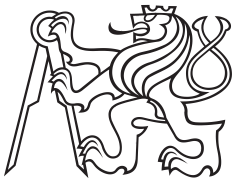
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Web aplikační framework pro Micropython

Martin Mašata

Vedoucí: Ing. Jiří Šebek
Oponent: Ing. Matěj Klíma
Studijní program: Softwarové inženýrství a technologie
Květen 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mašata** Jméno: **Martin** Osobní číslo: **474595**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Web aplikační framework pro Micropython

Název bakalářské práce anglicky:

Web application framework for Micropython

Pokyny pro vypracování:

Motivace:
V ekosystému micropython existuje omezené množství těchto typů knihoven. A žádná z nich nemá dostatečnou sadu funkcí, jsou špatně či téměř vůbec zdokumentované, z toho důvodu jsou nesnadno použitelné.
• picoweb:
o špatná dokumentace
o nesnadná implementace route
▪ nerozlišuje metody (GET, POST, PUT, DELETE)
▪ složité servírování statických souborů
▪ v requestech neřeší content-type: application/json a application/xml
o nepodporuje middleware (pre akce před zpracováním requestu pro konkrétní sadu route)
• microdot:
o žádná dokumentace
Mimo tyto dvě knihovny žádné jiné nejsou, proto zde je velký prostor k implementaci vlastního řešení, které bude lepší, bude obsahovat více funkcionalit a bude dobře zdokumentované.
Cíl práce:
Vytvořit novou knihovnu obsahující web aplikační framework, která bude dobře zdokumentovaná (standardizovaná dokumentace v pythonu a README.md), s dobrými examples a snadno použitelná (inspirace známými existujícími frameworky viz. Express, Spring, Flask, ...).
Požadavky na funkcionalitu:
• route (endpointy)
o seskupování route do controlleru
o rozlišování metod
o path params
o query params
• definice middleware(Express) funkcí
o json body
o form data
• Konfigurace
o pomoci konstruktoru
o pomoci envfile
• Logování
Výsledná práce bude obsahovat analýzu, návrh, implementaci frameworku a její testování.
Kompatibilní zařízení
Z důvodu velkého množství mikročipů na trhu, bude výsledná aplikace testována pouze na ESP8266.
Z tohoto důvodu je zaručena kompatibilita knihovny pouze pro tento zmiňovaný mikročip.

Seznam doporučené literatury:

Micropython [online]. [cit. 2021-01-11]. Dostupné z: https://micropython.org/
Flask [online]. [cit. 2021-01-11]. Dostupné z: https://flask.palletsprojects.com/en/1.1.x/
Spring Framework [online]. [cit. 2021-01-11]. Dostupné z: https://spring.io/
ExpressJS [online]. [cit. 2021-01-11]. Dostupné z: https://expressjs.com/
GRIDLING, Gunther; WEISS, Bettina. Introduction to microcontrollers. Vienna University of Technology

Institute of Computer Engineering Embedded Computing Systems Group, 2007.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **18.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **19.02.2023**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Velice rád bych tímto poděkoval Ing. Jiřímu Šebkovi za cenné připomínky a za vedení této bakalářské práce. Také bych rád poděkoval Milanu Kormoutovi za uživatelské, potažmo vývojářské testování frameworku a za jeho zpětnou vazbu k práci, díky které byla výsledná práce vylepšována. V neposlední řadě bych rád poděkoval mé rodině a přítelkyni za obrovskou podporu, které se mi během mého studia dostávalo.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 18. května 2021

Abstrakt

Tato bakalářská práce se zabývá vytvořením nového web aplikačního frameworku v Micropythonu, který bude nabízet více možností než ty stávající. Dokument obsahuje analýzu, návrh, implementaci a testování aplikace. Analýza se skládá z rozboru požadavků a průzkumu silných a slabých stránek konkurenčních řešení. V návrhu a implementaci je dbán důraz na omezenou paměť v mikročipech. Výsledný framework je otestován vývojářem, který jej použil pro svou chytrou domácnost. Z jeho zpětné vazby jsou získány poznatky pro budoucí vylepšení frameworku.

Klíčová slova: micropython, WAF, mikročipy, IoT

Abstract

This bachelor's thesis deals with the creation of a new web application framework in Micropython, which will offer more options than the existing ones. The document contains analysis, design, implementation of application. The analysis consists of an analysis of requirements and a survey of the strengths and weaknesses of competing solutions. Emphasis is placed on limited memory in microchips in the design and implementation. The resulting frame is tested by a developer who used it for his smart home. His feedback provides insights into future improvements to the framework.

Keywords: micropython, WAF, microcontrollers, IoT

Title translation: Web application framework for Micropython

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

1.1 Motivace

V ekosystému micropythonu existuje omezené množství web aplikačních frameworků. Žádný z nich nemá dostatečnou sadu funkcí, jsou špatně, či téměř vůbec zdokumentované, z toho důvodu nejsou snadno použitelné. To je zároveň důsledkem toho, že komunita Micropythonu je malá. Proto potencionální nový framework, který toho zvládne daleko více, by mohl přilákat nové lidi do komunity Micropythonu, a tím zlepšit stávající situaci [5, 6, 7].

1.2 Cíle bakalářské práce

Cílem této bakalářské práce je zanalyzovat web aplikační frameworky, pravidla a omezení Micropythonu, a z toho následně navrhnout framework, který bude naimplementován a otestován. Hotový framework by měl umět více než jeho konkurenti v Micropythonu, měl by být intuitivní a příjemný k používání vývojáři. Zároveň musí být minimalistický, aby se obešel bez výkonostních problémů, proto je potřeba balancovat mezi množstvím funkcí a výkonem. Musí být dobře zdokumentován a obsahovat věcné ukázky kódu. Výsledný framework bude nahrán do repozitáře PyPi jako knihovna volně přístupná každému.

1.3 Struktura bakalářské práce

Kapitola **Analýza 2** bude nejprve zaměřená na stejné frameworky, ale z jiných programovacích jazyků. Poté se budeme v kapitole zabývat obdobnými frameworky právě z Micropythonu. Následně se bude analyzovat používaný mikročip a přímo programovací jazyk Micropython. Na závěr budou popsány metodické postupy při psaní knihoven a funkční požadavky frameworku.

V kapitole **Konfigurace 3** budou popsány postupy, jakým způsobem a jak bylo připravováno prostředí pro vývoj na počítači. Také bude uveden používaný mikročip a jeho příprava, aby byl schopný fungovat s Micropythonem.

V kapitole **Návrh 4** bude navržen interface pro uživatele, doprovázen úryvky kódu, jak by přibližně mohl interface vypadat. Následně bude navržena struktura celého frameworku. Podkapitoly se poté budou zaměřovat na různé části logiky. Každá část bude popsána a doprovázena UML diagramy.

Kapitola **Implementace 5** bude obsahovat popis využívaných knihoven, jaké byly použity návrhové vzory v kódu a také, jaké problémy se objevili při implementaci, a jakým způsobem byly vyřešeny.

Kapitola **Testování 6** bude obsahovat seznam nalezených chyb, uživatelské testy vyvojáře, který implementoval na našem frameworku mikročipy do chytré domácnosti a jednotkové testy.

Na **Závěr 7** budou srhnuty výsledky práce, zhodnocení a splnění cílů práce, a jak bude dále framework rozšiřován.

Kapitola 2

Analýza

Tato kapitola je zaměřena na průzkum jiných, podobných frameworků (z jiných programovacích jazyků). Následovat budou konkurenční řešení, tedy frameworky psané také přímo v Micropythonu a budou vypíchnuty jejích silné a slabé stránky. Dále bude v kapitole věnována část samotnému mikročipu, jeho vlastnostem a případným omezením, které by mohly komplikovat návrh a implementaci, jaké budou možnosti k vyřešení problému, a která budou ta nejlepší pro navrhované řešení. Na závěr se analýza bude zabývat doporučenými způsoby psaní knihoven v Micropythonu, jaká pravidla se musí dodržovat, aby byly zachovány best practices a výsledná knihovna byla co nejkvalitnější.

2.1 Známé webové frameworky

V této části budou porovnány dostupné webové frameworky pro různé programovací jazyky. Budou zde vypsány a znázorněny jejich funkcionality, které chceme v našem řešení také naimplementovat. Daná funkcionalita bude porovnána napříč frameworky a následně zvolena nejlepší možná implementace naší vlastní funkcionality. V potaz také musíme brát, že programovací jazyky se liší, a proto nemusí být dané řešení pro nás technicky možné.

Celkově se analýza bude skládat ze tří webových frameworků. Prvním z nich bude Spring framework pro Javu [8]. Druhým bude Express pro Node.js [9]. A posledním bude Flask pro Python [10].

2.1.1 Route

Spring využívá pro routování anotace. Ta nadefinuje endpoint k funkci, která je vykonána po přijmutí HTTP requestu. Přístup ke query parametrům je umožněn tak, že u funkce nadefinujeme proměnnou a před ní oanotujeme. Path parametry jsou uvedeny v cestě ve Stringu. Spring je rozpozná, protože jsou ohraničeni složenými závorkami. Endpointy jsou zhlukovány pod třídu, která je definována jako Controller [11, 12]. V Listingu 2.1 je ukázka z routování v Spring frameworku.

```
1   @RestController
2   public class SomeControllerClass {
3
4       @GetMapping(value="/path/anotherPath/id/{id}")
5       public Response foo(@PathVariable("id") Long id){
6           //Path parameters example
7           //Request mapping new way
8       }
9
10      @RequestMapping(value="/path" method=RequestMethod.GET)
11      public Response foo2(@RequestParam("a") String a){
12          //Query parameters example
13          //Request mapping old way
14      }
15  }
```

Listing 2.1: Ukázka routování ve Springu.

Express funguje tak, že je nejprve nutné si jej zavolat a uchovat ho v proměnné. Z proměnné jsou poté volány metody, kterými určujeme HTTP metodu a další nastavení pro endpoint. Cesty mohou být stavěny i přes regulární výrazy. Path parametry lze zakomponovat pomocí zadefinování v cestě s prefixem dvojtečky a názvem proměnné. Ke query parametrům lze přistoupit přes parametr funkce [13, 14]. V Listingu 2.2 je ukázka z routování v Expressu.

```
1   var express = require('express');
2   var app = express();
3
4   app.get('/id/:id', function (req, res) {
5       //example of path parameters
6   });
7
8   app.put('/aaa*b', function (req, res) {
9       //example of regular expression path
10      var myParam = req.query.myParam;
11      // query parameters
12  })
```

Listing 2.2: Ukázka routování v ExpressJs.

Flask používá stejně jako Spring anotace, ale v Pythonu se nazývají dekorátory. V dekorátoru poté přijímá parametr s cestou a parametr s HTTP metodou či více HTTP metodami. Path parametry jsou uvedeny v cestě, označeny oboustranně v lomenných závorkách. Query parametry jsou uchovány v proměnné funkce. [15]. V Listingu 2.3 je ukázka z routování v Flasku.

```
1   from flask import Flask
2   app = Flask(__name__)
3
4   @app.route('/path/id/<id>' , methods = ['GET', 'POST'])
5   def foo():
6       #example of path parameters
7
8   @app.route('/aa')
9   def foo2():
10      myParam = request.args['myParam']
11      #example of query parameters
```

Listing 2.3: Ukázka routování ve Flasku.

2.1.2 Middleware funkce

Výrazem middleware jsou myšleny nějaké funkce, které probíhají ihned po příchodu HTTP requestu a slouží jako určitá pravidla, filtry, která vyhodnocují zda má být HTTP request vůbec nadále zpracováván. Buďto filtry projdou a bude vykonáváno zpracování response, nebo ne a spojení bude ukončeno s vrácením HTTP response či bez ní (záleží na definovaném filtru) [16]. Tento výraz je v každém jazyce nazýván trochu jinak. Ve Spring frameworku v Javě se to nazývá filtry, u ExpressJs a Flasku je zachován název middleware.

Ve Springu se nejčastěji filtry tvoří vytvořením nové třídy, který implementuje interface Filter. Logika je poté umístěna do metody *doFilter()*. Je zde také ještě varianta přes Beany [17]. V Listingu 2.4 je ukázka z filtrů v Spring frameworku.

```
1   @Component
2   public class MyFilter implements Filter{
3
4       @override
5       public void doFilter(
6           ServletRequest request,
7           ServletResponse response,
8           FilterChain chain){
9           //zde bude logika filtru
10      }
11  }
```

Listing 2.4: Ukázka filtrů ve Springu.

Express je registruje metodou *use()*, která má tři parametry. Objekt requestu, objekt responsu a referenci na další middleware funkci v pořadí. V podstatě jsou u Expressu middleware funkce uspořádány ve spojovém seznamu. Middleware funkci lze nadefinovat pro veškeré HTTP requesty, nebo pro volání určitých endpointů dle cesty k ní. Middleware funkcemi pak lze také vyvolat okamžité vrácení HTTP response. Většinou se tak řeší chybné vstupy z requestu [18]. V Listingu 2.5 je ukázka z filtrů v Expressu.

```
1   var express = require('express')
2   var app = express()
3
4   app.use(function (req, res, next) {
5       //vykona logiku a zavola dalsi middleware funkci v
    poradi
6       next()
7   })
8
9   app.use('/user/:id', function (req, res, next) {
10      //middleware funkce omezena jen pro urcite endpointy
11      next()
12  })
```

Listing 2.5: Ukázka middlewarů v Expressu. Příklad kódu převzat a upraven z dokumentace Expressu [18].

Na závěr je zde popsán Flask, který na to jde také jiným způsobem, než ostatní frameworky. Je nutné vytvořit třídu, která bude definovat daný middleware. Jako parametr v konstruktoru musí být reference na instanci Flasku. Pro samotnou logiku middlewaru je potřeba nadefinovat metodu *___call___* se vstupním parametrem *environ* a *start_response*, a do ní poté vepsat pravidla filtrování [19]. V Listingu 2.6 je ukázka z filtrů v Flasku.

```
1   class middleware:
2       def __init__(self, app):
3           self.app = app
4
5       def __call__(self, environ, start_response):
6           #vykonavana logika pro middleware
7
8   app = Flask()
9   app.wsgi_app = middleware(app.wsgi_app)
```

Listing 2.6: Ukázka middlewarů ve Flasku.

2.1.3 Konfigurace

Ve Springu lze nastavit konfigurace dvěmi způsoby. První je přes anotace, kde si vytvoříme třídu a využijeme anotaci Configuration. Dále je pak potřeba si nadefinovat anotaci Bean u metod třídy, které zajistí přes IoC, že budou nadefinovány do Springu [20]. Druhou možností je konfigurace vkládat do souboru *application.properties* [21].

V Expressu (respektivě v Node.js) je pravděpodobněji nejčastější přístup pro konfiguraci modul z npm - dotenv. Ten je potřeba nainstalovat a poté vytvořit soubor s příponou *.env*. Do něj budou nadefinovány proměnné a jejich hodnoty. K těm pak lze přistupovat již v implementovaném kódu [22]. Stejně jako Spring, tak i Flask nabízí dvě možnosti. Přes definování parametrů v kódu či přístup přes nějaký soubor, je to téměř to samé [23].

