Apple's Swift trifft Oracle's Datenbank
- Yes we can! -

Burak Bagci, Harm Knolle

Hochschule Bonn-Rhein-Sieg
Sankt Augustin

Schlüsselworte: API, Node.js, OCI, REST, SQL, Swift, Webservice

Abstract: *Apple's Swift* ist eine objektorientierte Multiparadigmen-Programmiersprache für *macOS/iOS* sowie *Linux*. *Apple* spricht von „C ohne Ballast“, denn die Speicherverwaltung erfolgt komplett automatisiert. Eine direkte Kopplungsmöglichkeit von *Swift* mit einer *Oracle*-Datenbank existiert jedoch nicht. Es stellt sich daher die Frage, ob sich *Swift* vor dem Hintergrund datenbankgestützter RESTful Micro-/Webservices behaupten wird? In diesem Zusammenhang muss zunächst der Frage nachgegangen werden, wie sich Datenbankzugriffe aus *Swift* grundsätzlich realisieren lassen und ob sich die Philosophie der Ballastfreiheit von *Swift* trotz Integration von Datenbankzugriffen aufrecht erhalten lassen kann.

Einleitung

Apple's Swift [Swi17a] ist eine objektorientierte Multiparadigmen-Programmiersprache für *macOS/iOS* sowie *Linux* und steht seit Dezember 2015 unter einer Open-Source-Lizenz. Nur kurze Zeit später verkündete *IBM*, *Swift* sowohl im Cloud- als auch im Client-/Server-Umfeld anzubieten. Was macht *Swift* so interessant? Benchmarks zeigen, dass *Swift's* Performance in der Lage ist

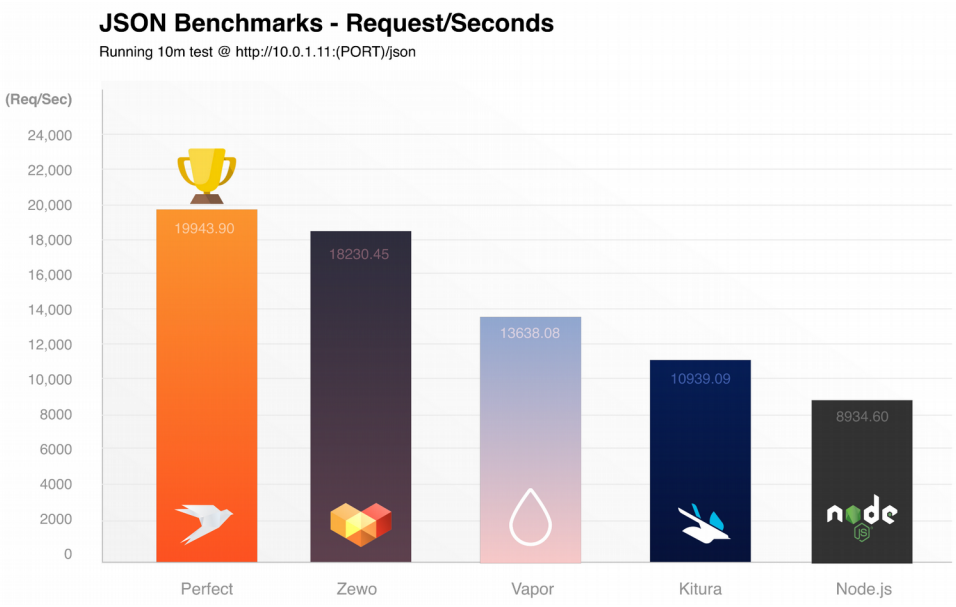

Abbildung 1: Linux Benchmarks for Server Side Swift Frameworks vs. Node.js [Col17a]

Platzhirsche wie *Node.js* um über 100% zu übersteigen [Col17a] (siehe *Linux* Benchmarks for Server Side *Swift* vs. *Node.js* zwischen den Swift-Frameworks *Perfect*, *Vapor*, *Kitura*, *Zewo* und *Node.js* in Abbildung 1). Hat *Swift* das Potenzial zum „next big thing“? Und wird sich *Swift* vor dem Hintergrund von RESTful Micro-/Webservices in Verbindung mit Datenbanken behaupten?

