UNIVERZA V MARIBORU

FAKULTETA ZA ELEKTROTEHNIKO,

RAČUNALNIŠTVO IN INFORMATIKO

Jan Haložan

PRIMERJAVA DIALEKTOV JEZIKA JAVASCRIPT

Diplomsko delo

Maribor, julij 2017

Jan Haložan

PRIMERJAVA DIALEKTOV JEZIKA JAVASCRIPT

Diplomsko delo

Študent:	Jan Haložan
Študijski program:	Univerzitetni študijski program
	Računalništvo in informacijske tehnologije UN
Mentor:	doc. dr. Tomaž Kosar
Lektorica:	mag. Nataša Koražija

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

Številka: E1064342
Datum in kraj: 03. 04. 2017, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Statut UM – UPB 11, Ur. l. RS, št. 44/2015) izdajam

SKLEP O ZAKLJUČNEM DELU

1. **Janu Haložanu,** študentu študijskega programa prve stopnje UN RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati zaključno delo.

2. Tema zaključnega dela je pretežno s področja Inštituta za računalništvo.

3. **MENTOR:** **doc. dr. Tomaž Kosar**

4. **Naslov zaključnega dela:**
   PRIMERJAVA DIALEKTOV JEZIKA JAVASCRIPT

5. **Naslov zaključnega dela v angleškem jeziku:**
   JAVASCRIPT TRANSPILER COMPARISON

6. Rok za izdelavo in oddajo zaključnega dela je 30. 09. 2017. Zaključno delo je potrebno izdelati skladno z "Navodili za izdelavo zaključnega dela" in ga v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) oddati v pristojnem referatu članice. Hkrati se odda tudi izjava mentor-ja/-ice (in morebitnega somentor-ja/-ice) o ustreznosti zaključnega dela.

Pravni pouk: Zoper ta sklep je možna pritožba na Senat članice v roku 10 delovnih dni od dneva prejema sklepa.

Dekan:
red. prof. dr. Borut Žalik

Obvestiti:
- kandidata,
- mentor-ja/-ico,
- odložiti v arhiv.

Primerjava dialektov jezika JavaScript

Ključne besede: javascript, coffeescript, babel, typescript, nodejs

UDK: 004.421(043.2)

Diplomsko delo predstavlja in opravi primerjavo različnih dialektov jezika JavaScript. Osredotočamo se na dialekte CoffeeScript, TypeScript in Babel. Najprej si ogledamo, kako je JavaScript prešel iz zgolj skriptnega jezika za brskalnike v jezik za strežniško programiranje. Nato za delo z različnimi dialekti vzpostavimo razvojno okolje in primerjamo različne urejevalnike. Preučimo okolje NodeJS in implementiramo strežnik v vsakem izmed izbranih dialektov. Vsak strežnik je performančno izmerjen in identificirani so razlogi za in proti uporabi posameznega dialekta. V sklepu podamo ugotovitev, katerega izmed dialektov je najbolj smiselno uporabljati glede na tip projekta.

JavaScript transpiler comparison

Keywords: javascript, babel, coffeescript, typescript, nodejs

UDK: 004.421(043.2)

This thesis focuses on a comparison between different JavaScript transpilers ie. dialects of the JavaScript programming language. We focus on CoffeeScript, TypeScript and Babel dialects. First we review how JavaScript moved from a client side scripting language to a server side programming language. We set up an integrated development environment for working with different dialects and compare different source code editors. We inspect the NodeJS runtime environment and implement a server in each dialect. We measure performance for each server and key pros and cons are identified. In conclusion we identify which dialect is most suited for use given a project type.

KAZALO

KAZALO SLIK

KAZALO TABEL

SEZNAM UPORABLJENIH KRATIC

URL – enotni naslov vira (Uniform Resource Locator)

HTML – jezik za oblikovanje večpredstavnih dokumentov (Hypertext Markup Language)

CSS – kaskadne stilske predloge (Cascading Style Sheets)

OS – operacijski sistem (Operating System)

RAM – delovni pomnilnik (Random Access Memory)

OOP – objektno orientirano programiranje (Object Oriented Programming)

API – aplikacijski programski vmesnik (Application Programing Interface)

JSON – notacija za zapis objektov JavaScript (JavaScript Object Notation)

JS – programski jezik JavaScript (JavaScript)

git – sistem za verzioniranje (version control system)

1 UVOD

Z iznajdbo svetovnega spleta leta 1989 so se pojavile nove razsežnosti v uporabnosti računalnikov. Kmalu po njegovi iznajdbi se je svetovni splet razširil iz zaprtih prostorov organizacije CERN[1] in našel svojo pot v svet. S porastom števila strežnikov in odjemalcev so se začeli pojavljati standardi, kako naj bo vsebina na spletu predstavljena.

Prvi tak standard je bil HTML, ki omogoča izgradnjo spletnih strani in narekuje, kako naj spletna stran izgleda. Jeziku HTML sta se hitro priključila še jezika CSS in JavaScript[2]. Skupaj tvorijo trojico, na kateri je postavljena velika večina svetovnega spleta.

Po razširitvi spleta se je hitro pojavila potreba za večjo dinamiko spletnih strani oziroma želja po tem, da spletne strani prikazujejo več kot samo slike in besedilo. Tukaj nastopi JavaScript. Omogočil je izvajanje kode na strani odjemalca (brskalnika) in tako dinamično spreminjanje vsebine dokumenta, interakcijo s strežnikom, validacijo podatkov ipd. Nastalo je veliko knjižnic za pomoč pri delu, ki olajšajo razvoj, krajšajo razvojni cikel, pohitrijo delovanje ipd. Zaradi priljubljenosti JavaScript jezika in zmogljivih interpreterjev zanj je kasneje svojo pot našel tudi na strežniško stran.

Namen diplomskega dela je predstaviti razlike med posameznimi dialekti JavaScript jezika in primerjava med njimi. Najprej si ogledamo, kako je JavaScript prešel na strežnik. Za tem raziščemo razvojna okolja in jih med sabo primerjamo. Na osnovi kriterijev izberemo najbolj ustreznega, ki ga uporabimo za razvoj. Ogledamo si, kako je zgrajeno okolje NodeJS in zakaj je nastalo. V vsakem izmed izbranih treh dialektov nato implementiramo preprost strežnik, ki izkorišča prednosti posameznega dialekta. Cilj je tako raziskati posamezne dialekte in med njimi poiskati prednosti in slabosti za določen tip projekta.

[1] Evropska organizacija za nuklearne raziskave.
[2] Skriptni programski jezik.

Diplomsko delo sestoji iz treh poglavij. Prvo je pričujoči uvod. V drugem poglavju si bomo najprej ogledali razvojna okolja in izbiro najboljšega glede na naše potrebe. Nato si pogledamo razloge za nastanek različnih dialektov jezika JavaScript in ključne razlike med njimi. Sledi uvod v NodeJS okolje in njegov pregled. V tretjem poglavju bomo implementirali strežnik v vsakem izmed treh dialektov in podrobneje pogledali razlike med njimi. Sledi merjenje hitrosti in vrednotenje rezultatov.

1.1 JavaScript na strežniku

Z eksplozijo knjižnic za JavaScript in njegove široke uporabnosti na strani odjemalca ter zaradi priljubljenosti jezika samega je JS kaj hitro našel svojo pot na strežniško stran. Vendar se na strežniku pojavlja precej drugačen nabor problemov in tehničnih težav kot na odjemalcu. Sam jezik je ostal popolnoma nespremenjen, bilo mu je pa dodanih precej knjižnic in funkcionalnosti, ki ga naredijo uporabnega za kodiranje na strežniku.

S porastom priljubljenosti strežniškega JavaScripta in s tem povezano rastjo baze uporabnikov so se za pomoč razvijalcem začele pojavljati povsem drugačne tehnologije in pristopi. To velja tako za razvojni proces kot funkcionalnosti, ki jih JavaScript na strežniku ponuja.

2 RAZVOJNA OKOLJA

Da lahko začnemo delo, si je zelo smiselno postaviti razvojno okolje, s pomočjo katerega si olajšamo razvojni proces. Ker smo v našem diplomskem delu delali z različnimi dialekti jezika JavaScript, smo postavili okolje, ki omogoča delo z vsemi izbranimi dialekti in ni omejeno na operacijski sistem oz. podpira vse večje distribucije (Windows, Linux in macOS).

Zahteve, ki smo jih imeli za razvojno okolje, so:

- barvanje sintakse,
- podpora razhroščevanja,
- sintaktična analiza kode,
- zaželena je podpora git,
- podpora vsem trem dialektom,
- kompatibilnost z različnimi operacijskimi sistemi.

Na spletu lahko najdemo veliko različnih okolij, ki ustrezajo zgornjim zahtevam. Načeloma vzpostavitev okolja, ki nam podpre omenjene zahteve, ni zahtevna. Bolj težavno je izbrati pravo okolje, da bo ustrezalo našemu razvojnemu procesu. Veliko programov podpira tudi tako imenovane vtičnike, s katerimi lahko razširimo osnovni nabor funkcionalnosti. Tako lahko določene programe dopolnimo, če nam ne nudijo vseh potrebnih funkcij. Druga možnost je poiskati nadomestni program, ki nam opravlja izbrano funkcijo.

