang mit der Lehrveranstaltung "Projektarbeit" der FH Joanneum Kapfenberg Studienrichtung Software Design, mit dem Thema Swift on Linux. Dazu wurde von uns recherchiert und zwei interessante Frameworks für die Entwicklung eines Servers ausgewählt. Diese beiden Frameworks sind Kitura.io und Perfect. Grundsätzlich sind beide Frameworks sehr ähnlich, wobei hierbei einige Punkte zu beachten sind [5] [6]:

- Kitura ist ein Framework, das von IBM erstellt wurde und auf MAC sowie auf Linux lauffähig ist.

Perfect wurde von PerfectlySoft Inc einem kleinem Startup Unternehmen, das sich auf die Entwicklung mit Swift spezialisiert hat, entwickelt.
- IBM's Kitura hat eine grundsätzliche Bibliothek, die verwendet wird, um einen HTTP Server umzusetzen sowie einige Packages, um die Funktionalität zu erweitern. Perfect setzt auf das selbe Prinzip wie Kitura und hält wie auch Kitura ein Template als Startpunkt für den Server zur Verfügung.
- Kitura ist für die Swift Versionen 3.1 entwickelt, Perfect für Swift Versionen der 3.0 Reihe. Leider gibt es für Perfect zu Swift 3.1 kein Update.
- Kitura und Perfect halten beide eine Documentation bereit, die ausführlich erscheint jedoch bei Perfect mehr Informationen bereit hielt.

Schlussendlich haben wir uns für Perfect entschieden. Gründe dafür waren, der eben erwähnte Punkt der Dokumentation, und das Perfect bereits länger auf dem Markt vorhanden ist. Viele Beispiele, Bibliotheken und Packages werden von PerfectlySoft auf deren Github Account gehostet: PerfectlySoft on Github

Installation von Perfect

Auch Perfect setzt einige Pakete auf Linux voraus, darunter openssl, libssl-dev und uuid-dev. Diese können sehr leicht mit

```
sudo apt-get install openssl libssl-dev uuid-dev
```

installiert werden[7].

Perfect Template

Das Perfect Template wurde von uns als Ausgangspunkt der Entwicklung des HTTP Servers im MVC-Pattern gewählt. Als Reverence wurde von uns der selbe Server zuvor mit Node.js und dem Framework Express.js implementiert. Das Perfect Template hält die Standard Baumstruktur des swift init-Prozesses bereit, die bereits in der Installation von Swift#Swift Package Manager vorgestellt wurde. Weiter befinden sich ein Datei names "main.swift" im Sources Verzeichnis, die die Initalisierung eines Servers enthält. Interressant ist jedoch, die in der Package.swift enhaltene Abhängigkeit [1]. Der Perfect-HTTPServer hat einige sehr nützliche Abhängigkeiten, die auch wir im Server verwendet haben.

Die wichtigsten Packages

Wie oben beschrieben sind einige Packete grundsätzlich notwendig um den Server ins Laufen zu bekommen. Diese Grund-Pakete sind PerfectLib COpenSSL, HTTP, HTTPServer, PerfectLib, LinuxBridge und Net[8].

- Foundation ist die Standardbibliothek von Swift und bietet die grunsätzlichen Funktionen und Definition wie Strings, Numbers usw. die von der Swift.org veröffentlicht wurden. https://developer.apple.com/documentation/foundation
- COpenSSL bietet verschiedene Verschlüsselungstechniken und SSL/TLS Methoden zur Verfügung, die auf Linux in C installiert werden (Dieses Paket wurde nicht verwendet). https://github.com /PerfectlySoft/Perfect-COpenSSL
- HTTP bietet verschiedene Strukturen und Methoden, um mit http Clients zu kommunizieren. Dieses Paket muss mit "import PerfectHTTP" im Projekt eingebunden werden. https://github.com /PerfectlySoft/Perfect-HTTP
- HTTPServer bietet die Hauptstrukturen für HTTP 1.1 und HTTP 2 Server und ist die Hauptabhängigkeit für das Projekt und muss mit "import PerfectHTTPServer" eingebunden werden https://github.com/PerfectlySoft/Perfect-HTTPServer

- PerfectLib ist das Herz des Perfect Frameworks und ist ohne nicht lauffähig. Muss ebenfalls mit "import PerfectLib" eingebunden werden. https://github.com/PerfectlySoft/Perfect
- LinuxBridge bildet die Brücke zwischen Perfect und den Linux Distriputionen https://github.com /PerfectlySoft/Perfect-LinuxBridge
- Net ist ein networking Paket welches TCP, SSL, UNIX Socket files und IO Event Handling zur Verfügung stellt. https://github.com/PerfectlySoft/Perfect-Net

Zusätzliche Packages

Einige für die Server wichtige zusätzliche Packages[8]:

