VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SQUIRREL: WEBOVÝ FRAMEWORK V JAZYCE SWIFT

SQUIRREL: WEB FRAMEWORK IN SWIFT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **FILIP KLEMBARA**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. ADAM HEROUT, PhD.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/21019/2017/xklemb00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Klembara Filip**
Obor: Informační technologie
Téma: **Squirrel: Webový framework v jazyce Swift**
Squirrel: Web Framework in Swift
Kategorie: Uživatelská rozhraní

Pokyny:

1. Seznamte se s vývojem v programovacím jazyce Swift.
2. Analyzujte existující frameworky pro vývoj webových aplikací; zaměřte se na ty jednoduché a intuitivní a na frameworky postavené na moderních principech a programovacích jazycích.
3. Definujte principy navrhovaného frameworku založeného na jazyce Swift: definujte cílovou funkčnost, principy vývoje, dekomponujte do modulů, atp.
4. Narhněte a implementujte řešený framework.
5. Navrhněte a implementujte vhodné demonstrační aplikace, které budou moci sloužit i jako tutoriály pro zájemce o vytvářený framework.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Matthew Mathias, John Gallagher: Swift Programming: The Big Nerd Ranch Guide (2nd Edition), Big Nerd Ranch Guides
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN-13: 978-0321965516
- Susan M. Weinschenk: 100 věcí, které by měl každý designér vědět o lidech, Computer Press, Brno 2012
- Jan Řezáč: Web ostrý jako břitva, Baroque Partners, 2014

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, prof. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cieľom tejto práce je vytvoriť framework v jazyku Swift, ktorý umožňuje tvorbu webových aplikácií a je jednoducho použiteľný, pracuje s databázou MongoDB, ponúka vlastný šablónovací systém, minimalizuje potrebu programovania typu kľúč-hodnota a využíva výhody jazyku Swift verzie 4.

Zvolený problém som vyriešil pomocou implementácie vlastného serveru, nad ktorým pracuje mnou vytvorený framework. V riešení bolo použité generické programovanie a protokol Codable, ktorý umožňuje jednoduchú serializáciu dát.

Vytvorené riešenie poskytuje jednoduché prostredie pre tvorbu webových aplikácií fungujúce na všetkých Swiftom podporovaných Linuxových distribúciách.

Prínosom tejto práce je zjednodušenie vývoja webových aplikácií Swift vývojárom v jazyku Swift.

Abstract

The goal of this thesis is to create a Swift framework that allows to create web applications and is easy to use, works with the MongoDB database, offers custom templates, minimizes the need for key-value programming, and uses Swift 4.

I have solved the problem by implementing server application, which is working with my framework. I used generic programming and Codable protocol, which allows simple data serialization.

The created solution provides an environment for creating web applications for all Swift-supported Linux distributions.

The benefit of this work is to simplify the development of web applications for Swift developers.

Kľúčové slová

Swift, webový framework, MongoDB, NutView, Squirrel, HTTP, serverový Swift

Keywords

Swift, web framework, MongoDB, NutView, Squirrel, HTTP, serverside Swift

Citácia

KLEMBARA, Filip. *Squirrel: Webový framework v jazyce Swift.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, PhD.

Squirrel: Webový framework v jazyce Swift

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána profesora Adama Herouta. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Filip Klembara
15. mája 2018

Poďakovanie

Ďakujem prof. Ing. Adamovi Heroutovi, Ph.D. za odborné vedenie bakalárskej práce a cenné rady pri jej vypracovaní. Ďalej by som chcel poďakovať Adamovi Valekovi za tvorbu grafického dizajnu dokumentačnej stránky a loga.

Obsah

Kapitola 1

Úvod

Táto bakalárska práca sa zaoberá implementáciou frameworku pre tvorbu webových aplikácii v jazyku Swift a knižníc potrebných pre jeho implementáciu. Framework je určený pre vývoj webových aplikácií pre operačné systémy MacOS a Ubuntu a niektoré vyvinuté knižnice je možné použiť aj pod operačnými systémami iOS, WatchOS a tvOS.

Existujúce riešenia sú zbytočne komplikované, a keďže sa jazyk Swift stále vyvíja, nevyužívajú najnovšie možnosti tohto jazyka. Z tohto dôvodu som sa rozhodol vytvoriť vlastné riešenie, vhodné aj pre vývojárov menej skúsených vo webových technológiách s využitím nových možností jazyka Swift.

Práca obsahuje päť kapitol. V druhej kapitole sa porovnávajú existujúce riešenia a popisujú sa vlastnosti jazyka Swift potrebné pre pochopenie kľúčových vlastností tohto jazyka a pre ľahšie pochopenie práce. Tretia kapitola sa zameriava na implementáciu abstrakcie nad protokolom HTTP, zaobalenie požiadaviek od klienta a odpovedí do objektov. Popisuje spôsob držania kontextu medzi požiadavkami a možnosti predspracovania požiadaviek a postspracovania odpovedí. Ďalej popisuje, ako je možné pridávať handlere pre oblúženie požiadaviek od klienta, spracovanie chýb pri tvorbe odpovede pre klienta, návrh vlastného šablonovacieho jazyka a implementáciu interpretéra pre tento jazyk. Porovnáva relačné a nerelačné databázové systémy a hodnotí ich vhodnosť pre implementáciu. V poslednej časti kapitoly sú spomenuté ďalšie implementované knižnice podporujúce funkčnosť frameworku. Štvrá kapitola popisuje tvorbu užívateľskej dokumentácie a návrh dizajnu webovej stránky, publikáciu na sociálnej sieti, testovanie a výsledky frameworku. V poslednej kapitole hodnotím možnosti pokračovania v práci a prípadné zlepšenia.

Kapitola 2

Prieskum existujúcich riešení a potrebných zdrojov

Pri tvorbe webovej aplikácie je možné použiť niekoľko rôznych postupov. Jedným z najľahších sú redakčné systémy, ktoré ponúkajú možnosť si doslova naklikať celú webovú stránku a jej správanie. Tento princíp je veľmi jednoduchý na používanie, ale neponúka veľkú možnosť zmeny kľúčového správania systému. Druhým spôsobom je napísanie si celej aplikácie doslova od „Hello, World!“ po webovú aplikáciu, ktorá spĺňa všetky požadované vlastnosti. I keď výhoda úplne vlastnej implementácie spočíva v celkom veľkej možnosti zmeny správania, vývoj takej aplikácie môže trvať príliš dlho a je potrebné vymýšľať kód, ktorý už bol napísaný niekym iným. Kompromisom je použitie knižníc zaobaľujúcich rutinnú prácu, ktoré stále dávajú možnosť širokej škály konfigurovateľnosti. Takéto riešenia nedosahujú rýchlosť ako vlastná implementácia kvôli niekoľkým vrstvám abstrakcie, ale znižujú čas potrebný pre vývoj aplikácie a programátor nemusí riešiť, ako zabezpečiť komunikáciu medzi serverom a klientom a pod., ale môže sa sústrediť na to, čo je pre neho dôležité – programovanie správania servera.

2.1 Porovnanie existujúcich frameworkov

Vo svete existuje niekoľko desiatok kvalitných a používaných frameworkov pre vývoj webových aplikácií. Medzi najpoužívanejšie patria NodeJS (Javascript), Laravel (PHP), Django (Python) a Spring MVC (Java). Skúmaním týchto frameworkov som stanovil kľúčové vlastnosti, ktoré by mali byť splnené mojím riešením: jednoduché pridávanie ciest a odpovedí pre klienta, predspracovanie vstupných parametrov klienta, objektová práca s databázou, spracovávanie požiadaviek, priama podpora pre držanie kontextu a konfigurovateľnosť pomocou konfiguračného súboru. Hlavnou inšpiráciou bol Laravel, Vapor (Swift) a Tris (Swift).

2.1.1 Laravel

Laravel[1] patrí medzi popredné PHP webové frameworky. Pre pridávanie odpovedí pre klienta používa statické metódy triedy `Route`. Podporuje priamu prácu s databázou cez objekty dediace od typu Model. Ponúka možnosť volať anonymné funkcie s požadovanými parametrami získanými z klientovej požiadavky a priamu kontrolu nad držaním kontextu pomocou pár riadkov.

[1] https://laravel.com

2.1.2 Tris

Tris[2] je nový framework pre jazyk Swift vyvíjany približne rovnako dlho ako moje riešenie. Zameriava sa na poskytnutie najvyššej možnej rýchlosti programátorovi. Podporuje iba jednu NoSQL databázu a dáva možnosť predspracovania parametrov troch kategorií. Parameter obsahujúci informácie o požiadavke, parameter vytvorený z dynamických častí cesty a parameter vytvorený z tela požiadavky. V dobe písania tejto práce neponúkal možnosti práce so session a pridávanie middlewarov nebolo intuitívne.

2.1.3 Vapor

V dnešnej dobe je asi najúspešnejší Swiftový webový framework Vapor[3], ktorý je vyvíjaný od roku 2016. Poskytuje prácu s databázou na základe volaní metód a hodnôt vo formáte kľúč-hodnota. Ponúka veľkú možnosť konfigurácie cez konfiguračné súbory. Jedná sa o štyri súbory, v ktorých je často potrebné konfigurovať vlastnosti, ktoré by mali vyplývať z kódu. Práca s parametrami je robená cez asociatívne polia. Validitu zložitejších štruktúr musí programátor kontrolovať sám.

2.1.4 Zhodnotenie vlasností

Najlepšiu prácu s databázou a session ponúka Laravel, ktorý oproti Tris a Vapor vyžaduje veľa zbytočného kódu. Tris jednoznačne ponúka najlepšie predspracovanie parametrov, ale je do veľkej miery nedoimplementovaný. Vapor ponúka veľké možnosti, ale je zbytočne komplikovaný pre malé až stredné projekty.

2.2 Jazyk Swift

Jazyk Swift vznikol ako náhrada za jazyk Objective-C v roku 2014. Prvý krát bol zmienený počas Keynote 2014 [5], kde bol predstavený ako rýchly, moderný, bezpečný a interaktívny s podporou pre closures (kap. 2.2.6), generické programovanie, typovým odvodzovaním pri kompilácii a ďalšími kľúčovými vlastnosťami, ktoré sa očakávajú od moderného jazyka [5]. Jazyk bol vývíjaný s myšlienkou: „Kompilátor je optimalizovaný pre výkon a jazyk pre vývoj, bez kompromisov“ [2]. Krátko po uvedení jazyka sa stal jedným z najrýchlejšie rastúcich programovacích jazykov v histórii. Tomuto značne prispelo aj vytvorenie verzie s otvoreným zdrojovým kódom, ktorá je spustiteľná pod niektorými distribúciami Linuxu. Vďaka svojej jednoduchej syntaxi a vyjadrovacej schopnosti podobnej ako pri skriptovacích jazykoch je vhodný aj pre začínajúcich programátorov. Swift je multiparadigmový, kompilovaný jazyk vhodný od „Hello, World!“ aplikácií až po veľké systémy. Tak ako Objective-C tak aj Swift je kompilovaný pomocou kompilátoru LLVM[4], vďaka čomu môže byť kompilovaný spolu s kódom písaným v jazyku Objective-C, C a C++[5].

Od vydania v roku 2014 prešiel jazyk veľkými zmenami, ako sú zmeny syntaxe alebo pridanie výnimiek. Aktuálna verzia počas písania tejto práce je 4.1, ktorá narozdiel od verzie 3 disponuje vo väčšine prípadov automatickým generovaním metód pre serializáciu a deserializáciu hodnôt a od verzie 4.0 sa líši automatickým generovaním kódu pre po-

[2] https://github.com/tris-foundation/universe
[3] https://vapor.codes
[4] https://llvm.org
[5] Pre kompiláciu s Objective-C a C++ je potrebný Objective-C runtime.

rovnávanie objektov a možnosťou kontrolovať, či je počas prekladu prístupná požadovaná knižnica.

Swift definuje niekoľko spôsobov ako zaistiť, aby sa programátor vyhol chybám v kóde a tým programoval bezpečný kód. Disponuje správou pamäte pomocou automatického počítania referencií. Pri preklade sa kontroluje inicializácia všetkých premenných pred tým, ako sa prvý krát použijú, počas behu programu prebieha kontrola indexov polí a pretečenie celých čísel. Jazyk vynucuje kontrolu, či hodnota premennej je skutočná hodnota a nie hodnota `nil` (`NULL`).

2.2.1 Výčtový typ (enum) s asociovanými hodnotami

Swift dáva programátorom možnosť asociovať ku každej možnosti z výčtu ľubovolný počet hodnôt rôznych typov. Ak chce programátor v jazyku C mať enum s hodnotou reprezentujúcou prácu s QR kódom obsahujúcim reťazec a hodnotu pre prácu s UPC kódom[6] tvoreným zo štyroch celých čisel, potrebuje si okrem informácie, o ktorý prípad sa jedná, držať osobitne aj požadovanú hodnotu kódu napríklad v type `union`. Vo swifte je možné nadefinovať pre prípady asociované hodnoty (pre QR kód reťazec a pre UPC štyri celé čisla), ktoré budú vystupovať vždy spolu s dotyčným prípadom výčtu [3].

```
1 enum Barcode {
2     case upc(Int, Int, Int, Int)
3     case qrCode(String)
4     case none
5 }
6 var productBarcode = Barcode.qrCode("ABCDEFGHIJKLMNOP")
7 productBarcode = .none
```

Výpis 2.1: Príklad použitia výčtového typu s acociovanými hodnotami.

2.2.2 Reprezentácia práznej hodnoty NULL – Optionals

Swift je jazyk vytvorený zo strachu pred hodnotou NULL.

Dr. Martin Hrubý

Vývojári jazyka Swift sa rozhodli, že v tomto jazyku nebude dovolené používať hodnotu `NULL`, ale bude existovať enum s dvoma prípadmi. Prvý prípad (`.none`) bude reprezentovať hodnotu `NULL` a druhý (`.some`) s generickou asociovanou hodnotou bude reprezentovať skutočnú hodnotu [4]. Tento koncept nazvali typ `Optional` a vytvorili priamu podporu v jazyku pre testovanie a prácu s hodnotami tohto výčtového typu. Väčšinou to znamená pridať jeden znak alebo slovo do kódu, pomocou ktorého je možné skontrolovať, akej je enum hodnoty a ďalej sa pracuje buď s asociovanou hodnotou, alebo sa ošetrí prípad, kedy enum reprezentuje `NULL`. Vďaka tomuto konceptu nie je pre programátora možné použiť optional s asociovanou hodnotou reťazec v kóde, kde sa očakáva konkrétne reťazec. Programátor je nútený skontrolovať hodnotu alebo prijať risk pomocou znaku výkričník (`!`), že konkrétny enum bude mať vždy prípad `.some`. V prípade, že sa mýli, je ochotný akceptovať pád programu [4]. Tento koncept pomáha programátorom nezabúdať kontrolovať, či pracujú so skutočnou hodnotou.

[6] Dvanásť miestny číselný čiarový kód.

```swift
// type Int? is shortcut for type Optional<Int>
let optionalInt: Int? = 4

// checking value for nil with switch
switch optionalInt {
case .some(let number):
    print("value is \(number)")
case .none:
    print("value is nil")
}

// checking value for nil with if let
if let number = optionalInt {
    print("value is \(number)")
} else {
    print("value is nil")
}
```

Výpis 2.2: Možnosti kontrolovania konštanty `optionalInt` na hodnotu

2.2.3 Protokolovo orientované programovanie

Podobne ako jazyk Java aj jazyk Swift ponúka možnosť definovať rozhranie (*interface*), ktoré vynucuje implementáciu požadovaných metód. Swift ponúka implementáciu rozhrania pomocou kľúčového slova `protocol` a na rozdiel od jazyka Java je možné okrem metód nadefinovať aj požadované atribúty alebo asociované typy.