2.1 Pregled razvojnih okolij

Od razvojnega okolja smo zahtevali, da podpira zgoraj naštete funkcionalnosti in da nam kot razvijalcu čim bolj ustreza. Obstaja mnogo kombinacij in v osnovi se moramo odločiti, ali želimo lažjo rešitev, ki na začetku ne podpira vsega in jo lahko dopolnimo, da ustreza našim zahtevam, ali pa poiščemo orodje, ki že v osnovi ponuja vse, kar potrebujemo. Pri lažjih rešitvah to pomeni hitrejše programe, s katerimi lahko imamo malenkost več težav, ko želimo dodati nekaj, česar program v osnovi ne podpira (npr. razhroščevanje). Večji in kompleksnejši programi pa so načeloma počasnejši in težje prilagodljivi.

Primer zgoraj omenjenih možnosti je:

- program PhpStorm[3], ki vključuje razhroščevalnik, sintaktično analizo, podpira več dialektov, vsebuje git sistem in vsebuje še precej dodatnih funkcionalnosti. Vendar smo pri taki izbiri omejeni na ponujen nabor funkcionalnosti in si način dela težko prilagodimo;
- kombinacijo manj zmogljivega programa, ki ga dopolnimo, da dosega naše zahteve. Kot primer lahko vzamemo Sublime Text, ki je zelo hiter in lahkoten urejevalnik (ne zaseda dosti resursov), mu dodamo podporo za naše dialekte preko vtičnikov, dodamo razhroščevalnik preko vtičnika, dopolnimo s podporo za git ali uporabljamo kateri drug program. Tovrsten način je zelo prilagodljiv in nam omogoča, da postavimo delovni proces, ki nam ustreza.

Obstaja veliko primernih programov, zato imamo res široko izbiro. Kot širše kandidate smo si ogledali:

- Atom,
- Sublime Text,
- Visual Studio,
- Spacemacs,
- TextWrangler,
- Coda,
- VS Code,
- Nodepad++,
- PhpStorm,
- NetBeans.

Za ožji izbor smo programe v glavnem izbrali glede na skupnost (angl. community), ki stoji za programom, ceno, ustreznostjo in preprostostjo uporabe. Orientirali smo se tudi na programe, ki so v glavnem mišljeni za strežniško programiranje, oz. si jih lahko prilagodimo,

[3] https://www.jetbrains.com/phpstorm/.

da kar se da dobro opravljajo omenjeno funkcijo. Pogosto se sicer zgodi, da zaledni (angl. back-end) razvijalci pomagajo tudi na čelni (angl. front-end) strani, zato je bil eden izmed začetnih kriterijev tudi široka uporabnost programa.

Programi, ki smo jih izločili v širšem izboru, so:

- Notepad++; gre že za precej star program, ki se ga uporablja vedno manj, saj nam konkurenčna orodja ponujajo precej več. Ne moremo si namestiti razhroščevalnika za NodeJS, ki je bil pogoj za izbiro programa.
- Spacemacs; precej priljubljen med vzdrževalci strežniških sistemov, saj gre za urejevalnik, zgrajen na poznanem terminalnem urejevalniku »emacs«, odprtokoden in vzdrževan s strani skupnosti. Zanj se nismo odločili, saj ne ponuja vtičnikov in v osnovi ne nudi vsega, kar smo potrebovali (npr. razhroščevalnika za NodeJS).
- TextWrangler; podobno kot Spacemacs žal ne ponuja dovolj vtičnikov, da bi ga lahko dodelali za podporo npr. razhroščevanju.
- Coda; glavni faktor za neizbor je sorazmerno visoka cena (program stane 100 €) ter usmerjenost za front-end razvoj s podporo back-endu.
- Visual Studio; edini razlog za neizbor tega razvojnega okolja v ožji izbor je njegova omejenost na Windowsov operacijski sistem. Naše delo je potekalo na operacijskem sistemu macOS, za katerega Visual Studio žal ni na voljo.

2.2 Primerjava razvojnih okolij

Za ožji izbor smo izbrali 5 programov. V spodnji preglednici lahko vidimo osnovne funkcionalnosti, na podlagi katerih smo izbrali okolje, s katerim smo delali.

Tabela 2.1: Primerjava razvojnih okolij glede na funkcionalnosti.

Program / Metrika	Atom	Sublime Text	VS Code	Net. Beans	PhpStorm
Cena	0	$70	0	0	$6 / mesec
Odprtokodni	Da	Ne	Da	Da	Ne
Barvanje sintakse	Da	Da	Da	Da	Da
Razhroščevanje	Vtičniki	Vtičniki	Vtičniki	Vtičniki	Da
Sintaktična analiza	Vtičniki	Vtičniki	Da	Da	Da
Git podpora	Da	Da	Da	Da	Da
Podpora več jezikom	Vtičniki	Vtičniki	Ne	Vtičniki	Da
Multi OS	Da	Da	Da	Da	Da

Razvidno je, da so vsi programi ustrezali našim zahtevam. V našem primeru smo se odločili za Atom [1]. Gre za sorazmerno nov urejevalnik[4], ki je pa zelo hitro pridobil širšo skupnost uporabnikov. Je izjemno prilagodljiv in zaradi njegove odprtokodnosti si lahko prilagodimo popolnoma vsak aspekt. Tudi razvit je bil z namenom, da si uporabniki sestavijo razvojno okolje po lastnih potrebah. Vgrajen ima sistem vtičnikov, kjer lahko kdorkoli razvije vtičnik, ki rešuje določen problem in ga deli z drugimi.

Kot zanimivost naj omenimo, da je Atom v celoti spisan s spletnimi tehnologijami, torej HTML, CSS in JavaScript. Zgrajen je v ogrodju Electron [2], ki omogoča spletnim aplikacijam

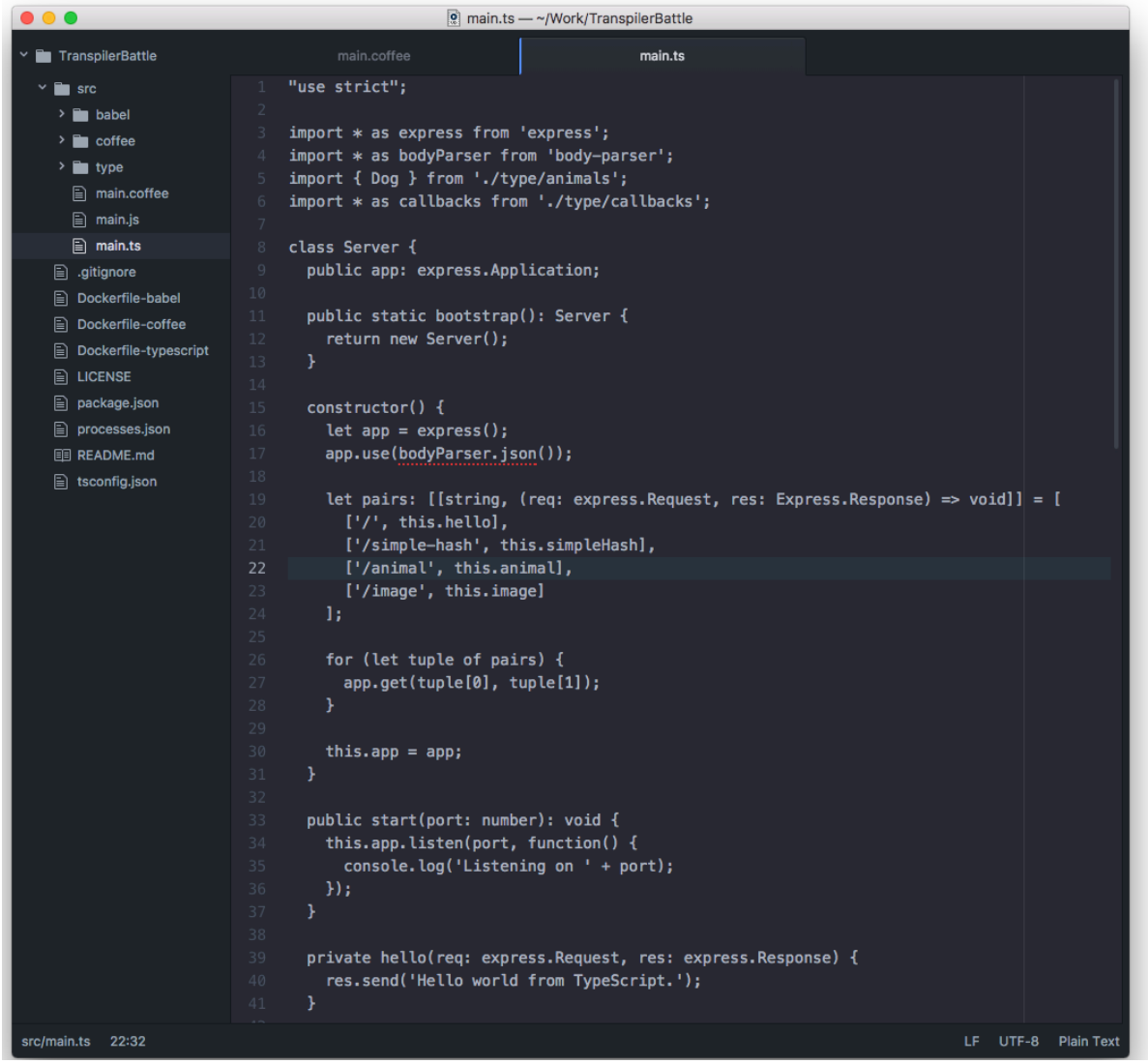
[4] Pojavil se je leta 2014.

delovanje kot namiznim programom. V ozadju je NodeJS, ki ponuja dostop do datotečnega sistema in skrbi za dostopnost sistemskih funkcij, medtem ko front-end poganja Chromium brskalnik, ki je prilagojen tako, da se tesno sklopi z NodeJS. Celotno ogrodje Electron je odprtokodno in uporabljeno v veliko drugih programih in aplikacijah, ki so spisane v spletnih tehnologijah, a delujejo na namizju in se obnašajo kot namizni programi.