- Perfect-Logger und Perfect-RequestLogger sind Pakete, die zum Loggen von Informationen auf die Konsole oder in ein Log File. Der RequestLogger ist eine Kindklasse vom Logger und fängt durch das Einbinden von Filtern alle Requests und logged diese mit, sieh auch: https://github.com /PerfectlySoft/Perfect-RequestLogger
- Perfect-Session wurde verwendet um einen Session zu starten, zu verwalten und wieder zu vernichten. https://github.com/PerfectlySoft/Perfect-Session
- Perfect-Websockets bietet eine Bibliothek zum Aufbauen von WebSocket-Verbindungen
- SwiftyBeaver[9] bietet verschiedene Verschlüsselungstechniken für Passwörter oder anderen empfindliche Daten.https://medium.com/swiftybeaver-blog/logging-in-server-side-swift-85bdecb6be80

Wichtige Punkte der Implementierung

Linux

Das Projekt soll nur auf Linux laufen, deshalb wurde am Anfang folgender Code eingebunden, welcher das Betriebssystem überprüft und beendet, sollte es kein Linux sein:

```
#if !os(Linux)
import Glibc
print("We are sorry this is only meant to be run on Linux")
exit(1)
#endif
```

Session Verwaltung

Bei der Session sind verschiede Konfigurationen möglich. Zu beachten ist jedoch, dass die cookieDomain, die festgelegt werden kann, während der Entwicklung nicht festgelegt werden darf. Erst wenn der Server gehostet wird, kann die Domain angepasst werden, was jedoch nicht nötig ist, damit das IPAddressLock und der userAgentLock eingeschalten sind.

```
SessionConfig.name = "mobileExtendSession" //Session name which is set as cookie
SessionConfig.idle = 86400 // idle time set to one day
// Optional
//SessionConfig.cookieDomain = "localhost"
SessionConfig.IPAddressLock = true //Session is bind to the IP address of the first request
SessionConfig.userAgentLock = true //Session is bind to the user
```

JSON

Objekte, die als JSON serialisiert werden sollen, müssen von der Klasse JSONConvertibleObject erben, sowie die Methoden "setJSONValues" und "getJSONValues" überschreiben. Zusätzlich zu den üblichen Attributen des Objektes kommt ein Registrierungsname zum Einsatz. Dieser dient als Schlüssel für die Objektklasse und als Registrierungsschlüssel im Register der JSON-Decodable-Object Referenz. Mit diesem Schlüssel kann die Decodierung eines JSON Strings erfolgen und wird dem Objekt, das erstellt werden soll, zugeteilt.

```
public class Profile: JSONConvertibleObject {
static let registerName = "profile"
   var username = ""
   var password = ""
....
....
override public func setJSONValues(_ values: [String: Any]) {
   self.username = getJSONValue(named: "username", from: values, defaultValue: "")
   self.password = getJSONValue(named: "password", from: values, defaultValue: "")
   self.firstname = getJSONValue(named: "firstname", from: values, defaultValue: "")
....
....
override public func getJSONValues() -> [String: Any] {
return [
   JSONDecoding.objectIdentifierKey: Profile.registerName,
   "username": username,
   "password": password,
   ...
   ]
....
....
```

Wichtig ist auch, dass die Objekt beim Start des Servers als JSONDecodable registriert werden müssen:

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

Working Directory

Beim Setzen des Working Directory ist zu beachten, dass, solange der Server nicht neu gestartet wird, nicht mehr im Verzeichnisbaum nach oben geändert werden kann, nur nach unten. Deshalb sollte das Working Direktory vom Standardwert Sources nur maximal in ein Unterverzeichnis, wie bei unserem Server, dem Verzeichnis "exe", geändert werden. Beim Setzen des Working Directory wird der Ausführungsprozess in ein darunterliegendes Verzeichnis verschoben.

```
func setupDir(_: Void) -> Void {
   let workingDir = Dir("./Sources/exe")
   if workingDir.exists {
      do {
         try workingDir.setAsWorkingDir()
         LogFile.debug("Working directory set to \(workingDir.name)")
      } catch {
         LogFile.debug("error in getFile() setting WorkingDir: \(error)")
         }
   } else {
      LogFile.error("Directory \(workingDir.path) does not exist. Main executable not started
```

```
from root of MVC cannot find resources?")
        exit(2)
    }
    issetup = true
}
var issetup = false
```