Swift kombiniert bewährte und moderne Konzepte wie Closures, Generics, Tupels und Nested Functions [Kof17, Sil17]. Eine direkte Kopplungsmöglichkeit mit einer *Oracle*-Datenbank existiert jedoch nicht. Das Manuskript führt in die grundlegende Entwicklung mit *Swift* ein und zeigt, wie sich eine solche Kopplung relativ einfach realisieren und in einen RESTful Webservice einbetten lässt.

Swift-Grundlagen

Mit *IBMs Swift Sandbox* existiert eine interaktive Webseite, mit der man *Swift* aus dem Stegreif heraus ausprobieren kann [IBM17]. Für die ersten Gehversuche wird lediglich ein moderner Webbrowser benötigt. Aller Anfang ist ein „Hallo Welt!“-Programm. Praktischerweise steht der Code eines „Hallo Welt!“-Programms bereits im Editor-Bereich, so dass sich der Code gleich online ausführen lässt.

```
print("Hello world!")
```

Variablen

Variablen werden mit `var` und Konstanten mit `let` deklariert. Dank Typinferenz erkennt der Compiler den Typ von Variablen und Konstanten. Ein Typ kann jedoch auch explizit angegeben werden. Die Nutzung von Semikolons ist fakultativ. Sie ist allerdings erforderlich, wenn mehrere Befehle in einer Zeile angegeben werden.

```
var frage = "nach dem Leben, dem Universum und dem ganzen Rest"
let antwort: Int = 42
print(frage); print(antwort)
```

Mit der sogenannten Initialisierungssyntax können Arrays erstellt werden. In den eckigen Klammern „`[ ]`“ wird der Datentyp angegeben. Die Initialisierung folgt in runden Klammern. Mit der Methode `append()` lassen sich neue Elemente ans Ende hinzufügen. Alternativ steht der Additionszuweisungsoperator „+=“ zur Verfügung.

```
var zutaten = [String]()
zutaten.append("Eier")
zutaten += ["Milch"]
print(zutaten)
```

Wenn in *Java* eine Objektreferenz kein Objekt referenziert, bekommt dieser den Wert `null` zugewiesen. In *Swift* ist dies direkt nicht möglich, da jede Variable und Konstante immer mit einem validen Wert belegt sein muss. Jedoch kennt *Swift* das aus *Java 8* bekannte Konzept der Optionals. Optionals sind an einem Fragezeichen hinter der Typdefinition erkennbar. Sie verpacken die Variable oder Konstante und ermöglichen diesen den Zustand „nicht belegt“ aufzuweisen. Dieser Zustand wird durch den Wert `nil` ausgedrückt. `nil` wird beim Zugriff auf Optionals ohne Werte zurückgegeben. Den Inhalt eines Optionals erhält man durch das Entpacken. Das explizite Entpacken erfolgt durch ein Ausrufezeichen hinter dem Variablennamen. Sollte sich herausstellen, dass der Wert nicht existiert, kann entsprechend reagiert werden. In bestimmten Situationen wie z. B. dem Inkrementieren sollte

sichergestellt sein, dass das Optional den Wert `nil` nicht enthält, da dies sonst zu einem Fehler führt. Mit Hilfe der Optionals lässt sich der Programmcode insgesamt sehr flexibel gestalten. Durch die Typensicherheit reduziert *Swift* zudem die Fehleranfälligkeit im Umgang mit Variablen und Konstanten.

```
var universum: String? = nil
universum = "Urknall"
print(universum!)
```

Funktionen

Mit dem Schlüsselwort `func` lassen sich Funktionen definieren, wobei ein Pfeil und ein Datentyp nach dem Funktionskopf den Rückgabetyp spezifizieren. Beim Funktionsaufruf müssen die Funktionsparameter angegeben werden, unter dem die Argumente innerhalb der Funktion auch bekannt werden. Im Rahmen der Funktionsdefinition kann jeder Funktionsparameter zwei Namen erhalten. Der Erste bestimmt den Parameternamen, der bei der Argumentübergabe genannt werden muss. Der Zweite legt fest, unter welchem Namen das übergebene Argument innerhalb der Funktion bekannt sein soll. Soll der Funktionsparameter beim Funktionsaufruf nicht genannt werden, wird der erste Name mit einen Unterstrich „_“ angegeben.

```
func vier(mal: Int) -> Int {
    return 4*mal
}
func quadrat(zahl n: Int) -> Int {
    return n*n
}
func hallo(_ text: String) -> String {
    return "Hallo " + text + "!"
}
print(vier(mal: 2)); print(quadrat(zahl: 42)); print(hallo("Welt"))
```