2.3 Namestitev in prilagoditev

Naše delo je potekalo na računalniku Apple Macintosh, ki poganja operacijski sistem macOS Sierra. Namestitev programov na sistemu macOS je preprosta, saj gre največkrat za paket, ki že sam po sebi vsebuje vse potrebno in ni potrebna nobena namestitev. Enako velja za Atom. S spleta ga lahko snamemo zastonj na uradni spletni strani[5]. Ko ga prenesemo, nas pričaka stisnjen arhiv, ki po ekstrakciji vsebuje Atom. Tega preprosto povlečemo med aplikacije in je pripravljen za uporabo. Brez dodatne namestitve vtičnikov je naš projekt razviden s slike 2.1.

[5] www.atom.io



Slika 2.1: Atom urejevalnik z odprtim projektom.

Po želji bi lahko tako začeli kodiranje. Vendar Atom ponuja precej več preko vtičnikov [1]. Za naše delo smo potrebovali naslednje:

- podporo JavaScript jezika,
- podporo TypeScript jezika,
- podporo CoffeeScript jezika,
- razhroščevalnik za JavaScript,
- če je mogoče, še razhroščevalnik za TypeScript in CoffeeScript,
- podporo git,
- razne dodatke, ki nam olajšajo razvoj.

Vse izmed naštetega je na voljo preko vtičnikov. Izjema sta TypeScript in CoffeeScript razhroščevalnika, ki na žalost ne obstajata. Razhroščevanje TypeScript je mogoče v orodju Visual Studio, a ta ne podpira ostalih dialektov na način, kot bi si ga želeli. Razhroščevalnik za CoffeeScript je v nastajanju, vendar je v preveč zgodnjih fazah, da bi ga lahko uporabili.

Namestitev vtičnikov je zelo preprosta; treba je iti med nastavitve in odpreti zavihek za nameščanje. Potem vpišemo ime vtičnika in ga namestimo. Za naše delo smo namestili naslednje vtičnike:

- atom-typescript; podpora za jezik TypeScript, preverjanje tipov, sintaktična analiza in pomoč pri pisanju;
- node-debugger; razhroščevalnik za NodeJS;
- language-coffee-script; podpora za jezik CoffeeScript in pomoč pri pisanju;
- language-javascript; podpora za jezik JavaScript in pomoč pri pisanju.

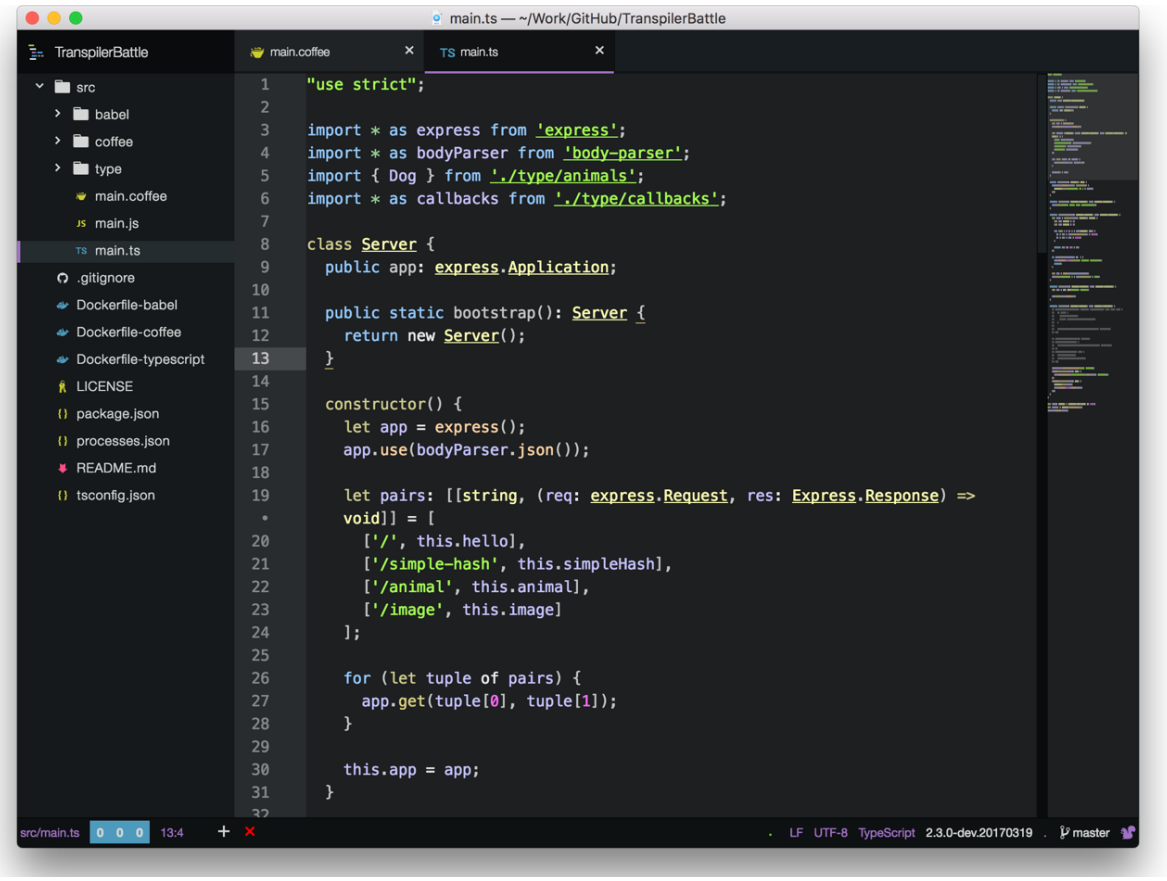
Za olajšanje dela smo dodatno namestili še:

- seti-ui; barvno temo za Atom;
- advanced-open-file; omogoča lažje delo z datotekami;
- language-docker; podpora pisanja datotek tipa Dockerfile;
- minimap; lažji pregled kode;
- pretty-json; lažje delo z JSON-podatki, omogoča formatiranje in podobno;
- tool-bar; komplet bližnjic za hitrejši dostop.

2.4 Namestitev razhroščevalnika za NodeJS

Vtičnik node-debugger za svoje delovanje potrebuje nameščeno okolje NodeJS. Namestitev obdelamo v razdelku 4.5.

Po nameščanju in konfiguraciji vseh vtičnikov je izgled našega okolja razviden s slike 2.2.



Slika 2.2: Atom urejevalnik s polno konfiguracijo.

Tako pripravljeno okolje je primerno za delo.

3 DIALEKTI JEZIKA JAVASCRIPT

Kot že omenjeno v uvodu, je prenos jezika, ki je bil sprva mišljen za preproste skripte na odjemalcu, na strežniško stran prinesel precej tehničnih težav in izzivov. Kot primer lahko pogledamo velikost projektov. V večini primerov je na strežniški strani bistveno več kode kot na odjemalcu. JavaScript sam po sebi ni tipiziran, kar nas lahko pogosto privede do težav.

Tako je nastalo precej dialektov, ki se trudijo premostiti težave osnovnega jezika JavaScript in mu dodati čim več gradnikov, ki naredijo razvoj lažji, hitrejši in vnesejo manj napak v proces razvoja.

JavaScript je verzioniran jezik, kar pomeni, da se dograjevanje funkcionalnosti dela z uvedbo nove verzije. Verzije so vodene s specifikacijami, imenovanimi ECMAScript.

3.1 ECMAScript

ECMAScript je standard, ki določa funkcionalnosti in zmogljivosti jezika, ki implementira ta standard [4]. Po pojavitvi jezika JavaScript leta 1995 se je ta prvič standardiziral leta 1997 v "ECMAScript first edition". Trenutno razvoj poteka tako, da se preko standarda ECMAScript določi nabor funkcionalnosti, in ko je ta popoln, se izda nova različica standarda. Za tem jo začnejo podpirati brskalniki (njihovi JavaScript interpreterji) in ostala okolja.