1. Apple Inc (2017f): Getting Started, [online] https://swift.org/getting-started/#installing-swift [27.07.2017].
2. Apple Inc (2017c): Using the REPL, [online] https://swift.org/getting-started/#using-the-repl [26.07.2017].
3. Apple Inc (2017d): Using the Package Manger, [online] https://swift.org/getting-started/#using-the-package-manager [26.07.2017].
4. Apple Inc (2017e): Compiler and Standard Library, [online] https://swift.org/compiler-stdlib /#compiler-architecture [26.07.2017].
5. IBM (o.J.): Setting Up, [online] http://www.kitura.io/en/starter/settingup.html [26.07.2017].
6. Jessup, Kyle / Stephens, Sean / Chang, Lucas / Guthrie, Jonathan (o.J.): What Is Perfect, [online] http://perfect.org/about.html [26.07.2017].
7. Jessup, Kyle / Stephens, Sean / Chang, Lucas / Guthrie, Jonathan (2017a): Getting Started, [online] https://github.com/PerfectlySoft/PerfectDocs/blob/master/guide/gettingStarted.md [27.07.2017].
8. Jessup, Kyle / Stephens, Sean / Chang, Lucas / Guthrie, Jonathan (o.J.): Documentation, [online] https://www.perfect.org/docs/ [27.07.2017].
9. Kreutzberger, Sebastian (2017): AES256CBC.swift, [online] https://github.com/SwiftyBeaver /AES256CBC/blob/master/Sources/AES256CBC.swift [27.07.2017].

Swift On Linux/Haupteil/Programmieren anhand eines Beispiels/

Aus SwiftWiki

Navigation

Inhaltsverzeichnis

Server installieren

Der Server wird auf Github ausgecheckt und compiliert. Dazu folgenden Link Github/Projekt mit Git ausführen oder als zip herunterladen. In diesem Repository befinden sich der Ordner NodeServer in dem der Reverenz-Server in JavaScript mit Node und Express.js geschrieben ist. Im Ordner SwiftServer befindet sich der in Swifft geschriebene Server, der für diese Installation relevant ist. Damit der Server läuft ist der Node-Server irrelevant, sodass dieser gelöscht werden kann oder ausgeführt werden kann wie im Kapitel "Installation Node Server" beschrieben ist.

Installation mit Git

Folgende Befehle sind auszuführen:

```
mkdir SwiftServer
cd SwiftServer
git clone https://github.com/hofchris15/Projektarbeit.git
swift build
```

oder

```
swift build --configuration release
```

Mit dem ersten Befehl wird in einem beliebigen Verzeichnis der Ordner SwiftServer erstellt, der als Projektordner dient. Danach wechselt man in das Verzeichnis und checkt den Server von Github aus. Danach liegt der Server in Form von Source Code im Verzeichnis "SwiftServer" auf und kann mit dem letzten Befehl compiliert werden. Mit diesem Kommando wird ein debug compilierung durchgeführt, das für die Entwicklung benötigt wird, da mehr Informationen während der Laufzeit an den Entwickler ausgegeben werden. Um den Server jemanden außerhalb des Entwicklungsbereichs zu übergeben wird empfohlen den compiler auf den release-Modus zu stellen. Dazu wird der Befehl `swift build --configration release` ausgeführt. Je nach compiler Konfiguration wird das Produkt im Verzeichnis .build/debug/exe abgelegt oder .build/release/exe abgelegt. Damit ist die Installation des Servers abgeschlossen und der Server kann gestartet werden.

Installation ohne Git

Der Ablauf ohne Git ist geringfühgig anders als bei der Installation mit Git. Dazu wird der Server als .zip-Archiv von Github heruntergeladen und in das zuvor erstellt Verzeichnis "SwiftServer" entpackt.

```
mkdir SwiftServer
cd SwiftServer
sudo tar -xcf /path/to/SwiftServer.zip ./
swift build
```

oder

```
swift build --configuration release
```

Danach ist die Installation auch ohne Git abgeschlossen und der Server kann gestartet werden.

Server starten

Der Server kann über ein Terminal gestartet werden nachdem dieser compiliert wurde siehe auch: Installation von Swift#Server installieren Danach kann der Server mit folgende Kommandos gestartet werden. Einzige Vorraussetzung ist, das man sich bereits mit dem Terminal im Projektordner befindet.

```
./.build/debug/exe <code>

oder mit

<code>./.build/release/exe
```

Danach sollte folgend Ausgabe auf dem Terminal angezeigt werden:

```
[DBG] init HTTPServer()
[DBG] makeRoutes()
[INFO] set config
[INFO] Starting HTTP server on 0.0.0.0:3000
```

Mit der letzen Zeile wird darüber informiert, dass der Server gestartet wurde und auf dem Port 3000 hört. Somit kann dieser mit einem bliebigen Browser angesteuert werden. Dazu wird in die Adressleiste localhost:3000 eingegeben und es sollte die Login-Seite des Services angezeigt werden.

Weiter mit einigen Details zum Server und den Packages die Verwendet werden.