Klassen und Strukturen

Ähnlich wie in *C++* lauten in *Swift* Klassen `class` und Strukturen `struct`. Der Unterschied zwischen Klassen und Strukturen besteht in *Swift* darin, dass Klassen Referenztypen sind. Bei einer Zuweisung wird eine neue Referenz erstellt, die auf das ursprüngliche Objekt zeigt. Dagegen sind Strukturen Wertetypen. Sie geben bei der Zuweisung den kopierten Inhalt des Objekts weiter. Konstruktoren, die überladen werden können, werden mit dem Schlüsselwort `init` definiert. `self` ist das *Swift*-Äquivalent zu *Java's* `this`.

```
class Universum {
    var inhalt: [Galaxie] = [Galaxie]()
}
struct Galaxie {
    var typ: String = "unbekannt"
    init() {}
    init(_ typ: String) {
        self.typ = typ
    }
}
```

```
let universum = Universum()
var galaxie0 = Galaxie()
var galaxie1 = Galaxie("elliptisch")
var galaxie2 = galaxie1
galaxie2.typ = "spiral"
universum.inhalt += [galaxie0] + [galaxie1] + [galaxie2]
print(universum.inhalt)
```

Kopplung von *Swift* mit *Oracle*-Datenbanken

Im vorangegangen Kapitel wurde die Einfachheit des grundsätzlichen Aufbaus von *Swift* gezeigt. Im Folgenden wird der Frage nachgegangen, wie sich Zugriffe von Swift aus auf eine *Oracle*-Datenbank realisieren lassen. Leider unterstützt *Oracle* keine unmittelbare Schnittstelle zu *Swift*. Auch unterstützt *Swift* keinen offenen Datenbank-Schnittstellen wie z. B. *Open Database Connectivity* (*ODBC)*.

Architektur

Betrachtet man *Swift* jedoch etwas genauer, so stellt man fest, dass es sich um eine interoperable Programmiersprache handelt. Damit ist *Swift* in der Lage, Programmcode zu verwenden, der in einer anderen Programmiersprache geschrieben wurde. Die Interoperabilität wird mit Hilfe der *LLVM*-Compiler-Infrastruktur (*Low Level Virtual Machine*) realisiert [Swi17b]. Die *LLVM* ermöglicht neben reinem *Swift*-Code auch die Verwendung von *C/Objective-C*-Routinen sowie den dort vorhandenen *C*-Sprachtypen -Funktionalitäten, -Konstrukten und -Mustern. Über diesen Weg lässt sich im Prinzip jede in *C* geschriebene Schnittstelle importieren (siehe Abbildung 2). *Oracle* bietet eine für *C* konzipierte API zum Zugriff auf eine *Oracle*-Datenbank an: das *Oracle Call Interface* (*OCI*) [Ora17a]. Somit ist es theoretisch von *Swift* aus möglich, über das *OCI* auf eine *Oracle*-Datenbank zuzugreifen (siehe Abbildung 3). Darüber hinaus existiert die *OCILIB* [Rog17]. Hierbei handelt es sich um eine API, die die leistungsstarke aber durchaus komplexe *OCI*-API kapselt und den Zugriff

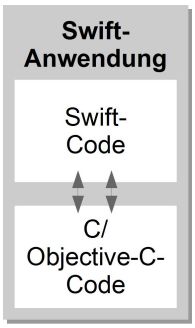

Abbildung 2: Integration von C-Code in das Swift-Umfeld

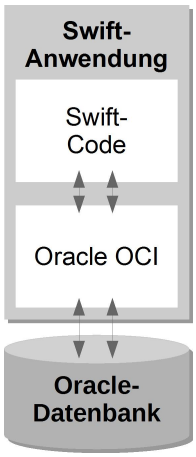

Abbildung 3: Nutzung der OCI in Swift

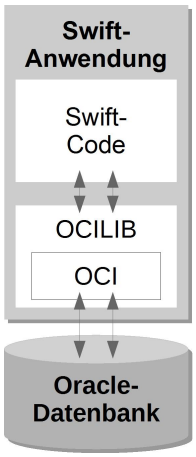

Abbildung 4: Nutzung der OCILIB in Swift

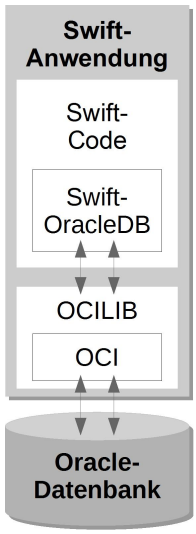

Abbildung 5: Integration des Swift-OracleDB Wrappers

auf die Datenbank insgesamt lesbarer und somit für den Programmierer einfacher gestalten lässt (siehe Abbildung 4).