Trenutno najširše podprta je različica ECMAScript 5 (skrajšano imenovana ES5), ki se je pojavila leta 2009. Z verzijo ES5 je JavaScript doživel razcvet in pojavilo se je veliko zahtev in želj po novih funkcionalnostih. Te so bile dodane v standardu ECMAScript 2015 (skrajšano imenovan ES6), ki velja za veliko spremembo glede na ES5.

Večje novosti vključene v ES6 so:

- razredi in moduli,
- obljube (promises),
- iteratorji in zanka `for of`,
- generatorji,
- kolekcije (collections),
- tipizirana polja,
- reflekcija (reflection).

Kot lahko vidimo, gre za veliko dograditev prejšnje verzije, kar je eden izmed glavnih razlogov, da se podpora standarda v brskalnikih in drugje premika počasi. Vse izmed naštetih funkcionalnosti nam precej olajšajo delo in nam pridejo zelo prav tako na odjemalcu kot na strežniški strani.

Eden izmed razlogov za pojavitev dialektov je podpora vseh funkcionalnosti, ki jih ima standard ECMAScript 2015, a še v trenutnih JavaScript interpreterjih niso podprte.

S pojavitvijo standarda ES6 so se prejšnji različici dodale precejšnje zmogljivosti, kar je velik razlog, da so si programerji želeli podporo na brskalnikih in strežniški strani čim prej. Ob nastanku tega diplomskega dela ni imel še noben brskalnik kompletne podpore za ES6, kar velja tudi za NodeJS. Se pa da posamezne funkcionalnosti pridobiti s poenostavitvijo koncepta in kodiranjem pomožnih funkcij, ki nadomeščajo funkcionalnosti ECMAScript 2015. To je bila glavna ideja pri nastanku Babel.

3.2 Babel

Babel je dialekt jezika JavaScript, ki izgleda popolnoma enako kot JavaScript, vendar doda vse funkcionalnosti, ki jih ponuja standard ECMAScript 2015 [5]. Pred izvajanjem je treba jezik prevesti, kar pomeni, da vse funkcionalnosti iz ES6 prevede v ekvivalente jezika JavaScript (standard ES5) [6].

Priljubljen je zato, ker je kodiranje z njegovo uporabo popolnoma enako kot pri JavaScriptu. Pišemo v bistvu JavaScript, ki ima podporo ES6. Zaradi tega se ni treba učiti nobenih novih lastnosti jezika, ampak le novosti ES6. Ko se na brskalnikih ali strežniški strani pojavi kompletna podpora za novejši standard, lahko začnemo uporabljati izvorne datoteke brez, da jih prevajamo z Babelom.

Razvojni proces z uporabo Babel lahko razdelimo:

- Babel prevajalnik namestimo preko upravitelja odvisnosti npm (podrobneje opisan v razdelku 4.2),
- datoteke imajo končnico `.js` kot navaden jezik JavaScript,
- z uporabo Babel prevajalnika prevedemo naše datoteke v njihov JavaScript (ES5) ekvivalent:
    - primer: `babel <vhodna_mapa> --out-dir <izhodna_mapa>`,
- generirane datoteke so pripravljene za uporabo.

3.3 CoffeeScript

CoffeeScript je dialekt, ki se je pojavil leta 2009. Njegov namen je skrajšati razvojni čas in premostiti manke pri funkcionalnosti jezika JavaScript [7]. Takoj ob pogledu na njegovo sintakso lahko opazimo precejšnjo razliko glede na JavaScript. Cilj drugačne sintakse je čim bolj izpostaviti funkcionalne dele jezika in odstraniti ves balast, ki se smatra za nepotrebnega. Tako na primer jezik odstrani oklepaje za začetek funkcij ali zank in uporablja zamike za določitev gnezdenja kode.

Primer, kako se skrajša koda z uporabo CoffeeScripta, lahko vidimo na preprosti funkciji, ki izračuna največji skupni delitelj dveh števil.

JavaScript:

```
function greatestCommonDivisor(x, y)
{
    var tmp;
    do
    {
        tmp = x % y;
        x = y;
        y = tmp;
    } while (y != 0);

    return tmp;
}
```

CoffeeScript:

```
greatestCommonDivisor = (x, y) ->
    [x, y] = [y, x % y] until y is 0
    x
```

Še primer, v kaj se zgornja koda dejansko prevede:

```
var greatestCommonDivisor;
greatestCommonDivisor = function(x, y) {
  var ref;
  while (y !== 0) {
    ref = [y, x % y], x = ref[0], y = ref[1];
  }
  return x;
};
```

Razlika je znatna, saj je CoffeeScript koda precej krajša, kar zmanjša možnost napak in pohitri razvojni proces. Poleg tega smo uporabili tudi funkcionalnosti, ki jih sam JavaScript ne podpira, kot je v tem primeru destrukturirano prirejanje spremenljivk.

Zanimivost CoffeeScript je tudi, da je popolnoma združljiv s klasičnim JavaScriptom, kar pomeni, da so vse uporabe CoffeeScript gradnikov opcijske. V skrajnem primeru tako prevajamo JavaScript v JavaScript brez sprememb.

Poleg prevajanja v JavaScript ima CoffeeScript priložen tudi interpreter, s katerim lahko poganjamo CoffeeScript kodo neposredno. V ozadju se še vedno zgodi prevod v JavaScript (datoteka se naloži v RAM) pred interpretacijo, ampak je tovrstno prevajanje uporabniku skrito. Tako lahko celoten projekt pišemo v CoffeeScript in ga neposredno tudi poženemo. To storimo z uporabo ukaza `coffee nasa_datoteka.coffee` namesto `node nasa_datoteka.js`.

Proces razvoja je podoben kot pri uporabi Babela:

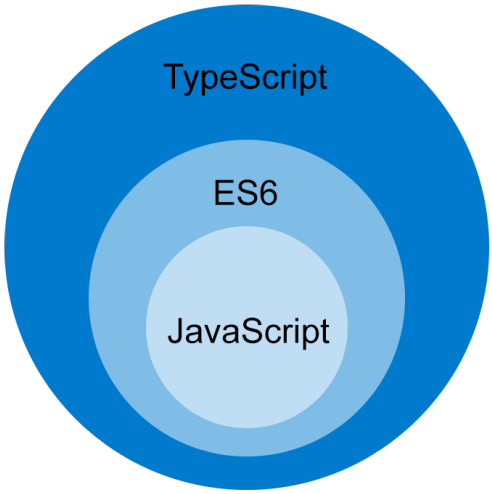
- namestimo CoffeeScript jezik preko uporabe npm,
- datoteke imajo končnico `.coffee`,
- prevedemo datoteke s CoffeeScript prevajalnikom:
    - primer: `coffee --output <izhodna> --compile <vhod>`,
- dobimo prevedene JavaScriptove ekvivalente, ki jih lahko poganjamo.

3.4 TypeScript

Morda največji preskok od osnovnega jezika JavaScript in standarda ECMAScript 2015 je naredil dialekt TypeScript [8]. Kot že ime pove, nam ponudi tipe oz. spremeni naše kodiranje tako, da nam ponudi strogo poimenovane tipe spremenljivk.

Preko sistema tipov nam tako prevajalnik že v času prevajanja pove, kje smo naredili napake, kar nam pri velikih projektih zelo pomaga pri omejevanju števila napak. Zelo pogosta napaka pri netipiziranih jezikih je prav ta, da programer pričakuje drugačen tip, kot se v spremenljivki nahaja v času izvedbe [9]. Tovrstne napake v TypeScriptu niso mogoče oz. jih moramo narediti zavestno [10].

Poleg omenjenega nam TypeScript ponudi vso funkcionalnost, ki se nahaja v standardu ES6. Abstrakten pogled funkcionalnosti dialekta je razviden s slike 3.1.



Slika 3.1: Nabor funkcionalnosti dialekta TypeScript

Zaradi sistema tipov je uporaba tretjerazrednih (angl. third-party) knjižnic malenkost bolj komplicirana, saj moramo poleg knjižnice namestiti še slovar tipov, ki nam jih knjižnica ponudi. To pomeni, da če želimo uporabljati tretjerazredne knjižnice, moramo prevajalniku povedati, kakšni tipi se v njih nahajajo in kakšne signature imajo funkcije. Za večino večjih in bolj razširjenih knjižnic je stvar preprosta, saj tovrstni slovarji že obstajajo, če pa uporabljamo kaj bolj eksotičnega pa si ga moramo pripraviti sami ali pa ne uporabljati tipov, kar je pa načeloma glavna prednost TypeScripta.

Na srečo je postopek uvoza slovarja tipov precej poenostavljen od TypeScript verzije 2 dalje. Prevajalniku pripravimo konfiguracijsko datoteko tipa json, ki vsebuje vse napotke, kje se omenjeni slovarji nahajajo. Prav tako lahko s tovrstno datoteko nastavimo tudi druge aspekte pri prevajanju.

Še ena večja razlika jezika TypeScript je uporaba objektno orientiranega programiranja. Leto ni obvezno, vendar je priporočeno in sam jezik nas napeljuje k uporabi objektov.