Jak lze vidět ze zmiňovaných frameworků, tak všechny z nich používají stejné přístupy pokud pomineme drobné rozdíly. Proto se jeví jako správná cesta, aby náš framework nabízel dvě možnosti také. Přes parametry v dictionary a druhý způsob přes nadefinovaný soubor. Je již pak na uživateli, jaký způsob si zvolí, který mu bude příjemnější či vhodnější pro danou implementaci. V Listingu 2.7 je ukázka z konfigurace v Spring frameworku.

```
1  spring:
2      datasource:
3          url: jdbc:postgresql://localhost:5432/
       example_application
4          username: postgres
5          password: postgres
6
7      jpa:
8          show-sql: true
9          hibernate:
10             ddl-auto: update
11             naming-strategy: org.hibernate.cfg.
       ImprovedNamingStrategy
12         properties:
13             hibernate:
14                 dialect: org.hibernate.dialect.
       PostgreSQL82Dialect
15
16 app:
17   auth:
18     tokenSecret: 926D96C90030ED58429D2851AC2BDBBC
19     tokenExpirationMsec: 864000000
```

Listing 2.7: Ukázka konfigurace ve Spring Bootu v souboru *application.yml*.

2.2 Konkurenční řešení

Pro micropython existují v tuto chvíli celkem tři knihovny podobného charakteru. Pro analýzu byly vybrány tyto dva frameworky - picoweb [3] a microdot [24]. Tím třetím je noggin [4], ale nebude zde analyzován z několika důvodů:

- Téměř nikdo ho nepoužívá.
- Téměř žádné zmínky ani na fórech micropythonu.
- Přes 3 roky nebyl aktualizován.

V tabulce 2.1 jsou zobrazeny statistiky ostatních frameworků na gitu.

Knihovna	Stars	Forks
picoweb	349	75
microdot	117	13
noggin	13	4

Tabulka 2.1: V tabulce jsou uvedeny statistiky z git repozitářů jednotlivých knihoven. Podle toho lze přibližně určit jejich používanost [3, 4].

2.2.1 Picoweb

Začneme s picowebem [3], ten je nejpoužívanějším webovým frameworkem v Micropythonu. Obsahuje dokumentaci, která popisuje knihovnu do detailů, ale chybí tam zdokumentované funkce, jaké parametry přijímají a co dělají. To je sice nevýhoda, ale mírně to kompenzuje existence vzorových ukázek kódu, ze kterých si lze některé funkcionality odvodit a domyslet.

Mezi jeho výhody poté patří standartní používání logování, které je nastavitelné. Knihovna je založená na uasyncio (dokáže zpracovávat více requestů naráz), to je další výhoda. Silnou stránkou knihovny jsou mimo jiné dekorátory, které usnadňují uživatelům implementaci.

Nevýhod má několik, hlavní z nich je absence parametru metod v dekorátoru. Nelze nadefinovat stejný endpoint, který se chová různě při jiných metodách, musí se to řešit uvnitř jediné funkce pomocí jejího rozvětvení. To není ideální a kód poté může ztrácet na přehlednosti. Kromě toho také nepodporuje middleware. Tím máme na mysli sadu funkcí, které jsou vykonané ještě před samotným zpracováním HTTP requestu. Na to se váže i absence validace headerů (například Content-type).

2.2.2 Microdot

V Micropythonu existuje po picowebu ještě druhá možná varianta - microdot [24]. Ten je jistými aspekty napřed oproti picowebu, ale také má svá úskalí. Mezi jeho výhody patří rozdělování metod přímo v dekorátoru přes parametr. Umí také logovat informace v knihovně. Dokáže vracet soubory (obrázky, HTML soubory, CSS soubory, ...), statusy a headers dle parametrů.

Velkou nevýhodou, která má negativní dopad na jeho používání je zcela chybějící dokumentace. Kdyby obsahoval dokumentaci byl by zcela jistě konkurencí picowebu. Knihovna není implementována asynchronně, při více požadavcích bude vznikat fronta.

2.2.3 Shrnutí

Pro přehlednější porovnání je zde uvedena ke srovnání tabulka, co jednotlivé knihovny nabízejí a v čem se liší mezi sebou. Naše výsledná implementace by měla splňovat všechny zmiňované body v tabulce. V tabulce 2.2 je graficky znázorněno, co který framework umí, a co ne.

Funkcionalita/Framework	**picoweb**	**microdot**
Dokumentace	✓	×
Ukázky kódů	✓	✓
Metody v dekorátoru	×	✓
Asynchronnost	✓	×
Logování	✓	✓
Middleware	×	×
Query params	×	×
Path params	×	×

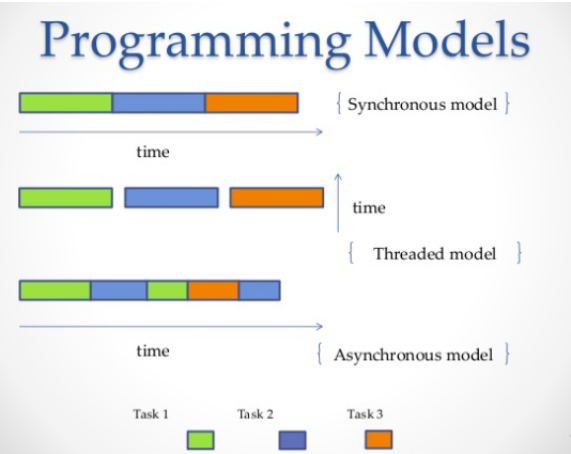
Tabulka 2.2: Shrnutí funkcionalit jednotlivých knihoven a porovnání mezi sebou.

2.3 Omezení mikročipu

Definovaná omezení bude potřeba brát v potaz při návrhu, protože mají zásadní roli. První hlavní omezení je velikost flash paměti. Ta v našem případě činí 4MB. Výsledný produkt musí být ovšem mnohokrát menší, protože musíme počítat s tím, že někdo bude stavět na naší knihovně svoji chytrou domácnost a bude potřebovat většinu paměti svého mikročipu pro svoji implementaci. Řešením tohoto problému je klást vysoký důraz na optimalizaci při návrhu. Dále do knihovny dávat jen důležité věci a psát čistý kód.

Druhým velkým problémem je přítomnost pouze jednoho jádra. Z tohoto důvodu může být problém, když bude přicházet na webový server více dotazů v jeden moment. Jako jediné vhodné řešení je použití asynchronního přístupu. Pro tyto účely existuje v Micropythonu knihovna uasyncio [25, 26], která je podmnožinou CPython knihovny asyncio [27]. Tuto knihovnu budeme využívat i my v naší implementaci, aby webový server byl schopný obsloužit co nejvíce akcí.

Přístup asyncia je takový, že máme takzvanou "event loop"[28], ve které jsou námi definované funkce (budeme definovat generátory - příkaz yield či await). Pokud funkce v danou chvíli nic nevykonává a pouze čeká na nějaký výsledek (například HTTP response, nebo odpověď z databáze), pak asyncio přejde na jinou funkci a bude vykonávat jí. Tímto přístupem se omezí čekání a hlavní proces (v našem případě i jediný) bude neustále něco vykonávat. Na obrázku 2.1 je znázorněn rozdíl mezi synchronním, asynchronním a vícevláknovým programováním.

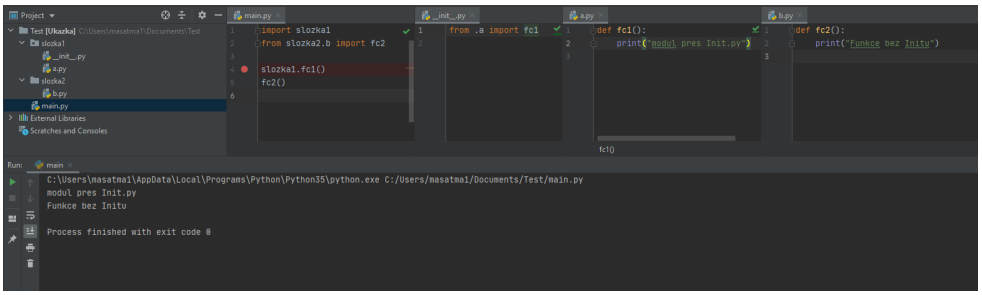


Obrázek 2.1: Rozdíl mezi synchronním, asynchronním a vícevláknovým programováním [1].

2.4 Metodika psaní knihoven v Micropythonu

Před návrhem je potřeba si vydefinovat pravidla, podle kterých se budeme řídit při návrhu knihovny. Tato pravidla budou respektována a dodržována, abychom se drželi best practices.

Každý balíček bude obsahovat *___init___.py* soubor. Tento soubor v sobě nebude obsahovat žádnou logiku, ale bude importovat ze všech ostatních modulů ve stejném balíčku, kde se nachází. Děláme to z důvodu přehlednosti při vývoji, a také proto, aby výsledný import byl pro uživatele knihovny jednodušší k používání. Modulem je myšlen soubor s *.py* příponou [29]. Na obrázku 2.2 je ukázán rozdíl použití a nepoužití importovacího souboru.



Obrázek 2.2: Ukázka rozdílu mezi používáním *___init___.py* v balíčku a bez něj.

Balíček bude rozdělen na moduly podle logiky, aby každý vykonával svou unikatní funkci. Balíček bude také obsahovat soubor *setup.py*. V tomto souboru jsou definovány informace jako třeba jméno balíčku, verze, autora a další. Slouží pro konfiguraci celé knihovny a bude potřebný pro nahrání do balíčkovacího systému PyPi [30, 31].

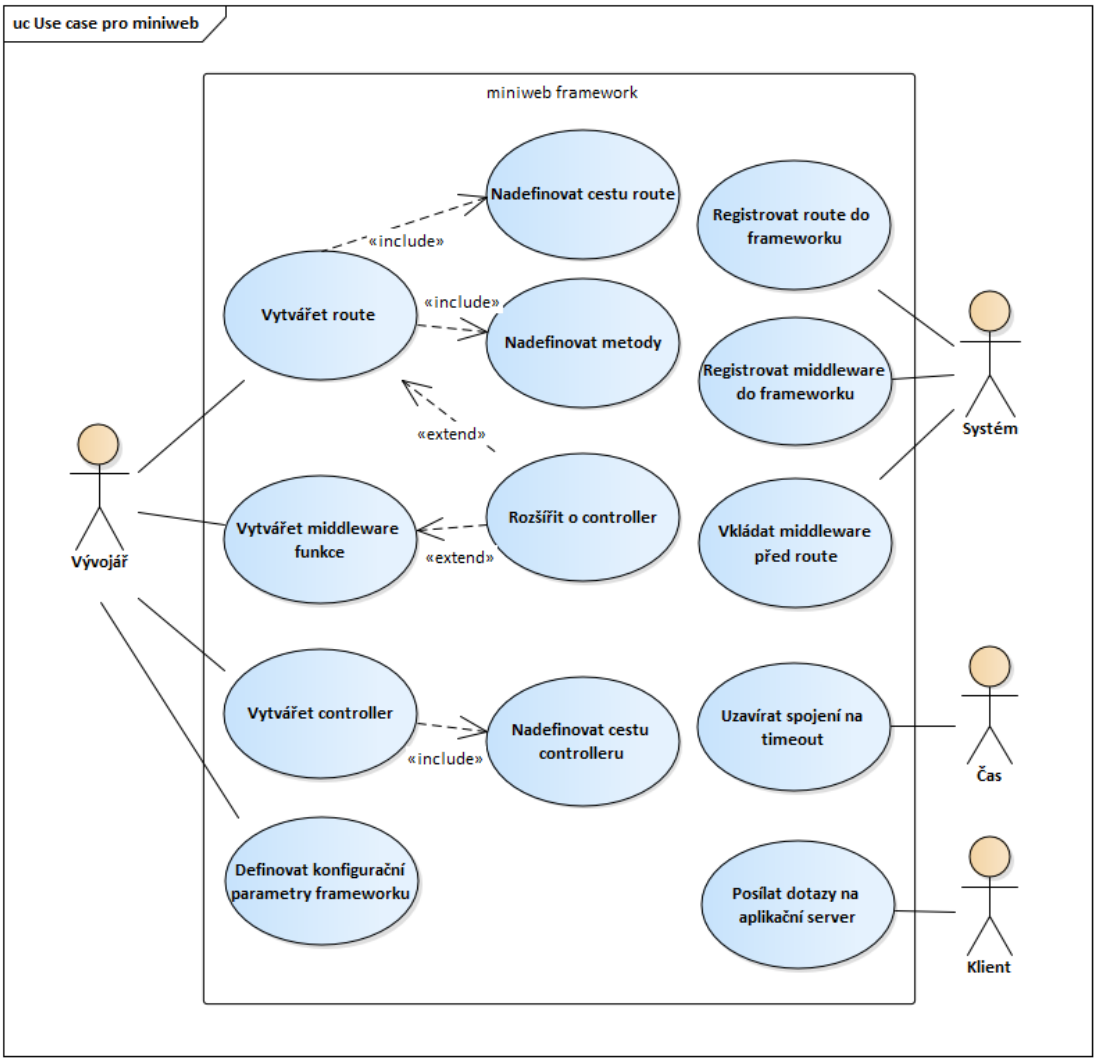
2.5 Funkční požadavky

Byly vydefinovány tyto požadavky, díky kterým bude mít výsledný naimplementovaný framework více funkcionalit než stávající dva existující (picoweb [3], microdot [24]).

- Route
  - Seskupování Route do Controllerů
  - Rozlišování metod
  - Path params
  - Query params
- Definice middleware funkcí
  - JSON body
  - Form data
- Konfigurace
  - Konfigurace přes konstruktor
  - EnvFile
- Logování

2.6 Use case

Use case diagram bude v tomto případě pro čtyři aktéry: vývojáře, systém, klienta a čas. V diagramu lze vidět, které všechny moduly může vývojář upravovat. Dle konkrétní implementace vývojáře bude rozmanitost funkcionality aplikačního serveru veliká. Systém se mezitím stará o to, aby všechny věci naimplementované vývojářem byly uloženy do běhu aplikace a řádně byly využívány při dotazování od klienta. Klient pouze posílá své dotazy a čeká na odpověď. Čas se stará pouze o timeout, v případě že žádná odpověď z aplikačního serveru nepřijde. Na obrázku 2.3 je Use Case diagram, který znázorňuje výše popsané věci.



Obrázek 2.3: Use case diagram pro miniweb. Diagram byl vytvořen v EA.

Kapitola 3

Konfigurace

V této kapitole jsou popsány všechny nezbytně důležité konfigurace, které je potřeba vykonat před implementační částí práce. Jedná se zejména o zmínění konkrétních využívaných verzí aplikací a zdůvodnění. A také příprava prostředí a zařízeních.