Protokolovo orientované programovanie je spôsob programovania, v ktorom programátor dekoruje triedy, štruktúry a výčtové typy pomocou protokolov. Napríklad objekty `Včela` a `Vták` majú spoločnú vlastnosť, že „vedia lietať“. Tento fakt je možné modelovať tým, že budú dediť z triedy `Letec`. Problém nastáva, ak objekt `Včela` má dediť z inej triedy (napríklad `Hmyz`) ako objekt `Vták`. Riešením je definovať protokol `Letec` a ten implementovať v objektoch, ktoré majú reprezentovať vlastnosť, „vie lietať“ [8]. Protokoly modelujú abstrakciu pomocou definovania, čo by implementujúce typy mali implementovať [1]. Swift ponúka rozšírenie definovaných typov o metódy, protokoly a dynamicky počítané atribúty pomocou kľúčového slova `extension`. Pri rozšírení protokolu o implementovanú metódu získa každý implementujúci typ možnosť použiť túto metódu alebo prepísať jej správanie podobne ako pri dedičnosti, ale bez problémov ako je diamantová dedičnosť.

```swift
class Insect { }

// Bee is a subclass of Insect
class Bee: Insect { }

struct Bird { }

protocol Flier {
    func fly()
}

```

```swift
// Bee and Bird implement Flier protocol
extension Bee: Flier {
    func fly() {
        print("Bee is flying")
    }
}

extension Bird: Flier {
    func fly() {
        print("Bird is flying")
    }
}
```

Výpis 2.3: Riešenie problému s viacnásobnou dedičnosťou pomocou protokolu.

2.2.4 Serializácia typov pomocou protokolu `Codable`

Bežnou potrebou je serializovať hodnoty do určitého formátu a potom ich deserializovať. Táto činnosť sa stala tak bežnou, že v jazyku Swift od verzie 4.0 je protokol `Codable` a automatické generovanie serializácie a deserializácie objektu. Protokol `Codable` je alias pre kombináciu dvoch protokolov `Decodable` a `Encodable`. Protokol vyžaduje implementáciu dvoch metód, ktoré môžu byť často automaticky dogenerované pri preklade.

Pravidlá serializácie a deserializácie sú popísané už v konkrétnom kodéri, ktorý je predaný do implementovaných metód ako argument. Swift ponúka niekoľko hotových kodérov a dekodérov (napr. `JSONDecoder`, `JSONEncoder`), pričom užívateľ si môže vytvoriť vlastný kodér a serializovať objekty pomocou neho.

2.2.5 Pomenované parametre v jazyku Swift

V jazyku Swift sú všetky parametre implicitne pomenované. To znamená, že pri volaní funkcie je nutné zadať okrem argumentov aj názvy parametrov (označenie parametra, anglicky *label*). Mená parametrov môžu byť lokálne (pod týmto menom bude parameter vystupovať v tele funkcie) alebo externé (externé meno je potrebné písať pri volaní funkcie pred argumentom). Externé a lokálne mená sa definujú v hlavičke funkcie v tomto poradí, pričom sú oddelené medzerou. Pokiaľ je zadané iba jedno meno, použije sa ako lokálne a aj ako externé meno. Ak chce programátor definovať parameter, ktorý nebude vyžadovať použitie externého mena, musí ako externé meno určiť _ (znak podčiarkovník[7]). Pri vhodnom pomenovaní externých označení parametrov sa zvyšuje čitatelnosť kódu a programátor vie od externého mena parametru odhadnúť, s akými argumentami treba volať funkciu [8]. Z tejto konvencie vychádza aj dokumentačný popis funkcie. Namiesto len samotného názvu funkcie sa používa názov funkcie aj s externými názvami parametrov oddelenými dvojbodkou.

```swift
// function signature greet(person:)
// 'person' is local and external name
func greet(person: String) -> String {
    return "Hello, \(person)!"
}
```

[7] V jazyku Swift sa používa podčiarkovník ako signál, že programátor nemá záujem o daný parameter alebo hodnotu.

```
6
7  greet(person: "Leo") // Hello, Leo!
8
9  // function signature divide(_:by:)
10 // ‘a‘ is local name with silent external name
11 // ‘by‘ is external name and ‘b‘ is internal name
12 func divide(_ a: Int, by b: Int) -> Int {
13     return a / b
14 }
15
16 divide(6, by: 3) // 2
```

Výpis 2.4: Použitie pomenovaných parametrov vo funkcii.

2.2.6 Funkcie, metódy a closures

Closures sa dajú prirovnať k blokom v jazyku C. Sú považované za bežné objekty (anglicky *first-class objects*), ktoré môžu byť vnorené do seba alebo odkazované premennými ako hodnota. Closures môžu používať premenné deklarované v nadradených blokoch. Tento koncept je známy ako uzatváranie nad (closing over) premennými a konštantami, z čoho pochádza ich názov – closures. Tieto bloky môžu prijímať parametre a mať návratový typ. Parametrom je n-tica obsahujúca ľubovolné typy. Funkcie sú vo Swifte špeciálnym prípadom closures.

Používanie closures je tak bežný koncept, že Swift ponúka pre prácu s nimi „syntaktický cukor“. Ak funkcia berie ako posledný parameter closure, môže tento parameter „vynechať“ a definuje ho za volaním funkcie. Pri práci s closures môže využiť automatické odvodzovanie návratového typu, vďaka čomu nemusí explicitne písať kľúčové slovo `return` alebo použiť skrátené pomenovanie premenných pre jednoduchú čitateľnosť.

```
1  // last argument is a closure
2  [3, 1, 2].sorted(by: { (left: Int, right: Int) -> Bool in
3      return left > right
4  })
5
6  // return expression and type can be inherited
7  [3, 1, 2].sorted { left, right in left > right }
8
9  // automatically provided shorthand argument names
10 [3, 1, 2].sorted { $0 > $1 }
11
12 // function is a special case of closure, > is a function
13 [3, 1, 2].sorted(by: >)
```

Výpis 2.5: Možnosti predania closure ako posledného argumentu funkcii.

V jazyku Swift sa za funkciu považuje pomenovaná closure definovaná na globálnej úrovni. Pokiaľ sa definuje pomenovaná closure v neglobálnej úrovni, jedná sa o metódu. Podobne ako v jazyku C++ tak aj vo Swifte je možné pridať metódy nielen triedam, ale aj štruktúram. Narozdiel od C++, Swift podporuje metódy aj pre výčtový typ (enum).

2.2.7 Open source verzia swiftu

Od verzie 2.2 je swift otvorený komunite. Vďaka tomuto ťahu sa podarilo dostať Swift aj na určité Linuxové distribúcie. Swift, ktorý je v oficiálnom repozitári na stránke github[8] nie je rovnaká verzia Swiftu, aká sa inštaluje s Xcode[9] na operačný systém Mac OS. Jedná sa iba o odnož, do ktorej môže prispievať komunita a ktorá funguje pod Linuxovými distribúciami.

Vďaka otvoreniu kódu Swiftu pre iné platformy bolo len otázkou času, kedy sa Swift dostane na Linux. Jedinou oficiálne podporovanou distribúciou je Ubuntu (verzie: 14.04, 16.04, 16.10). Vzhľadom k tomu, že Swift s otvoreným kódom je navrhnutý aby fungoval bez Objective-C, je možné nájsť niektoré funkcie, ktoré majú v tele príkaz na vyhodenie kritickej chyby s dôvodom, že funkcia nie je implementovaná, ale je možné, že bude implementovaná v budúcnosti. Prevažne sa jedná o špecifické funkcie určené pre súborový systém Mac OS.

2.2.8 Určovanie závislostí pomocou Swift Package Manager

Tak ako je zbytočné vymýšľať koleso, tak je zbytočné písať kód, ktorý už napísal niekto predtým a zverejnil ho pre použitie druhými. Do verzie Swiftu 3.0 nebola žiadna Swiftom podporovaná možnosť, ako takýto kód integrovať. Existujúce riešenia boli buď stiahnuť celý zdrojový kód a ručne ho pridať vedľa zdrojových kódov alebo použiť programy tretích strán. Pri použití prvej možnosti sa autor dobrovoľne vzdáva možnosti automaticky zistiť, či vznikla aktualizácia nejakej z použitých závislostí. Swift Package Manager ponúka jednoduchý spôsob, ako sa odkázať na git repozitár a určiť od akej verzie po akú chce požadovanú závislosť použiť. Pri použití príkazu `swift package update` sa stiahnu najnovšie verzie všetkých závislostí vyhovujúce kritériam a integrujú sa do kódu vo forme knižníc. Ako závislosť sa môže použiť ľubovoľný verzovaný kód, ktorý má git tagy vo formáte major-version.minor-version.revision (major-version = číslo verzie, minor-version = číslo podverzie, revision = číslo opravy).

2.2.9 Serverový Swift

Krátko po otvorení Swiftu komunite začali vznikať knižnice a frameworky umožnujúce jednoduchú tvorbu serverových aplikácií. Medzi popredné frameworky sa dostal Perfect, Vapor a Kitura od IBM. Vďaka tomu, že je Swift kompilovaný, je rýchlejší ako skriptovacie jazyky a vďaka modernejšej syntaxi je jednoduchší na použitie ako C++. Toto viedlo k využívaniu Swiftu prevažne pre náročné operácie, ako je spracovávanie objemných dát alebo videí.

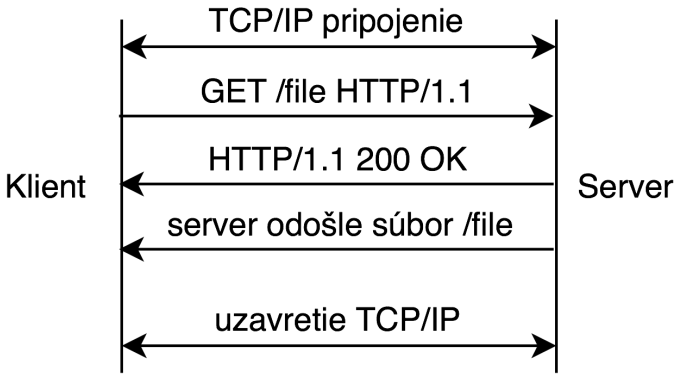
2.3 Protokol HTTP

Protokol HTTP je určený primárne na prepravu informácií na WWW. Definuje formát požiadavky a odpovede medzi klientom a serverom. Pri komunikácii protokolom HTTP má zvyčajne server otvorený port 80, na ktorý sa pripája klient s použitím TCP spojenia. Po ustanovení spojenia klient posiela požiadavku na server, ktorá je spracovaná a klientovi sa pošle odpoveď s informáciou o úspešnosti spracovania (grafický popis komunikácie je na obrázku 2.1). Po odoslaní odpovede sa väčšinou uzavrie spojenie. Táto vlastnosť robí z HTTP bezstavový protokol. V HTTP verzii 1.1 je možné udržať spojenie, v ktorom môže klient spraviť viac požiadaviek. Pre udržanie informácií o stave a kontexte, v ktorom sa klient nachádza, sa používajú prevažne cookies a občasne skryté atribúty v HTTP formulároch.

[8] https://github.com/apple/swift
[9] https://developer.apple.com/xcode/

Správa protokolu sa skladá z dvoch hlavných častí: hlavička a voliteľná časť – telo. Hlavička drží informácie o odosielateľovi a ak telo nie je prázdne, tak aj typ odosielaného obsahu a jeho dĺžku. Formát hlavičky požiadavky a odpovede sa líši iba prvými tromi informáciami. V požiadavke sa udáva HTTP metóda, cesta k požadovanému dokumentu a verzia protokolu HTTP. Odpoveď začína verziou protokolu, číselným kódom odpovede a krátkou správou popisujúcou význam kódu. Pokračovanie hlavičky odpovede aj požiadavky sa skladá zo záznamov typu kľúč hodnota oddelenými dvojbodkou, kde za každou dvojicou je kombinácia znakov CRLF. Druhá časť obsahuje priamo odosielané informácie v kódovaní špecifikovanom v hlavičke [7].

HTTP metódy slúžia k označeniu typu požiadavky. Protokol HTTP používa deväť typov, pričom najčastejšie podporované sú iba štyri – GET, POST, PUT, DELETE. GET sa používa pri žiadosti o dokument. Požiadavka tohto typu by nemala zmeniť vnútorný stav systému, ale iba vrátiť požadovaný dokument. POST sa používa k posielaniu dát na server, ktoré menia vnútorný stav. PUT a DELETE slúžia k manipulácii dokumentov a vždy zmenia vnútorný stav serveru. PUT nahradí všetky reprezentácie dokumentu s požadovanou adresou a DELETE slúži k zmazaniu dokumentu.



Obr. 2.1: Grafický popis komunikácie protokolom HTTP medzi klientom a serverom.

2.4 Vlastnosti webového serveru

Server predstavuje software, ktorý je zameraný na sprostredkovanie služieb pre klientov [9]. Server, ktorý sa venuje sprístupneniu HTML dokumentov a ich zdrojov klientovi s použitím protokolu HTTP [10] sa nazýva webový server. Tento typ serverov využíva prevažne TCP socket pripojený na port 80. Na tento port sa môžu pripojiť klienti a odoslať požiadavku. Webový server by mal mať vytvorený buffer, ktorý je prispôsobený na čítanie obsahu po dvojicu znakov CRLF, keďže táto dvojica znakov slúži ako oddelovač v protokole HTTP. Server by mal na základe prvej časti tohoto protokolu rozhodnúť, ako získa odpoveď pre požiadavku.

Kapitola 3

Návrh a implementácia frameworku a potrebných knižníc

3.1 Abstrakcia nad protokolom HTTP

3.1.1 Hlavička protokolu HTTP

Keďže v protokole HTTP je hlavička tvorená niekoľkými záznamami typu kľúč-hodnota, bolo potrebné dať užívateľovi možnosť pracovať s touto štruktúrou rovnako, ako keby pracoval s asociatívnym poľom. Používanie kľúčov typu reťazec môže viesť k vzniku chýb, pri ktorých sa programátor pomýli v jednom znaku a tým sa požadovaná vlastnosť neprejaví. Aby sa predišlo týmto chybám, je vo frameworku implementovaný výčtový typ (`HTTPHeaderKey`) obsahujúci všetky kľúče hlavičiek. Využitím preťaženia metódy `subscript` sa získala možnosť používať ako kľúč hodnoty tohto výčtového typu, čo zabráni preklepom v reťazcových literáloch.

Celé toto správanie zaobaľuje štruktúra typu `HTTPHeader`, a teda práca s ňou sa javí ako práca s asociatívnym poľom, kde kľúčom môže byť reťazec alebo hodnota z `HTTPHeaderKey`. Táto štruktúra implementuje protokol `Collection`, vďaka ktorému je možné iterovať cez túto štruktúru.

Niektoré kľúče HTTP, ako napríklad dĺžka tela alebo typ obsahu, si držia presný počet parametrov. Táto skutočnosť je využitá pridaním ďalšieho výčtového typu obsahujúceho vnútorné výčtové typy. Tento výčtový typ pomáha pri čitateľnosti kódu a zvyšuje bezpečnosť kódu proti chybám v reťazcových literáloch použitých ako hodnota. Táto štruktúra sa stará aj o tvorbu dát reprezentujúcich hlavičku protokolu HTTP pri odosielaní odpovede.

```
1 var header = HTTPHeader()
2
3 header["Connection"] = "keep-alive"
4 header[.version] = "v1.0"
5 header.set(to: .contentType(.html))
```

Výpis 3.1: Možnosti nastavenia HTTP hlavičky.