Uporaba in prevajanje TypeScript datotek sta naslednja:

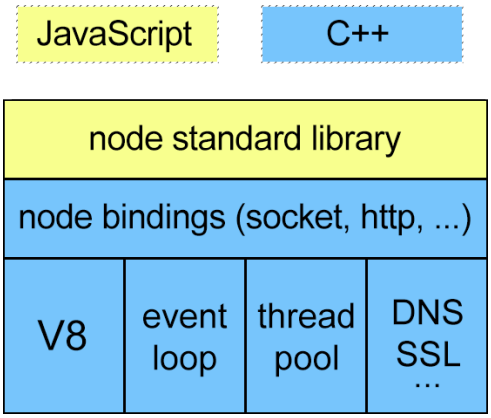
- namestimo TypeScript jezik z uporabo npm,
- datoteke imajo končnico `.ts`,
- pred prevajanjem moramo poskrbeti za pravilno nastavljen tsconfig.json,
- prevedemo datoteke s TypeScript prevajalnikom:
  - primer: `tsc --outDir <izhod>. Ostale nastavitve so v tsconfig.json`,
- dobimo JavaScript ekvivalent, ki je pripravljen na uporabo.

4 NODEJS

Že večkrat smo dejali, da se je JavaScript razvil z namenom izvajanja logike na strani odjemalcev. To pomeni, da je bila koda JavaScript vgnezdena v HTML-kodo spletne strani, ki jo je odjemalec izvedel, ko je prejel spletni dokument s strežnika. NodeJS prenese JavaScript na strežnik, kjer se izvede, preden se rezultat poizvedbe vrne odjemalcu [11]. Združuje front-end del, ki v glavnem vključuje JavaScript interpreter, in back-end, ki omogoča dostop do sistemskih funkcij in datotečnega sistema.

Front-end temelji na Googlovem JavaScript interpreterju Chrome V8[6], ki je del odprtokodnega projekta Chromium. V8 je tako kot Chromium odprtokoden in uporabljen tudi na drugih večjih projektih, kot so MongoDB, Couchbase in Electron. V8 prevaja JavaScript direktno v strojno kodo, ki jo nato izvaja. Prevedena strojna koda je pred izvedbo še dodatno dvakrat optimizirana na osnovi hevristik. Podprt je na arhitekturah, ki temeljijo na IA-32, x86-64, ARM, MIPS ISA, PowerPC in IBM s390.

Back-end deluje na principu modulov, ki ponujajo funkcije, kot so: dostop do datotečnega sistema, kriptografija, mreženje (HTTP, TCP/UDP, TLS/SSL, DNS ...), tokovi podatkov (angl. data streams) in podobno. Tako lahko programer vključi in uporablja zgolj pakete oz. module, ki jih potrebuje. Groba struktura je razvidna na sliki 4.1.



Slika 4.1: Struktura okolja NodeJS

[6] JavaScript interpreter, https://developers.google.com/v8/.

Glavna razlika NodeJS in ostalih strežniških okolij je teženje k temu, da so vse funkcije neblokirajoče, kar pomeni, da se, ko pokličemo funkcijo, ki npr. bere iz datoteke, naša koda izvaja dalje. Ko se branje datoteke konča, dobimo rezultat v anonimni funkciji kot povratni klic (angl. callback). Razliko lahko vidimo, če NodeJS kodo primerjamo s PHP. Za primer spodaj predpostavimo, da napak pri branju ne bo.

NodeJS:

```
function parseFileIntoJson(path, doneCallback)
{
    fs.readFile(path, 'utf8', function(data, error) {
        var json = JSON.parse(data);
        doneCallback(json);
    });
}
```

Ob klicu funkcije `parseFileIntoJson` se v njeni implementaciji pokliče funkcija `readFile` na modulu `fs`. Funkcija `readFile` je asinhrona, kar pomeni, da se dejansko branje zgodi v ozadju in o rezultatu smo obveščeni v anonimni funkciji, ki smo jo podali kot tretji parameter klica. Če bi za klic funkcije `readFile` dodali še več kode, bi se ta začela izvajati nemudoma po klicu `readFile`.

PHP:

```
function parseFileIntoJson($path)
{
    $data = file_get_contents($path);
    $json = json_decode($data, true);
    return $json;
}
```

Za razliko od NodeJS je klic funkcije `file_get_contents` blokirajoč, kar pomeni, da naša koda stoji, dokler se branje ne zaključi. Po zaključku branja se pokliče naslednja vrstica, ki v našem primeru podatke pretvori v PHP-objekt.

Če premislimo in zgornji primer preslikamo na spletni strežnik, bi to pomenilo, da ko strežnik bere datoteko, čaka na zaključek branja in vmes ne more servirati novih zahtevkov. Za podporo serviranja več zahtevkom hkrati bi morali implementirati nitenje, kar pri NodeJS ni potrebno.

4.1 Namestitev

Da lahko začnemo z uporabo NodeJS, ga moramo najprej namestiti. Najlažji način namestitve je, da z uradne spletne strani[7] snamemo namestitveni paket, vendar smo za potrebe naše naloge ubrali drugačen način. Delali smo na sistemu macOS Sierra, ki privzeto nima sistemskega upravitelja paketov (angl. package manager system), zato smo najprej namestili le-tega. To nam je poenostavilo nameščanje NodeJS in nam pogosto koristi tudi v primerih, ko moramo namestiti še kak drug program (npr. git, wget, vim, httperf ...). Upravitelj paketov se imenuje homebrew[8] in je na voljo brezplačno. Je odprtokoden in vzdrževan s strani skupnosti [12].

Po namestitvi Homebrew NodeJS namestimo z naslednjim ukazom:

```
brew install node
```

Ukaz nam namesti NodeJS in njegovega upravitelja odvisnosti imenovanega npm (node package manager – več o njem v naslednjem podpoglavju). Preprost test, s katerim preverimo delovanje, predstavljamo v nadaljevanju:

```
node -e "console.log('hello world');"
```

Če je vse nameščeno pravilno, se nam bo v konzoli izpisal tekst »hello world«. Tako nameščen NodeJS je pripravljen za uporabo.

[7] www.nodejs.org.
[8] www.brew.sh.

4.2 Upravitelj odvisnosti npm

Kot omenjeno, se nam je pri namestitvi NodeJS dodatno namestil še upravitelj odvisnosti npm. Ta je priročen, saj ima NodeJS vzpostavljen velik ekosistem paketov, ki nam ponudijo številne funkcionalnosti, ki jih sam NodeJS nima in so na voljo preko npm.

Ker smo za potrebe naloge implementirali spletni strežnik v vsakem izmed dialektov, smo preko npm namestili nekaj paketov, ki so nam olajšali delo.

Npm podpira dva tipa namestitve paketa. Prvi je lokalni, ki pomeni, da je paket nameščen za uporabo v trenutnem projektu. Medtem ko je drugi globalni in nam namesti paket za uporabo v konzoli (podobno kot homebrew). Za potrebe lokalnega nameščanja npm podpira tudi konfiguracijsko datoteko, v kateri lahko povemo, katere pakete želimo namestiti in lahko tako namestimo več stvari hkrati. Po navadi jo poimenujemo "package.json" in vsebuje vse metapodatke o našem projektu. Poleg namestitve paketov lahko preko npm zaženemo tudi naš projekt ali ga damo na voljo drugim v npm skupnosti.

Za potrebe naše naloge smo namestili naslednje pakete:

- `express`; olajša delo s spletnimi strežniki in usmerjanjem zahtev (endpoint routing);
- `body-parser`; olajša delo s pridobivanjem podatkov iz zahtevkov (GET in POST);
- `request`; olajša ustvarjanje zahtevkov iz našega strežnika na druge;
- `sharp`; podpre delo s slikami (pomanjševanje, spreminjanje formata ...);
- `bluebird`; podpre uporabo obljub (angl. promises) na paketih, ki jih ne podpirajo.

Pakete smo nameščali z uporabo ukaza:

```
npm install --save <ime_paketa>
```

Zgornji ukaz nam poleg namestitve paketa doda tudi zapis v `package.json` (`--save` zastavica) v razdelek `dependencies`.

Kot smo že omenili, moramo za pravilno delovanje TypeScript dialekta dodatno namestiti še slovarje tipov za module, ki jih uporabljamo. Ker tovrstne slovarje potrebujemo samo, ko razvijamo našo aplikacijo, smo jih dodali v `devDependencies`. Poleg nameščanja slovarjev za zgoraj naštete pakete smo morali namestiti še slovar za NodeJS. To smo storili z ukazom:

```
npm install --save-dev types/<ime_paketa>
```

Zastavica `--save-dev` povzroči, da se zapis v `package.json` doda v razdelek `devDependencies`.