3.1 Vývojové prostředí

Pro vývoj je používán PyCharm Community Edition 2020.2.3 64-bitová verze od společnosti JetBrains [32]. Do IDE byl také stažen plugin MicroPython [33], který umožňuje nahrávat a mazat soubory do připojeného mikročipu k počítači. Jako interpreter IDE poté používáme verzi Pythonu 3.5, a to z toho důvodu, že má k MicroPythonu nejblíže (až na menší rozdíly) [34].

Místo balíčkovacího nástroje pip budeme používat micropip [35]. Ten je potřeba nejdříve stáhnout na zmiňovaném pipu. Poté již budeme knihovny instalovat přes micropip soubor.

3.2 Mikročip ESP8266

Pro zprovoznění Micropythonu na mikročipu ESP8266 bylo postupováno dle oficiálního návodu [36]. Nejprve byl stažen firmware (jelikož disponujeme 4M zařízením, musela být použita Stable verze pro 2M a více) [37]. Poté bylo potřeba stáhnout esptool (přes balíčkovací systém pip) pro následné vymazání paměti mikročipu. Na závěr se nahrál stažený firmware na mikročip a zároveň byl nadefinován sériový port (v našem případě na operačním systému Windows 10 byl použit COM3).

3.3 Unix port

Pro snadnější zkoušení a debugování knihovny využijeme Unix port pro micropython [38]. Jelikož jako primární operační systém je používán Windows 10, tak si stáhneme VirtualBox [39] a v něm si vytvoříme virtuální distribuci linuxu - Ubuntu (Image stáhneme na oficiálních stránkách) [40]. Poté stáhneme snap a nainstalujeme micropython. Nainstalujeme PyCharm [32] a do něj si nastavíme Shell script. Přes něj jsme schopni přímo v Ubuntu spouštět kód micropythonu a nemusíme neustále nahrávat kód do čipu.

```
1    sudo apt update
2    sudo apt install snapd
3    sudo snap install micropython
```

Listing 3.1: Příkazy pro stažení micropythonu přes snap

3.3.1 WSL

Původní záměr bylo používat WSL2 [41] namísto VirtualBoxu, bohužel WSL2 se jevilo jako velice problémové. Při restartu prostředí se vždy u snapu promazali důležité meta soubory a nic nefungovalo. Zabrala až úplná reinstalace, ale po restartování Ubuntu jsme byli opět na začátku. Jako dočasné řešení byl použit bashový skript, který přeinstaloval celý snap a micropython po spuštění, aby bylo prostředí provozuschopné. Z tohoto řešení bylo upuštěno a raději se přešlo na VirtualBox, kde funguje vše bez problému.

Kapitola 4

Návrh

V této kapitole bude popsán celkový návrh frameworku, jako je definování tříd, jak a co si mezi sebou budou předávat, a jak se budou chovat. Nebude také chybět část o tom, jakým způsobem budou funkcionality vystavovány pro uživatele, aby používání našeho frameworku bylo co nejintuitivnější. To vše bude doprovázeno názornými ukázkami kódu či diagramy pro grafické znázornění funkcionalit.

4.1 API frameworku

Jak vyplývá z analýzy, tak je téměř ve většině obdobných frameworků využíváno právě dekoratorů jako API pro uživatele. V našem případě bude jejich používání také vhodné, protože je přehledné a snadné k používání. Proto budou využívány právě tam, kde to v našem případě bude dávat smysl.

4.1.1 Jak fungují dekorátory v Pythonu

Jak již bylo zmíněno v nadkapitole, tak budou značně využívány dekorátory. Proto v této části bude popsáno, jak vlastně dekorátory fungují, a z toho budeme dále vycházet. Dekorátory v Micropythonu jsou totožné jak v Pythonu, zde nebylo nic pozměněno. Jedná se zpravidla o nějakou funkci, ta je označena zavináčem před samotným názvem. Pod tím je poté nějaká další funkce, která je dekorovaná oným dekorátorem. V podstatě to znamená, že je povolána funkce dekorátoru s parametrem vnitřní funkce [2]. V Listingu 4.1 je ukázka zápisu dekorátoru s anotací, v 4.2 bez použití anotace a v 4.3 dekorátor s parametry.

```
1  @decorator
2  def foo():
3      pass
```

Listing 4.1: Zápis dekorátoru přes anotaci.

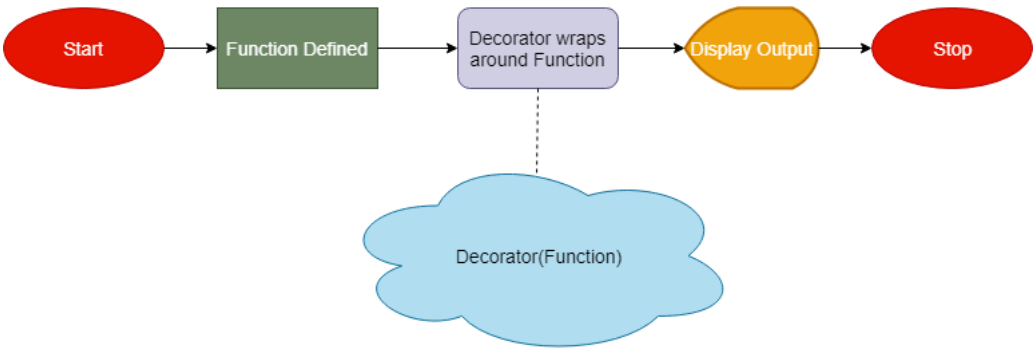
```
1  decorator(foo)
```

Listing 4.2: Zápis jednoduchého dekorátoru bez anotace.

```
1  decorator(arg)(foo)
```

Listing 4.3: Zápis dekorátoru s parametry bez anotace.

Z těchto přepisů bez zavináčů lze vypozorovat, že je funkce zavolána. Zde se aplikuje paradigma procedurálního programování, to je v případě našeho frameworku žádoucí. Budeme totiž chtít, aby se dekorované funkce okamžitě při spuštění začali vykonávat, a tímhle toho snadno docílíme. Na obrázku 4.1 je vyzobrazeno fungovaní dekorátorů.



Obrázek 4.1: Grafická ukázka, jak dekorátory v Pythonu fungují [2].

4.1.2 Route

Základním stavebním kamenem každého web aplikačního frameworku jsou endpointy. Na daný endpoint budeme chtít navázat funkci, která bude nadefinována uživatelem. Z frameworku by se potom do funkce měli dostat request headery, path a query parametry či content v těle requestu. Všechno toto bude obstarávat dekorátor - route. Ten bude přijímat 3 parametry - cestu, metody a controller. Poslední parametr, controller, je nepovinný parametr. Pokud budeme chtít nastavit na daný endpoint právě jednu základní metodu bude možné použít dekorátory get, post, put či delete. Ty pouze přezavolají původní dekorátor route s již nadefinovanou hodnotou. Nebudou mít tedy žádnou přidanou funckionalitu, pouze zjednodušený a příjemnější zápis.

Prvním parametrem bude cesta, která definuje endpoint, na který bude funkce reagovat. Do cesty budou moci býti nadefinovány také path parametry. Ty budou označeny složenými závorky a uvnitř nich bude název daného path parametru. Druhým parametrem jsou HTTP metody. Pokud uživatel bude definovat přes dekorátor route, bude mu umožněno vkládat více hodnot. Pro metody budou vydefinovány enumerátory, které by měl uživatel využívat. Třetí, nepovinný a poslední parametr - controller. Ten přijímá referenci na stejnojmennou instanci třídy. Controller definuje cestu jako prefix k cestě route a bude hrát roli při middleware funkcích (bude popsáno v další kapitole). V Listingu 4.4 je ukázka, jak by přibližně mohlo routování vypadat po implementaci v našem řešení.

```
1   controller = Controller("/controller")
2
3   @route("/path", [Method.GET, Method.POST], controller)
4   def route_function(req, res)
5       #vysledna cesta bude "/controller/path"
6       pass
7
8   @get("/path/subpath")
9   def get_function(req, res)
10      pass
```

Listing 4.4: Návrh API pro routování.

Framework bude vkládat do routových funkcí dva nebo tři parametry (v případě použití path parametrů) - request, response a variable. Request bude instancí stejnojmenné třídy, která se naplní po příchodu dat na aplikační server od klienta. Bude možné skrze ní přistoupit k contentu, headerům a query parametrům. Response poté bude také instancí stejnojmenné třídy, která bude prozatím s prázdnými parametry (defaultně vyplněny na hodnoty None). Do té uživatel bude nadefinovávat odpověď, kterou dostane klient. Třída bude řešena přes návrhový vzor builder, a aby klientovi dorazila response bude

nutné nadefinovat HTTP status, MIME type odesílaných dat a nakonec data samotná. Status a MIME budou omezeny přes předdefinované enumerátory našeho frameworku. Uživatel nemusí parametry vyplnit, tím říká, že nebude reagovat na klienta, a zároveň nebude chtít uzavřít spojení. Variable bude třetí a posledním příchozím parametrem. Bude se jednat o dictionary, kde přes klíč (název) bude možné přistoupit k hodnotě konkrétního path parametru z cesty. V Listingu 4.5 je návrh přístupu k parametrům.

```
1  @get("/user/{name}/id/{id}")
2  def foo(req, res, var):
3      #pristup k path parametrum
4      name = var["name"]
5      id = var["id"]
6
7      #pristup ke query parametrum a contentu v body requestu
8      query = req.params
9      content = req.content
10
11     #response pres builder a enumeratory
12     res.status(Status.OK).type(Mime.HTML).entity("<body>foo
   </body>").build()
```

Listing 4.5: Parametry routové funkce a Response builder

4.1.3 Middleware funkce

V našem frameworku budou middleware funkce jakési filtry, které se budou vykonávat vždy před tím, než začne být vykonávána logika route funkce, která je naimplementována vývojářem. Pokud filtry neprojdou, k závolání funkce nedojde. To bude řešeno tím, že každá middleware funkce bude vracet datový typ Boolean. True bude definovat úspěšnost middleware funkci, když projde, framework spustí další middleware v pořadí. Pokud neprojde, již žádnou další middleware spouštět nebude a skončí. V middleware funkci je možné nadefinovat HTTP response. Pokud middleware funkce neprojde a response nebude nadefinována, bude úmyslně udržováno spojení s klientem, dokud nespadne na timeout. V případě nadefinování bude vrácena HTTP response klientovi a spojení bude následně ukončeno. V Listingu 4.6 je návrh, jak by mohla výsledná forma filtrů vypadat.

```
1     @filter()
2     def foo(req, res):
3         #middleware funkce projde
4         return True
5
6     @filter()
7     def bar(req, res):
8         #middleware neprojde a vraci response
9         res.status(Status.FORBIDDEN).type(Mime.HTML).entity("
      Neprosel middleware!").build()
10        return False
```

Listing 4.6: Příklad middleware funkcí ve frameworku

Filtry bude možné rozdělit do dvou kategorií - globální filtry a controller filtry. Globální filtry se vykonávájí pro všechny zaregistrované route ve frameworku. Filtry pro controllery budou vykonávány jenom u route, které spadají pod tentýž controller. Filtry, stejně jako route bude dávat smysl definovat přes dekorátory. Dekorátory zařídí, že daná filter funkce bude uložena do běhu frameworku a až přijde HTTP request, tak bude zavolána.

Dalším a posledním velice specifickým filtrem bude dekorátor consumes. Ten přijímá MIME type, který by měla daná route přijmout. Pokud nebude tento dekorátor nadefinován, musí se uživatel sám vypořádat s příchozím datovým typem. Tento filtr při špatném MIME typu ukončí spojení s klientem. V Listingu 4.7 je návrh route s dekorátorem consumes.

```
1     @consumes(MIME.JSON)
2     @post("/path")
3     def foo(req, res):
4         #pokud prijde neco jineho nez json, samo ukonci spojeni
5         pass
```

Listing 4.7: Příklad route, která smí přijímat pouze json.

4.1.4 Controllery

Controllery budou mít speciální funkci, a to shlukování více route pod sebe. Tím bude pak možné přiřadit daným routám společný prefix před cestou, který je definován právě controllerem. Dále pod controllery budou uloženy reference na middleware funkce, které patří jen pod daný controller, ty budou volány jen nad routami controlleru.

Controllery samostatné nejsou užitečné, musí se k middleware funkci do dekorátoru předat jejich reference na instanci třídy, nebo se do route dekorá-

toru musí předat reference na jejich instanci třídy. Tím vznikne mezi těmito entitami asociace s controllerem. Názornou ukázku lze vidět v Listingu 4.8.

```
1   user_controller = Controller("/user")
2
3   @filter(user_controller)
4   def is_logged_in(req, res):
5       #zde lze napriklad definovat pouze pro user route, zda
    jsou uzivatele prihlaseni
6       pass
7
8   @get("/data", user_controller)
9   def get_user_data(req, res):
10      #vysledna cesta je "/user/data"
11      #zkontroluje user_controller filtry
12      pass
```

Listing 4.8: Definice controlleru a předání do filteru a route.

4.1.5 Konfigurace

Konfigurování bude umožněné dvěmi způsoby. Prvním způsobem bude přímo v kódu přes kontruktor při vytváření frameworku, a druhým bude skrze konfigurační soubor. Primární varianta je přes konstruktor, takže pokud budou vyplněny oba způsoby, pak parametry konstruktoru mají přednost. Při konfiguraci přes soubor bude stačit vytvořit jakýkoliv soubor, který má příponu *.env*, díky tomu bude framework vědět, že se jedná o soubor, ze kterého bude číst. Bude možné v něm nastavit úroveň logů, hosta či port pro komunikaci (popřípadě další parametry). Ukázka, jak by přibližně mohla konfigurace vypadat, je v Listingu 4.9 pro konstruktor a v Listingu 4.10 pro soubor.

Varianta přes konstruktor bude řešena skrze datový typ dictionary, který se bude předávat konstruktorem. V daném dictionary pak budou nadefinovány parametry skrze klíč a jejich hodnotu.

```
1   from miniweb import miniweb, Log
2
3   params = {
4       "port": 8000,
5       "log": Log.DEBUG,
6       "host": "localhost"
7   }
8
9   app = miniweb(params)
```

Listing 4.9: Ukázka konfigurace skrze konstruktor

V konfiguračním souboru bude každý řádek mít vlastní parametr s hodnotou. Na řádku pak bude název parametru, rovnítko a hodnota, vše bez mezer (pokud se nejedná o daný název parametru či její hodnoty).

```
1 #nadefinovany config.env soubor a data uvnitr
2 port=8000
3 log=DEBUG
4 host=127.0.0.1
```

Listing 4.10: Ukázka konfigurace skrze konfigurační soubor

4.1.6 Servírování statických souborů

V rámci komunikace s uživatelským testerem bylo zjištěno, že by bylo dobré, kdyby framework nabízel i možnost servírovat statické soubory. To znamená, že skrze definovaný endpoint bude framework vyhledávat v složkách zařízení (mikročipu) cílový soubor a ten vracet. Dále by měl být schopný pracovat i s podsložkami. Tato zpětná vazba je trefná, a opravdu dává smysl tuto funkcionalitu přidat, přestože je nad rámec požadavků. Bude tedy dodatečně navržena a doimplementována.