3.1.2 Reprezentácia požiadavaky – `Request`

Obsah požiadavky je zaobalený v objekte typu `Request`. Do jeho konštruktoru sa predáva referencia na socket, z ktorého načítava potrebné informácie. Pre lepšiu prácu so socketom

sa používa štruktúra slúžiaca ako buffer. Nad touto štruktúrou sa ďalej volajú metódy podľa potreby. Táto štruktúra poskytuje metódy na čítanie po určitý znak, sekvenciu znakov alebo čítanie určitého počtu bajtov. Pokiaľ buffer nevie vyhovieť požiadavkám volanej metódy (napríklad je prázdny), čaká sa maximálne určený, čas kedy klient môže dodať potrebné data. Ak ich neposkytne a čas vyprší, spojenie sa ukončí.

Po načítaní všetkých potrebných dát zo socketu sa úspešne inizializuje objekt typu `Request` reprezentujúci požiadavku. Tento objekt poskytuje prístup k hlavičke požiadavky a špecificky aj k niektorým jej parametrom. Najbežnejším spôsobom ako poslať nejaké parametre serveru, je zakódovať ich do samotnej URI, alebo ak sa jedná o požiadavku typu POST, tak sa zakódujú do tela požiadavky. Pokiaľ má požiadavka v URI parametre, tak sú dekódované a uložené oddelene. Podobne sa obslúžia aj parametre v tele, pokiaľ to vyžiada programátor a je možné data dekódovať do požadovanej podoby. Pokiaľ sa požiadavka odkazuje na dynamickú URI tak sa jej dynamicky substituované časti uložia v objekte (dynamické URI sú viac popísané v sekcii 3.3.2). Priamy prístup má programátor aj k hodnotám cookies, ktoré slúžia k ukladaniu hodnôt v klientovom prehliadači. Pokiaľ medzi klientom a serverom existuje session, reprezentujúca udržiavanie kontextu, je prístupná aj referencia na tento objekt (kap. 3.2.2).

3.1.3 Reprezentácia odpovede – `ResponseProtocol`

Protokol `ResponseProtocol` popisuje rozhranie potrebné pre odoslanie odpovede klientovi. Pre splnenie jeho požiadaviek treba implementovať dve metódy určené pre odoslanie, premennú typu `HTTPHeader` a premennú typu `HTTPStatus`. Programátor má možnosť vytvoriť vlastný typ popisujúci tvorbu HTTP odpovede. Framework ponúka dve predimplementované triedy, ktoré pokrývajú všetky bežné potreby pre tvorbu odpovede. Trieda `Response` je určená k vytváraniu odpovedí menších až stredných rozmerov. Pri jej inicializácii je potrebné predať parametrom telo správy, ktoré je ďalej nemenné. Hlavička ostáva meniteľná. Z popisu vyplýva, že sa táto správa odosiela naraz, a teda nedáva možnosť streamovania dát. Pre tento prípad je k dispozícii trieda `StreamResponse`, ktorá sa inicializuje predaním closure v ktorej má programátor prístup ku štruktúre predstavujúcej stream. Volaním metódy `send(_:)` môže programátor odosielať dáta v streame klientovi. Keďže pri streamovaní nie je predom známa veľkosť odpovede, používa sa špecialny typ kódovania. Jeho princíp spočíva v odoslaní hlavičky, za ktorou idú dáta vo formáte: veľkosť odosielaných dát v bajtoch; CRLF; dáta; CRLF. Po odoslaní všetkých dát, server pošle číslo nula následované dvomi dvojicami CRLF.

Programátorovi je umožnené predať na odoslanie ľubovoľný typ, ktorý je ďalej transformovaný do typu odvodzujúceho protokol `ResponseProtocol`. Pokiaľ použije typ `String`, výsledkom bude HTML dokument. Inak bude transformovaný do formátu JSON a následne odoslaný klientovi. Táto vlastnosť dáva programátorovi možnosť jednoducho serializovať štruktúry bez kódu naviac. Pokiaľ chce programátor zmeniť toto správanie, stačí mu rozšíriť daný typ pomocou `extension` o protokol `SquirrelPresentable`.

```
1 // Custom structure
2 struct CustomHeadline {
3     let headline: String
4 }
5
6 // Implementing ‘SquirrelPresentable‘ protocol to override
7 // representation of this structure
```

```
8  extension CustomHeadline: SquirrelPresentable {
9      // Data used in HTTP body
10     func present() throws -> Data {
11         return "<h1>\(headline)</h1>".data(encoding: .utf8)!
12     }
13
14     // Content-Type used in HTTP header
15     var representAs: Representation = .html
16 }
```

Výpis 3.2: Predefinovanie finálnej reprezentácie štruktúry v HTTP odpovedi pomocou protokolu `SquirrelPresentable`.

3.2 Držanie kontextu medzi požiadavkami a middleware

Pri práci s klientom je dôležité mať možnosť držať informáciu o stave, v ktorom sa práve klient nachádza. Protokol HTTP, ako bezstavový protokol, priamo nepodporuje držanie informácie o stave, v ktorom sa klient nachádza. Jednoduchým príkladom je držanie informácie o nákupnom košíku v eshopoch medzi viacerými požiadavkami. Riešením je využívať položku cookies v hlavičke protokolu HTTP.

Cookies sú dáta typu kľúč-hodnota, ktoré sa posielajú medzi serverom a klientom v každej požiadavke. Používajú sa k držaniu informácie, v akom stave je klient, k získavaniu štatistických údajov (napr. koľkokrát daný klient navštívil stránku) alebo k reklamným účelom. Ukladajú sa na strane klienta ako malé textové súbory.

3.2.1 Predspracovávanie požiadaviek a postspracovávanie odpovedí – *Middleware*

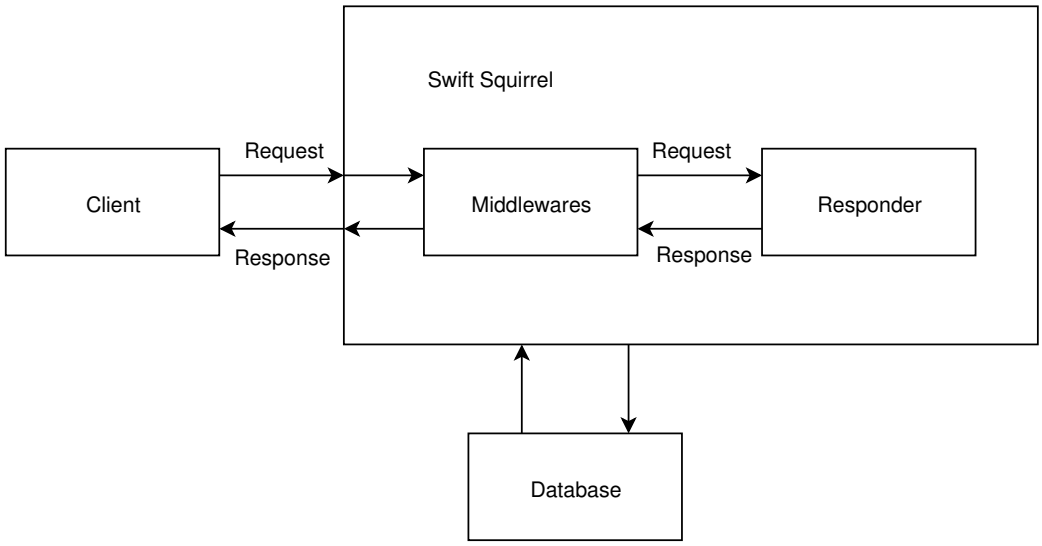
Webový programátor často potrebuje spracovávať niekoľko typov požiadaviek podobne. Aby sa vyhol duplikácii kódu, môže definovať kód, ktorý sa vždy vykoná pred tým ako on pristúpi k požiadavke alebo po jej vybavení. Tento princíp sa používa pri kontrole, či je klient prihlásený alebo k úprave hlavičiek (napr. pridanie verzie API).

Pre vytvorenie middleware slúži protokol `Middleware` požadujúci implementáciu jednej metódy prijímajúcej dva argumenty s ľubovoľným návratovým typom. Prvý argument je objekt typu `Request` držiaci informácie o požiadavke a druhým je odkaz na funkciu, ktorá má ďalej spracovať požiadavku a vrátiť odpoveď. Programátor môže vyhodnotiť informácie z požiadavky a odpovedať priamo v middlewari alebo nechať odpoveď vyhodnotiť, ktorú môže následne zmeniť. Pri reťazení viacerých middlewarov sa využíva redukcia pomocou vnorovania closures obsahujúcich volanie príslušného middlewaru.

Programátor má možnosť použiť dva predimplementované middleware, z nich prvý slúži na označenie odpovede, aby nebola ukladaná. Tento systém bráni klientom získať prístup k obsahu po odhlásení klienta použitím spätného tlačidla v prehliadači. Pre držanie informácii o klientovi slúži `SessionMiddleware`, ktorý kontroluje, či k požiadavke existuje valídna session (kap. 3.2.2).

3.2.2 Držanie kontextu – *Session*

Pre jednoduché sprístupnenie možnosti udržania si kontextu, v ktorom sa klient nadhádza, bolo potrebné vytvoriť dva protokoly. Prvý popisuje, ako sa bude dať pracovať s dátami



Obr. 3.1: Grafický popis komunikácie klient server s použitím middleware.

uloženými v session (protokol `Session`) a druhý popisuje, ako vytvoriť a pracovať so session (`SessionBuilder`).

Typ implemenujúci `Session` slúži ako asociatívne pole, kde kľúč je reťazec a hodnota je štruktúra JSON (kap. 3.7.1). Tomuto typu je možné nastaviť príznak symbolizujúci zmazanie všetkých dát a získať čas, dokedy je táto štruktúra valídna. Framework poskytuje predimplementovanú štruktúru, ktorá spĺňa požiadavky protokolu. Dáta v štruktúre sa ukladajú do súboru vo formáte JSON. Názov súboru sa zhoduje s identifikátorom klientovej session. Súbor obsahuje dáta uložené programátorom, čas expirácie a informácie o prehliadači, aký použil. Pri načítavaní súboru sa používa JSON dekodér a pokiaľ načítavaný súbor má expiračnú dobu v minulosti, je zmazaný.

O vytváranie, získavanie a čistenie sa stará predimplementovaná štruktúra odvodzujúca `SessionBuilder`. Pri vytváraní novej session je potrebné jej priradiť identifikátor. Ten je spočítaný ako md5 hash[1] z informácií o použitom prehliadači, aktuálnej časovej známky a náhodných 32 bajtov. Pokiaľ klient disponuje identifikátorom session, prebehne kontrola, či sa jedná o neexpirovanú session a či sa zhoduje IP adresa a informácie o prehliadači s údajmi uloženými v súbore.

Pre vytvorenie session je potrebné zaslať klientovi identifikátor v cookies. Pre pridanie referencie na session je potrebné overiť, či je možné priradiť ku klientovi session na základe jeho požiadavky. Táto činnosť môže byť automatizovaná pomocou predimplementovaného middlewaru `sessionMiddleware`. Jediné, čo musí programátor spraviť, je inicializovať túto štruktúru, pridať ju medzi používané middlewari a framework sa postará o to, aby mal vždy prístup k session, ak existuje, o správne mazanie a vytváranie novej session. Pri inicializácii je možné nastaviť voliteľné dva argumenty. Prvý je štruktúra, ktorá implementuje protokol `SessionBuilder` a druhý určuje, ako často sa majú mazať expirované sessions. Session sa často ukladá do databázy namiesto do súborového systému kvôli prístupu z viacerých serverov [11]. Tieto argumenty dávajú možnosť programátorovi použiť vlastný systém pre správu session a nebyť odkázaný iba na predimplementované štruktúry využívajúce súborový systém.

[1] https://www.ietf.org/rfc/rfc1321.txt

Programátor má aj prístup ku konfiguračnej štruktúre, v ktorej je možné nastaviť dobu, po akej sa označia session súbory ako expirované a kľúč, pod ktorým bude uložený identifikátor session v súboroch cookies.

3.3 Spravovanie smerovaní – Routing

Routing je proces, pri ktorom sa vyberá vhodná cesta v smerovači. Podobným princípom sa dá popísať aj routing vo webových frameworkoch. Jedná sa o proces, pri ktorom sa na základe požadovanej cesty (URI) a použitej HTTP metódy hľadá funkcia, ktorá sa použije pre vytvorenie odpovede (anglicky „handler“).

Pre efektívne vyhľadávanie handleru je zvolená stromová štruktúra (príklad stromu na obrázku 3.2), ktorej každý uzol je tvorený jednou časťou[2] cesty. Jedinou výnimkou je koreňový uzol reprezentovaný symbolom lomítko. Pre reprezentáciu uzlu sa použije jedna z dvoch tried. Statický uzol, reprezentujúci časť cesty s konkrétnou hodnotou, alebo dynamický uzol reprezentujúci mennú časť cesty, ktorá môže byť substituovaná. Každý uzol si drží pole statických poduzlov, pole dynamických poduzlov a dve asociatívne polia, ktorých kľúč je HTTP metóda a hodnota closure predstavujúca handler. Framework podporuje 5 metód – GET, POST, PUT, DELETE, PATCH[3].

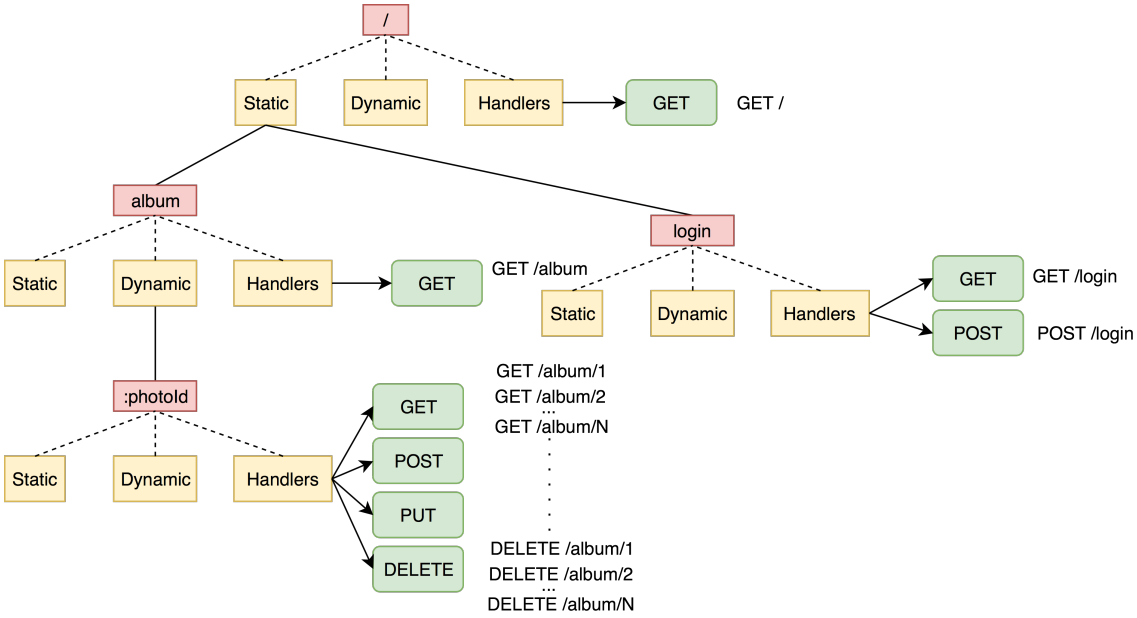
Pri vyhľadávaní správneho handleru sa cesta vo formáte URI rozdelí na podreťazce podľa oddeľovača lomítko. Výsledné pole sa predá koreňovému uzlu, ktorý začne vyhľadávanie v strome. Uzol najskôr kontroluje dĺžku poľa. Pokiaľ je pole prázdne, vyhodnotí sa vyhľadávanie ako neúspešné. Ak pole obsahuje iba jeden prvok, vráti sa handler pre požadovanú metódu ak existuje, inak sa vyhodí chyba typu `notAllowed` obsahujúca podporované metódy pre daný uzol. Ak pole obsahuje viac prvkov, vymaže sa z neho prvý prvok a rekurzívne sa prehľadávajú jeho statické a dynamické uzly.