4.3 Zagon programov

Z NodeJS lahko izvajamo naše skripte na več različnih načinov. Najlažji je preko direktne uporabe ukaza node v ukazni vrstici. Primer:

```
~:node nasa_skripta.js
```

Ta način je popolnoma zadovoljiv, ko izvajamo lažje skripte, ki ne trajajo dolgo. Pri uporabi tega načina moramo le paziti, da smo namestili vse potrebne odvisnosti, ki jih naš program uporablja. Zato obstaja malenkost bolj optimalen način preko uporabe npm, in sicer lahko z njim poganjamo tudi skripte. Tako lahko v `package.json` dodamo razdelek scripts:

```
»scripts«: {
    »prestart«: »npm install«,
    »start«: »node nasa_skripta.js«
}
```

Sedaj lahko našo skripto zaženemo kot:

```
~:npm start
```

Vsakič, ko to storimo, npm najprej izvede polje `prestart`, ki v našem primeru poskrbi, da so nameščene vse odvisnosti, ki jih uporabljamo. Za tem se izvede polje `start`, ki je enako kot v prvem primeru, in sicer požene skripto preko ukaza `node`. Za naše potrebe smo polja prestart uporabili za prevod dialektov v JavaScript.

Večja težava nastopi, ko želimo na zgornja načina poganjati spletni strežnik. Medtem ko bi tovrstna načina delovala povsem normalno, nista optimalna, saj v primeru napake v kodi naš strežnik pade in se samodejno ponovno ne vzpostavi. Za rešitev tega problema lahko uporabimo še en paket, ki ga ponuja NodeJS skupnost, konkretneje pm2[9]. Pm2 je upravitelj procesov (angl. process manager), ki ga lahko nastavimo, da v primeru napake samodejno ponovno zažene strežnik.

Namestili smo ga z naslednjim ukazom:

```
npm install -g pm2
```

Zaradi zastavice `-g` bo pm2 nameščen globalno. Torej nam je po namestitvi na voljo v ukazni vrstici pod ukazom `pm2`. Za uporabo moramo le malenkost spremeniti `start` razdelek skript za naš npm.

```
 »scripts«: {
    »prestart«: »npm install«,
    »start«: »pm2 start nasa_skripta.js«
}
```

Tako ob primeru napake pm2 samodejno poskrbi za ponovni zagon. Po želji bi lahko uporabo ukaza `pm2 start` prilagodili še z dodatnimi zastavicami, vendar osnovni ukaz poskrbi za vse, kar smo potrebovali.

4.4 Izolacija spletnih strežnikov in virtualizacijsko okolje Docker

Večkrat se nam lahko zgodi, da ko postavimo naš strežnik za lokalni razvoj ter ga nato prestavimo na produkcijo, nekaj ne deluje povsem enako, kot je delovalo lokalno. Tovrstni problemi so nadvse nadležni in včasih vzamejo veliko časa, da jih odpravimo. Problem je preprosto rešljiv z uporabo virtualizacije. Če uporabimo npr. VirtualBox in v njem vzpostavimo Linux strežnik, ki deluje po naših zahtevah, smo lahko precej prepričani, da se bo, če enake ukaze za vzpostavitev uporabimo na drugem Linux strežniku, okolje obnašalo enako. Zelo narašča trend uporabe programa Docker kot izbiro za poganjanje virtualnih operacijskih sistemov, na katerih delujejo servisi [13].

[9] process manager 2, http://pm2.keymetrics.io.

Docker je virtualizacijsko okolje, s katerim našo arhitekturo razbijemo na t. i. mikro servise, kjer vsak opravlja svoj del. Vsak servis je izoliran v svoj vsebnik, ki je popolnoma neodvisen od okolja. Vzpostavitev vsebnika je preprosta in hitra, da nam čim bolj poenostavi proces razvoja.

Ker smo za potrebe testiranja strežnikov želeli čim bolj realne rezultate, smo vsak strežnik izolirali v svoj vsebnik, na katerem smo izvajali teste. Tako je imel vsak strežnik na voljo enako količino virov, kot sta RAM, procesor ipd.
Ustvarjanje vsebnika narekuje posebna datoteka z imenom `Dockerfile`. Primer ukazov za vzpostavitev CoffeeScript testnega vsebnika je naslednja:

```
FROM node:7.0.0
RUN npm install -g pm2@latest coffee-script

COPY . /usr/app
WORKDIR /usr/app
RUN npm install

CMD ["npm", "run", "startcoffee"]
```

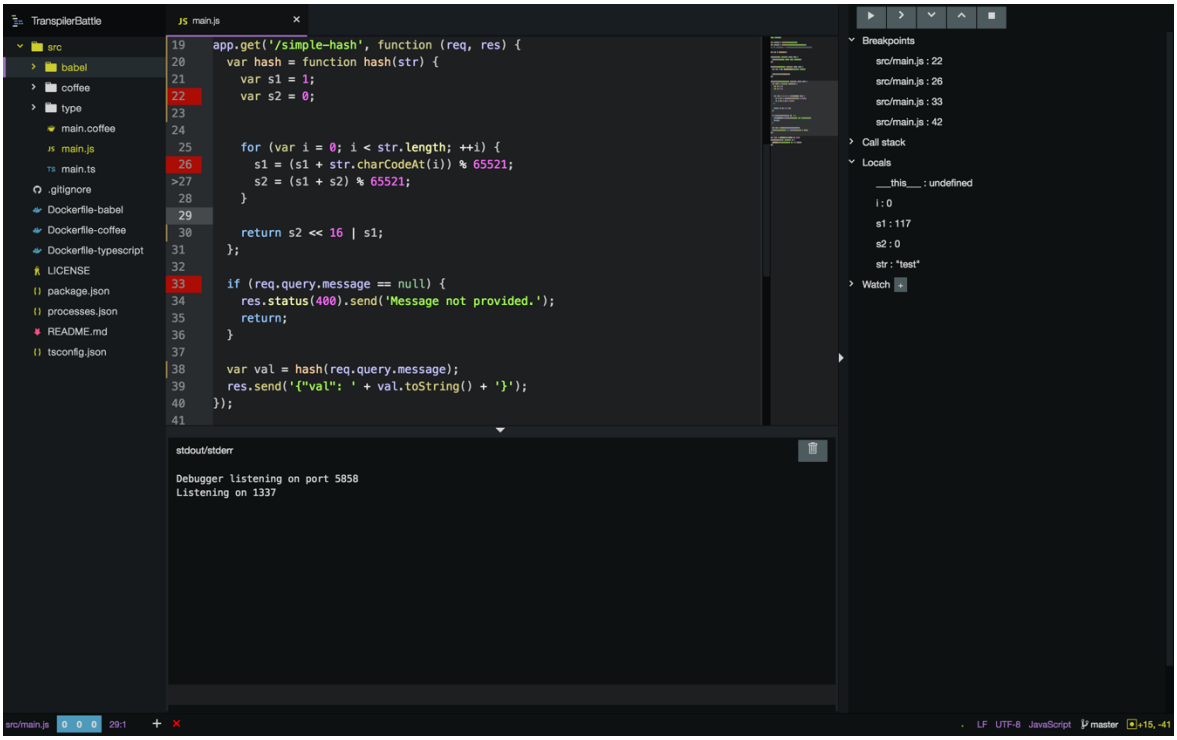
Zgornji ukazi pomenijo naslednje:

- Naš vsebnik se ustvari iz node vsebnika (ta že vsebuje NodeJS in postavljeno okolje) verzije 7.0.0. V ozadju lahko najdemo podobno datoteko, ki postavi vsebnik node. Ta v ozadju temelji na operacijskem sistemu Debian.
- Globalna namestitev pm2 in CoffeeScript (več o pm2 najdemo v razdelku 4.3, CoffeeScript pa v 3.3).
- Vsebino trenutnega direktorija (naš projekt) kopiramo v mapo `/usr/app`.
- Spremenimo privzeti direktorij vsebnika na `/usr/app`.
- Namestimo odvisnosti preko npm.
- Zadnji ukaz je poseben, in sicer zato, ker se izvede, ko naš vsebnik poženemo. V tem primeru bomo zagnali ukaz oblike `npm run startcoffee`. To povzroči izvedbo `startcoffee` ukazov v `package.json` datoteki.

4.5 Razhroščevanje

V procesu razvoja pogosto nehote implementiramo tudi napake. Včasih te lažje identificiramo, včasih pa lahko zanje porabimo tudi več dni. Zato nam pogosto pride prav razhroščevalnik, s katerim lahko gremo skozi naš program, vrstico po vrstico, in preverjamo, kaj se ne obnaša, kot bi se moralo. Pri strežniških jezikih je problem razhroščevanja včasih še bolj pereč, saj je naš strežnik povezan na bazo in se koda izvaja drugje, ne na našem računalniku. Tukaj nam lahko zelo koristi Docker, saj nam omogoča poganjanje celotnega okolja, kot ga imamo na strežniku, lokalno.

NodeJS privzeto že vsebuje podporo za razhroščevanje, vendar je ta na voljo le iz ukazne vrstice. Da smo lahko uporabljali razhroščevanje v urejevalniku Atom, smo namestili paket `node-debugger`. Ta se v ozadju poveže na NodeJS. Razhroščevanje je razvidno s slike 4.2.