Vzhledem k tomu, že k přidání této nové funkcionality dochází již ve fázi, kdy máme hotový návrh a z větší části i implementaci, pak je třeba hledat přepoužití nějaké stávající funkcionality. Díky tomu lze i ověřit, zda je naimplementovaná aplikace navržena tak, aby byla dále rozšiřitelná. Jako nejvhodnější řešení se jeví přepoužití třídy Route, a tedy i její funkcionality. Bude vydefinován potomek třídy Route, který se bude starat o statické servírování souborů. Lze také předem určit, že nebude potřeba ho mít jako dekorátor, jelikož funkce pod Route je jasná - najít soubor.

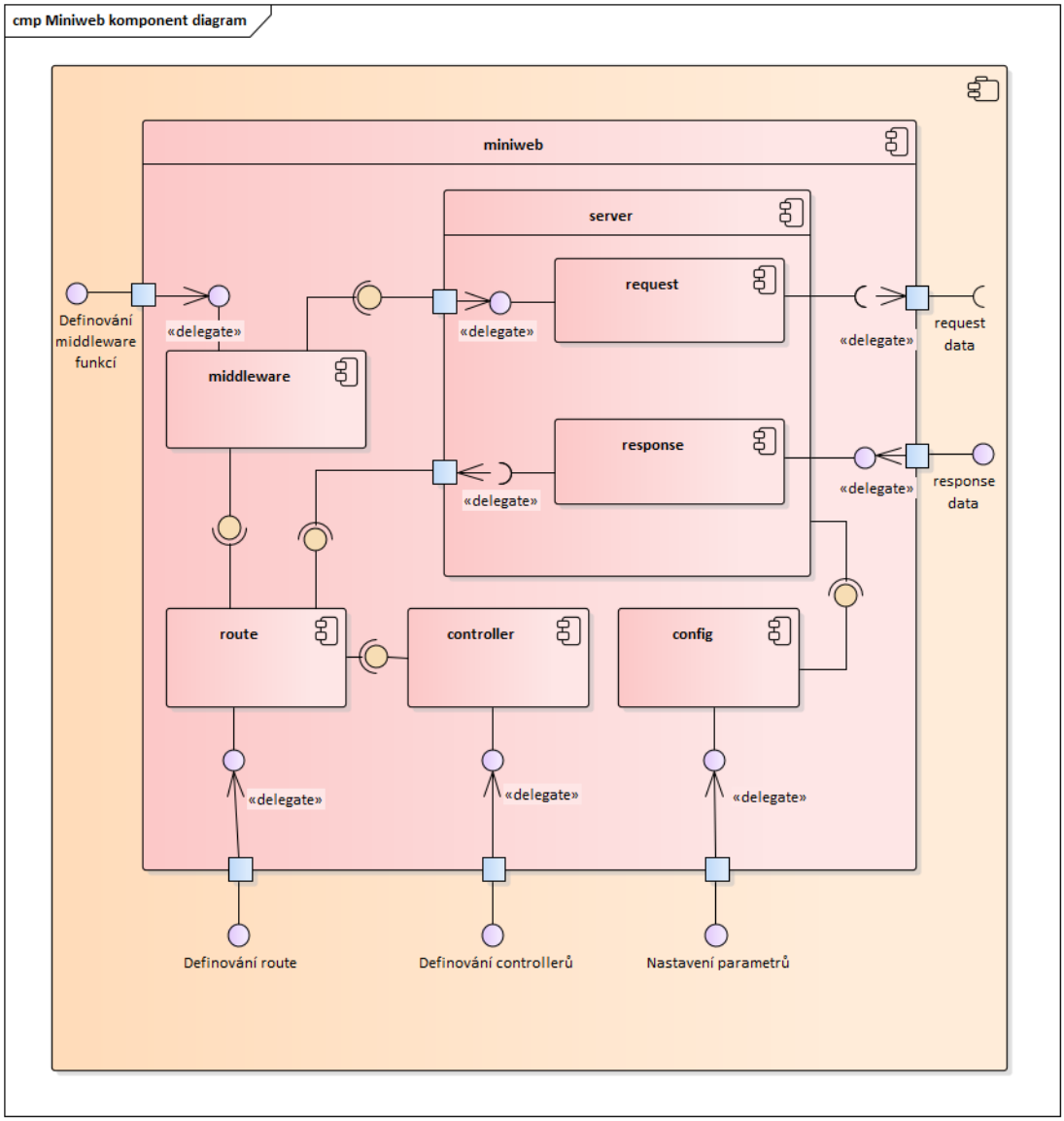
Stejně jako běžnou route, bude možné seskupovat statické routy do Controllerů. To znamená, že pro ně budou prováděny middleware funkce, jak globální, tak controllerové. Jako další parametry zde dávají smysl tyto dva: root cesta a cesta endpointu. Root cesta bude definovat, kde v zařízení začneme hledat soubor. Cesta v endpointu bude jakousi maskou pro root cestu, aby se do URL adresy nemuseli zapisovat celé cesty v adresářové struktuře daného zařízení. Níže v Listingu 4.11 je ukázka statického routování.

```
1  from miniweb import miniweb, Controller
2
3  app = miniweb()
4
5  file_controller = Controller("/file")
6
7  #pres staticky route lze pristupovat k podslozkam, proto zde "/
       subf" muze byt, ale je redundantni
8  app.static_router(root="/var/www/", path="/home/",
       file_controller)
9  app.static_router(root="/var/www/subfolder/" path="/subf/",
       file_controller)
10
11 #controller je volitelny parametr
12 app.static_router(root="/etc/", path="/etc/")
```

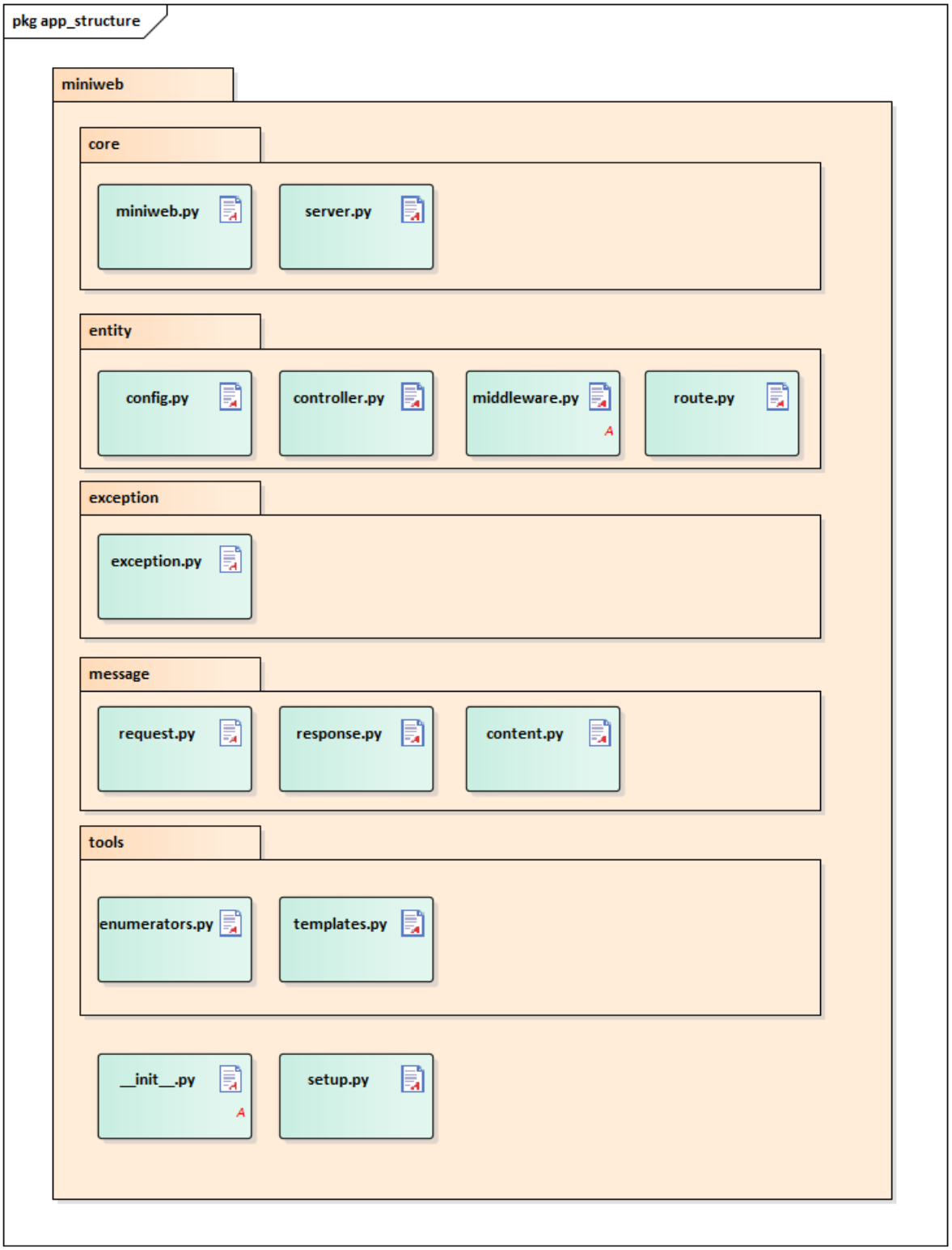
Listing 4.11: Ukázka definování statických route

4.2 Struktura frameworku

Framework bude rozdělen do modulů (*.py* souborů) dle logiky. Pro lepší přehlednost v kódu budou moduly rozčleněny do podadresářů. Obecný pohled nad celým frameworkem je zde graficky znázorněn v komponent diagramu 4.2 a v package diagramu 4.3. Detailnější znázornění bude v sekvenčních diagramech, které budou rozkouskované do konkrétních funkcionalit v podkapitole - **Logika funkcionalit**.



Obrázek 4.2: Komponent diagram frameworku. Diagram byl vytvořen v EA.



Obrázek 4.3: Package diagram frameworku pro přehled, jak jsou soubory rozděleny do složek v projektu. Diagram byl vytvořen v EA.

4.2.1 Moduly a jejich popis

Jak již bylo v kapitole **Analýza 2** zmiňováno, tak moduly myslíme přímo soubory v pythonu. Jelikož jsme pro přehlednost kódu rozdělili modul do více souborů, tak v tomto kontextu budeme mít na mysli podsložky, které zhlukují několik *.py* souborů.

První a zároveň největší modul je nazván core - jádro. Tam je vykonávána většina operací, a ostatní moduly jsou spíše dopomocné pro tento. Je rozdělen na soubor pro Server a pro Miniweb. Server bude zpracovávat asynchronně dotazy a bude vracet odpovědi. V tomto souboru bude vše začínat a zároveň končit. Poté bude předávat data právě do Miniwebu, který obsahuje route dekorátory, a drží v sobě reference na všechny důležité objekty. Miniweb je středobodem celého frameworku, vše jde skrze něj. Zajišťuje předávání informací či volání do dopomocných modulů a nakonec zpět vrací Serveru odpověď.

Dalším modulem je entity. Do něj patří objekt s konfiguračními parametry, Route objekt, Controller objekt či middleware funkce. Tyto objekty jsou často používány přímo Miniwebem, a jsou drženy v paměti za celou dobu běhu aplikace. Uchovávají informace, které jsou definované vývojářem a ovlivňují chování frameworku. Lze laicky říct, že dávají aplikaci "život", konkrétní implementaci.

Následuje modul message. Obsahuje objekt pro HTTP request, HTTP response a Content. Data se přijímají a odesílají v bytech, ale jelikož chceme mít data nějak uspořádaná, budou v běhu aplikace uloženy do objektu. Request obsahuje hlavičky a objekt Content, který má v sobě například přijatý JSON, či XML. Response poté obsahuje Status, data a jejich MIME typ, které vracíme.

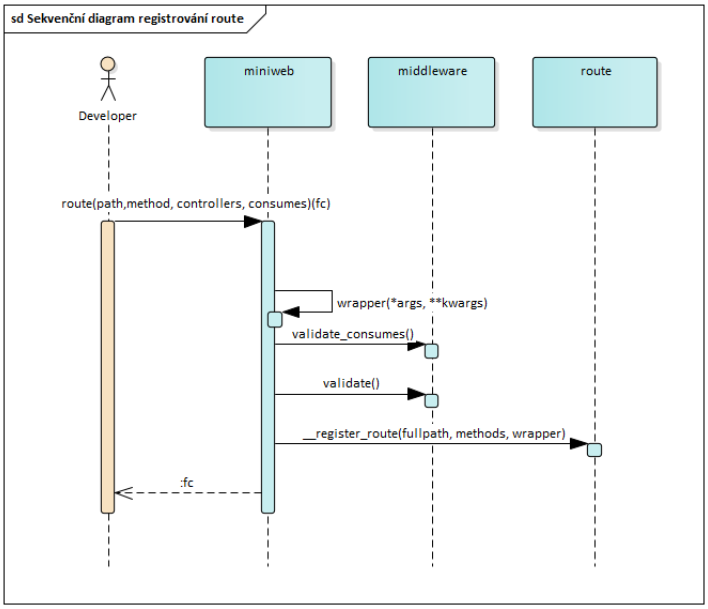
Zbývající moduly exception a tools obsahují dopomocné funkce a objekty. Soubor Exception má v sobě potomky třídy Exception, přímo pro framework. Tools poté obsahují dodatečné funkce a enumerátory. Bez modulů, na stejné úrovni jako složky modulů jsou Init a Setup. Init obsahuje import všech souborů, funkcí a parametrů, které předáváme ven z frameworku. Setup poté obsahuje informace o projektu, autora, email a další informace.

4.3 Logika funkcionalit

Pro hlubší pochopení, jak nástroj funguje, budou v této sekci popsány a následně graficky znázorněny zapomocí UML diagramů jednotlivé části frameworku. Některé z průchodů budou z pohledu vývojáře a uživatele frameworku, jiné naopak z pohledu klienta, který provolává server a chce pouze dostat zpátky odpověď.