Details zum Server

Der Server entstand im zusammenhang mit einer Projektarbeit mit dem Thema Swift on Linux. Dazu wurde von uns rechachiert und zwei interessante Frameworks für die Entwicklung eines Servers ausgewählt. Diese beiden Frameworks sind Kitura.io und Perfect.org . Grundsätzlich sind beide Frameworks sehr ähnlich, wobei hierbei einige Punkte einzubringen und beachten sind. Kitura ist ein Framework das von IBM erstellt wurde und auf MAC sowie auf Linux lauffähig ist. Perfect wurde von PerfectlySoft Inc. einem kleinem Startup Unternehmen, das sich auf die Entwicklung mit Swift spezialisiert hat. IBM's Kitura hat einen Grundsätzliche Bibliothek die verwendet wird um einen HTTP Server umzusetzen und einige Packages um die Funktionalität zu erweitern. Perfect setzt auf das selbe Prinzip wie Kitura und hält wie auch Kitura ein Template als Startpunkt für den Server zur Verfügung. Einziges manko bei Kitura ist auf den ersten Blick, dass die Dokumentation nicht so ausführlich erscheint wie bei Perfect. Schlussendlich haben wir uns für swift Perfect entschieden. Grund dafür waren der eben erwähnte Punkt der Dokumentation, die von Perfect sehr ausführlich ist und auch einige Beispiele für verschiedenen Bereiche eines Servers bereithält. Die Beispiel und viele Bibliotheken und Packages werden von PerfectlySoft auf deren Github Account gehostet: PerfectlySoft on Github

Installation von Perfect

Auch Perfect setzt einige Packet auf Linux voraus, darunter openssl, libssl-dev und uuid-dev. Diese können sehr leicht mit

```
sudo apt-get install openssl libssl-dev uuid-dev
```

installiert werden.

Perfect Template

Das Perfect Template wurde von uns als Ausgangspunkt der Entwicklung des HTTP Servers im MVC-Pattern gewählt. Als Reverence wurde von uns der selbe Server zuvor mit Node.js und dem Framework Express.js implementiert. Das Perfect Template hält die Standard Baumstruktur des swift init-Prozesses bereit, die bereits in der Installation von Swift#Swift Package Manager vorgestellt wurde. Weiter befinden sich ein Datei names "main.swift" enthält, die die Initalisierung eines Serves bereithält. Interressant ist jedoch das in der Package.swift enhaltene Abhängigkeit [1]. Der Perfect-HTTPServer hat einige sehr nützliche Abhängigkeiten die auch wir im Server verwendet haben.

Die wichtigsten Packages

Wie oben beschrieben sind einige Packete grunsätlich notwendig um den Server ins laufen zu bekommen. Diese Grund-Pakete sind PerfectLib COpenSSL, HTTP, HTTPServer, PerfectLib, LinuxBridge und Net.

- Foundation ist die standard Bibliothek von Swift und bietet die grunsätlichen Funktionen und Definition wie (Strings, Numbers,...) die von der Swift.org veröffentlicht wurden. https://developer.apple.com/documentation/foundation
- COpenSSL bietet verschieden Verschlüsselungstechniken und SSL/TLS Methoden zur Verfügung die auf Linux in C installiert werden (Dieses Paket wurde nicht verwendet) https://github.com /PerfectlySoft/Perfect-COpenSSL
- HTTP bietet verschiedene Strukturen und Methoden um mit http Clients zu kommunizieren. Diese Paket muss mit "import PerfectHTTP" im Projekt eingebunden werden https://github.com /PerfectlySoft/Perfect-HTTP
- HTTPServer bietet die Hauptstrukturen für HTTP 1.1 und HTTP 2 server und ist die Hauptabhängigkeit für das Projekt und muss mit "import PerfectHTTPServer" eingebunden werden

https://github.com/PerfectlySoft/Perfect-HTTPServer
- PerfectLib ist das Herz des Perfect Frameworks und ist ohne nicht lauffähig. Muss ebenfalls mit "import PerfectLib" eingebunden werden. https://github.com/PerfectlySoft/Perfect
- LinuxBridge bildet die Brücke zwischen Perfect und den Linux Distriputionen https://github.com /PerfectlySoft/Perfect-LinuxBridge
- Net ist ein networking Paket welches TCP, SSL, UNIX Socket files und IO Event Handling zur Verfügung stellt. https://github.com/PerfectlySoft/Perfect-Net

zusätzliche Packages

- Perfect-Logger und Perfect-RequestLogger sind Pakete die zum Loggen von Informationen auf die Konsole oder in ein Log File. der RequestLogger ist eine Kindklasse vom Logger und fängt durch das einbinden von Filtern alle Request und logged diese mit. https://github.com/PerfectlySoft/Perfect-RequestLogger
- Perfect-Session wurde verwendet um einen Session zu starten, verwalten und wieder zu vernichten. https://github.com/PerfectlySoft/Perfect-Session
- Perfect-Websockets bietet eine Funktionen zum Aufbauen von WebSocket-Verbindungen
- SwiftyBeaver bietet verschiede Verschlüsselungstechniken für Passwörter oder anderen empfindlichen Daten https://medium.com/swiftybeaver-blog/logging-in-server-side-swift-85bdecb6be80