Slika 4.2: Razhroščevanje projekta v urejevalniku Atom

Razhroščevanje preko NodeJS je možno na dva načina. Prvega smo že omenili, in sicer gre za razhroščevanje preko ukazne vrstice. Da razhroščevalniku povemo, kje želimo, da se ustavi, v naši kodi preprosto dodamo ukaz `debugger;`. Za vse nadaljnje ustavitve (angl. breakpoints) lahko uporabimo enak pristop ali pa jih nastavimo v ukazni vrstici za razhroščevalnik. Razhroščevanje začnemo z uporabo ukaza:

```
node debug ime_skripte.js
```

Drugi način je pa razhroščevanje »na daljavo«, kar pomeni, da lahko razhroščujemo že delujoč proces. Za tovrsten način moramo ob zagonu NodeJS povedati, da želimo vključeno podporo razhroščevanja. V starejših verzijah je to pomenilo, da smo ob zagonu dodali zastavico `–debug`, vendar tovrsten način več ni podprt. Trenutno to storimo tako, da ob zagonu podamo zastavico `--inspect`, ki podpira razhroščevalni protokol CDP[10]. Ob tovrstnem zagonu nam bo node ob začetku izpisal URL, ki ga lahko prilepimo v Chrome brskalnik, kjer lahko razhroščujemo. Ker smo naše okolje postavili lokalno, drugega načina nismo potrebovali. Zadostoval je le zagon razhroščevalnika preko programa Atom. Za ostalo je poskrbel paket `node-debugger`.

[10] Chrome Debugging Protocol (slov. Razhroščevalni protokol Chrome), omogoča povezavo razhroščevalnika.

5 PRIMERJAVA DIALEKTOV

Pri vsakem izmed dialektov smo že podali nekaj ključnih razlik ter prednosti in slabosti. Vsak izmed njih se je težav in pomanjkljivosti JavaScripta lotil na svoj način. Z vsemi pa lahko ob pravilni uporabi precej pridobimo pri razvoju. Glede na podano lahko poskusimo poiskati najprimernejše področje uporabe za posamezen dialekt.

Babel:

- Še vedno navaden JavaScript, kar pomeni, da se ni treba učiti novega jezika.
- Podpira vse funkcionalnosti ECMAScript 2015.
- Ko bo podpora ECMAScript 2015 na odjemalcih popolna, lahko le izpustimo korak prevajanja.

CoffeeScript:

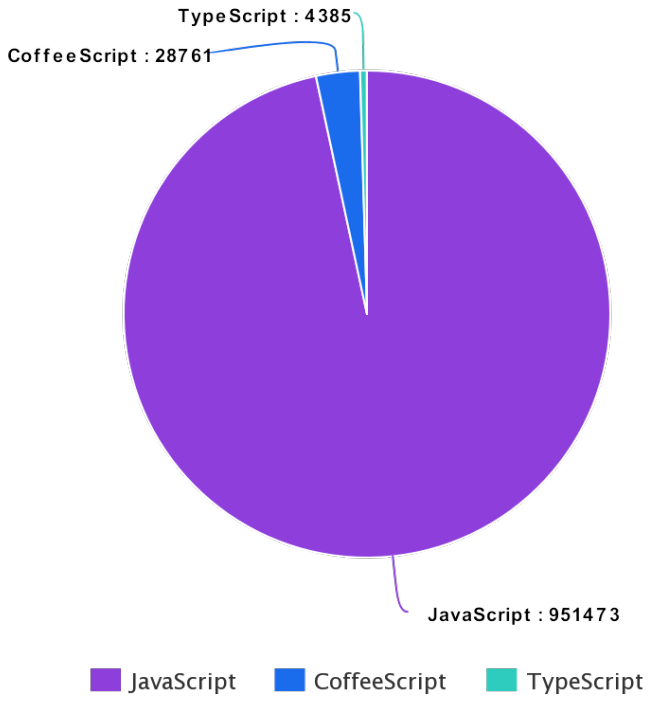
- Popolnoma združljiv z navadnim dialektom JavaScript.
- Podpira vse funkcionalnosti ECMAScript 2015.
- Drastično skrajša kodo in vsebuje precej dodatkov (skrajšane zanke, nestrukturirano prirejanje ...), ki jih v ostalih dialektih ne najdemo.
- Treba se je naučiti njegovo sintakso.

TypeScript:

- Vsebuje sistem tipov, ki nam pri večjih projektih drastično zmanjša količino napak.
- Podpira vse funkcionalnosti ECMAScript 2015.
- Treba se je le malenkost priučiti dodatke k jeziku JavaScript.
- Za vse knjižnice potrebujemo slovar tipov. Včasih je lahko priprava le-teh težavna.

Ko izbiramo programski jezik oz. v našem primeru dialekt, nam je lahko v veliko pomoč tudi skupnost, ki stoji za njim. Pri tem velja omeniti tako nabor ogrodij, ki so za naš dialekt na voljo, in tudi skupine ljudi, pri katerih lahko najdemo odgovore na morebitna vprašanja in težave.

Med spletnimi ponudniki, ki omogočajo gostovanje repozitorijev, je najbolj priljubljen GitHub[11]. Na njem smo poiskali število repozitorijev, ki so spisani pretežno v določenem dialektu. To smo storili tako, da smo v iskalno vrstico dodali filter `language:dialekt`. Težava nastopi pri Babel, ki je v bistvu JavaScript, pisan po specifikaciji ES6, kar pomeni, da nismo mogli točno izluščiti repozitorijev, pisanih po novejšem standardu. Rezultati so vidni na sliki 5.1.



Slika 5.1: Distribucija repozitorijev glede na dialekt

Pričakovali smo, da je JavaScript najpopularnejši dialekt. Je pa opaziti precejšnjo razliko med CoffeeScript in TypeScript, pri čemer je repozitorijev, pisanih v slednjem skoraj sedem krat manj. Upoštevati je potrebno tudi, da se je TypeScript pojavil tri leta za CoffeeScript. Za malenkost boljši vpogled smo zato izluščili repozitorije, ustvarjene v letu 2017.

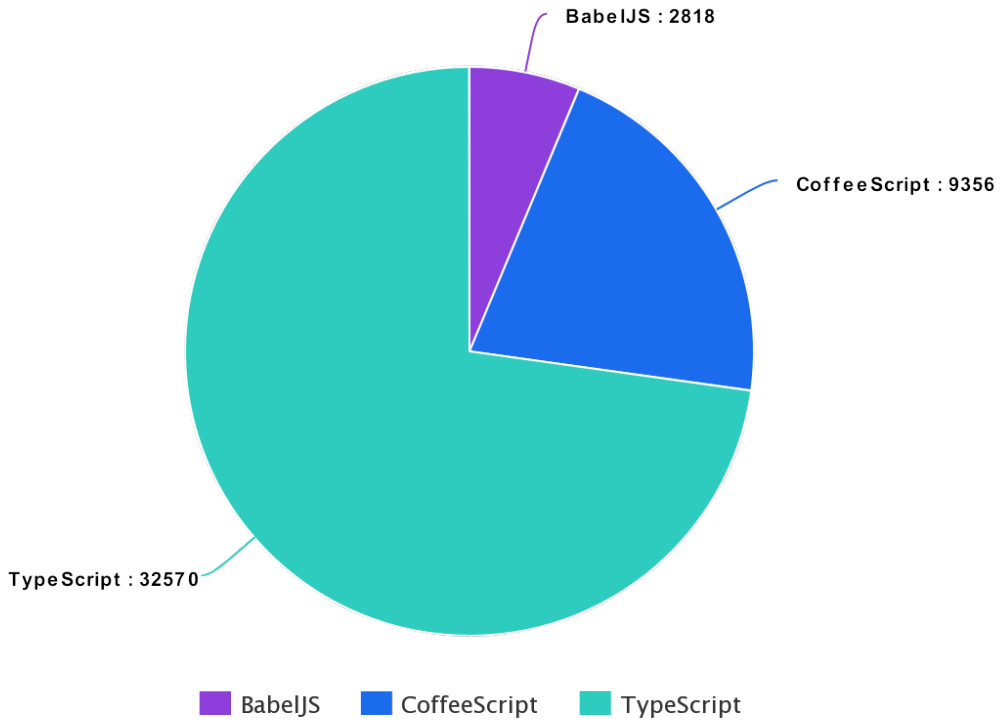
Podatki so naslednji:

- JavaScript: 434.575,
- TypeScript: 3.949,
- CoffeeScript: 1.338.

[11] www.github.com.

Vidimo lahko, da je v zadnjem letu nastalo precej manj repozitorijev, pisanih z dialektom CoffeeScript kot TypeScript, kar kaže na porast slednjega. Glede na to, da je možno vse dialekte prevesti v navaden JavaScript, nas načeloma ne rabi pretirano skrbeti, v katerem je ogrodje, ki ga želimo uporabljati, napisano.

Za postavljanje raznoraznih vprašanj povezanih s programiranjem in podobnimi problemi je najbolj priljubljena spletna stran StackOverflow[12]. Na njej smo poiskali število vprašanj, povezanih s posameznim dialektom. Vprašanja so označena in po oznakah lahko iščemo, tako da jih v iskalniku zapišemo v oglatih oklepajih (npr. `[coffeescript]`). StackOverflow ponuja vprašanja označena z Babel dialektom, vendar je treba omeniti, da vsi uporabniki vprašanj vedno ne označijo primerno, kar pomeni, da obstaja še precej vprašanj, ki slonijo na ECMAScript 2015, a niso ustrezno označena.