3.3.1 Routing statických súborov

Pre obsluhovanie statických súborov slúžia dva priečinky. Priečinok *Public* je určený pre súbory, ktoré majú byť voľne prístupne a nie sú menené alebo vytvárané za behu programu. Patria sem súbory HTML, CSS, JS a ďalšie zdroje, ako sú obrázky a videá. Nie je vhodné sem ukladať dočasné súbory, ktoré sú generované počas behu. Pre tento účel je vyhradený priečinok *Storage/Public*. Pri sputení sa framework pokúsi vytvoriť symlink *Public/Storage* → *Storage/Public*, vďaka ktorému bude možné pristúpiť k súborom pomocou prefixu `Storage`. Pokiaľ sa klient odkazuje na priečinok, vyhľadá sa v ňom súbor s názvom *index.html*.

- */* → *Public/index.html*
- */index.html* → *Public/index.html*
- */Images/squirrel.png* → *Public/Images/squirrel.png*
- */Storage/file.txt* → *Storage/Public/file.txt*

[2] Pod časťou sa myslí reťazec medzi dvomi lomítkami
[3] Metóda PATCH sa používa pri nahratí zmien pre daný dokument



Obr. 3.2: Ukážka možného stromu smerovaní, v ktorom je najvyšší uzol koreňový, **červená:** uzol stromu, **žltá:** atribúty uzlu obsahujúce statické a dynamické uzly a handlere, **zelená:** handler pre konkrétnu HTTP metódu.

3.3.2 Pridanie nových smerovaní

Pridávať nové uzly do stromu môžu typy implementujúce protokol `Router`. Tento protokol je rozšírený o sadu metód pre správu stromu a pre zoskupovanie ciest s podobnými vlastnosťami. Jeho dôležitým atribútom je pole middlewarov, ktoré sa aplikujú na všetky cesty pridané daným typom. Pre zoskupovanie ciest je možné použiť metódu `group`, ktorá prijíma closure s definovanými novými cestami a ich handlermi. Týmto spôsobom sa dajú čitateľne pridávať nové cesty s podobnými vlastnosťami, ako sú napríklad rovnaký prefix v URI alebo spoločné middleware.

Pri pridávaní novej cesty s handlerom sa podľa znaku lomítko rozdelí pridávaná cesta do poľa. Každá časť môže nadobudnúť jeden z troch sémantických významov. Pokiaľ časť cesty má prefix : (dvojbodku), jedná sa o dynamickú časť cesty, ktorá môže byť substituovaná za ľubovoľnú hodnotu. Pokiaľ je časť rovná znaku * (hviezdička) jedná sa o dynamickú časť, ktorá môže byť nahradená za ľubovoľný počet hodnôt. Tretí prípad reprezentuje statickú časť cesty, a teda neprebieha žiadna substitúcia.

```
1  // exact match
2  // /albums
3  server.get("/albums") { ... }
4
5  // dynamic match
6  // /albums/12
7  // /albums/summer
8  // /albums/anything
9  server.get("/albums/:id") { ... }
10
11 // dynamic prefix match
```

```
// /photos/
// /photos/something/
// /photos/something/longer/and/longer
server.get("/photos/*") { ... }
```

Výpis 3.3: Príklady možných substitúcii v ceste.

3.3.3 Možnosti parametrov pri spracovávaní požiadaviek

Protokol `Router` je rozšírený o metódy pre pridávanie nových handlerov pre určenú cestu[4]. Každá z týchto metód má 64 preťažení prevažne líšiacich sa v poslednom parametri predstavujúcom closure pre obslúženie požiadavky (handler).

Preťaženia slúžia k automatickej kontrole parametov potrebných pre správne vyhodnotenie odpovede. Vo väčšine iných frameworkov si programátor musí získať parametre z požiadavky a potom skontrolovať ich validitu (typ, hodnotu), čo vedie k množstvu zbytočného kódu. V mojom riešení som sa inšpiroval frameworkom Tris a túto kontrolu je možné spraviť automaticky. Na rozdiel od Trisu moje riešenie ponúka viac možností pre kontrolu parametrov. Programátor musí definovať štruktúru, s akou chce pracovať, a framework sa postará o to, aby ju mal vždy prístupnú. Pokiaľ požiadavka neobsahuje dostatočné informácie potrebné pre inicializáciu danej štruktúry, framework vráti HTTP odpoveď *400 Bad Request*. V handleri môže použiť nula až 6 argumentov. Objekt typu `Request` pre prístup k všetkým údajom z požiadavky, vlastný typ implementujúci protokol `Decodable`, ktorý bude vytvorený z URI parametrov, vlastný typ implementujúci `BodyDecodable`, ktorý je vytvorený z parametrov v tele požiadavky, objekt reprezentujúci session (protokol `Session`), vlastný typ implementujúci `SessionDecodable`, ktorý je vytvorený z dát uložených v session a posledný parameter je vlastný typ. Pri použití posledného parametru musí programátor poskytnúť closure, ktorá tento typ inicializuje z požiadavky. Vo väčšine prípadov nie je potrebné použiť všetky parametre. Veľké množstvo preťažení dáva možnosť vynechať nepotrebné parametre, čo vedie k lepšej čitateľnosti kódu. Tieto preťaženia sú generované pomocou skriptu v jazyku Python, vďaka čomu je jednoduché pridávať nové parametre v prípade potreby. Programátor takisto môže vytvoriť vlastné preťaženia metód pomocou kľúčového slova `extension`.

```
// Attributes expecting from html registration form
struct UserRegistrationForm: Decodable {
    let login: String
    let password: String
}

// Automatic check for login and password attribute
server.post("/register") { (form: UserRegistrationForm) in
    print(form.login)
    return "ok"
}
```

Výpis 3.4: Využitie automatickej kontroly atribútov a typov pri registácii klienta. Pokiaľ požiadavka nebude obsahovať atribúty `login` a `password`,odošle sa klientovi HTTP odpoveď *400 Bad Request*.

[4]Metódy: `get`, `post`, `put`, `delete`, `patch`.

Ak programátor využije v handleri typ implementujúci `Decodable`, framework pred zavolaním handlera skúsi vytvoriť tento typ pomocou vlastného dekóderu určeného na dekódovanie záznamov typu kľúč-hodnota. Tento typ podporuje tvorbu základných typov[5], štruktúr a polí. Dekóder prijíma asociatívne pole získané z parametrov v požiadavke. Do prázdneho asociatívneho pola sa najprv pridajú dynamicky substituované parametre z cesty[6] a parametre dekódované z URI. Pokiaľ chce dekódovať parametre z tela požiadavky, musí použiť typ `BodyDecodable`, ktorý sa postará o správne dekódovanie obsahu tela, pokiaľ je podporované. Pri použití objektu typu odvodzujúceho od protokolu `Session`, sa kontroluje, či je možné priradiť k požiadavke valídnu session. V prípade, že požiadavka nemá session, vráti sa užívateľovi HTTP odpoveď *400 Bad Request*. Táto kontrola prebehne, aj keď programátor využije v handleri typ `SessionDecodable`. Okrem overenia validity session sa daný typ skúsi vytvoriť z uložených dát v session a pri neúspechu vráti odpoveď *400 Bad Request*.

3.3.4 Mazanie existujúcich smerovaní

Programátorovi je umožnené zmazať zo stromu ľubovoľný handler. Pre tento účel je vyhradená metóda `drop`, ktorej sa ako argumenty predá cesta a metóda. Pri mazaní sa kontroluje prázdnosť uzlov a mažú sa uzly, ktoré už nemajú žiadny handler.

3.4 Implementácia triedy server

Pre obsluhu klienta je určená trieda `Server`, ktorá má na starosti prijímať požiadavky a priradiť k nim správne odpovede. Tento objekt je možné nastaviť s určitými parametrami pri inicializácii alebo konfigurovať pomocou konfiguračného súboru. Inštancia tejto triedy slúži pre spravovanie routovania (implementuje protokol spomenutý v kapitole `Router` 3.3.2) a k ovládaniu chodu servera.

3.4.1 Adresárová štruktúra serveru

Pre správne správanie servera je potrebné dodržiavať určitú štruktúru adresárov. Pokiaľ nejaký adresár chýba, je automaticky vytvorený pri prvom spustení frameworku. Adresár *Resources* je určený pre zdroje, ktoré by nemali byť viditeľné klientom. V adresároch *Public* a *Storage/Public* sa nachádzajú súbory, ktoré sú voľne prístupné pre všetkých klientov. Adresár *Storage* ďalej obsahuje ďalšie podadresáre, ktoré už klientom nie sú prístupné.

- *Storage/Cache* – vyhradené pre ukladanie cache
- *Storage/Fruits* – vyhradené pre generované súbory z NutView (kap. 3.5)
- *Storage/Logs* – vyhradené pre ukladanie logov
- *Storage/Sessions* – vyhradené pre ukladanie session

3.4.2 Konfigurácia objektu server

Inštanciu triedy `Server` je možné nastavovať staticky pomocou konfiguračného súboru alebo priamo pri inicializácii. Pri inicializácii je možné nastavovať port, na ktorom má server čakať na pripojenia, URI prefix, ktorý bude automaticky pridaný všetkým pridaným smerovaniam, a middleware, ktoré budú použité pre každé pridané smerovanie.

[5] Typy Int, String, Bool, Double a ich odvodené.
[6] Časti cesty s prefixom dvojbodka.

Pre písanie konfiguračných súborov pre webové frameworky sa najčastejšie používa formát JSON a jazyk YAML. Formát JSON je populárny vďaka širokému použitiu (napríklad pre komunikovanie medzi API), ale obsahuje veľa zbytočných znakov, ako sú napríklad zložené zátvorky vďaka čomu je „čitatelnejší“ pre programy. V jazyku YAML už záleží na počte a umiestení bielych znakov. Jeho popularita je dôsledkom lepšej čitateľnosti pre ľudí ako JSON. Aj z tohto dôvodu sa stal jazyk YAML konfiguračným jazykom pre vyvíjaný framework.

Pri spustení frameworku sa skontroluje existencia súboru *.squirrel.yaml* alebo *.squirrel.yml* a v prípade existencie sa dekóduje obsah a uložia sa podporované nastavenia[7]. Pomocou tohoto súboru je možné konfigurovať názov domény, aký bude zobrazovaný v odpovediach, port, na ktorom má server prijímať nové pripojenia, a maximálny počet čakajúcich pripojení. Pomocou statickej metódy `getDBData` triedy `Config` je možné získať nastavenia pre databázu. V konfiguračnom súbore je možné sa odkázať na premenné z prostredia pomocou prefixu $ (znak dolár).

Kompletnú konfiguráciu a dôležité konfiguračné premenné drží jedináčik triedy `Config`. V inštancii tejto triedy sa držia informácie, ako sú cesty k dôležitým priečinkom (priečinky *Public*, *Storage* apod.), číslo portu, maximálny počet čakajúcich pripojení a atribúty slúžiace pre konfiguráciu logovacích správ. Pri inicializovaní jedináčika prebehne inicializácia všetkých premených podľa konfiguračného súboru alebo podľa východzích hodnôt. Inicializácia jedináčika prebehne automaticky pri spustení frameworku.

Logovanie prebieha pomocou knižnice *SwiftyBeaver*[8], ktorá ponúka možnosť logovať päť druhov správ. Ako prednastavený súbor pre ukladanie logov sa použije súbor s cestou *Storage/Logs/server.log*. Trieda `Config` obsahuje atribút `log`, pomocou ktorého je možné logovať nové správy, pridávať nové logovacie destinácie (prednastavené sú spomínaný súbor a terminál) alebo meniť ich minimálnu úroveň dôležitosti pre zobrazenie v logovacej destinácii.

3.4.3 Ovládanie chodu servera

Po inicializácii inštencie triedy `Server` je možné spustiť obluhovanie metódou `run`. Táto metóda sa pomocou neblokujúceho socketu pripojí na určený port, na ktorom bude prijímať nové spojenia. Po prijatí sa pridá do konkurenčne odbavujúcej fronty blok kódu s obslúžením pripojenia, ktorý bude vykonaný asynchrónne. Pre zastavenie prijímania je možné použiť metódu `stop`, ktorá prijíma jeden argument, určujúci či treba nechať dokončiť rozpracované požiadavky alebo ich okamžite ukončiť. Metóda `pause` pozastaví prijímanie požiadaviek, vykoná určitý blok kódu, po ktorom pokračuje v prijímaní nových požiadaviek. Tento blok kódu je predaný ako closure pri volaní. Táto metóda prijíma argument na určenie, či treba počkať na dokončenie rozpracovaných požiadaviek alebo ich násilne ukončiť rovnako ako metóda `stop`. Táto metóda je určená pre vykonanie kódu, počas ktorého nie je vhodné prijímať nové požiadavky a programátor vie, že po vykonaní tohto kódu chce znova povoliť prijímať nové požiadavky.

3.4.4 Obslúženie požiadavky od klienta

Pri obsluhovaní klienta sa inicializuje inštancia triedy `Request` (kapitola 3.1.2), ktorej sa ako argument predá objekt obaľujúci socket. Po úspešnej inicializácii sa prejde k vyhľadaniu

[7] Načítané nastavenia je možné prepísať argumentami pri inicializácii inštancie triedy `Server`.
[8] https://swiftybeaver.com

správneho handleru. Vyhľadávanie prebieha v niekoľkých fázach. V prvej fáze sa prehľadá strom obsahujúci všetky handlere. Pokiaľ vyhľadávanie skončí neúspešne, vznikne predpoklad, že požadovaná cesta vedie k súboru v priečinku *Public* alebo *Storage/Public*. K týmto súborom je povolené pristupovať iba HTTP metódou `GET`. Ak požadovaná cesta nevedie k súboru ani k priečinku, vyhodnotí sa vyhľadávanie ako neúspešné. Pokiaľ je požadovaná cesta priečinok, skúsi sa v nej vyhľadať súbor *index.html*. V prípade, že cesta vedie k súboru, odošle sa. Po nájdení požadovaného handlera sa vykoná jeho kód a získa sa požadovaná odpoveď. Ak vznikne chyba počas vyhľadávania (vyhľadávanie skončilo neúspešne) alebo počas tvorby odpovede, je vyhľadaná náhradná odpoveď podľa vzniknutej chyby (kap. 3.4.5).