4.3.1 Průchod registrací route

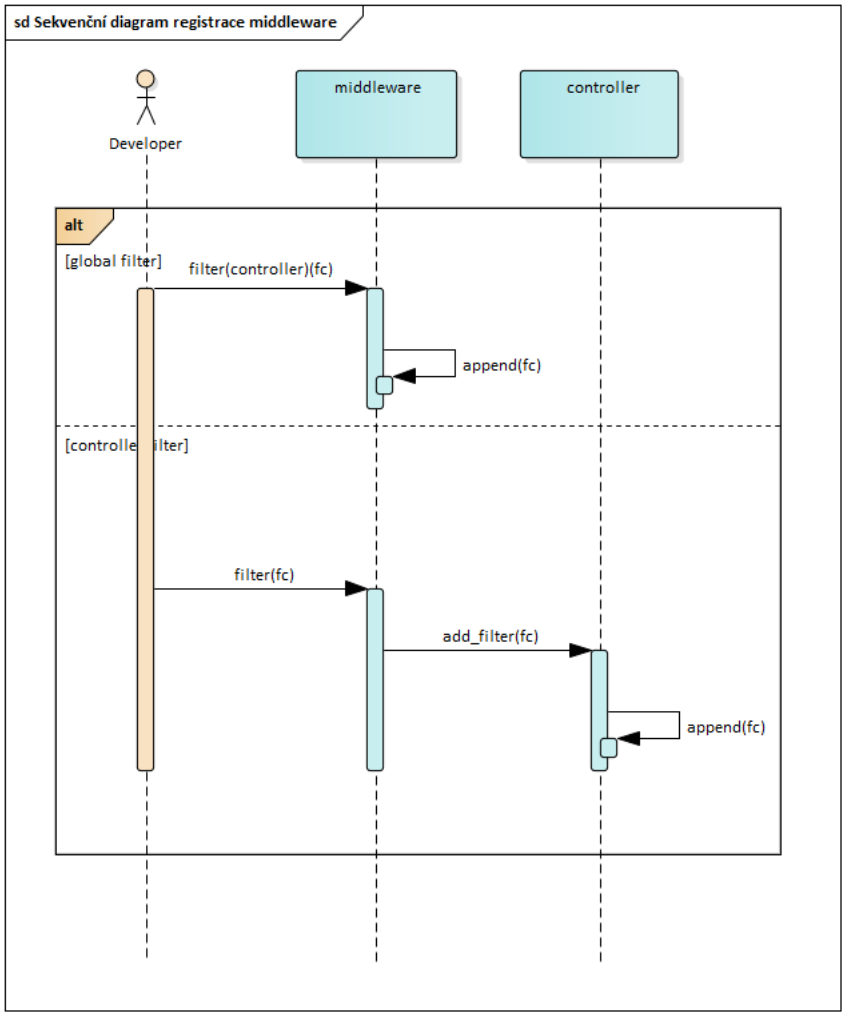
Route dekorátor pod vývojářovou funkcí zajistí, že při spuštění aplikace se všechny jeho vydefinované dekorátory spustí. Ty mají referenci na funkci přímo pod ní. Díky tomu se do běhu frameworku dostanou přes dekorátor informace o routě a k jaké funkci jsou asociované. Dekorátor podle toho v podstatě upraví stávající funkci a před ní vloží middleware funkce. Vzniká nová funkce, která vykoná middleware funkce a následně volá funkci původní. Nyní framework vytvoří instanci třídy route, do ní vloží onu referenci na funkci, zkompiluje cestu a uloží metodu. Instance route je poté uložena do pole v miniwebu. Uložena je pro pozdější dotaz od klienta, aby našla shodu v cestách a metodách, a mohla provolat danou funkci, co je k ním asociovaná. Výše popsané je ukázané i v sekvenčním diagramu - obrázek 4.4.



Obrázek 4.4: Sekvenční diagram znázorňující průchod route a její registraci do běhu aplikace. Diagram byl vytvořen v EA.

4.3.2 Průchod registrací middleware

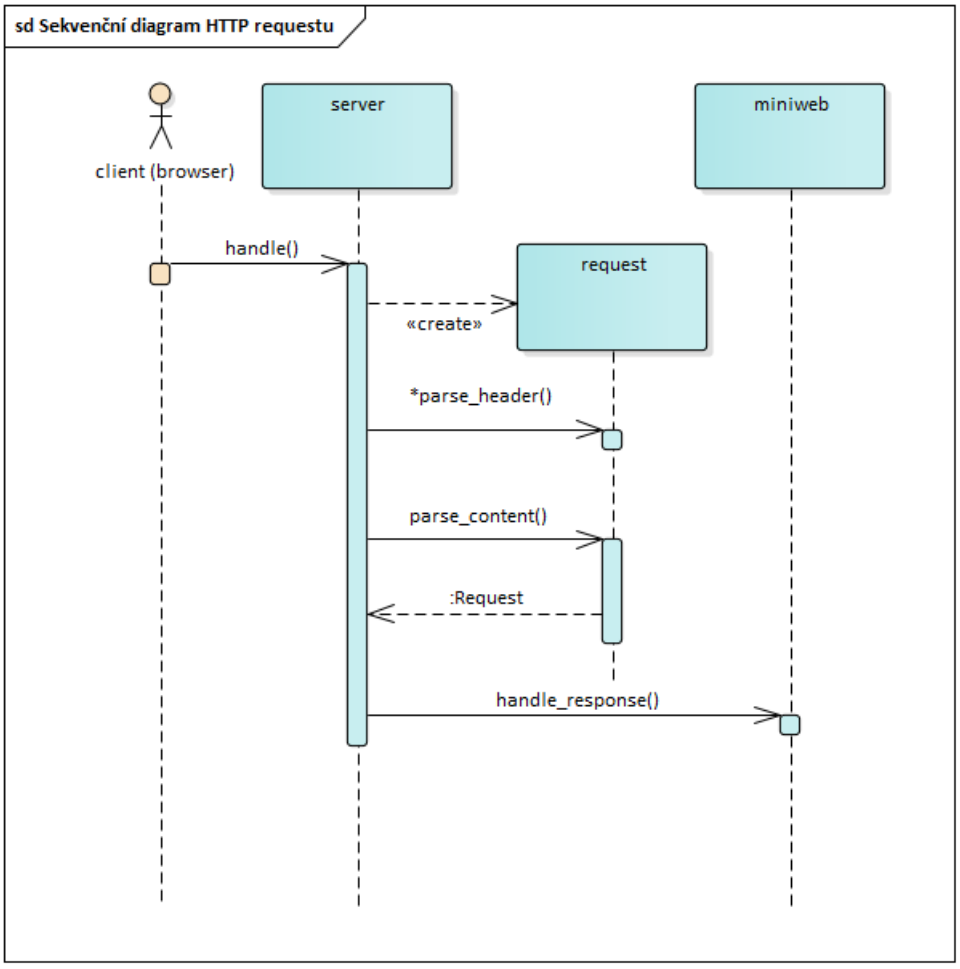
Vývojář nadefinuje middleware funkci a přes dekorátor přes funkci filter z ní vytvoří middleware funkci frameworku. Ta bude přijímat jeden nepovinný parametr, controller. Pokud bude controller vyplněn, pak se uloží reference na funkci do instance controlleru, který si bude uchovávat všechny svoje middleware funkce. V opačném případě se uloží do pole, kde jsou uchovány všechny globální middleware. Globální middleware máme na mysli takové middleware, které se budou vykonávat u všech route. Při spuštění aplikace až do jejího vypnutí jsou drženy reference na tyto funkce, které čekají, že budou zavolány při příchozím HTTP requestu. Lze vidět i v grafické podobě na obrázku 4.5.



Obrázek 4.5: Sekvenční diagram znázorňující průchod middleware a její registraci do běhu aplikace. Diagram byl vytvořen v EA.

4.3.3 Průchod HTTP requestu

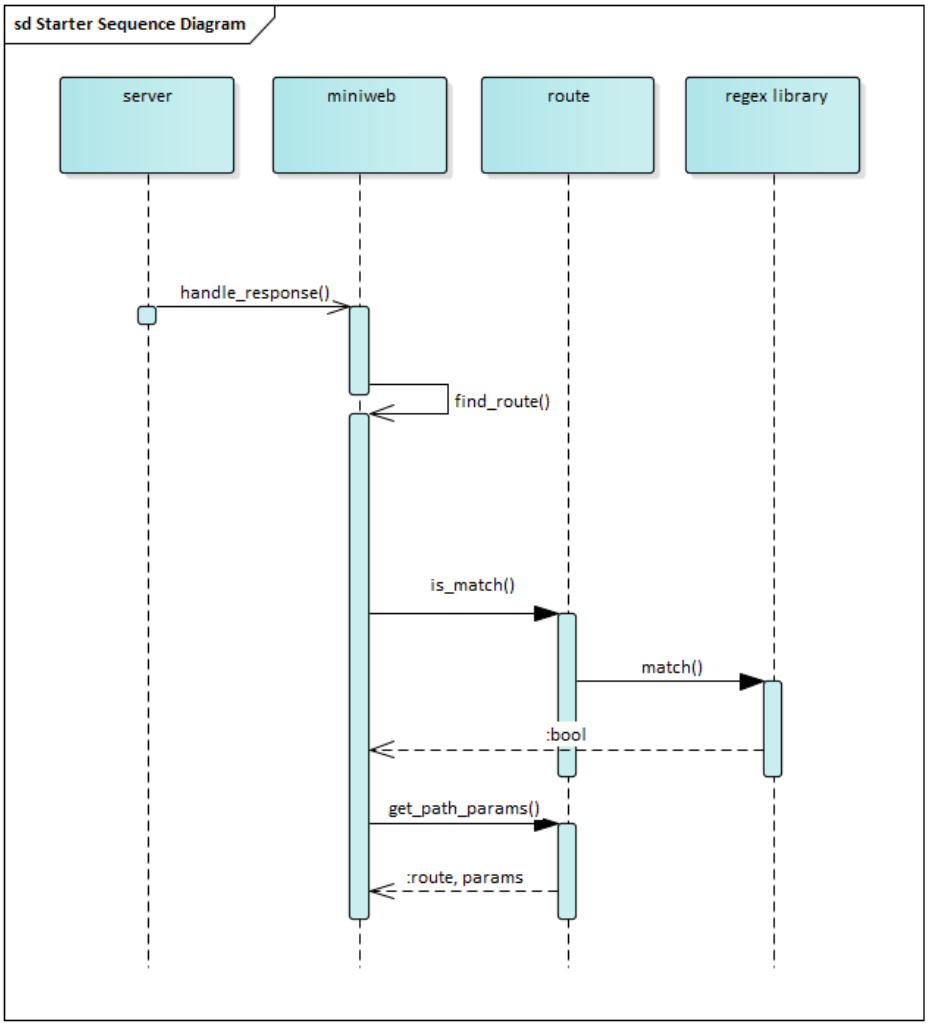
Klient pošle svůj HTTP request. Ten je přijat na serveru, který zpracovává příchozí a odchozí zprávy asynchronně. Framework vytvoří instanci třídy Request. Následně se začne číst příchozí HTTP request. Nejprve budou čteny headers, ty budeme zpracovávat řádek po řádku a zapisovat do objektu Request. Jakmile dočteme headers, tak zbydou pouze data, která můžeme přečíst již celá najednou, ta jsou zabalena do objektu Content, který je také uložen do objektu Request. Následně je zavolána metoda miniwebu, do které je poslán daný objekt Requestu. Metoda se dále postará o validaci cest, middleware, a vrácení příslušné odpovědi. To vše je znázorněné na obrázku 4.6.



Obrázek 4.6: Sekvenční diagram znázorňující průchod HTTP requestu od klienta k zpracování serverem a předání pro následné zpracování HTTP response. Diagram byl vytvořen v EA.

4.3.4 Průchod hledáním route

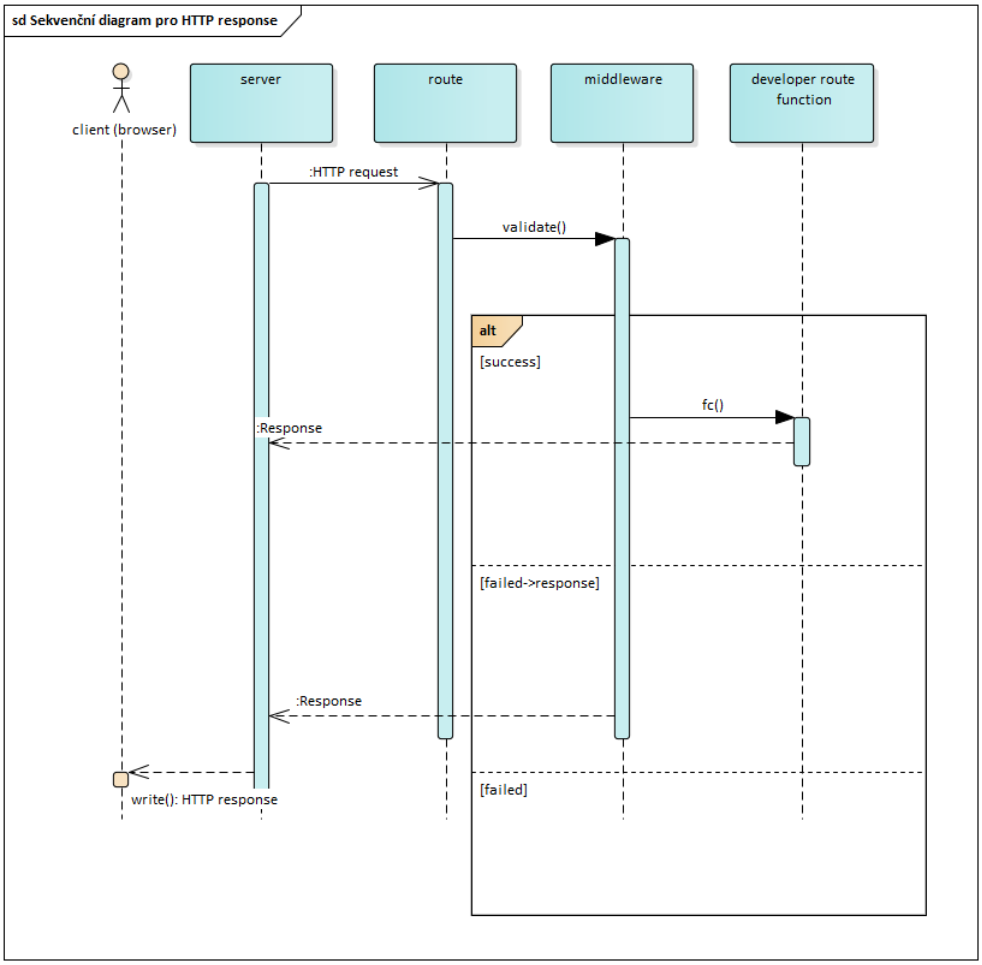
Průchod hledaním route a parametrů navazuje na předchozí sekvenční diagram průchodu HTTP requestu (obrázek 4.6). Tady již máme zpracovaná příchozí data do objektu. V miniwebu je zavolána metoda pro nalezení správné route. Metoda kontroluje dvě věci, zda cesta odpovídá nějaké cestě routy a zda sedí HTTP metody. HTTP metody jsou kontrolovány jednoduchým způsobem - zda odpovídájí. U cesty je to komplikovanější, tam je kontrola prováděna přes regulární výrazy, kvůli případným path parametrům. Pokud nějaké path parametry obsahuje, tak metoda objektu Route také vrací ony parametry ve formátu klíče a hodnoty (v dictionary). Sekvenční diagram tohoto scénáře je na obrázku 4.7.



Obrázek 4.7: Sekvenční diagram znázorňující průchod hledání route a popřípadě i path parametrů a jejich předání. Diagram byl vytvořen v EA.