Wrapper „Swift-OracleDB“

Swift wurde speziell entwickelt, um den Entwicklern das Schreiben und Pflegen von korrekten Programmen zu erleichtern. Die Gefahr ist groß, über die Interaktion mit *C*-APIs auch den „Ballast“ von *C* wieder mit nach *Swift* zu importieren. Es stellt sich daher die Frage, ob sich *Swift*-Entwickler mit den Herausforderungen der *C*-Programmierung auseinandersetzen wollen. Dieses trifft mit großer Wahrscheinlichkeit auch auf die Akzeptanz der *OCILIB* zu, die trotz ihrer Vereinfachung der *OCI*-API den für *Swift*-Programmierer immer noch den ungewohnten „Ballast“ wie den Umgang mit Zeigern mit sich bringt. Auch stellen komplexe anwendungsspezifische Datentypen und -strukturen in *C*, die zur Nutzung in *Swift* mit entsprechenden Strukturen nachgebildet werden müssen, eine Herausforderung dar.

Kapselung der OCILIB

Bei *Swift-OracleDB* handelt es sich um eine erste prototypische Entwicklung der Autoren, die die *OCILIB* transparent in das Swift-Umfeld einbettet. *Swift-OracleDB s*tellt somit eine komfortable und einfach nutzbare *Swift*-API speziell für den Zugriff auf *Oracle*-Datenbanken dar. Hierbei lehnt sich das Design von *Swift-OracleDB* an das der bereits etablierten *Swift*-Wrapper für andere Datenbanken an [Prf17b]. Die folgende Swift-Methode `query` dient der zur Ausführung von SQL-Anweisunen. Sie verdeutlicht exemplarisch die Kapselung der in *C* programmierten *OCILIB*-Funktionen `OCI_StatementCreate` und `OCI_ExecuteStmt` [Bag17]. Seinerseits verbirgt `OCI_StatementCreate` die Arbeit mit dem Statement-Handling der originalen *OCI*-API, während `OCI_ExecuteStmt` auf den Execute-Funktionen der originalen *OCI*-API basiert [Rog17].

```
@discardableResult public func query(sql: String) -> Bool {
    self.stmt = OCI_StatementCreate(self.con)
    var vSql = sql
    if (vSql[vSql.index(before: vSql.endIndex)] == ";") {
        print("warning: please remove ';' from the end of the sql
        statement")
        vSql.remove(at: vSql.index(before: vSql.endIndex))
    }
    return (OCI_ExecuteStmt(self.stmt, vSql) == 1)
}
```

Integration in das Swift-Umfeld

Der Wrapper *Swift-OracleDB* steht in einem *GitHub*-Repository zur Verfügung [Bag17]. Nachdem die Installation von *Swift-OracleDB* entsprechend der Anleitung durchgeführt wurde, steht eine Basis für die *Swift*-Entwicklung mitsamt der nötigen Einrichtung für den Zugriff auf eine *Oracle*-Datenbank zur Verfügung. Über den ebenfalls mitgelieferten *Swift Package Manager* lassen sich *Swift*-Projekte einfach initialisieren. Das folgende Beispiel zeigt die Erstellung eines *Swift*-Projektordners. Mit dem *Swift Package Manager* folgt anschließend die Erstellung und Initialisierung der internen Ordnerstrukturen einschließlich des Beispielprojekts „MyExecutable“, das ein „Hallo Welt!“-Programm enthält. Die Kompilierung erfolgt über einen `build`-Aufruf. Anschließend kann `MyExecutable` im Verzeichnis `.build/debug/` ausgeführt werden.

```
$ mkdir MyExecutable
$ cd MyExecutable
$ swift package init --type executable
$ swift build
$ .build/debug/MyExecutable
Hello, world!
```