Wichtige Punkte der Implementierung

Linux

Das Projekt soll nur auf Linux laufen deshalb wurde am Anfang folgender Code eingebunden, welcher das Betriebssystem überprüft und beendet sollte es kein Linux sein:

```
#if !os(Linux)
import Glibc
print("We are sorry this is only meant to be run on Linux")
exit(1)
#endif
```

Session Verwaltung

Bei der Session sind verschieden Konfigurationen möglich. Zu beachten ist jedoch, dass die cookieDomain, die festgelegt werden kann, während der Entwicklung nicht festgelegt werden darf. Erst wenn der Server gehostet wird, kann die Domain angepasst werden, was jedoch nicht nötig ist, da mit das IPAddressLock und der userAgentLock eingeschalten sind.

```
SessionConfig.name = "mobileExtendSession" //Session name which is set as cookie
SessionConfig.idle = 86400 // idle time set to one day
// Optional
//SessionConfig.cookieDomain = "localhost"
SessionConfig.IPAddressLock = true //Session is bind to the IP address of the first request
SessionConfig.userAgentLock = true //Session is bind to the user
```

JSON

Objekte die als JSON serialisiert werden sollen, müssen von der Klasse JSONConvertibleObject erben, sowie die Methoden "setJSONValues" und "getJSONValues" müssen überschrieben bzw definiert werden.

Zusätlich zu den üblichen Attributen des Objektes kommt ein Registrierungsname zum Einsatz. Dieser dient als Schlüssel für die Objektklasse und als Registrierungsschlüssel im Register der JSON-Decodable-Object Referenz. Mit diesem Schlüssel kann die Decodierung eines JSON Strings erfolgen und wird dem Objekt, dass erstellt werden soll zugeteilt.

```
public class Profile: JSONConvertibleObject {
static let registerName = "profile"
   var username = ""
   var password = ""
....
....
override public func setJSONValues(_ values: [String: Any]) {
   self.username = getJSONValue(named: "username", from: values, defaultValue: "")
   self.password = getJSONValue(named: "password", from: values, defaultValue: "")
   self.firstname = getJSONValue(named: "firstname", from: values, defaultValue: "")
....
....
override public func getJSONValues() -> [String: Any] {
return [
   JSONDecoding.objectIdentifierKey: Profile.registerName,
   "username": username,
   "password": password,
   ...
   ]
....
....
```

Wichtig ist auch das die Objekt beim Start des Servers als JSONDecodable registriert werden müssen:

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

working Directory

Beim setzen des Working Directory ist zu beachten, dass solange der Server nicht neu gestartet wird, nicht mehr im Verzeichnisbaum nach oben gändert werden kann, nur nach unten. Deshalb sollte das Working Direktory vom Standardwert Sources, nur maximal in ein Unterverzeichnis, wie bei unserem Server, dem Verzeichnis "exe", geändert werden. Beim Setzen des Working Directory wird der Ausführungsprozess in ein darunterliegendes Verzeichnis verschoben.

```
func setupDir(_: Void) -> Void {
   let workingDir = Dir("./Sources/exe")
   if workingDir.exists {
      do {
         try workingDir.setAsWorkingDir()
         LogFile.debug("Working directory set to \(workingDir.name)")
      } catch {
         LogFile.debug("error in getFile() setting WorkingDir: \(error)")
         }
   } else {
      LogFile.error("Directory \(workingDir.path) does not exist. Main executable not started
from root of MVC cannot find resources?")
      exit(2)
```

```
    }
    issetup = true
}
var issetup = false
```

Zurück zur Startseite des Buches

Abgerufen von „http://192.168.42.2/mediawiki/index.php?title=Swift_On_Linux/Haupteil/Programmieren_anhand_eines_Beispiels/&oldid=70“

Swift On Linux/Haupteil/Vergleich mit anderen Sprachen/

Aus SwiftWiki

Inhaltsverzeichnis

JavaScript als Reverenz

JavaScript und Node.js

Ursprünglich wurde JavaScript als Teil vom Webbrowsern implementiert, um clientseitig Skripte auszuführen. Dabei soll dem Anwender eine Schnittstelle für interaktives Web geboten werden, asynchrone Anfragen gestllt und der Dokumenteninhalt verändert werden können. Die Sprache war in der Vergangenheit sehr negativ besetzt und hat auch heute noch bei vielen einen negativen Beigeschmack. Leider wird oft die Entwicklung von JavaScript in den letzten Jahren außer Acht gelassen, wo sich JavaScript immer mehr Richtung "Unverzichtbarkeit" gearbeitet hat. Derzeit ist JavaScript beinahe auf jeder WebSite und Webanwendung vertreten. Früher als Spielzeug bezeichnet, ist es heute ein beliebtes Werkzeug, um dynamische WebSites zu erstellen und es Usern zu ermöglichen, interaktiv im Browser und im Internet unterwegs zu sein. Auch große Firmen wie Mozilla, Google, Microsoft und Apple setzen mittlerweile auf JavaScript[1]. Mittlerweile ist die Sprache prototypenbasiert, dynamisch und typensicher. Grundlage für JavaScript bildet die Skriptsprache C sowie sehr viele Begriffsstandards von Java[2]. JavaScript am Server auszuführen, ist seit Entstehung der Sprache eine laufende Idee von JavaScript Entwicklern und wurde bereits ein Jahr nach Entwicklung 1996 umgesetzt. Der Druchbruch gelang aber erst am 8. November 2009 als Ryan Dahl Node.js vorstellte, dass bis heute immer beliebter wird. Mittlerweile gibt es sehr viele Tutorials, die es ermöglichen, JavaScript und Node.js zu lernen und auch sehr viele Beispiele um diese Sprache und das Framework zu erlernen.