4.3.5 Průchod HTTP response

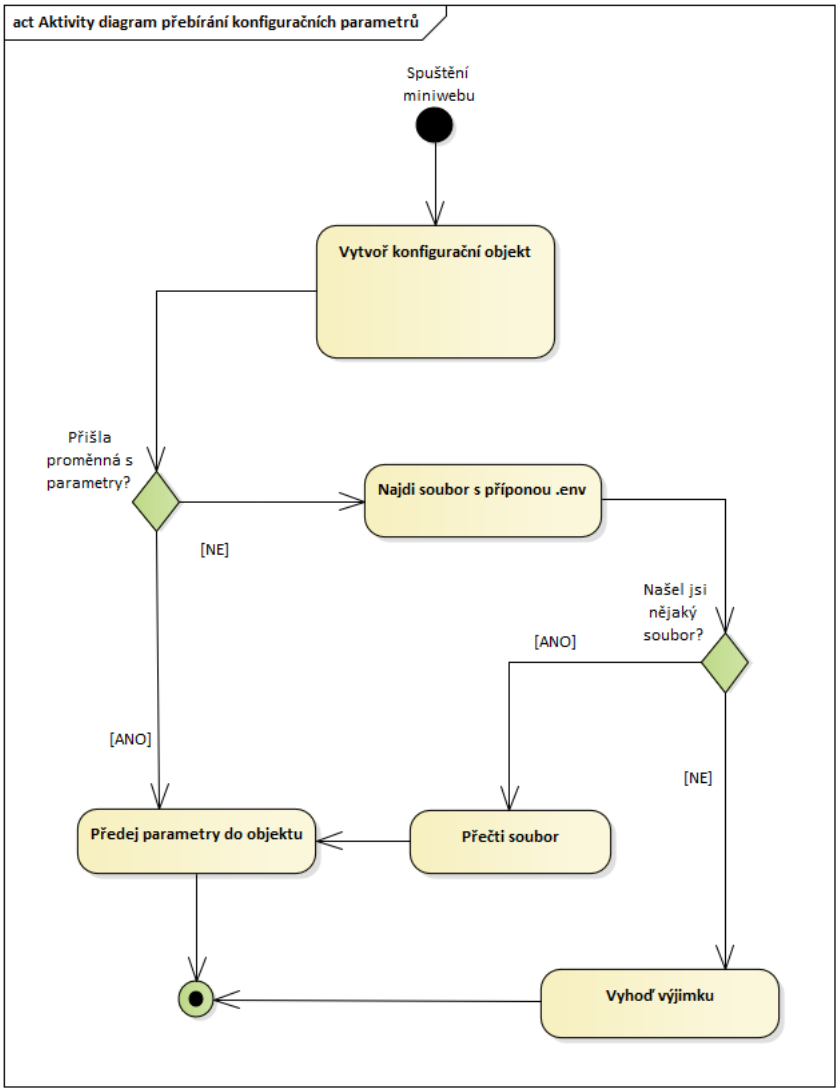
Tento sekvenční diagram navazuje na předchozí s HTTP requestem (obrázek 4.6) a následně diagram pro nalezení Route (obrázek 4.7). V případě, že máme nalezený objekt Route, tak v něm máme uchovanou referenci na vývojářovu nadefinovanou funkci, ta bude provolána s parametry. Ta nejprve začne vykonávat své middleware funkce. Ty buďto projdou a může se vykonávat samotná funkce, a nebo ukončí běh a mohou vrátit Response. Po úspěšném průchodu middleware funkcí je ve vývojářově funkci zpracována HTTP response, která je poté vrácena zpět na server, který ji přes metodu asyncia pošle zpět klientovi. Diagram k tomuto průchodu je na obrázku 4.8.



Obrázek 4.8: Sekvenční diagram znázorňující zpracovávání HTTP response pro klienta. Diagram byl vytvořen v EA.

4.3.6 Přijímání konfigurace

V tomto diagramu aktivit (obrázek 4.9) je znázorněno přebírání konfiguračních parametrů nadefinované vývojářem do frameworku. Vytvoří se objekt Config, který primárně přebírá z nepovinné proměnné. Pokud je proměnná vyplněna, předá informace do objektu. V opačném případě se bude hledat soubor s příponou *.env*. Pokud se nenajde vrací výjimku, pokud ano, prečte data ze souboru a zapíše do objektu. Framework pak již celou dobu pracuje pouze s objektem Config, kde má všechny parametry uchované po zbytek běhu aplikace.



Obrázek 4.9: Diagram aktivit znázorňující přebírání konfiguračních parametrů. Diagram byl vytvořen v EA.

4.4 Zpracování formátů dat z těla requestů

Při příchodu HTTP requestu může v některých případech dorazit i nějaký Content. Ten je rozpoznán dle toho, že předchází příchodem headers s názvem Content-length, který definuje počet znaků daného Contentu. Ten je poté přístupný z route funkce, a je potřeba jej zformátovat do nějaké příjemné podoby pro vývojáře, aby s ním mohl manipulovat dle své potřeby. To je důvodem, proč framework bude umět pársovat vydefinované datové typy do objektů.

Framework bude schopen pracovat s příchozím JSON objektem (ukázka v Listingu 4.12) či Form Daty (multipart/form-data - ukázka v Listingu 4.13). To jsou nejčastěji používané datové typy, které by měli pokrýt většinu případů užití. Ke zvážení by ještě pravděpodobně stálo XML, ale pro tuto chvíli nebude v plánu k implementaci. V případě, že se bude jednat o jiný typ než je JSON či Form Data, bude uživateli předán pouze textový řetězec s daty.

JSON bude zformátován do objektu, aby bylo možné přistupovat k různým hodnotám klíčů přes ně. Objekt zvládne více vnořených JSONů do sebe, a také pole hodnot.

```
1    @get("/example")
2    def example(req, res):
3        #pristup k hodnote dle nazvu klice
4        key = req.content["key"]
5        id = req.content["id"]
```

Listing 4.12: Ukázka práce s JSON objektem.

Form Data budou ukládána do objektu, kde každý klíč bude uložen ve stejnojmenném parametru objektu. Pokud se bude jednat o soubor, pak bude parametr odkazovat na instanci třídy File, která si bude držet název, příponu a data samotná.

```
1    @get("/example")
2    def example(req, res):
3        #pristup k hodnotam dle parametru objektu
4        file = request.content.file
5        file_name = file.name
6        file_format = file.type
7        file_data = file.data
8
9        value = request.content.value
10       multilinevalue = request.content.multilinevalue
```

Listing 4.13: Ukázka práce s Form Daty.

Kapitola 5

Implementace

V této implementační kapitole budou doplněny zbylé informace, které navazují na návrh. Tím je myšleno, které knihovny a pro co byly využity, zda museli být nějaké funkce dodělány, či co jim chybělo. Dále budou popsány návrhové vzory, které byly uplatněny při implementaci frameworku a v jakých modulech. Na konec budou popsány funkce, které byly přidány navíc, a pak také problémy, které při implementaci nastali, a jak byly řešeny.

5.1 Využívané knihovny třetích stran

Knihovny Pythonu nejsou zpravidla kompatibilní v Micropythonu, proto je potřeba na PyPi hledat konkrétní Micropython knihovny. Dále pak existuje mnoho Micropython knihoven, které zpravidla umějí jen omezeně to, co ta samá knihovna v Pythonu. Jsou případy, kdy žádná taková knihovna není, a to je zároveň motivací, proč bylo právě toto téma práce zvoleno - vytvořit tento framework. Tímto shrnutím se chtělo říct, že bude nutné si mnohé doprogramovat.

Celkem bylo použito 5 knihoven, do kterých počítáme ty, které jsou vestavěné přímo v Micropythonu, tak ty z PyPi. Jedná se o microasyncio (také uasyncio), micropython-logging (ulogging), micropip (upip), microjson (ujson) a ure (regulární výrazy).

5.1.1 uasyncio

Jedná se o pravděpodobně nejdůležitější knihovnu, kterou používáme v implementaci. Je použita v modulu Server a obstarává asynchronní zpracovávání příchozích a odchozích požadavků. V asynciu máme tzn. event loop, to je jádro celého asyncia, které se stará o onu asynchronnost. V naší implementaci předáme do event loop nový task, a do něj vložíme coroutine, což v našem případě je spuštění serveru s callback funkcí. Danou event loop necháme běžet neustále, dokud nebude zavolána metoda *stop()*. Poté veškerá logika probíhá v oné callback funkci, v našem případě se jmenuje *___handle()*. Tato funkce čte příchozí data, předává je dál do frameworku a následně vrací data, to vše dělá asynchronně. Konkrétně z příchozích dat vytvoří objekt Request, a ten předá dál do jádra frameworku. A přichází ji naopak objekt Response, ze kterého vezme parametry a posílá je klientovi [25, 28, 42].

5.1.2 ulogging

Jak již název vypovídá, stará se o logování. Funguje jako ve všech logovacích knihovnách, tedy v kódu jsou log zprávy, definované různými úrovněmi. Na vývojáři je poté, kterou úroveň logů si nastaví. Úrovně jsou definovány číselnými hodnotami, ale pro lepší zápis je dodán enumerátor [43].

5.1.3 upip

Tento skript je důležitý pro stahování ostatních micropython knihoven z softwarového repozitáře - PyPi. Všechny knihovny micropythonu jsou stahovány za pomoci upipu [35].

5.1.4 ujson

Knihovna je používána v modulu Content. Pokud je přijat Content-Type application/json, pak jej knihovna zpracuje a přeparsuje do dictionary. Objekt je poté uložen jako parametr do objektu Request, který je pak předán do route funkce vývojáře. Knihovnu může využít i vývojář pro svoje účely, nemusí ji ani stahovat, protože je tato knihovna přímo builtin v micropythonu.

5.1.5 ure

Jedná se o knihovnu pro regulární výrazy. Používáme jí v modulu Route, kde nejprve při registraci route zkompiluje cestu. Poté, když přijde nějaký HTTP request, tak hledá shodu přes metodu *match()*. Bohužel ostatní funkce jako třeba *search()* nefungují dobře. Pak také mnoho funkcí v této micro verzi knihovny chybí, a z toho důvodu je nutné si řešit path parametry jiným způsobem. To bylo vyřešeno zapomocí pozic zpětných lomítek, díky kterým rozpoznáváme path parametry.

5.2 Použité návrhové vzory a metody

Ze standartních návrhových vzorů je zde využit zejména Singleton, a pak také Builder.

Jako Singleton jsou zde navrženy třídy Miniweb, Server a Config. Hlavním důvodem využití tohoto vzoru je optimalizace, nechceme vývojáři dovolit tvořit více instancí miniwebu v aplikaci. Miniweb je kontejner pro celý framework, Server obstarává zprávy a Config uchovává konfigurační data. V Listingu 5.1 lze vidět ukázku Singletonu v kódu.

```
1  class Miniweb:
2      __instance = None
3
4      @staticmethod
5      def get_instance(params=None):
6          if Miniweb.__instance == None:
7              Miniweb(params)
8          return Miniweb.__instance
9
10     def __init__(self, params=None):
11         if Miniweb.__instance != None:
12             raise SingletonExpcetion()
13         else:
14             self.config = config(params)
15             Miniweb.__instance = self
16             self.routes = []
17             self.server = None
18             self.__init_logging()
```

Listing 5.1: Příklad Singletonu z třídy Miniweb.

Dalším používaným standartním návrhovým vzorem je Builder, který je využíván vývojářem u stavby Responce objektu. Jeho definice je jednoduchá jako je v případě Javy EE ze stejnojmenného objektu Response [44].

Kromě návrhových vzorů jsou zde aplikovány další metody vývoje, které zlepšují kvalitu kódu. Jednou z nich jsou Enumy, ty slouží pro lepší zápis opakujících se hodnot. Jsou udělány pro MIME, HTTP statusy a logy. Uživatel si proto nemusí kupříkladu pamatovat HTTP statusy dle kódů, ale může zadat název dle předdefinovaného Enumu.

Pak samozřejmě již dříve mnohokrát zmiňované dekorátory. Zde by bylo dobré uvést na pravou míru, a říct rozdíl mezi dekorátory v pythonu a návrhovým vzorem dekorátor, aby se tyto pojmy nepletly. U návrhového vzoru jde o změnu funkčnosti objektu bez použití dědičnosti. To je dosaženo vložením dalšího objektu. Zatímco dekorátor v pythonu pozměňuje funkci či objekt za běhu aplikace [45, 46]. Jsou využívány jako rozhraní pro vývojáře pro definici route a middleware.

5.3 Funkcionality nad rámec zadání

V průběhu implementace jsme dostali zpětnou vazbu od vývojáře, který testoval framework. Na základě toho byl návrh na přidání nové funkcionality a to statické servírování souborů. Tento návrh i argumenty, proč jej přidat, dávaly smysl, a proto byla ihned funkcionalita přidána. Následně byla doplněna i kapitola do dokumentu a to **4.1.6 Servírování statických souborů**, kde je vše popsáno. Přidání této funkce proběhlo bez komplikací.

Další funkcí navíc lze brát i to, co vše umí middleware funkce. V původním zadání jasně nebylo specifikované, co přesně mají umět. Tam jsou zmíněny pouze parsery pro JSON a Form Data, a to řeší parametr consumes v route. Ony samotné *filter()* dekorátory tedy umějí něco navíc, vývojář si může nadefinovat lehce svá vlastní pravidla. Zde bylo čerpáno z toho, jak funguje právě express (Javascript knihovna), s tím rozdílem, že express middleware funkce prochází jakýmsi iterátorem, a lze snadno nechat provolat další. Toto u našeho frameworku není možné. Pouštění dalších filterů v řadě probíhá automaticky.

Navíc byl také přidán konfigurační parametr - buffer. Ten slouží k definování, jak velké pole bytů bude naplněno čtením souboru. Parametr je proměnlivý, a je třeba ho nastavit dle používaného mikročipu.

5.4 Problémy při implementaci a řešení

V této kapitole budou zmíněny problémy, které se naskytly během implementace. Jelikož větší část vývoje probíhala na UNIX portu a na mikročipu to bylo zkoušeno až v pozdější fázi vývoje, tak jsou problémy rozděleny do podkapitol. To především proto, že zmíněný UNIX port nedokáže na 100 % nasimulovat mikročip. Může to znamenat, že něco, co funguje na UNIX, nemusí fungovat v mikročipu. S tím souvisí i důvod, že port neobsahoval všechny builtin knihovny jako mikročip a museli se stahovat dostupné verze z pipu, které nemuseli být totožné jako v mikročipu.

5.4.1 Problémy s mikročipem

První problém nastal ihned při samotném nahrávání kódu do čipu. Nahrávání padalo u souboru *enumerators.py*, přestože v souboru chyba nebyla. Po pátrání se přišlo na to, že čipu vadil název složky, ve které soubory byly - utils. Tento název byl pravděpodobně nějakým způsobem rezervovaný pro účely micropythonu. Řešením bylo přejmenovat složku na tools, pak se již celý projekt v pořádku nahrál na čip.