Odpoveď je možné odoslať dvomi spôsobmi. Jeden odošle odpoveď celú a druhý slúži k odoslaniu iba určitej časti odpovede. Ak má požiadavka určený rozsah a odpoveď má HTTP status rovný *200 OK*, odošle sa klientovi iba požadovaná časť. V opačnom prípade sa odosiela klientovi celá odpoveď.

Po odoslaní sa kontroluje, či prijatá požiadavka mala nastavenú hlavičku `connection` na `keep-alive`, ktorá sa používa, keď klient nechce uzatvoriť spojenie okamžite po obdržaní odpovede, pretože predpokladá, že bude potrebovať načítať ďalšie súbory (napr. CSS, Javascript, obrázky, a pod.). Ak klient požaduje udržanie spojenia, je toto spojenie držané maximálne 20 sekúnd (túto hodnotu je možné zmeniť). Po vypršaní času sa socket uzatvorí.

3.4.5 Spracovávanie chýb pri získavaní odpovede pre klienta

Pri odpovedaní klientovi môže vzniknúť niekoľko chýb. Medzi bežné chyby patrí: vypršanie času určeného na čítanie zo schránky, chyba pri vyhľadávaní handlera (neúspešné vyhľadávanie, použitie nepodporovanej metódy, a pod.) a chyba vyhodená programátorom v handleri. Pri vzniku takýchto chýb je stále potrebné odoslať odpoveď popisujúcu čo sa stalo, prípadne čo spravil klient zle alebo akú požiadavku mal poslať, aby dostal odpoveď bez chyby. Pre jednoduché spracovávanie týchto chýb framework ponúka možnosť rozšíriť typ chyby o protokol `HTTPConvertibleError` v ktorom sa implementuje spôsob, ako z danej chyby vytvoriť správnu odpoveď pre klienta.

Pre získanie odpovede z chyby je určená trieda `ErrorHandler`. Obsahuje pole handlerov chýb, ktoré dostávajú postupne možnosť prejaviť záujem o určitú chybu a vrátiť odpoveď. Ak nejaký handler vráti odpoveď, odošle sa klientovi. V prípade, že na danú chybu nevznikne odpoveď, klientovi sa odošle odpoveď s textovou reprezentáciou chyby a statusom HTTP *500 Internal Error*, nakoľko nie je možné chybu reprezentovať iným statusom. Pri inicializácii inštancie triedy `ErrorHandler` obsahuje pole handlerov jeden prvok, ktorý spracuje všetky chyby odvodzujúce protokol `HTTPConvertibleError` a o iné chyby neprejaví záujem (namiesto odpovede vráti hodnotu `nil`). Programátor môže pomocou metódy `addError(handler:)` pridať vlastné chybové handlere. Handlerom sa môže stať ľubovoľný typ implementujúci protokol `ErrorHandlerProtocol`. Okrem možnosti určiť odpoveď pre skupiny chýb, môže programátor týmto spôsobom predefinovať reprezentáciu frameworkom nadefinovaných chýb. Napríklad pri chybe reprezentujúcej nenájdenie požadovaného zdroja, môže programátor prepísať frameworkom definovanú odpoveď a vytvoriť odpoveď vhodnejšiu pre jeho potreby.

3.5 Šablónovací jazyk – *NutView*

Šablónovacie jazyky sú veľmi populárne vo webových aplikáciach. Väčšinou sa jedná o špeciálne príkazy zapísané priamo v kóde HTML. Pri čítaní tohto kódu sa interpretujú tieto

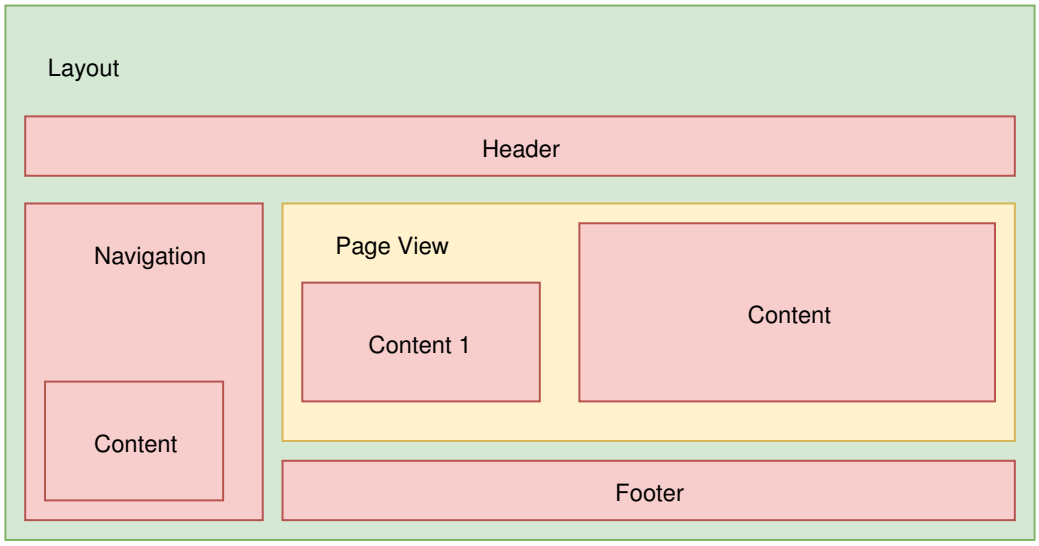
príkazy a tým sa pridáva dynamickosť pri tvorbe HTML dokumentu. Medzi najčastejšie príkazy patria riadiace príkazy (`if`, `for`) a príkazy na umiestnenie hodnôt vytvorených za behu programu do HTML dokumentu. Pre väčšinu týchto jazykov platí, že sa snažia priblížiť svojou syntaxou jazyku, pre ktorý sú určené (Laravel (PHP) – Blade šablony). Z tohto dôvodu som navrhol šablónovací jazyk so syntaxou príkazov čo najviac podobnou jazyku Swift a implementoval interpretér ako samostatnú knižnicu – *NutView*[9].

Pre získanie interpretovaného obsahu je určená trieda `View`, ktorá zo zadaného názvu súboru a voliteľných dát zostaví požadovaný HTML dokument. Pri zostavovaní dokumentu sa využívajú dva typy súborov. Zdrojové „Nut“ súbory s príponou *.nut.html* obsahujú HTML kód spolu so šablonovacím kódom. Tie sa pri interpretácii skompilujú do „Fruit“ súborov s príponou *.fruit*, ktoré obsahujú kód pripravený pre interpretáciu bez potreby lexikálnej, syntaktickej a sémantickej analýzy.

Knižnica *NutView* je konfigurovateľná pomocou statických metód a premenných triedy `NutConfig`. Ponúka možnosť zmeniť priečinky, v ktorých sa nachádzajú *Nut* a *Fruit* súbory, prednastavený formát dátumu, veľkosť a umiestnenie cache súborov a možnosť zmazať všetky vygenerované *Fruit* súbory.

3.5.1 Koncept – *Layout-View-Subview*

Koncept vychádza z toho, že každá webová stránka sa dá rozdeliť do viacerých častí spadajúcich iba do troch tried (obrázok 3.3): hlavný obsah stránky, ktorý je na každej stránke iný (*View*); rozloženie stránky popisujúce ako sú rozmietnené časti stránky okolo hlavného obsahu, ktoré je rovnaké pre viacej stránok (*Layout*); časti stránky, ktoré sa môžu opakovať na viacerých stránkach (*Subview*). Toto dáva programátorovi možnosť rozdeliť web stránku do viacerých súborov. Pre tento účel sa využívajú tri priečinky (*Layouts*, *Views*, *SubViews*), ktoré udávajú, do akej triedy spadajú súbory.



Obr. 3.3: Príklad rozloženia webovej stránky a rozdelenie jej častí do troch tried, **zelená:** rozloženie stránky (*Layout*), **žltá:** hlavný obsah stránky (*View*), **červená:** opakovateľné časti stránky (*Subview*).

[9] https://github.com/Swift-Squirrel/NutView

3.5.2 Hlavný obsah stránky – *View*

View reprezentuje hlavný obsah stránky. Každá stránka by mala mať vlastný obsah, a teda všeobecne platí, že pre každú stránku by mal existovať práve jeden *View*. V tomto súbore je možné určiť titulok stránky (text v HTML tagu `<title>`) a v akom rozložení má byť *View* uložené. Programátor sa môže v kóde odkazovať iba na *View* súbory, pretože stránka je tvorená obsahom obklopeným ďalšími častami, a teda nie je dôvod mať možnosť priamo volať nepodstatné časti stránky.

```
1 <!-- file Views/SomeView.nut.html -->
2 \Layout("SomeLayout")
3 \Title("Index")
4
5 <B>Hello, \(name)!</B>
```

Výpis 3.5: Príklad obsahu *View* súboru, ktorý má byť vložený do rozloženia dokumentu s názvom *SomeLayout.nut.html*, s titulkom *Index* a v obsahu bude pozdravenie osoby s menom uloženým v premennej `name`.

3.5.3 Rozloženie stránky – *Layout*

Súbor typu *Layout* by mal okrem kódu HTML určujúceho rozloženie stránky, obsahovať aj príkaz `\View()`, ktorým určí, kde sa má vložiť hlavný obsah. Súbor by mal obsahovať HTML tagy `<html>`, `<head>` a `<body>`.

```
1 <!-- file Layouts/SomeLayout.nut.html -->
2 <HTML>
3     <HEAD></HEAD>
4     <BODY>
5         \View()
6     </BODY>
7 </HTML>
```

Výpis 3.6: Príklad súboru s rozložením stránky, ktorý do HTML tagu `<body>` vloží odkazovaný súbor *View*.

3.5.4 Opakovateľné časti stránky – Subview

Pre zníženie potreby programátora písať viackrát ten istý kód, je možné nejaké časti kódu presunúť do samostatných súborov, na ktoré sa potom odkáže. Klasickým príkladom sú rôzne banery, hlavičky a pätičky webových stránok. Je vhodné presunúť do samostatného súboru obsah HTML tagu `<head>`, hlavičku a pätičku stránky.

3.5.5 Zaujímavé príkazy

Pri návrhu príkazov bolo treba navrhnúť spôsob, akým bude možné rozoznať príkaz od kódu HTML. Zvolil som pridanie znaku spätné lomítko (\) ako prefix pre každý príkaz. Príkazy boli navrhované tak, aby sa riadili syntaxou jazyka Swift.

Knižnica ponúka príkaz `\Head(_:)` slúžiaci pre pridanie určeného reťazca do tagu `<head>` v ľubovoľnej triede súboru, vďaka čomu si môže programátor pripraviť CSS súbory špecifické pre rôzne časti webovej stránky a na tieto súbory sa odkáže priamo zo

súboru obsahujúceho časť stránky, ktorej sa daný súbor CSS týka. JavaScriptové súbory sa často dávajú pred koncový tag tela dokumentu HTML (`<body>`). Toto je možné docieliť príkazom `\Body(_:)`, ktorý je k tomuto účelu určený.

Pre overenie, či existuje hodnota určitej premennej, je možné použiť príkaz `\if let <variable> = <expression> { ... \}` , podobne ako v jazyku Swift. Tento príkaz vyhodnotí `<expression>`, a pokiaľ výsledkom nie je `nil`, uloží výsledok do `<variable>` a vykoná určený blok kódu. Tento príkaz podporuje oddelenie viacerých podmienok pomocou znaku čiarka (`,`). Pre vykonanie bloku kódu musia byť všetky podmienky pravdivé.

Pre spracovávanie dátumov slúži funkcia `Date(_:format:)`, ktorá berie dva argumenty. Prvý je čas od 1.1.1970 v sekundách alebo premenná typu `Date` z knižnice `Foundation`. Druhý argument je voliteľný a predstavuje formát dátumu. Ak je druhý argument nezadaný, použije sa prednastavená hodnota uložená v `NutConfig.defaultDateFormat`.

3.5.6 Kompilovanie *Nut* súborov

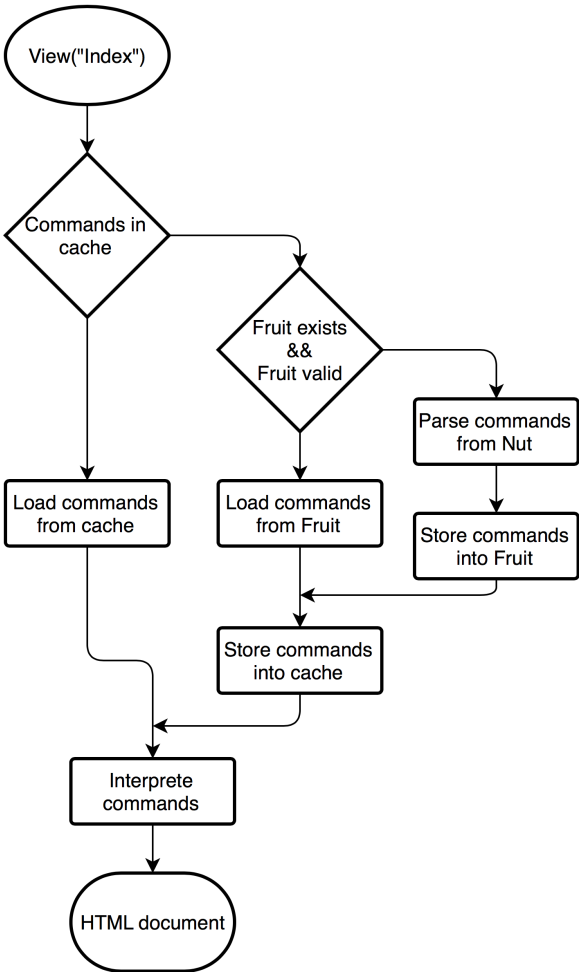
Nut súbory obsahujú HTML kód zmiešaný so šablonovacím kódom. Obsah týchto súborov je potrebné pred interpretovaním podrobiť lexikálnej a syntaktickej analýze, aby sa odhalili možné chyby, ktoré zabraňujú spracovaniu príkazov v súbore. Lexikálna analýza prebieha počas syntaktickej analýzy.

Tokeny lexikálnej analýzy sa dajú rozdeliť do dvoch nadskupín. Prvá skupina obsahuje iba jeden typ tokenu reprezentujúci HTML text. Druhá skupina je niekoľko typov tokenov, ktoré reprezentujú šablónovací príkaz. Cyklicky je lexikálna analýza dotazovaná otázkou, aká skupina tokenov nasleduje, až kým lexikálna analýza neprejde celý súbor. Po každom zodpovedanom dotaze sa podľa typu tokenu začne kontrolovať syntax nasledujúcich tokenov. Keď sa začne spracovávať príkaz, ktorý obsahuje telo s ďalším kódom (`if`, `for`), uložia sa na vrch zásobníka informácie o tejto štruktúre. Tieto informácie sú vybrané zo zásobníka po spracovaní celého tela a spracované telo sa uloží do štruktúry. Po úspešnom spracovaní šablónovacieho príkazu alebo HTML textu sa uloží štruktúra s informáciami o konštrukcii do poľa. Po spracovaní celého súboru je toto pole uložené do štruktúry typu `ViewCommands`, ktorá popisuje príkazy daného šablónovacieho súboru. Z tejto štruktúry je možné interpretovať všetky príkazy. Výsledná štruktúra sa ukladá ako *Fruit* súbor vo formáte JSON.