Pros and Cons

Pros of JavaScript	Cons of JavaScript	Pros of Node.js	Cons of Node.js
Benutzer freundlichere	JavaScript kann clientseitig deaktiviert	Open source server Framework[3]	Unübersichtlichkeit der Community und der Entwicklung, durch sehr

Websites	werden		schnelles Wachstum[4]
Dynamischer Inhalt von Websites	Wenn deaktiviert, oft negative Auswirkungen	lauffähig auf Windows, Linux, Unix, Mac OS X, ...[3]	Modulsytem verbraucht viel Speicher[4]
objekt-basiert, prototyping für Vererbung	ursprünglich keine Vererbung	Verwendung von serverseitigen JavaScript[4]	nur ein Prozess mit einem Speicher von 1,7 GB [4]
Objekt-Referenzen werden erst zur Laufzeit geprüft[5]	schwach typisierte Variablen[5]	Asynchron, event basiert[4]	benötigt clustering um große Mengen abzuarbeiten[4]
einfache prozeduale Sprache[5]	kein natives Modulsystem[4]	einbinden von C und C++ libraries[4]	nicht alle libraries von C und C++ verfügbar[4]

ECMAScript

JavaScript wurde in der ECMA (European association for standardizing information and communication systems) spezifiziert. Der Name "ECMAScript" wurde deshalb gewählt, weil Netscape und Microsoft sich nicht auf einen gemeinsamen Namen für die eigenständigen Sprachen "JScript und JavaScript" einigen konnten. Jedoch wurde dieser Name nie weitläufig zur Gewohnheit, weil Brendan Eich, Erfinder von JavaScript meinte, dass dies "wie eine Hautkrankheit" klinge[6]. Entwickler können die offene Sprache zum Entwickeln von Programmen nutzen und dabei sicher gehen, dass der Code unterstützt wird, wenn dieser den Standard einhält.

Swift vs. JavaScript

Ziel der Entwicklung

Ein großr Unterschied bei den beiden Sprachen liegt im Ziel der Entwicklung der Sprachen. JavaScript wurde entwickelt mit dem Ziel, statisches HTML und CSS im Web, dynamischer zu gestalten und dem Client ein interaktives Web zu bieten. JavaScript ist beinahe auf jeder Website zu finden und auch nicht mehr wegzudenken[7]. Hingegen zu JavaScript ist Swift als eine Allzwecksprache erfunden worden, mit der Absicht die beste, einheitliche Sprache für die Systemprogrammierung, zu mobilen Applikationen und Desktop Anwendungen bis hin zum Cloud Service zu bieten. Dabei wurde noch auf drei Punkte sehr großer Wert gelegt[8]:

- Sicherheit: Dabei wird davon ausgegangen, dass nicht definiertes Verhalten der Grund für unsichere Programmierung ist. So muss in Swift jedes mögliche Ende bedacht und definiert werden da sonnst eine Kompiliertungsfehler auftritt.
- Performance: Swift soll alle C-basierten Sprachen ersetzen. Dazu muss Swift vergleichbar in der Abarbeitung verschiedener Aufgaben sein wie diese Sprachen und diese genau so schnell umsetzen können und gleichzeitig die Ressourcen schonen.
- Ausdruck: Die Syntax[9] wurde mit Jahrzehnten von Programmiererfahrung entwickelt und wird auch immer weiter entwickelt werden.

Compiling

JavaScript iste eine dynamische typisierte, prototypenbasierte, interpretierte Programmiersprache, wobei die Betonung auf "interpretierte" liegt. Der Unterschied liegt darin, dass Swift zu den kompilierten Sprachen zählt.