Um den *Swift*-Wrapper für die *OCILIB* im Projekt nutzen zu können, muss dieser in der *Swift*-Manifest-Datei `Package.swift` im Wurzelverzeichnis des Projekts angegeben werden. Das folgende Beispiel zeigt, wie die *GitHub*-Quelle des Wrappers über die Angabe der URL im Rahmen der Abhängigkeiten bekannt gegeben werden muss.

```
let package = Package(
    name: "MyExecutable",
    dependencies: [
        .Package(url: "https://github.com/bbagci/Swift-OracleDB",
        majorVersion: 0, minor: 1)
    ]
)
```

Programmierung der Datenbankzugriffe

Die folgenden Beispiele basieren auf dem Schema „Human Resources“ (`HR`) der *Database Sample Schemas* von *Oracle* [Ora17b]. Die Datenbankzugriffe sind stark vereinfacht und verdeutlichen insbesondere die grundsätzliche Anwendbarkeit. Auf eine Auswertung potenzieller Return-Codes im Rahmen der Fehlerbehandlung wird daher zunächst verzichtet. Der Einsprungpunkt einer *Swift*-Applikation ist die `main.swift`-Datei im Verzeichnis `Sources`. Der Wrapper wird mit dem Schlüsselwort `import` importiert, wonach eine Instanz der Klasse `OracleDB` erstellt wird. Der Verbindungsaufbau mit einer *Oracle*-Datenbank erfolgt mit der `connect`-Methode sowie den für die Verbindung erforderlichen Argumenten.

```
import OracleDB
let oracledb = OracleDB()
oracledb.connect(db: "127.0.0.1:1521/xe", user: "hr", pwd: "oracle")
```

Einfache SQL-Anweisungen werden mit der `query`-Methode abgesetzt. Transaktionen lassen sich mit der `commit`-Methode festschreiben.

```
oracledb.query(sql: "INSERT INTO regions VALUES (5, 'Oceania')")
oracledb.commit()
```

Die `storeResults`-Methode liefert das Ergebnis mengenorientierter `SELECT`-Anweisungen in Form eines Optionals. Das ist erforderlich, da die Ergebnismenge auch leer sein kann. Das Ergebnis-Objekt kann nun mit einer `for-in`-Schleife zeilenweise ausgegeben werden. Die `close`-Methode am Ende dient der Freigabe der Verbindung.

```
oracledb.query(sql: "SELECT * FROM regions")
let results = oracledb.storeResults()!
for row in results {
    print(row)
}
oracledb.close()
```

Kompilierung und Ausführung

Da der *Swift*-Code des Wrappers die in *C*-programmierten *OCILIB*-Aufrufe enthält, ist beim Kompilieren die Angabe des Pfads zu den Programmbibliotheken erforderlich, in denen die *C*-Libraries enthalten sind. Die Ausführung des nun erweiterten „Hallo Welt!"-Programms erfolgt wie oben bereits beschrieben. Die Ausgabe der Tabellendaten ist Key-Value-orientiert, wobei der Spaltenname als Key fungiert.

```
$ swift build -Xlinker -L/usr/local/lib/
$ .build/debug/MyExecutable
Hello, world!
["REGION_ID": "1", "REGION_NAME": "Europe"]
["REGION_ID": "2", "REGION_NAME": "Americas"]
["REGION_ID": "3", "REGION_NAME": "Asia"]
["REGION_ID": "4", "REGION_NAME": "Middle East and Africa"]
["REGION_ID": "5", "REGION_NAME": "Oceania"]
```

Swift-OracleDB und RESTful Webservices

Im letzten Kapitel wurde gezeigt, dass ein Zugriff von *Swift* auf *Oracle*-Datenbanken relativ einfach möglich ist. Es stellt sich nun die Frage, wie sich eine solche Kopplung in einen RESTful Webservice einbetten lässt.