3.5.7 Interpretácia šablónovacích súborov

Interpretovanie súboru začína súborom triedy *View*, ktorý je zadaný ako argument. Pokiaľ sa súbor nachádza v ďalšom podpriečinku, je použitá bodková konvencia, podľa ktorej sa namiesto znaku / (lomítko) píše znak . (bodka) pre lepšiu čitateľnosť v kóde. Pri interpretovaní súboru sa skontroluje, či požadovaný súbor existuje. Pokiaľ je pokus o interpretovanie neexistujúceho súboru, vyhodí sa chyba. Pokiaľ súbor existuje, získa sa jeho časová známka poslednej modifikácie a časová známka vytvorenia príslušného *Fruit* súboru (ak existuje), a pokiaľ je čas modifikácie novší, je potreba znova skompilovať daný súbor, pretože existuje jeho novšia verzia. Po prípadnej kompilácii a uložení do vyrovnávacej pamäte alebo po načítaní obsahu z *Fruit* súboru je prístupná štruktúra typu `ViewCommands`, obsahujúca všetky príkazy pripravené k interpretácii daného súboru.

Príkazy sa interpretujú sekvenčne za sebou. Počas tohto procesu sa interpretujú iba príkazy, ktoré môžu ovplyvniť obsah tela výsledného html dokumentu. Príkazy ako `\Title(_:)`, `\Body(_:)` a `\Head(_:)`, ktoré sú určené k zmene hlavičky a konca tela dokumentu, sú uložené nabok k neskoršej interpretácii. Pokiaľ je interpretovaný príkaz `\Layout(_:)`, ktorý obsahuje názov súboru popisujúceho rozloženie HTML dokumentu, interpretuje sa iba výraz



Obr. 3.4: Algoritmus popisujúci proces interpretovania šablónovacieho súboru s názvom *Index.nut.html*.

pre získanie názvu tohto súboru a uloží sa na neskoršiu interpretáciu, podobne ako príkazy meniace hlavičku a koniec tela. Pri príkaze `\Subview(_:)` sa hneď interpretuje obsah tohto súboru okrem príkazov meniacich hlavičku, ktoré sa podobne ako pred tým, ukladajú nabok. Po interpretácii tela sa vykoná interpretácia rozloženia dokumentu (pokiaľ je nejaké určené) a interpretujú sa aj príkazy meniace hlavičku a koniec tela, ktorých výsledok sa pomocou regulárnych výrazov pridá pred koniec HTML hlavičky a tela v dokumente.

3.5.8 Evaluácia výrazov

Pri návrhu šablónovacieho jazyka vznikla potreba mať možnosť vyhodnotiť výraz zapísaný v reťazci. Pre tento účel som navrhol samostatnú knižnicu, ktorá ponúka vyhodnocovanie výrazov podobným spôsobom, ako ich vyhodnocuje jazyk Swift. Celkovo podporuje pätnásť operátorov a šesť funkcií určených k pretypovaniu hodnoty.

Pri vyhodnocovaní sa výraz rozdelí do viacerých tokenov a podľa precedencie tokenu sa infixový výraz transformuje do postfixového výrazu, ktorý je nakoniec vyhodnotený. Vo vyhodnocovanom výraze sa môže vyskytnúť aj identifikátor premennej. Pre sprístupnenie hodnoty premennej vo výraze je možné predať pri vyhodnocovaní argument obsahujúci

všetky premenné. Tento argument je asociatívne pole, kde kľúč je názov premenej (typ `String`) a ľubovoľná hodnota (typ `Any`). Pokiaľ je hodnotou ďalšie asociatívne pole, je možné k nemu pristupovať cez bodku ako k premenným v štruktúrach.

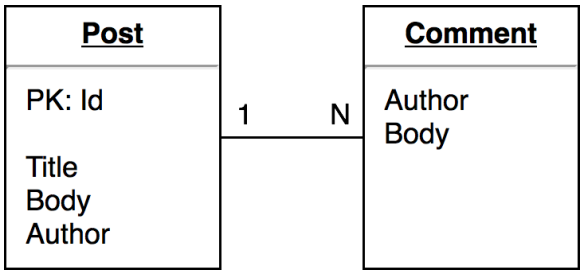
Postfix je možné serializovať do formátu JSON, z ktorého je možné znova deserializovať postfixový výraz a vyhodnotiť ho. Tento princíp bol implementovaný s myšlienkou, že keď sa prvýkrát skompiluje *Nut* súbor, tak sa do *Fruit* súboru uloží výraz v postfixovej forme a pri vyhodnocovaní výrazu už bude treba len naplniť dáta a priamo vyhodnotiť. Tento systém sa neosvedčil, nakoľko réžia spôsobila časovo náročnejšie vyhodnocovanie dokumentov HTML ako keď sa pri každom použití výrazu vytvoril postfix nanovo.

3.6 Práca s databázou – Knižnica *Squirrel Connector*

Databáza je základným kameňom mnohých webových aplikácií, preto je dôležité, aby práca s ňou bola čo najladnejšia. Existujúce riešenia pre jazyk Swift sú založené na práci pomocou asociatívnych polí, a keď chce programátor mať z výsledku štruktúru alebo objekt, musí sa o to postarať sám. Laravel ponúka prácu s databázou cez metódy objektu, a teda programátor sa nemusí starať o dekódovanie asociatívnych polí do požadovaných štruktúr. Na základe tejto inšpirácie som sa rozhodol implementovať knižnicu, ktorá bude ponúkať podobné správanie.

3.6.1 Porovnanie relačných a nerelačných databáz

V súčastnosti dominujú svetu dva typy databáz. Jedná sa o relačné databázy a ich alternatívu, nerelačé databázy. Hlavný rozdiel je v spôsobe, akým uchovávajú dáta. Kým relačné databázy sú štruktúrované a vyžadujú dodržiavanie určitej schémy, nerelačné databázy ukladajú svoje dáta v dokumentoch, ktoré môžu byť dynamicky menené – nemusí sa udržiavať žiadna schéma.

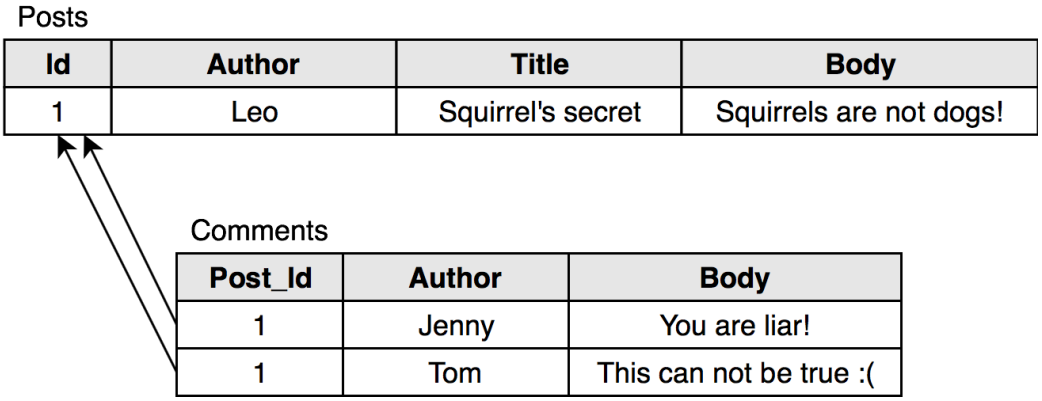


Obr. 3.5: UML diagram modelujúci vzťah medzi príspevkom a jeho komentármi. Tento diagram bude použitý v nasledujúcich príkladoch.

Relačné databázy – *SQL*

Jedná sa o štruktúrovanú databázu určenú k rozoznávaniu relácií medzi uloženými informáciami. Informácie sú organizované do jednej alebo viacerých relácií záznamov a atribútov s unikátným identifikátorom pre každý záznam. V praxi sa bežne používajú termíny tabuľka (relácia), riadok (záznam) a stĺpec (atribút), keďže sa tieto relácie zobrazujú v tabuľkovej forme. V nasledujúcom texte budú používané práve tieto výrazy, keďže je jednoduchšie pochopiť ich význam aj menej znalým čitateľom.

Každý stĺpec tabuľky musí mať predom určený typ a hodnoty v stĺpcoch by mali byť normalizované. Kvôli normalizácii musia existovať medzi stĺpcami a riadkami vzťahy, ktoré sú vytvárané pomocou cudzích kľúčov. Cudzí kľúč obsahuje hodnotu jednoznačného identifikátora (primárny kľúč) riadku. Systém integrity dát v relačných databázach sa stará o to, aby nebolo možné sa pomocou cudzieho kľúča odkazovať na neexistujúci riadok.



Obr. 3.6: Príklad uloženia príspevku s dvomi komentármi na základe UML diagramu na obrázku 3.5 v SQL databáze, pre uloženie do relačnej databázy sú potrebné dve tabuľky, pričom tabuľka *Comments* sa pomocou cudzieho kľúča odkazuje na riadok v tabuľke *Posts*.

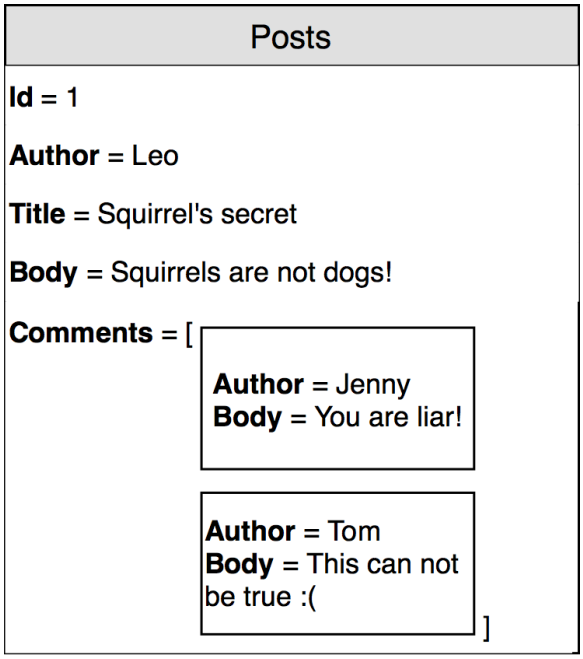
3.6.2 Nerelačné databázy – *NoSQL*

Tieto databázy boli navrhnuté, aby prekonali obmedzenia relačných databáz. Nejedná sa však o náhradu, ale o alternatívu. Na rozdiel od relačných databáz, nerelačné databázy nevyžadujú dodržiavanie konkrétnej schémy. Je možné pridať atribút ľubovoľného typu do ľubovoľnej časti databázy bez obmedzení. Tieto databázy bývajú na rozdiel od relačných nenormalizované a vznikajú v nich duplicity dát. Nerelačné databázy sa delia do štyroch skúpín.

- **Dokumentovo orientovaná databáza** – Dáta sú uložené vo formáte JSON ako dokumenty. Hodnoty môžu byť ľubovoľného typu.
- **Databázy typu kľúč-hodnota** – Databáza založená na asociatívnych poliach.
- **Stĺpcové databázy** – Dáta sú uložené v stĺpcoch na rozdiel od riadkov. Stĺpce môžu byť zoskupované a agregované podľa potrieb dotazov.
- **Grafové databázy** – Dáta sú reprezentované ako sieť alebo graf entít a vzťahov medzi nimi.

Porovnanie relačnej a nerelačnej databázy

Relačné databázy boli navrhnuté, aby obetovali rýchlosť požiadaviek pre vlastnosti, ako sú napríklad normalizácia. Nerelačné databázy zasa obetujú všetko ostatné pre rýchlosť. Kým v relačných databázach je pevná schéma, ktorá musí byť správne navrhnutá, keďže jej zmena môže byť problematická, nerelačné databázy sú vďaka svojej dynamickosti vhodné už od skorej fázy vývoja softwaru. Relačné databázy sú navrhované so zameraním na čo najlepší návrh, kým nerelačné sú navrhované podľa toho, aké dotazy bude potrebné obsluhovať [6].



Obr. 3.7: Príklad uloženia príspevku s dvomi komentármi na základe UML diagramu na obrázku 3.5 v dokumentovo orientovanej NoSQL databáze. Komentáre sú uložené v poli ako atribút v príspevku.

3.6.3 Implementácia abstrakcie nad databázou

Pred začiatkom implementácie som si ujasnil, čo by malo implementované riešenie spĺňať. Moja predstava požadovaných vlastností bola, že programátor si vytvorí vlastnú triedu, ktorá bude dediť z predimplementovanej triedy alebo v nej implementuje určitý protokol a vďaka tomu bude môcť pracovať s triedou ako s abstrakciou nad SQL tabuľkou. Ďalšia podmienka bola, že ak to bude protokol, musí mať maximálne 5 požiadaviek na implementáciu nejakej metódy alebo atribútu, pričom cieľom bolo nutnosť implementovať iba premennú reprezentujúcu identifikátor objektu v databáze a nič viac.

Prvé pokusy o implementáciu knižnice boli určené pre prácu s relačnou databázou MySQL. V tomto návrhu bolo treba vyriešiť, ako pracovať so vzťahmi medzi tabuľkami, ako odvodiť dátové typy v stĺpcoch tabuliek a ako z triedy získať názov tabuľky. Pomocou zrkadlenia objektov sa mi podarilo získať meno triedy, podľa ktorého bola potom pomenovaná tabuľka a mená a typy premenných. Vzťahy medzi tabuľkami bolo pôvodne potrebné explicitne určiť. Tento spôsob nespĺňal moje očakávania, keďže programátor musel explicitne zadať všetky vzťahy a aj ich kardinalitu. Pomocou zrkadlenia sa mi podarilo odstrániť nutnosť explicitne zadať vzťahy, ale bolo možné ich odvodiť podľa typu premennej. Pokiaľ premenná bola pole ďalších objektov, jednalo sa o 1-N väzbu, typ optional znamenal 1-0...1 a premenná, ktorej typ bol objekt, reprezentoval väzbu 1-1. Táto implementácia spôsobila problémy, kedy po vyhľadaní riadku v tabuľke sa vyhľadávali aj všetky riadky iných tabuliek, ktoré boli vo vzťahu s týmto riadkom. Riešením bolo označiť každú premennú s väzbou ako optional a vytvoriť vyhľadávanie, ktoré po vyhľadaní riadku nezačne vyhľadávať ďalšie riadky, ale nastaví vo výslednom objekte ich hodnotu na `nil` a pomocou inej metódy bude možné ich dohľadať. Tento koncept sa neosvedčil, pretože bolo nutné vždy kontrolovať, či hodnota prenenej je `nil` alebo nie. Ďalším riešením bolo označiť atribúty

objektu ako lazy a pridať mu ako inicializáciu closure, ktorá bude pracovať s databázou. Toto nespĺňalo požiadavku, že užívateľ nemusí veľa doimplementovať, keď chce, aby bol daný objekt uložiteľný v databáze.

Po neúspechoch s obalením relačnej databázy som sa na odporúčanie profesora Herouta pokúsil vytvoriť abstrakciu s požadovanými vlastnosťami nad NoSQL databázou. Pri relačných databázach boli najväčší problém relácie. NoSQL databáza je nerelačná databáza, takže najväčší problém odpadol. Mnou implementované riešenie využíva existujúcu knižnicu MongoKitten[10], ktorá ponúka prácu s dokumentovo orientovanou databázou *MongoDB* na základe asociatívnych polí. Bolo nutné vytvoriť spôsob, ako z asociatívneho poľa vytvoriť štruktúru a naopak. Riešením je serializácia objektu pomocou protokolu `Codable` a využitie zrkadlenia objektov. Výsledkom je protokol `Model`, ktorý vynucuje implementáciu jednej premennej (`_id`) a typom, ktoré ho implementujú dáva jednoduchý prístup pre prácu s databázou.