'**Interpretierte Sprachen:**'

In dem Moment wo das Programm gestartet wird, wird der Code in Instruktionen übersetzt und auch ausgeführt. Diese Übersetzung übernimmt ein zusätzliches Tool, der Interpreter. Dieser läuft wie eine virtuelle Maschine auf dem PC und nimmt Eingaben sowie den Quellcode und wandelt diesen in einen hardwareunabhängigen Bytecode. Dies passiert während der Laufzeit, wobei pro Prozessor ein Interpreter benötigt wird.
Vorteil: Leichter bei der Entwicklung, da bereits während der Entwicklung getestet werden kann. JavaScript wird im Plaintext an den Client übertragen und kann von anderen Entwicklern gelesen und gelernt werden.
Nachteil: Langsamer und ineffizienter als kompilierte Sprachen, da kontrollflusssteuernte Funktionen (Schleifen, Funktionen) immer wieder übersetzt werden müssen. [10]

'**Kompilierte Sprachen:**' Der Compiler übersetz den Quellcode in ein maschinenlesbares Programm, sodass es vom Menschen nicht mehr gelesen werden kann, jedoch sofort ausgeführt werden kann. Dieses Programm bzw. die Anweisungen im Programm werden direkt vom Prozessor ausgeführt. Jedesmal wenn sich im Programmcode etwas ändert, muss der gesamte Code neu kompiliert werden.
Vorteil: Der Code wird durch den Compiler optimiert. Kompilierte Programme sind sehr schnell in der Ausführung.
Nachteil: Der Aufwand bei der Entwicklung ist ist zeitraubender, da das Programm vor einem Testlauf jedesmal neu kompiliert werden muss. [10]

Warum JavaScript mit Node.js als Reverenz

JavaScript hat mit der immer größer werdenden Nachfrage und der schnellen Weiterentwicklung eine sehr interessanten Geschichte. Auch wird es zusehends in der Webentwicklung serverseitig immer öfters eingesetz. Damit kann man neben PHP davon ausgehen, dass JavaScript mit Frameworks wie Node.js eine solide Basis für einen Vergleich bietet. Da Swift auch unter anderem die beste Sprache für die Webentwicklung werden möchte, kann somit ein Fazit gezogen werden, in wie weit Swift dieses Ziel bereits erreicht hat und in welchen Bereichen noch Nachholbedarf besteht.

Installation des Node.js Servers

Der Server ist wie bereits bei der Installation des Swift-Servers von Github auszuchecken oder als .zip-Datei herunterzuladen. Nach dem Auschecken oder Entpacken des Projekts, muss mit einem Terminal ins Verzeichnis "NodeServer" gewechselt werden und der Server mit npm initalisiert werden. Dabei werden alle notwendigen Packages heruntergeladen und der Server kann mit node gestartet werden.

```
cd NodeServer
npm init
node index.js
```

Die Server im Vergleich

Beide Server sind im MVC (Model View Controller) Pattern gehalten und haben folgenden Ablauf: Der Request wird empfangen und an einen der jeweiligen Handler weitergegeben. Die Handler geben entweder den Request direkt an eine View-Klasse/Methode oder an eine Model-Klasse/Methode weiter. Die Model Dateien speichern User-Daten wie Usernamen, Passwort, Vorname, Nachname und E-Mail in einem JSON File. Das Passwort wird bevor es gespeichert wird verschlüsselt. Die Profile der User werden in beide Servern als eigenen Objekt angelegt und können als JSON serialisiert werden. Als Sammelverzeichnis wird der Ordner "profiles" angelegt in dem wiederum ein Verzeichnis mit dem Usernamen erstellt. Erst in diesem wird das JSON File gespeichert. Diese Verzeichnisse sind notwenig sollten Bilder oder ähnlichen zu den Usern gespeichert werden. Als zweites Aufgabengebiet sind in den Model Dateien Methoden und

Funktionen enthalten, die diese Daten aus den JSON Files wieder ausliest, und das Passwort z.B. für den Login überprüfen. Spätestens nachdem die Model-Funktionen fertig sind wird der Request an die View übergeben. Hier werde die HTMLs zusammengebaut und an den Client zurückgeschickt. Die Funktionalität ist für Studenten der FH Joanneum gedacht. Die Studenten können ihre tätsächlichen Noten mit dem Usernamen und Passwort der FH-Domain abrufen.

Die Server unterscheiden sich ausschließlich durch Sprach- und Framework spezifischen Vorgaben. Weiter haben wir versucht ähnliche Packages zu verwenden und so wenig Workarounds zu implementieren, die Packages ersetzen.