Dalším problémem bylo to, že servírování některých statických souborů neprocházelo, či se posílali jen z části. Jako příklad lze uvést obrázky, které se klientovi vraceli neúplné. Problém byl způsoben tím, že čip je schopen číst jen omezené pole bytů jednorázově, zatímco framework se snažil číst všechno. Bylo nutné číst postupně byty souboru a ihned je poslat klientovi zpět, a takto opakovat, dokud nebude soubor odeslán celý. V případě námi používaného čipu - ESP8266, je pole bytů omezeno na 512B [47]. Jelikož je velice pravděpodobné, že bude framework využíván na ESP32, či jiných čipech, pak bylo toto nastavení zprostředkováno jako další konfigurační parametr. Sice by ničemu nevadilo, kdyby zůstala hodnota pevně daná například na 128B, ale tím by se mohl omezovat čip, který zvládne podstatně více.

5.4.2 Problémy s knihovnami

Co se týče chybovosti knihoven, tak je nutno uznat, že byly doopravdy výjimečné. Spíše než chyby byl problém v funkčnostech, kde v podstatě knihovny neuměly to, co jejich plnohodnotná obdoba v CPythonu. To mělo za následek, že bylo nutné si funkce dopsat, či vymyslet jiný způsob, jak dosáhnout výsledku.

Nejkritičtější pro framework, byla knihovna na regulární výrazy - **ure 5.1.5**. Uměla kompilovat a hledat shodu v patternech, takže route byli nalezeny správně, ale problém nastal v path parametrech v cestě dotazu. Zde nefungovala metoda *search()*, a proto bylo nutné vymyslet jiný způsob. To bylo vyřešeno za pomoci zpětných lomítek, kde bylo při kompilaci cesty route zároveň zaznamenáno, za jakými zpětnými lomítky se nachází path parametry. Číselná vyjádření umístění lomítek byla uložena do pole a poté při příjmutí HTTP requestu byly parametry nalezeny.

Kapitola 6

Testování

Tato kapitola je rozdělena do tří částí. Tou první a základní jsou unit testy, které kontrolují zda základní funkce doopravdy fungují a popřípadě zda implementované změny nic nerozbijí. Druhou částí jsou uživatelské testy, kde vývojář implementuje na našem frameworku svou chytrou domácnost a hlásí nám chyby, dává přípomínky či náměty na vylepšení. Kapitolu uzavíráme seznamem nalezených chyb v průběhu testování.

6.1 Unit testy

Unit testy jsou spouštěny pod UNIX prostředím. Pro testování je používána přímo knihovna pro tento typ testů v Micropythonu [48]. Testy jsme dále rozdělili do modulů po testovaných oblastech. Máme zde testy pro Controllery, Middleware, Route a zprávy. Všechny tyto komponenty spolu úzce spolupracují, proto v jednom modulu mohou být využívany i prvky jiných komponent. U route jsou testovány jak všechny metody, tak obecná route funkce a statické route. Kontrolujeme registraci route, tvorbu výsledných cest, včetně přidávání prefixů controllery. Kromě registrací a špatných vstupů jsme se zaměřili také na matchování dynamických cest a také parametrů v cestách. V controlleru testujeme téměř to samé s tím rozdílem, že kontrolujeme zda data uvnitř Controller objektu odpovídají. V middleware modulu prověřujeme, zda filter funkce jsou správně volány před zavoláním route funkce, zda mají přístup k Request a Response objektům, a zda dokáží dotazy ukončit. Jako poslední máme modul na zprávy, kde především zkoušíme správnost parsování příchozích dat. Seznam testů je v tabulce 6.1.

Route testy	
test_add_get_route	Test správné registrace GET route.
test_add_post_route	Test správné registrace POST route.
test_add_put_route	Test správné registrace PUT route.
test_add_delete_route	Test správné registrace DELETE route.
test_add_route	Test správné registrace obecné route.
test_add_static_route	Test správné registrace statické route.
test_controller_prefix_path	Test přidání prefix cesty controllerem do route.
test_route_none_method	Test route bez metody. Neměla by být zaregistrována.
test_route_without_path	Test route bez cesty. Neměla by být zaregistrována.
test_match_path_success	Test o správnosti shody cesty s dynamickou cestou.
test_match_path_fail	Test o nesprávnosti shody cesty s dynamickou cestou.
test_path_variables	Test správně převzatých parametrů z dynamické cesty.
Controller testy	
test_controller_path	Test správně uložené controllery cesty.
test_add_none_fc	Test přidání None filter funkce. Neměla by být uložena.
test_add_fc	Test přidání filter funkce do controlleru.
test_empty_path_default	Test controlleru bez cesty, bude nastavena na "/default/".
Middleware testy	
test_add_global_filter	Test přidání globálního filteru.
test_filter_not_global	Test o nepřidání controller filtru do globálních filterů.
test_add_controller_filter	Test přidání controller filtru.
test_middleware_is_called	Test provolání filteru před route funkcí.
test_middleware_success	Test úspěšnosti filter funkce.
test_middleware_fail	Test neúšpěšnosti filter funkce. Route není zavolána.
test_consumes_success	Test úspěšnosti kontroly konzumovaného content-type.
test_consumes_fail	Test neúspěšnosti kontroly konzumovaného content-type.
Message testy	
test_response_status_set	Test nastavení HTTP response statusu.
test_response_status_default	Test defaultní hodnoty pro HTTP response status.
test_response_type	Test nastavení response MIME.
test_response_data	Test vložení response dat.
test_response_build	Test sestavení výsledné HTTP. response
test_response_build_default	Test o defaultním stavu Response objektu.
test_parse_content_json	Test parsování JSON stringu do dictionary.
test_parse_unknown_type	Test parsování neznámého formátu.
test_form_data_single_line	Test parsování formulářových dat. Jednoduchý atribut.
test_form_data_multi_line	Test parsování formulářových dat. Víceřádkový atribut.
test_form_data_file	Test parsování formulářových dat. Atribut se souborem.

Tabulka 6.1: Seznam unit testů a popis testované oblasti.

6.2 Uživatelské testy

Narozdíl od unit testů, které zachytili základní funkčnost, tak uživatelské testování dokázalo odhalit již konkrétní problémy, spojené s implementací. Některé z problémů byly odhaleny až v této fázi testování, a to z důvodu práce přímo s čipem, které unit testy odhalit nemohli. Především jsou tím myšleny problémy s pamětí (například obrázek 6.1 s logem o chybě), které byly nalezeny. Přestože Micropython v čipu používá Garbage collector, tak v některých případech byl problém s alokováním paměti.

```
DEBUG:miniweb:Response was marked as builded.
DEBUG:miniweb:Response arrived back to server.py
DEBUG:miniweb:Sending response headers.
DEBUG:miniweb:Header
 Connection: keep-alive
Task exception wasn't retrieved
future: <Task> coro= <Task>
Traceback (most recent call last):
  File "uasyncio/core.py", line 1, in run_until_complete
  File "miniweb/core/server.py", line 67, in __handle
  File "miniweb/message/request.py", line 35, in parse_header
MemoryError: memory allocation failed, allocating 128 bytes
DEBUG:miniweb:Header
 Pragma: no-cache
DEBUG:miniweb:Sending response data
DEBUG:miniweb:Header
 Cache-Control: no-cache
DEBUG:miniweb:Closing communication with client.
```

Obrázek 6.1: Log s výstupem MemorryErroru

Chyby byly založeny do GitLabu (označeny prefixem "MK"- iniciály testera). Spousta z nich byla opravena, na dalších se postupně pracuje. Tester vyzkoušel routování, a pak především statické pro servírování souborů. Jednalo se o HTML, CSS, JS (konkrétně SvelteJS) soubory.

6.3 Nalezené chyby

V této kapitole bude seznam nalezených chyb v průběhu implementace, nálezy z uživatelského testování, či jednotkových testů. Veškeré chyby byly zadávány na GitLab jako issue/incident. V issue je uveden vždy popis chyby, a pokud je uzavřena, tak je zapsána i reference na commit, ve kterém byla chyba opravena (pokud to byla chyba v kódu). Seznam všech chyb je níže v tabulce 6.2.

Název	**Popis**	**Stav**
Routovani problem s matchovanim requestů	Pokud příchozí request odpovídá nějaké routě, ale pokračuje dál (request), pak ho uzná za shodu a začne ho vykonávat.	CLOSED
Dictionary - osetreni klicu	Pokud v dictionary daný klíč vůbec neexistuje, pak spadne na KeyError. Je potřeba toto ošetřit.	CLOSED
Konfiguracni soubor s priponou .env - chyba	V tuto chvíli je pouze podmínka že musí obsahovat *.env*, nikoliv že se jedná o příponu. Může tedy projít i jiný soubor se shodou charaketrů v názvu.	CLOSED
uasyncio - POST bez body	Pokud se dotazuji na HTTP POST metodu a v requestu nepošlu nic v body, pak v některých případech neprojde a Insomnia vrací: "Error: Failed sending data to the peer"Podle logů z miniwebu ovšem byla HTTP response v pořádku odeslána. Je potřeba prozkoumat, kde by mohl být problém.	OPEN
opravit exceptiony	Pokud na něco program spadne, vzdy vrací ConfigParamsException. Je nutné opravit vyhazováný výjimek.	CLOSED

Problém s flashnutím frameworku do mikročipu	Při flashnutí kodu frameworku do mikročipu hází chybu u souboru enumerators.py, ten se nechce nahrát do mikročipu. Je potřeba prověřit a vyřešit tento problém.	CLOSED
Vyšší verze micropythonu	Vyšla nová verze micropythonu - 1.14. Je potřeba současný kod refaktorovat, aby byl funkční v nejnovější verzi.	CLOSED
Import modulů - problém	Import modulu miniweb funguje v UNIXovém portu bezproblémově, ale přímo na mikročipu se některé moduly nedotahují. Problém je způsoben pravděpodobně tím, že čip prohledává složky primitivněji než UNIX. Je potřeba projekt poupravit tak, aby fungoval i v čipu.	CLOSED
MK-1 - static routing ukoncuje provadeni requestu kdyz nenalezne soubor	Pokud statické routování nenalezne soubor, měl by pokračovat program v hledání vhodné routy. Nyní ukončí provádění routy responsem 404.	CLOSED
MK-2 - exception je pozrana v pripade static routingu	Exceptions by měly být logované tak, jak se objeví. Nyní při chybě vracení souboru, server odpoví 404. Soubor se přitom nalezne, ale vyskytne se chyba při vracení obsahu.	CLOSED
MK-3 - respektovat fs omezeni micropython esp8266	Micropython na ESP8266 omezuje velikost pole na 512B při načítání ze souboru. Vracení souboru při statickém routování je potřeba ošetřit na postupné (ve smyčce) načítání a vracení klientovi.	CLOSED
MK4 - response - automaticky prevod dictionary na string	Při sestavování response přidat schopnost automaticky serializovat předané dictionary na JSON.	CLOSED
MK-5 - občasný problém s alokováním paměti	Při vracení statických souborů v některých případech se vrací MemoryError, který říká, že se nealokuje paměť. Tato chyba se děje přímo na čipu, v UNIX portu toto nelze nasimulovat.	OPEN

Tabulka 6.2: Seznam nalezených chyb.

Je zapotřebí také uvést, že chyb by bylo mnohem více, kdyby nebylo jednotkových testů, které se spouštěli vždy před nahráním na GIT. Ty odhalili několik chyb, které byli opraveny okamžitě, a nemusela být na ně zakládána issues v GitLabu. Výsledkem je, že v rámci testování byly nalezeny kritické a vážné chyby, které byly okamžitě opraveny. Díky tomu je aktuální stav frameworku stabilní a vhodný k používání. Určitě se v budoucnu najdou další chyby, ale bude se jednat o méně závažné chyby, které nezabraňují v používání a nebo nastávají jen ve výjimečných případech.

Kapitola 7

Závěr

7.1 Výsledky práce

Cílem práce bylo vytvořit web aplikační framework v Micropythonu, který bude nabízet více funkcionalit než stávající nástroje. Výsledná práce obsahuje analýzu, návrh, samotný kód, testy, soubor dokumentací (readme v angličtině a češtině, pdoc) a ukázky kódu doplňující dokumentaci. Framework je zcela funkční. Umí route, metody, path parametry, query parametry, přijímat FormData či JSON. Dále umí shlukovat route do controllerů, spouštět middleware funkce, statické routery a je do budoucna navržen tak, že rozšiřitelnost bude snadná.

7.2 Splnění cílů

Práce splňuje všechny požadavky ze zadání, tudíž všechny požadované cíle byly dosaženy. Mimo to bylo přidáno několik funkcionalit navíc, které nástroj činí ještě univerzálnejším. Funkcemi jako jsou statické routování, nebo filter funkce, které zvládají daleko více, než měli.

Lze také říci, že výsledný framework umí daleko více, než kombinace konkurenčních řešení a je daleko lépe zdokumentován. Z toho vyplývá, že

je veliká šance, že by mohl zmiňované frameworky nahradit a stát se číslem jedna.

7.3 Další rozšiřitelnost a doporučení

Do budoucna určitě dává smysl framework rozšiřovat, ale pravděpodobně v podobě dodatečných knihoven a s celkovou větší modularitou nástroje. To z důvodu, že se stále musí dbát kódu na velikost kvůli mikročipu.

Ohledně konkrétních vylepšení do budoucna stojí za uvážení SMTP spojené s odesíláním emailů z Micropythonu. Pak také časté a náročné operace, jako je například matchování route, napsat v C a poté zkompilovat do binárního souboru *.mpy*, tím by bylo možné framework mírně zrychlit.

Literatura

[1] "Obrázek rozdílu synchronního, asynchronního a vláknového programování," [cit. 2021-01-31]. [Online]. Available: https://www.velotio.com/engineering-blog/asynchronous-programming-python-an-introduction

[2] "Popis funkce dekorátorů v pythonu," [cit. 2021-03-12]. [Online]. Available: https://morioh.com/p/6a3c60dac526

[3] "Picoweb: Stránka v pypi repozitáři," [cit. 2021-01-28]. [Online]. Available: https://pypi.org/project/picoweb/

[4] "noggin," [cit. 2021-01-31]. [Online]. Available: https://forum.micropython.org/viewtopic.php?t=4061

[5] M. Khamphroo, N. Kwankeo, K. Kaemarungsi, and K. Fukawa, "Micropython-based educational mobile robot for computer coding learning," in *2017 8th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*. IEEE, 2017, pp. 1–6.

[6] N. H. Tollervey, *Programming with MicroPython: embedded programming with microcontrollers and Python*. "O'Reilly Media, Inc.", 2017.

[7] C. Bell, *MicroPython for the Internet of Things*. Springer, 2017.