Pri práci s databázou je potrebné sa na konkrétnu databázu pripojiť. Knižnica ponúka tri spôsoby, ako sa pripojiť k databáze. Pripojenie bez užívatelského mena a hesla, s menom a heslom a pomocou URL adresy obsahujúcej všetky potrebné informácie pre pripojenie. Tretí spomenutý spôsob je vhodný pre pripojenie k databáze na základe premennej z prostredia.

3.6.4 Protokol `Model`

Aby sa stal nejaký typ uložiteľný v databáze, je potrebné, aby implementoval protokol `Model` a ním vyžadujúci protokol `Codable`. Požiadavky protokolu `Codable` sú vo väčšine prípadov automaticky generované počas kompilácie a protokol `Model` vyžaduje implementovanie iba jednej premennej – `_id`. Táto premenná je typu `ObjectId?` [11], ktorú treba inicializovať na hodnotu `nil`. Premenná nie je určená k priamej práci programátora a nemal by ju meniť (premmenná má prefix _ (podčiarkovník), ktorý symbolizuje fakt, aby sa ju programátor nesnažil zmeniť sám). Ak je identifikátor hodnota `nil` a zavolá sa metóda `save()`, slúžiaca k uloženiu objektu do databázy, objekt sa uloží a premenná `_id` sa nastaví na identifikátor objektu z databázy. Ak sa metóda `save()` zavolá a premmená `_id` má hodnotu identifikátora, objekt v databáze sa aktualizuje. Pri mazaní pomocou metódy `remove()` sa premenná `_id` nastaví na hodnotu `nil`.

Trieda implementujúca tento protokol predstavuje abstrakciu nad kolekciou v databáze, ktorej meno je odvodené od mena triedy. Všetky veľké písmená triedy sú prevedené do malých písmen a na koniec je pridané písmeno *s*[12]. Konkrétna inštancia triedy je abstrakciou dokumentu v databáze.

3.6.5 Vyhľadávanie v databáze

Vyhľadávať objekty v databáze je možné pomocou metód `find` a `findOne`, ktoré podporujú projekciu. Týmto metódam je možné predať ako parameter filter s podmienkami, ktoré musia spĺňať nájdené objekty, spôsob usporiadania a veľkosť offsetu (metóda `find` prijíma aj maximálny počet výsledkov). Pri vytváraní filtra sa používa typ `Query` z knižnice *MongoKitten*, ktorý preťažením operátorov zlepšuje čitateľnosť kódu[13]. Pri vyhľadávaní sa

[10] http://mongokitten.openkitten.org

[11] Typ `Optional` s asociovanou hodnotou typu `ObjectId`.

[12] Ak sa trieda volá *Car*, kolekcia bude mať názov *cars*.

[13] Napríklad `"name" == "Bob"` sa nevyhodnotí ako false, ale ako filter s podmienkou, že atribút `name` sa musí rovnať reťazcu `"Bob"`.

volá funkcia knižnice *MongoKitten*, ktorá vráti výsledok ako asociatívne pole. Toto pole sa pomocou typu `JSONDecoder` dekóduje a inicializujú sa požadované objekty (`find`: pole objektov, `findOne`: `Optional` s asociovanou hodnotou výsledného objektu).

Aby nebolo potrebné sťahovať z databázy celé objekty, keď je potrebný len nejaký atribút, bola implementovaná metódam `find` a `findOne` možnosť projekcie. Na rozdiel od ostatných riešení, ktoré projekciu riešia pomocou poľa atribútov, moje riešenie implementuje projekciu na základe typu odvodzujúceho protokol `Projectable` a jeho atribútov. Kvôli možnosti zrkadliť tento typ je protokolom vynútená možnosť inicializovať typ bez argumentov. Pri tvorbe projekcie sa pomocou zrkadlenia získajú názvy atribútov a z nich sa vytvorí projekcia. Aby sa ušetril čas, používa sa vyrovnávacia pamäť pre rýchly prístup už vytvorených projekcií. Z výsledkov vyhľadávania je inicializovaný výsledný typ podobne, ako pri vyhľadávaní bez projekcie.

```
1  struct User: Model {
2      var _id: ObjectId? = nil
3      var username: String
4      var age: Int
5      var images: [Image]
6  }
7
8  // Finds all users with age equal or higher as 18
9  let adults = try User.find("age" >= 18)
10
11 struct UsernameAge: Projectable {
12     let username: String
13     let age: Int
14
15     init() {
16         username = ""
17         age = 0
18     }
19 }
20
21 // Projection of all users with age equal or higher as 18
22 let adultsProjection: [UsernameAge] = try User.find("age" >= 18)
```

Výpis 3.7: Príklad vyhľadávania užívateľov s vekom aspoň osemnásť bez a s projekciou. Rozdiel medzi vyhľadávaním s projekciou je iba nutnosť definovania výsledného typu.

Knižnica dáva možnosť vyhľadať rozličné hodnoty pre daný atribút pomocou metódy `distinct`, ktorej výstupom je pole určeného typu. Výsledok je potom kotrolovaný na tento typ a sú zobrazené iba hodnoty s požadovaným typom. Pre tento účel je možné použiť iba primitívne typy.

3.7 Implementovanie podporujúcich knižníc

3.7.1 Zaobalenie formátu JSON

Pre prácu s formátom JSON bola navrhnutá knižnica `SquirrelJSON`, ktorá dáva možnosť serializovať aj typy, ktoré neimplementujú protokol `Codable` ani žiadny podobný. Seriali-

zácia sa deje najmä vďaka zrkadleniu typov, pomocou ktorej je možné získať typ, hodnotu a aj meno premennej.

V knižnici je implementovaná štruktúra JSON, ktorá ponúka bezpečnú prácu s hodnotami. Pre každý primitívny typ, pole a asociatívne pole ponúka niekoľko dvojíc dynamicky vyhodnocovaných atribútov. Jedna skupina sú atribúty slúžiace k získaniu hodnoty konkrétneho typu. Pokiaľ nie je možné požadovanú hodnotu reprezentovať požadovaným typom, vráti sa hodnota `nil`. Druhá skupina atribútov má postfix *Value* a slúži k získaniu hodnoty nejakého typu. Pokiaľ nie je možné reprezentovať hodnotu požadovaným typom, vráti sa namiesto hodnoty `nil` predvolená hodnota. Pre číselné typy je predvolená hodnota nula, pre reťazec, prázdny reťazec a pre polia sa vráti ich prázdna varianta. Štruktúru `JSON` je možné, vďaka implementácii protokolov `ExpressibleBy*`, inicializovať literálmi.

3.7.2 Preimplementovanie knižnice *Cache*

Pre zvýšenie rýchlosti spracovania častých operácií bolo potrebné implementovať vyrovnávaciu pamäť. Jedno z najlepších riešení, aké sa mi podarilo nájsť, je knižnica *Cache*[14], ktorá v dobe vývoja nepodporovala kompiláciu pre platformu Linux. Z tohto dôvodu som pomocou operácie *fork* vytvoril vlastnú kópiu repozitára a upravil som knižnicu tak, aby bolo možné ju plne využívať aj pod operačným systémom Linux. V knižnici bolo treba hlavne vylúčiť volania funkcií, ktoré nie sú plne implementované v Linuxe a nahradiť ich volaním funkcií jazyka C.

Knižnica vytvára vyrovnávaciu pamäť zloženú z prednej vrstvy a zadnej vrstvy. Predná vrstva drží dáta v pamäti RAM a zadná slúži k ukladaniu dát z RAM do súborov na disk pre neskoršie načítanie. Je možné nastaviť maximálnu veľkosť dát týchto vrstiev a aj dôležitosť dát v nich. Dáta s menšou dôležitosťou sú v prípade nedostatku pamäte mazané ako prvé.

3.7.3 Program pre podporu práce s frameworkom

Pre uľahčenie práce s frameworkom je vytvorený aj program pre podporu vývoja aplikácií (*toolbox*). Tento program je schopný spravovať programy napísané v jazyku Swift (spúšťanie, zastavovanie a sledovanie programov, ktoré boli spustené pomocou Toolboxu) a generovať súbory slúžiace ako šablóny pre nový projekt alebo pre súbory používané knižnicou *NutView*.

[14] https://github.com/hyperoslo/Cache

Kapitola 4

Dokumentácia, publikovanie frameworku, a výsledky

Pred samotným publikovaním bolo potrebné vyvinuté knižnice otestovať a vytvoriť dokumentáciu. Okrem unit testov bolo potrebné vyskúšať vytvoriť nejaké projekty, ktoré overia celkovú funkčnosť frameworku. Framework bol publikovaný na sociálnej sieti *Twitter*[1] a bola vytvorená žiadosť na pridanie odkazu na framework v github repozitári *TheList*[2], určenom pre publikovanie serverovo orientovaných knižníc napísaných v jazyku Swift.

4.1 Testovanie knižníc

Testovanie slúži k overeniu výsledného produktu a k odhaleniu chýb zanesených do kódu pri vývoji. Knižnice sú automaticky testované samostatne, ale prebehlo aj testovanie frameworku ako celku.

4.1.1 Unit testy

Pre všetky kritické časti mnou vyvinutých knižníc boli vytvorené testy. Celkovo sa jedná o vyše 100 testov obsahujúcich 660 assertov. Všetky knižnice používajú automatizované testovanie pomocou *CircleCI*[3], ktoré sa spustí automaticky pri každej zmene v repozitári. Testovanie je rozdelené do troch fáz. Prvá fáza kontroluje možnosť získať všetky požadované závislosti potrebné ku kompilácii. V druhej fáze prebieha kontrola, či je možné kód skompilovať. Knižnica sa skompiluje raz s optimalizáciami a raz bez nich. V poslednej fáze prebehnú unit testy. Testy sa spúšťajú sekvenčne a aj paralelne, aby sa otestovala funkčnosť knižnice v paralelnom prostredí. Výsledky týchto fáz sú priamo viditeľné v repozitári knižnice, takže každý navštevník repozitára vie skontrolovať, v akom stave je knižnica.

4.1.2 Použitie knižníc v projektoch

Aby sa overila celková funkčnosť riešenia ako celku, bolo vo vyvíjanom frameworku napísaných nieľko projektov. Niektoré boli použité ako tutorialy v dokumentácii a niektoré priamo vo väčších projektoch. Medzi významné patrí tvorba kompletného informačného systému do predmetu *Informační systémy – IIS* a použitie frameworku v bakalárskej práci

[1] https://twitter.com
[2] https://github.com/Awesome-Server-Side-Swift/TheList
[3] https://circleci.com

Martina Střálku. V jeho bakalárskej práci využíva hlavne knižnicu *Squirrel Connector* pre prácu s *MongoDB* a knižnicu *NutView* pre tvorbu dokumentov HTML. Projekty boli publikované pomocou cloudovej služby Heroku, ktorá umožňuje spustenie aplikácie v prostredí potrebnom pre jazyk Swift zadarmo.

4.2 Dokumentácia

Pri tvorbe dokumentácie bolo dôležité vyriešiť niekoľko problémov. Prvým bolo zakúpiť doménu a hosting, na ktorom bude dokumentácia umiestnená. Pre tento účel bola zakúpená doména *squirrel.codes* a využité platené služby Heroku. Potom bolo treba vyriešiť, ako bude dokumentácia vytvorená. Možnosti boli vytvoriť dokumentáciu na základe dokumentačných komentárov v kóde a tú sprístupniť na webovej stránke alebo ručne napísať dokumentačnú stránku. Vybral som si druhú variantu, nakoľko je prijateľnejšia pre užívateľov aj napriek tomu, že nemusí byť vždy aktuálna oproti dokumentačným reťazcom. Dokumentácia je písaná v značkovacom jazyku Markdown, nakoľko je tvorba dokumentov v ňom jednoduchá a intuitívna.

4.2.1 Jazyk Markdown

Tento jazyk bol navrhnutý v roku 2004. Zameriava sa na možnosť jednoducho písať textové dokumenty, ktoré je jednoduché aj čítať a prípadne konvertovať do HTML dokumentu. Tento jazyk sa rozšírene používa pre písanie takzvaných *readme* dokumentov. Jazyk sa rokmi overil ako veľmi jednoduchý a intuitívny a začal sa široko používať spoločnosťami ako *Github* a *Slack*.

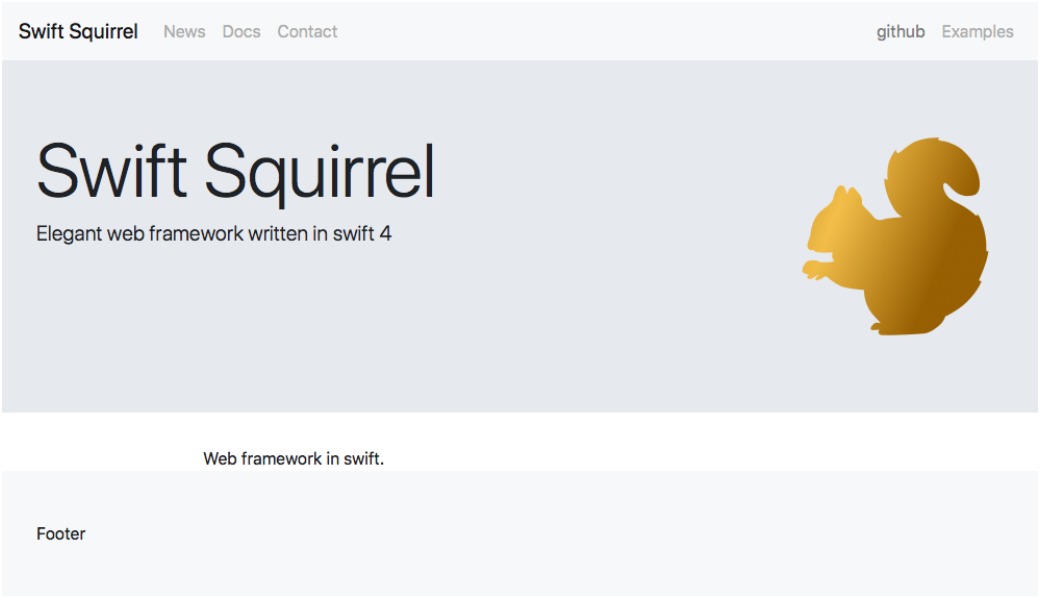
4.2.2 Návrhy dizajnu stránky

Pri návrhu dizajnu bolo treba určiť, čo bude na stránke viditeľné, aké podstránky bude ponúkať, či sa otvorí hneď dokumentácia a aké farby a logo sa použijú. V počiatočnom návrhu sa uvažovalo, že úvodná stránka a dokumentácia budú oddelené. Zvolila sa jemne sivá farba s bielou ako hlavné farby pozadia a veverička s hnedým gradientom ako logo (obrázok 4.1). Tento dizajn nebol na základe menšej ankety zvolený ako graficky „pekný“ a z tohto dôvodu bol oslovený Adam Valek[4], ktorý prerobil grafický návrh. Návrh sa týkal iba dokumentačnej stránky (obrázok 4.2). V návrhu sa vyskytoval gradient, ktorý spôsoboval malú čitateľnosť v menu a kontrast medzi farbou textu a pozadia obsahu nebol dostatočne veľký. V ďalších návrhoch (obrázok 4.4) sa podľa ankety zlepšila čitateľnosť textu a nové návrhy boli prijateľnejšie ako pôvodné. Počas dizajnovania stránky bolo dizajnované aj logo (obrázok 4.3 vpravo), ktoré sa podľa ankety tiež zlepšilo.