Das Swift-Framework Perfect

Im *Swift*-Umfeld hat sich u. a. *Perfect* als Framework zur Entwicklung von Anwendungen und RESTful Services etabliert [Prf17a]. Das unter einer Open-Source-Lizenz stehende Framework *Perfect* beinhaltet einen Webserver sowie entsprechende Werkzeuge die es den Entwicklern erlauben, sowohl die Client- als auch Server-Seite ihrer Projekte ausschließlich mit *Swift* zu programmieren. Mit *Perfect* entwickelte Applikationen sind leichtgewichtig, wartbar und skalierbar. Daher ist *Perfect* eine gute Wahl für die schnelle Entwicklung eines RESTful Webservices mittels *Swift* [Col17b]. Um *Perfect* nutzen zu können, müssen zuerst die erforderlichen Pakete installiert werden, um dann, wie oben beschrieben, ein *Swift*-Projekt mit dem Namen `MyAwesomeProject` zu initialisieren.

```
$ apt-get install openssl libssl-dev uuid-dev
$ mkdir MyAwesomeProject
$ cd MyAwesomeProject
$ swift package init --type executable
```

In der Datei `Package.swift` wird nun neben dem bereits bekannten Wrapper *Swift-OracleDB* auch die Quelle des *Perfect*-HTTP-Servers im Rahmen der Abhängigkeiten angegeben.

```
let package = Package(
    name: "MyAwesomeProject",
    dependencies: [
        .Package(url: "https://github.com/bbagci/Swift-OracleDB",
        majorVersion: 0, minor: 1),
        .Package(url: "https://github.com/PerfectlySoft/Perfect-
        HTTPServer.git", majorVersion: 2)
    ]
)
```

Der Import der benötigten Komponenten des *Perfect*-Frameworks und des Wrappers erfolgt in der Datei `main.swift`.

```
import PerfectLib
import PerfectHTTP
import PerfectHTTPServer
import OracleDB
```

Danach wird der HTTP-Server, die Container-Variable für die Routen und eine Instanz des Wrappers erstellt sowie eine Verbindung mit der *Oracle*-Datenbank etabliert.

```
let server = HTTPServer()
var routes = Routes()
let oracledb = OracleDB()
oracledb.connect(db: "127.0.0.1:1521/xe", user: "hr", pwd: "oracle")
```

Im Folgenden werden der Container-Variablen zwei Anwendungsfälle hinzugefügt, die jeweils über eine Route beschrieben werden. Der erste Anwendungsfall behandelt HTTP-Anfragen auf das Wurzelverzeichnis, die mit `Hallo Welt!` als HTML-Code beantwortet werden. Dieser Anwendungsfall soll lediglich die Funktionalität testen und enthält noch keine Datenbankzugriffe. Das Objekt `request` der Route beinhaltet die Eingabeinformationen während das Objekt `response` die Ausgabe verwaltet.

```
routes.add(method: .get, uri: "/", handler: {
        request, response in
        response.setHeader(.contentType, value: "text/html")
        response.appendBody(string: "<!doctype html><title>Hallo Welt!
        </title><h1>Hallo Welt!</h1>")
        response.completed()
    }
)
```

Der zweite Anwendungsfall selektiert eine Tabellenzeile anhand ihrer Schlüsselnummer. Die entsprechende HTTP-Anfrage der Route wird über einen variablen Verzeichniszugriff formuliert. Die `next`-Methode liefert den nächsten Datensatz der Ergebnismenge `results`. Eine Schleife zur Abarbeitung der Ergebnismenge ist nicht erforderlich, da über den Primärschlüsselzugriff auch nur ein Datensatz erwartet wird. In einem `do-catch`-Konstrukt ohne weiteren Ausdruck steht die Konstante `error` immer zur Verfügung. In diesem Beispiel würde `error` Informationen über Fehler enthalten, sofern die Ausführung der Methode `setBody` fehlschlägt. Die `response`-Methode `setBody` formatiert die Ausgabe als *JSON*-Objekt.

```
routes.add(method: .get, uri: "/region_id/{id}", handler: {
        request, response in
        oracledb.query(sql: "SELECT * FROM regions WHERE region_id = "
        + request.urlVariables["id"]!)
        let results = oracledb.storeResults()!
        response.setHeader(.contentType, value: "application/json")
        do {
            try response.setBody(json: results.next()!)
        } catch {
            print(error)
```

```
            }
            response.completed()
        }
    )
```

Ausführung des Webservices

Zur Anwendung der Webservices muss die Container-Variable dem HTTP-Server hinzugefügt werden. Der Port, unter dem man den Webservice erreichen kann, wird auf 8181 festgelegt. Anschließend folgt der Start des HTTP-Servers.

```
server.addRoutes(routes)
server.serverPort = 8181
do {
    try server.start()
} catch {
    print(error)
}
```