[8] "Spring framework," [cit. 2021-01-27]. [Online]. Available: https://spring.io/

[9] "Express js," [cit. 2021-01-27]. [Online]. Available: https://expressjs.com/

[10] "Flask," [cit. 2021-01-27]. [Online]. Available: https://flask.palletsprojects.com/

[11] "Baeldung: Spring routing tutorial," [cit. 2021-02-04]. [Online]. Available: https://www.baeldung.com/spring-new-requestmapping-shortcuts

[12] "Spring dokumentace," [cit. 2021-02-04]. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

[13] "Geeks for geeks: req query property," [cit. 2021-02-04]. [Online]. Available: https://www.geeksforgeeks.org/express-js-req-query-property/

[14] "Express js dokumentace," [cit. 2021-02-04]. [Online]. Available: https://expressjs.com/en/guide/routing.html

[15] "Flask dokumentace," [cit. 2021-02-04]. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/quickstart/

[16] "Popis middleware," [cit. 2021-02-08]. [Online]. Available: https://dzone.com/articles/what-is-http-middleware-and-best-practices

[17] "Filtry ve springu," [cit. 2021-02-08]. [Online]. Available: https://www.baeldung.com/spring-boot-add-filter

[18] "Middleware funkce v expressu," [cit. 2021-02-08]. [Online]. Available: https://expressjs.com/en/guide/using-middleware.html

[19] "Middleware funkce ve flasku: článek na portálu medium," [cit. 2021-02-08]. [Online]. Available: https://medium.com/swlh/creating-middlewares-with-python-flask-166bd03f2fd4

[20] "Spring boot configuration: článek na baeldung," [cit. 2021-02-11]. [Online]. Available: https://www.baeldung.com/configuration-properties-in-spring-boot

[21] "Spring boot properties: článek na javatpoint," [cit. 2021-02-11]. [Online]. Available: https://www.javatpoint.com/spring-boot-properties

[22] "Nodejs environment variables," [cit. 2021-02-11]. [Online]. Available: https://www.twilio.com/blog/working-with-environment-variables-in-node-js-html

[23] "Flask konfigurace," [cit. 2021-02-11]. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/config/

[24] "Microdot: Stránka v pypi repozitáři," [cit. 2021-01-28]. [Online]. Available: https://pypi.org/project/microdot/

[25] "uasyncio: dokumentace na oficiálních stránkách micropythonu," [cit. 2021-01-31]. [Online]. Available: http://docs.micropython.org/en/latest/library/uasyncio.html

[26] "uasyncio: stránka v pypi repozitáři," [cit. 2021-01-31]. [Online]. Available: https://pypi.org/project/micropython-uasyncio/

[27] "asyncio: Oficiální dokumentace k cpython knihovně," [cit. 2021-01-31]. [Online]. Available: https://docs.python.org/3.5/library/asyncio.html#module-asyncio

[28] "Popis funkce event loop v asyncio," [cit. 2021-01-31]. [Online]. Available: https://docs.python.org/3.5/library/asyncio-eventloop.html

[29] "Moduly python - oficiální dokumentace," [cit. 2021-01-25]. [Online]. Available: https://docs.python.org/3.5/tutorial/modules.html

[30] "Balíčky a jejich distribuce v pythonu," [cit. 2021-01-25]. [Online]. Available: https://packaging.python.org/guides/distributing-packages-using-setuptools/

[31] "Setup script v pythonu," [cit. 2021-01-25]. [Online]. Available: https://docs.python.org/3.5/distutils/setupscript.html

[32] "Pycharm - vývojové prostředí," [cit. 2021-01-24]. [Online]. Available: https://www.jetbrains.com/pycharm/

[33] "Micropython plugin do pycharmu," [cit. 2021-01-24]. [Online]. Available: https://plugins.jetbrains.com/plugin/9777-micropython

[34] "Odlišnosti pythonu a micropythonu," [cit. 2021-01-24]. [Online]. Available: http://docs.micropython.org/en/latest/genrst/index.html

[35] "Micropip: knihovna v pypi repozitáři," [cit. 2021-02-08]. [Online]. Available: https://pypi.org/project/micropython-upip/

[36] "Návod k instalaci micropythonu pro esp8266," [cit. 2021-01-22]. [Online]. Available: http://docs.micropython.org/en/latest/esp8266/tutorial/intro.html

[37] "Micropython firmware pro esp8266," [cit. 2021-01-22]. [Online]. Available: http://micropython.org/download/esp8266/

[38] "Micropython pro ubuntu," [cit. 2021-02-04]. [Online]. Available: https://snapcraft.io/install/micropython/ubuntu

[39] "Virtualbox," [cit. 2021-02-04]. [Online]. Available: https://www.virtualbox.org/

[40] "Ubuntu: distribuce linuxu," [cit. 2021-02-04]. [Online]. Available: https://ubuntu.com/

[41] "Wsl," [cit. 2021-02-04]. [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/install-win10

[42] "Coroutine v pythonu," [cit. 2021-03-21]. [Online]. Available: https://www.geeksforgeeks.org/coroutine-in-python/

[43] "Používaná logovací knihovna," [cit. 2021-03-21]. [Online]. Available: https://pypi.org/project/micropython-ulogging/

[44] "Java ee: Response," [cit. 2021-03-24]. [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.html

[45] "Algoritmy: Popis dekorátor vzoru," [cit. 2021-03-24]. [Online]. Available: https://www.algoritmy.net/article/1629/Decorator

[46] "voho: Popis dekorátor vzoru," [cit. 2021-03-24]. [Online]. Available: http://voho.eu/wiki/decorator/

[47] "Tipy a návody k optimalizaci micropythonu," [cit. 2021-04-10]. [Online]. Available: https://docs.micropython.org/en/latest/reference/speed_python.html

[48] "Micropython unittest knihovna," [cit. 2021-04-03]. [Online]. Available: https://pypi.org/project/micropython-unittest/

[49] "Nástroj pro generování api dokumentace z pythonu," [cit. 2021-04-11]. [Online]. Available: https://pdoc.dev/

Příloha A

Příručka k frameworku

V této kapitole bude popsáno, jak užívat výsledný framework. Popis instalace bude zkrácený, protože se předpokládá, že vývojář je obeznámen s Micropythonem. Kapitola bude popisovat pouze ty nejzákladnější informace. Pro vývojáře je poté sepsána *README.md* dokumentace (dostupná v čestině i angličtině), ukázky kódů a pdoc[49] v *.html* formátu, kde najdou veškeré informace a detaily.

A.1 Instalace

Do projektu, který bude nahrán do čipu je potřeba stáhnout framework. V tuto chvíli je jako jediná varianta stažení přes gitlab (v budoucnu bude přidána možnost přes micropip). Postačí pouze složka miniweb, ostatní složky nejsou potřebné. Také není nutné stahovat knihovny třetích stran, vše je obsaženo v Micropythonu, jedná se o builtin knihovny. Je nutné používat na mikročipu verzi Micropython 1.14 (v tuto chvíli nejnovější verze), z důvodu kompatibility využívaných builtin knihoven. Starší verze nemusí fungovat správně.

A.2 Použití

V tuto chvíli je miniweb ve stejném projektu jako bude implementovaný kód. Do své implementace naimportujte z miniwebu dostupné funkce, třídy, enumerátory. Lze importovat app, filter, Controller, Mime, Log, Status, Method.

Route se definují skrze dekorátory. Lze použít 4 univerzální funkce, a nebo jednu obecnou - get(), post(), put(), delete() a route(). U obecné je nutné nadefinovat HTTP metodu (či více HTTP metod). Povinným parametrem pro všechny je path. Nepovinný parametr je controller, který může shlukovat route do skupin. Dalším nepovinným parametrem je consumes, které nadefinuje pole akceptovatelných příchozích Content-Type.

```
1  import miniweb
2
3
4  app = miniweb.app()
5
6
7  @app.route("/foo", [Method.GET, Method.POST])
8  def foo(req, res):
9      pass
10
11
12 @app.get("/bar")
13 def bar(req, res):
14     pass
15
16
17 @app.post("/bar")
18 def bar(req, res):
19     pass
20
21
22 @app.put("/bar")
23 def bar(req, res):
24     pass
25
26
27 @app.delete("/bar")
28 def bar(req, res):
29     pass
```

Listing A.1: Ukázka route funkcí.

Do route cesty lze nadefinovat path parametry.

```
from miniweb import app, Log


params = {
    "port": 8000,
    "host": "0.0.0.0",
    "log": Log.INFO,
    "buffer": 128
}


app = app(params)


#path parameters are wrapped in {}
@app.get("name/{name}/surname/{surname}")
def example(req, res, var):
    #if route has path parameters, then route function will have 3 parameters
    #path parameters values will be stored inside of var
    name = var["name"]
    surname = var["surname"]
    pass


#route without path parameter
@app.post("/foo")
def foo(req, res):
    #this route is without path parameter, so route function will have only 2 parameters - req and res
    pass


app.run()
```

Listing A.2: Ukázka path parametrů.

Do route funkce jsou vloženy miniwebem objekty Request a Response. Z Requestu lze přistoupit ke všem HTTP headers, query parametrům či přijatému Contentu z těla HTTP requestu, ten umí zpracovávat JSON a FormData do objektu, aby byla lehce použitelná pro vývojáře. Když miniweb příchozí Content-type nezná, jsou data přístupná pouze jako String.

```
from miniweb import app


#/foo?key=1
#body: {id: 2, arr: ["a", "b", "c"]}

@app.post("/foo"):
def foo(request, res):
    query_parameter_key = request.params["key"]
    request_content = request.content

    #pokud je content json lze pristupovat pres klice

    json_key_id = request.content["id"]


#Content-Type: "multipart/form-data"
# file: image.jpg
# value: 22
# multilinevalue: line1
#                 line2

@app.post("/bar")
def bar(request, res):
    #pokud je content multipart/form-data
    #pristupuje se pres atributy objektu

    file = request.content.file
    value = request.content.value
    multilinevalue = request.content.multilinevalue

    #pristup k headers

    headers = request.headers
  content_type = headers["Content-Type"]
```

Listing A.3: Ukázka přístupu k Request datům.

Response objekt nadefinuje HTTP status, Content-type a samotná data, která se budou vracet. Vše se děje přes builder, když je Response finální (připraven k odeslání), musí být použita metoda build(). Vracet HTTP response není povinné, nemusí být tedy vyplněna.

```
from miniweb import Status, Mime, app
    #ukazka mimo funkci

    r = Response()
    ok = Status.OK
    html = Mime.HTML
    r.status(ok).type(html).entity("<h1>Hi!</h1>").build()


    @app.get("/foo")
    def foo(req, res):

       #lze poslat dictionary, framework se postara
       #o poslani jako json string

       dict = {
         "key": "value",
         "arr": [1, 2, 3, 4]
       }
       json = "application/json"
       res.status(200).type(json).entity(dict).build()
```

Listing A.4: Ukázka nastavení Response dat.

Statické routery slouží k spravování servírování statických souborů na serveru. Lze jich nadefinovat více. Přijímají dva povinné parametry - root, path a jeden nepovinný parametr- controller. Parametr root definuje, která složka na používaném zařízení je ta výchozí. Parametr path naopak definuje jakýsi alias, pod kterým bude daná root cesta zadávána v endpointu.

```
from miniweb import app
    app = app()


    app.static_router(root="/var/www/", path="/file/")
    #http://localhost/file/subfolder/index.html
    # vrati
    #/var/www/subfolder/index.html


    app.static.router(root="/var/user/www/", path="/")
    #http://localhost/index.html
    # vrati
    #/var/user/www/index.html
```

Listing A.5: Ukázka použití statických route.

Middleware funkce se definuje srkze dekorátory. Obsahuje jeden nepovinný parametr - controller. Middleware funkce jsou funkce, které jsou vykonávány před samotným vykonáváním route funkce, slouží pro kontrolu a ochranu. Pokud nemají controller, pak jsou volány před všemi routami. U controllerů naopak jen u route, které patří pod controller.

```
1 from miniweb import filter
2
3 @filter()
4 def middleware(req, res):
5     return True
6
7 @filter(userController)
8 def is_logged(req, res):
9     return False
```

Listing A.6: Ukázka middleware funkcí.

Controllery slouží pro shromažďování route do skupin. Definuje se jejich cesta. Výsledná cesta route s controllerem je cesta controlleru + cesta route. Mimo prefix cesty také slouží pro definice middleware funkcí právě jen pro controllery.

```
1 from miniweb import Controller, app
2
3 app = app()
4 user_controller = Controller("/user")
5
6 @app.get("/id", controller=user_controller)
7 def get_user_id(req, res):
8     #cesta je "/user/id"
9     pass
```

Listing A.7: Ukázka controlleru.

Konfigurovat lze dvěmi způsoby - v kódu přes konstruktor, nebo přes konfigurační soubor. Pokud jsou vyplněny obě varianty, přednost má parametrizace skrze konstruktor. Při konfiguraci přes soubor je nutné aby nadefinovaný soubor měl koncovku .env.

Název	Popis	Povinný
port	Port serveru, na kterém bude naslouchat.	✓
log	Úroveň logů frameworku.	✓
host	IP adresa pro server. (Pokud se připojuje k wi-fi přes boot.py, pak zadat "0.0.0.0")	✓
buffer	Velikost pole pro čtení souboru. (Závislé na používaném čipu - 128,256,512, ...)	✓

Tabulka A.1: Seznam konfiguračních parametrů.

```
1  from miniweb import app, Log
2
3  params = {
4      "port": 8000,
5      "log": Log.DEBUG,
6      "host": "localhost",
7      "buffer": 512
8  }
9
10 app = app(params)
```

Listing A.8: Konfigurace přes konstruktor.

```
1 #config.env soubor
2 port=8000
3 log=DEBUG
4 host=127.0.0.1
5 buffer=512
```

Listing A.9: Konfigurace přes soubor.

Pro snadnější práce jsou vytvořeny Enumerátory pro Log, HTTP metody, Mime typy a HTTP statusy. Do miniweb funkcí lze uvádět jak běžné hodnoty, tak i hodnoty skrze Enumerátory.

```
#misto Log.DEBUG lze zapsat 10
params = {
    "log": 10
}

#misto Mime.JSON lze zadat "application/json"
#To stejne plati pro Statusy, misto Status.OK lze vyjadrit
   cisly kodu - 200
}
```

Listing A.10: Enumerátory v miniwebu.

Jednotlivé funkce vypadají jednoduše, ale kombinací lze vytvořit složitější konstrukty, které utvoří jeden velký funkční celek.

Příloha B

Seznam použitých zkratek

- miniweb - název implementovaného frameworku
- IDE - Vývojové prostředí
- PyPi - Python Package Index, softwarový repozitář
- npm - Node Package Manager, správce javascriptových balíčků
- HTTP - Hypertext Transfer Protocol
- CSS - Kaskádové styly
- Image - instalační obraz
- WSL/WSL2 - Windows subsystem for Linux
- IoC - Inversion of Control
- API - Application programming interface, aplikační rozhraní
- UML - Unified Modeling Language, grafický jazyk pro vizualizaci
- EA - Enteprise Architect, nástroj pro tvorbu modelů a diagramů
- MIME - Multipurpose Internet Mail Extensions, typ datového souboru
- JSON - JavaScript Object Notation, formát pro zápis dat
- XML - Extensible Markup Language, formát pro zápis dat
- pdoc - Nástroj pro automatickou generaci API dokumentace z Pythonu
- SMTP - Simple mail transfer protocol, protokol pro přenos zpráv elektronické pošty