1. Gelsendörfer, Felix (2010): Wie Node.js JavaScript auf dem Server revolutioniert: Schubrakete für JavaScript, [online] http://t3n.de/magazin/nodejs-javascript-server-revolutioniert-schubrakete-226177/ [27.07.2017].
2. Fuchs Media Solutions (o.J.): JavaScript Begriffserklärung und Definition, [online] https://www.seo-analyse.com/seo-lexikon/j/javascript/ [27.07.2017].
3. Refnes Data (2017b): Node.js Introduction, [online] https://www.w3schools.com/nodejs /nodejs_intro.asp [27.07.2017].
4. Elke, Gregor (2014): Ist Node.js ein Superheld?, [online] https://blog.codecentric.de/2014/06/ist-node-js-ein-superheld/ [27.07.2017].
5. o.V. (o.J.): Gefahren und Anwendungsmöglichkeiten druch JavaScript, [online], http://www.mathematik.uni-ulm.de/sai/ws01/portalsem/wiede/ [27.07.2017].
6. Vorlage:Cite book Günster, Kai (2013): *Schrödinger lernt HTML5, CSS3 & JavaScript: das etwas andere Fachbuch*, Bonn: Galileo Press.
7. Mozilla Developer Network and individual contributors (2017): JavaScript Guide - Introduction, [online] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction [27.07.2017].
8. Apple Inc (2017a): About Swift, [online] https://swift.org/about/ [27.07.2017].
9. Apple Inc (2017b): Swift - Language Guide, [online] https://developer.apple.com/library/content /documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref /doc/uid/TP40014097-CH5-ID309 [27.07.2017].
10. Schnabel, Patrick (2012): Compiler und Interpreter, [online] https://www.elektronik-kompendium.de /sites/com/1705231.htm [27.07.2017].

Swift On Linux/Conclusion/

Aus SwiftWiki

Inhaltsverzeichnis

Conclusion

Mit dem Ziel die beste Sprache der Entwicklung zu werden, hat sich Swift ein sehr hohes Ziel gesteckt, dass beinahe nicht zu erreichen sein wird. Bei der Entwicklung des Swift Servers sind einige Probleme aufgetaucht, die wir hier in unserer subjektiven Meinung beschreiben möchten.

Die Sprache

Unter anderem ist uns aufgefallen, dass die Syntax anfangs ungewohnt ist und etwas Zeit in Anspruch nimmt, sich mit dieser vertraut zu machen. Die Dokumentation bzw. die Guides, die auf Swift.org oder auch auf anderen Seiten zur Verfügung gestellt werden, sind eine Unterstützung, jedoch kommt man leider nicht darum herum, sehr vieles mit "try and error" auszutesten. Sehr hilfreich war der auf Apples Developer Seite zu findende Language Guide, welcher die Basics und auch viele fortgeschrittene Themen behandelt. Für Kontrollfluss-Funktionen gibt es die Möglichkeit z.B. Initialisierung, Condition-Überprüfung und "else"-Block in eine Zeile zu schreiben. Leider wird der Code dadurch schwer lesbar und für Entwickler anderer Sprachen beinahe unmöglich zu lesen, sollte keine Zeit zum Investieren vorhanden sein.

Das Framework

Zum verwendeten Framework ist zu sagen, dass es bereits sehr gute Ansätze für die Abhandlung von Request und Response bereithält, jedoch noch einen sehr weiten Weg zum Referenzprodukt Node.js hat. Das Handling im Allgemeinen ist sehr unausgereift und muss mit vielen kleinen Workarounds dazu gebracht werden, das Verhalten von Node.js zu imitieren. Leider verführt das vorgeschlagene Requesthandling dazu, dass viele Methoden angelegt werden, die alle das gleiche Verhalten aufweisen, jedoch auch nur schwer refactored werden können. Leider gab es für Perfect auch keinen Beispiel-Server mit mehreren Seiten und Funktionen, der veranschaulicht, wie die Entwickler des Frameworks sich die Umsetzung bzw. die Verwendung des Frameworks gedacht haben. Bei den vorhandenen Beispielen war oft die Dokumentation unzureichend, sodass die Anwendung verschiedener Packages in der begrenzten Zeit nicht möglich war. Als Beispiel lässt sich das Perfect-Crypto Package anführen, wo die Beschreibung nicht reichte, um ein Passwort zu verschlüsseln. Dazu wurde von uns das Packet SwiftyBeaver nach längerem Studieren der Klassen und Methoden von Perfect-Crypte als Ersatz verwendet. Einige Male mussten die Packages selbst analysiert werden, um zu verstehen, wie die Pakete angewandt werden, was natürlich wiederum zu einem besseren Verständnis führte, jedoch eine Menge Zeit verschlang. Grundsätzlich ist das Framework weiter zu empfehlen, jedoch muss bis zum sicheren Anwenden des Frameworks sehr viel Zeit und Ärger investiert werden.

Abschließend bleibt noch zu sagen, dass diese Sprache für sich genommen eine sehr schöne und lesbare Sprache ist, wenn nur wenige Kurzschreibweisen verwendet werden. Der Ansatz der Sicherheit und der daraus schließenden "absoluten Definition" ist nachvollziehbar und auch sehr vorteilhaft, da man bei jeder Methode und Funktion das Verhalten genau erkennt. Jedoch ist es bis zum Einholen des Node.js Frameworks noch viel Arbeit, da das Node.js Framework ausgereifter und mehr Funktionen in Form von Modulen bietet. Leider ist Perfect für die Version 3.0 entwickelt worden, der Server bereits auf 3.1 entwickelt und während der Entwicklung Version 4.0 von Swift veröffentlicht worden, wodurch sich Fehler in der Dokumentation eingeschlichen haben.

Einzelnachweise