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].