4.2.3 Tvorba dokumentácie

Keďže dokumentácia bola písaná v jazyku Markdown, bolo potrebné vytvoriť nástroj, ktorý by vedel dokumentáciu preložiť do HTML dokumentu. Pre tento účel bola vytvorená šablóna pomocou knižnice *NutView*. Dokumentácia je umiestnená vo verejnom repozitári na stránke github[5]. Hocikto môže spraviť fork repozitára, upraviť chyby v texte alebo pridať

[4] https://dribbble.com/adamvalek
[5] https://github.com/Swift-Squirrel/Docs



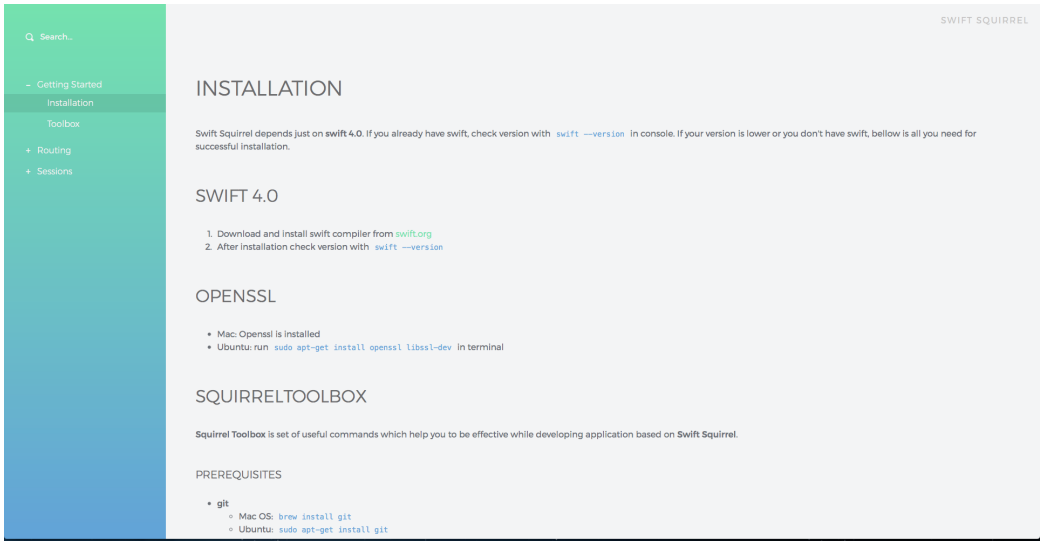
Obr. 4.1: Prvý grafický návrh pre webovú stránku.

nové dokumentačné texty a požiadať o spojenie s pôvodným repozitárom. Aby bola dokumentácia vždy synchronizovaná s dokumentáciou v repozitári, bolo nastavené automatické upozornenie pri každej zmene v repozitári. Toto upozornenie spôsobí stiahnutie novej verzie, pregenerovanie dokumentácie a aktualizovanie webovej stránky. Pregenerovanie dokumentácie je možné jedenkrát v priebehu piatich minút. Pri pokuse o častejšie pregenrovanie sa vráti odpoveď *202 Accepted*, ktorá symbolizuje, že server zaznamenal požiadavku, ale vybaví ju až po uplynutí piatich minút.

Pre generovanie dokumentácie je dôležitý súbor *pages.yml*, ktorý obsahuje poradie a popis podstránok. Obsahuje dve hlavné časti: prvá je pole sekcií a druhá pole priečinkov so súbormi, ako sú obrázky a podobné zdroje. Každá sekcia má informácie o názve, priečinku, v ktorom sú dokumentačné súbory a pole podstránok s názvami stránky a príslušnom dokumentačnom súbore.

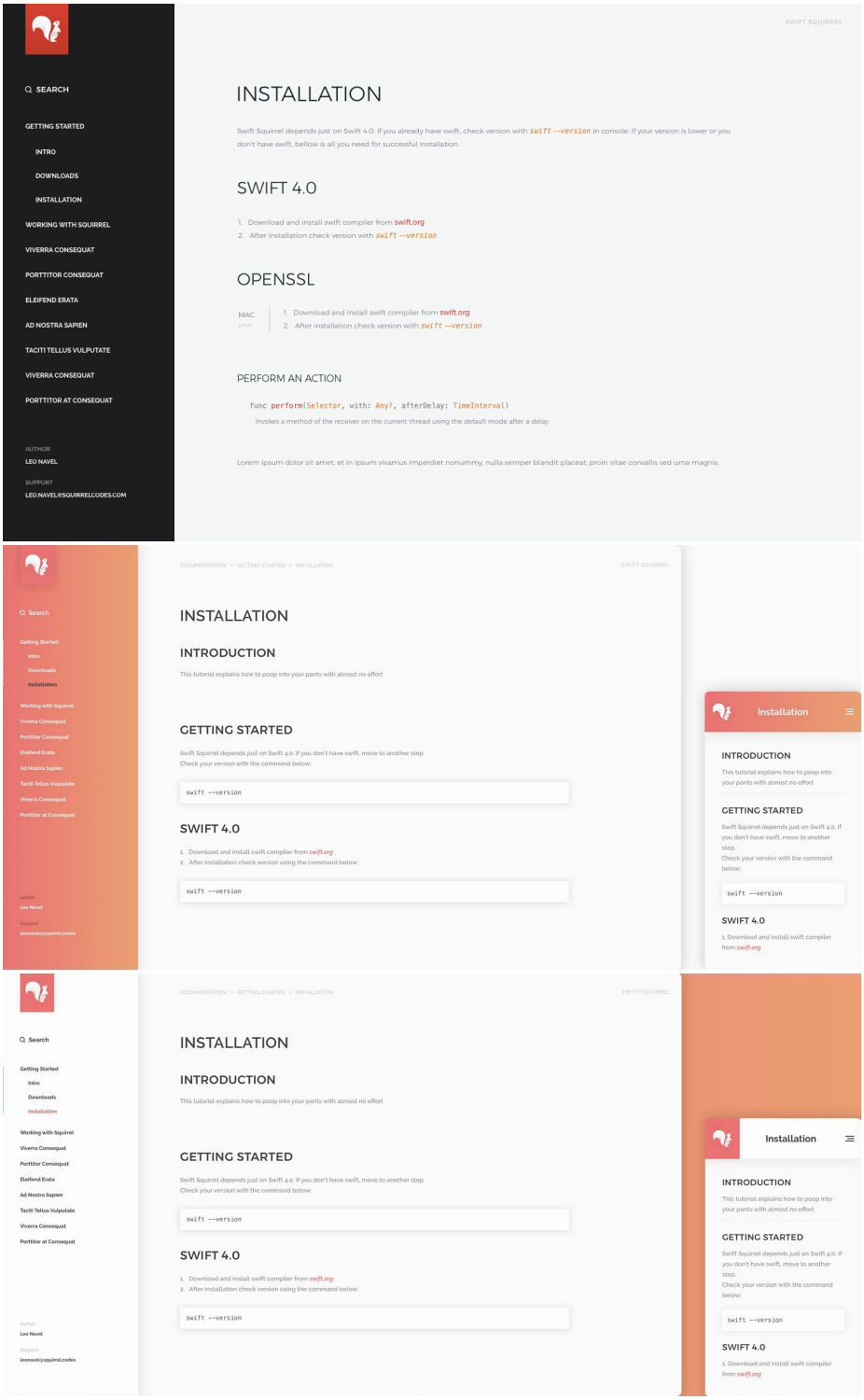
4.3 Výsledky frameworku

Počas vývoja som vo frameworku napísal niekoľko webových aplikácii a našiel som ľudí ochotných tento framework vyskúšať. Na základe ich spätnej odozvy je možné odhadnúť, že moje riešenie sa dá považovať za framework vhodný pre tvorbu týchto aplikácií. Dokumentačné texty sa dolaďovali takisto podľa spätnej väzby tak, aby bolo jasné, čo sa snaží konkrétna dokumentačná stránka povedať čitateľovi. Na github-ovských stránkach projektu je v čase písania tejto práce dokopy 16 hviezdičiek.



Obr. 4.2: Prvý grafický návrh dokumentácie vytvorený Adamom Valekom.

Obr. 4.3: Vývoj loga frameworku, **vľavo:** prvá verzia loga, **v strede:** logo k zeleno-modrému návrhu, **vpravo:** aktuálne logo.



Obr. 4.4: Ďalšie grafické návrhy dokumentačnej stránky použité v ankete. Ako výsledný návrh sa použil posledný (spodný) návrh na základe ankety.

Kapitola 5

Záver

V práci sa podarilo vytvoriť webový framework vhodný pre vývoj webových aplikácií a knižnice podporujúce jeho funkčnosť. Framework priamo podporuje prácu s databázou MongoDB a poskytuje možnosť písania HTML šablón pomocou príkazov so syntaxou podobnou jazyku Swift. Framework je plne funkčný pod operačnými systémami Mac OS a Ubuntu. Pre framework a implementované knižnice je vytvorená dokumentácia aj s tutorialom. Po implementovaní základných vlastností bol publikovaný na sociálnej sieti Twitter a na dokumentačnú stránku bola pridaná služba *Google analytics*, ktorá sleduje návštevnosť stánky. Framework bol úspešne využitý v bakalárskej práci Martinom Střálkom a bol v ňom úspešne implementovaný informačný systém ako projekt do predmetu *Informační systémy*.

Do frameworku by sa v budúcnosti dala pridať podpora pre multijazyčnosť pomocou databázy alebo súborov vo formáte JSON reprezentujúcich konkrétny jazyk, podpora pre architektúru MVC alebo generovanie HTML formulárov s možnosťou pridávať pravidlá pre kontrolu vstupu v HTML formulári pomocou jazyka JavaScript. Ďalej je práca pripravená aj pre podporu automatickej aktualizácie objektov pomocou požiadavky s HTTP metódou PATCH a telom vo formáte JSON Patch[1]. Pomocou paralelizmu by bolo možné zvýšiť rýchlosť interpretácie šablónovacích súborov, kde by každý odkazovaný súbor bol interpretovaný samostatne vo vlastnom vlákne. Aktuálne riešenie zatiaľ nepodporuje protokol HTTPS, na ktorého podpore pracujem v čase písania tejto práce.

[1] https://tools.ietf.org/html/rfc6902

Literatúra

[1] Alicanbatur: *Protocol Oriented Programming in Swift. Medium*, 10 2017.

[2] Apple: *About Swift*, 2018, official documentation.
URL https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/

[3] Apple: *Enumerations*. 2018.
URL https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Enumerations.html

[4] Eidhof, C.: *Advanced Swift*. Amazon, 2016, ISBN 9781523831715.

[5] Federighi, C.: *WWDC 2014 - Session 101 - macOS, iOS*, 2014, wWDC 2014 Keynote.
URL https://developer.apple.com/videos/play/wwdc2014/101/

[6] Holubová, I.; Kosek, J.; Minařík, K.; aj.: *Big Data a NoSQL databáze*. Praha: Grada, 2015, ISBN 978-80-247-5466-6.

[7] Miller, P. M.: *TCP/IP: the ultimate protocol guide vol. 2*. Boca Raton: BrownWalker Press, 2009, ISBN 978-1-59942-493-4.

[8] Neuburg, M.: *IOS 11 programming fundamentals with Swift: Swift, Xcode and Cocoa basic. Fourth edition*. Sebastopolo, CA: O Reilly, 2018, ISBN 978-1-491-99931-8.

[9] Rouse, M.: *Server. WhatIs*, 1 2018.

[10] Roy T. Fielding and James Gettys and Jeffrey C. Mogul and Henrik Frystyk Nielsen and Larry Masinter and Paul J. Leach and Tim Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616, RFC Editor, June 1999, http://www.rfc-editor.org/rfc/rfc2616.txt.
URL http://www.rfc-editor.org/rfc/rfc2616.txt

[11] Shiflett, C.: *Storing Sessions in a Database. PHP Magazine*, 12 2004.

Príloha A

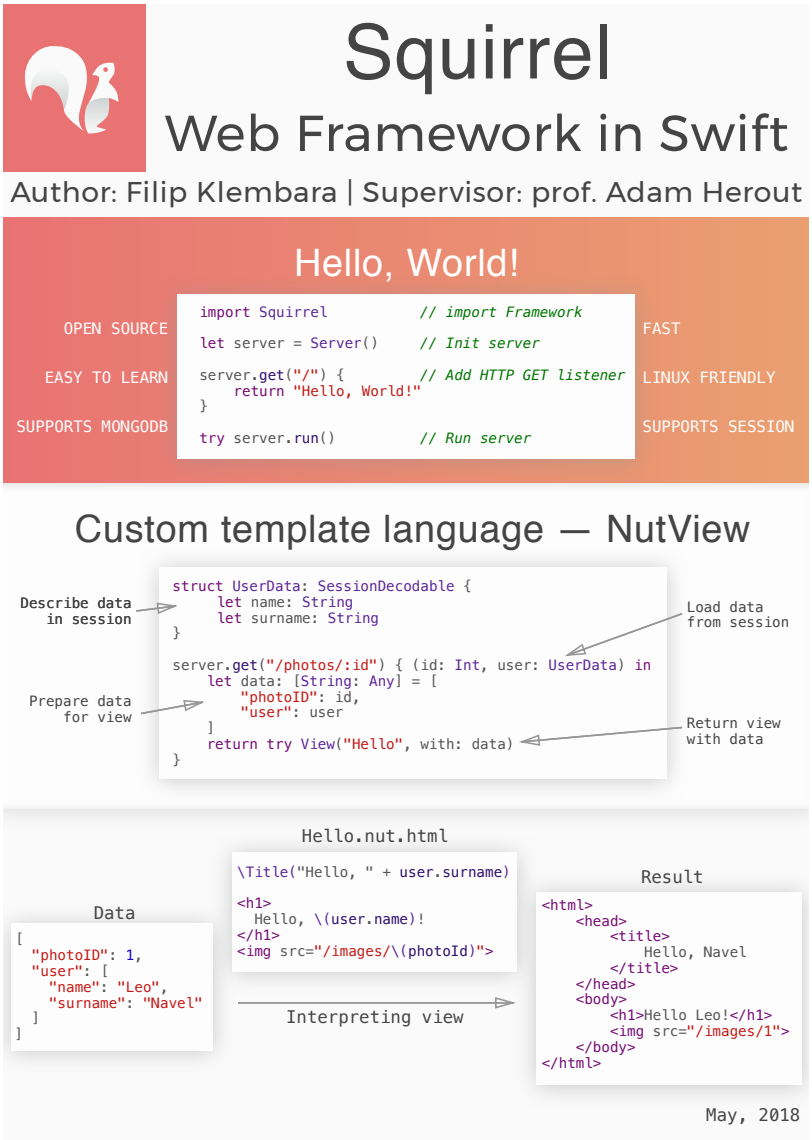
Obsah priloženého CD

Priložené CD obsahuje následúce adresáre a súbory:

- *docs/* - priečinok obsahujúci dokumentáciu
- *libraries/* - priečinok obsahujúci vytvorené knižnice
- *LICENSE* - súbor obsahujúci licenciu projektu
- *poster.pdf* - plagát prezentujúci framework
- *presentation.mp4* - videoprezenácia práce
- *SwiftSquirrel.pdf* - textová časť práce
- *tex/* - priečinok obsahujúci zdrojové texty textovej časti práce
- *README.md* - popis obsahu CD vo formáte markdown

Príloha B

Plagát



Obr. B.1: Plagát pre prezentovanie frameworku.