Aufgrund der in *C*-programmierten *OCILIB*-Aufrufe, darf beim Übersetzen des Programms wie oben nicht vergessen werden, den Pfad zu den *C*-Programmbibliotheken anzugeben.

```
$ swift build -Xlinker -L/usr/local/lib/
$ .build/debug/MyAwesomeProject
Hello, world!
[INFO] Starting HTTP server  on 0.0.0.0:8181
```

Die Ausführung der URL

```
http://0.0.0.0:8181/
```

sollte nun eine Webseite mit „Hallo Welt!“ zeigen und somit die generelle Funktionalität des Webservices beweisen. Die URL

```
http://0.0.0.0:8181/region_id/1
```

wird den als JSON formatierten Datensatz der Tabelle `regions` mit der Schlüsselnummer 1 anzeigen.

```
{"REGION_ID":"1","REGION_NAME":"Europe"}
```

Fazit

Apple's Swift trifft *Oracle*'s Datenbank … und sogar relativ einfach. Die ersten Schritte wurden prototypisch im Wrapper *Swift-OracleDB* implementiert und stehen im *GitHub* zum Download bereit [Bag17]. Es gibt aber noch viel zu tun. Im Prototyp wurden zunächst ausgewählte *OCILIB*-Funktionen umgesetzt. Zudem wurde die Fehlerbehandlung zu Gunsten der Funktionalität zunächst zurückgestellt. Schließlich muss bewiesen werden, dass der verwendete „*Swift-OCILIB-OCI-C*-Stack“ auch in Sachen Performance mithalten kann. Dennoch, die Kopplung von Swift mit *Oracle*-Datenbanken wurde vollzogen - *Yes we can!*

Literatur

[Bag17] Burak Bagci: *Swift-OracleDB*. GitHub Repository, https://github.com/bbagci/Swift-OracleDB, 2017

[Col17a] Ryan Collins: *Linux (Ubuntu) Benchmarks for Server Side Swift vs Node.js.* Webseite, https://medium.com/@rymcol/linux-ubuntu-benchmarks-for-server-side-swift-vs-node-js-db52b9f8270b, 2017

[Col17b] Ryan Collins: *Current Features & Benefits of the Top Server-Side Swift Frameworks.* Webseite, https://medium.com/@rymcol/current-features-benefits-of-the-top-server-side-swift-frameworks-b15b4f2d7bc3, 2017

[IBM17] IBM: *IBM Swift Sandbox.* Webseite, https://swift.sandbox.bluemix.net/#/repl, 2017

[Kof17] Michael Kofler: *Swift 3: Das umfassende Handbuch*. Rheinwerk Verlag, 2017

[Ora17a] Oracle: *Oracle Call Interface Programmer's Guide, 12c Release 2 (12.2).* Part Number E49723-25, Webseite, https://docs.oracle.com/database/122/LNOCI/toc.htm, 2017

[Ora17b] Oracle: *Oracle Database Sample Schemas, 12c Release 2 (12.2).* Part Number E49970-09, Webseite, https://docs.oracle.com/database/122/COMSC/toc.htm, 2017

[Prf17a] Swift: *Server-Side Swift - Perfect*. Webseite, http://perfect.org/, 2017

[Prf17b] Swift: *Database Connectors*. Webseite, http://perfect.org/docs/databaseConnectors.html, 2017

[Rog17] Vincent Rogier: *OCILIB - C and C++ Driver for Oracle*. Webseite, http://vrogier.github.io/ocilib/, 2017

[Sil17] Thomas Sillmann: *Swift 3 im Detail: Einführung und Sprachreferenz.* Carl Hanser Verlag, 2017

[Swi17a] Swift: *Welcome to Swift.org.* Webseite, https://swift.org/, 2017

[Swi17b] Swift: *Compiler and Standard Library*. Webseite, https://swift.org/compiler-stdlib/#compiler-architecture, 2017

Kontaktadressen

Burak Bagci, Prof. Dr. Harm Knolle
Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
Grantham-Allee 20
53757 Sankt Augustin

Telefon: +49 (0) 2241 865 201
Fax: +49 (0) 2241 865 8253
E-Mail: burak.bagci@smail.inf.h-brs.de, harm.knolle@h-brs.de
Web: www.inf.h-brs.de