Slika 5.2: Distribucija vprašanj glede na dialekt

[12] www.stackoverflow.com.

Kot lahko vidimo na sliki 5.2, v številu vprašanj prevladuje TypeScript. Omenjeno gre pripisati težavam pri uporabi slovarjev tipov, ki so bili še bolj pereč problem v verziji 1. Poleg tega se TypeScript po funkcionalnostih najbolj razlikuje od ostalih dialektov.

5.1 Razlike

Glede na to, da se je vsak dialekt težav jezika JavaScript lotil na svoj način, jih je med sabo težko neposredno primerjati. Vsi nudijo polno podporo standarda ECMAScript 2015, objektno orientirano programiranje ipd. Poskušali pa smo identificirati nekaj ključnih prednosti in slabosti za posamezen dialekt.

TypeScript

Prednosti:

- Tipiziran

Zaradi te lastnosti vnese bistveno manj napak v (večje) projekte.

- Objektno orientiran.

OOP vnaša lažji pregled in spodbuja ponovno uporabo kode.

- Stremenje k urejeni kodi.

TypeScript nas napeljuje k enotnemu načinu programiranja, kar olajša delo, ko na projektu dela več razvijalcev.

Slabosti:

- Težaven za vzpostavitev.

Izmed treh preizkušenih dialektov je TypeScript najtežji za namestitev (konfiguracijska datoteka ...).

- Nedosegljivost slovarjev tipov.

Medtem ko za večje projekte obstajajo vnaprej pripravljeni slovarji tipov, jih za manjše projekte ni na voljo. To pomeni, da nam glavna prednost dialekta (tipiziranost) v takem primeru ne pomaga.

- Brez razhroščevalnika za Atom.

Razhroščevanje TypeScript je trenutno mogoče samo v Visual Studio.

CoffeeScript

Prednosti:

- Ogromno konstruktov, ki skrajšujejo kodo in pospešujejo proces razvoja.

Kratek primer zelo skrajšane zanke, ki jo je tudi precej preprosto razumeti:

```
eat food for food in ['toast', 'cheese', 'wine']
```

- Preprost dostop do razrednih spremenljivk

Razredne spremenljivke preprosto predznačimo z `@`. Na njih se ni treba sklicevati preko `this`, ki se večkrat tudi spreminja, ko smo v povratnih funkcijah (angl. callbacks).

- Več različnih načinov programiranja

Glede na problem lahko uporabimo objektno orientirano, funkcijsko ipd. programiranje.

- Še vedno JavaScript.

Kljub vsem dodatkom in spremembam lahko v `.coffee` datotekah še vedno pišemo povsem klasičen JavaScript.

- Polna podpora preko vtičnika v urejevalniku Atom.

Slabosti:

- Težaven za razumevanje ob prvem srečanju.

Ker se lahko povsem razlikuje od ostalih dialektov, smo lahko na začetku malenkost zmedeni.

- Težko razumevanje argumentov pri klicih funkcij.

Kot smo že pokazali na primeru, lahko pride do problemov pri podajanju parametrov funkcije.

- Brez podpore za razhroščevanje.

Če ga želimo razhroščevati, ga moramo najprej prevesti v JavaScript, ki pa lahko izgleda precej drugače.

- Možnost težko izsledljivih napak zaradi obsega (angl. scope) spremenljivk.

CoffeeScript bo v osnovi spremenljivke delal globalne, razen če smo v lokalnem obsegu. Lahko pride tudi do neželenih prekrivanj, če nismo pozorni.

Babel

Prednosti

- Preprost za učenje.

Ker v bistvu pišemo navaden JavaScript, se nam ni treba učiti posebnosti ostalih dialektov.

- Podprt razhroščevalnik

Čeprav še podpora ni uradna, že večina JavaScript razhroščevalnikov podpira funkcionalnosti standarda ECMAScript 2015.

- Dobra podpora v urejevalnikih kode.

Kot rečeno, gre za navaden JavaScript, kar pomeni, da je Babel implicitno podprt povsod, kjer imamo JavaScript podporo.

Slabosti

- Spreminjanje in dodajanje novih funkcionalnosti.

Bolj kot je ECMAScript 2015 standardiziran, bolj bo Babel dodajal novosti iz že novejših standardov, kar pomeni, da moramo ohraniti korak prevajanja.

5.2 Integracija ostalih instrumentov

Ko naši projekti zrastejo, pogosto zanje uvedemo tudi teste in pripomočke, ki nam pomagajo zagotavljati določen nivo kvalitete naše kode. Govorimo lahko o testih enot (angl. unit testing) ali kakšnem drugačnem načinu testiranja ali merjenja kvalitete. Malenkost podrobneje smo si pogledali ogrodje nodeunit[13], ki je namenjeno testiranju posameznih komponent v naši kodi (angl. unit testing).

Nodeunit je preprosto ogrodje s katerim lahko vzpostavimo testiranje komponent za naš projekt. Omogoča testiranje asinhronih funkcij, podpira uporabo navideznih podatkovnih virov (angl. mocking), omogoča nadaljno integracijo (izdelava poročil, ipd.), deluje za NodeJS ali brskalnik, itd. Za njegovo uporabo ga moramo dodati med odvisnosti – natančneje med `devDependencies`, saj ga ne potrebujemo v produkcijskem okolju. Za zagon testov lahko v `package.json` dodamo le ukaz:

```
./node_modules/.bin/nodeunit test
```

[13] https://github.com/caolan/nodeunit

Teste lahko pišemo v katerem koli izmed dialektov, saj jih lahko vedno prevedemo v JavaScript in poženemo prevedene datoteke.
Podobno je z itegracijo poljubnih ogrodij, saj lahko naše projekte vedno vzpostavimo tako, da se izvorne datoteke prevajajo v JavaScript in poganjajo od tam dalje.

5.3 Implementacija strežnikov

V vsakem izmed omenjenih treh dialektov smo za potrebe testiranja in identifikacije ključnih prednosti ter slabosti jezikov implementirali spletni strežnik. Pri implementaciji smo skušali najti primere, ki čim bolje izrabljajo aspekte posameznih dialektov, da se čim lepše pokaže razlika.

Za potrebe zmogljivostnih testov smo v vsakem strežniku implementirali preprosto sekljalno funkcijo. Performančni test smo izvajali z orodjem httperf, s katerim smo izvedli 10.000 zahtevkov na naš strežnik in merili čas. Orodje nam izpiše statistko preizkušanja, pri kateri smo se najbolj osredotočili na povprečen odzivni čas, saj nam le-ta nakaže na razliko hitrosti izvajanja kode posameznega dialekta.

Za ostale potrebe smo implementirali nekaj splošnih primerov, da vsak dialekt pokaže, kje se čim bolj odlikuje. Ti primeri vključujejo:

- osnovna implementacija strežnika,
- preprost primer »hello world«,
- uporabo razredov in dedovanje,
- uporabo asinhronih funkcij in gnezdenje kode,
- izraba specifičnih funkcionalnosti posameznega dialekta.

Za poganjanje naših strežnikov smo uporabili že omenjeno datoteko package.json, v kateri smo za vsak dialekt definirali, kateri ukazi naj se zaženejo. Razdelek scripts je tako:

```
"scripts": {
    "prestartcoffee": "coffee --output dist --compile src",
    "startcoffee": "pm2 start dist/main.js --no-daemon",
    "prestarttype": "tsc --outDir dist",
    "starttype": "pm2 start dist/main.js --no-daemon",
    "prestartbabel": "babel src --out-dir dist",
    "startbabel": "pm2 start dist/main.js --no-daemon",
    "test": "echo \"no tests\""
  },
```

Vsak dialekt lahko poženemo z uporabo `npm run`, kjer kot zadnji parameter podamo še dialekt, ki ga želimo npr. `starttype`. Ob zagonu npm samodejno izvede še skripte, ki so predznačene s »pre«, ki v našem primeru prevedejo vse izvorne datoteke v JavaScript. Prevedena koda je nato pognana preko pm2.

Ker so naši strežniki izolirani v vsebnike, tudi samega npm ne kličemo neposredno, vendar za to poskrbi ukaz CMD, ki se nahaja v posamezni datoteki Dockerfile. Tako lahko naš vsebnik poženemo z ukazom:

```
docker run -p 1337:1337 --name tbc transpiler-battle-coffee
```

V implementaciji našega strežnika privzeto uporabljamo vrata 1337, ki jih nato izpostavimo ob zagonu vsebnika. V našem primeru ohranimo vrata vsebnika na številki 1337, lahko bi jih prestavili tudi drugam.

Strežniki so ustvarjeni z uporabo ogrodja ExpressJS, ki nam poenostavi vzpostavitev in implementacijo. Najbolj nam pride prav pri uporabi usmerjanja (angl. routing) zahtevkov. Preprosto je implementirati različne končne točke (angl. endpoints), ki smo jih uporabljali za demonstracijo različnih funkcionalnosti.

Preprost primer implementacije strežnika z uporabo ExpressJS je:

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
   res.send('Hello world!');
});
app.post('/error', function(req, res) {
   res.status(500).send('Server error');
});

var port = 1337;
app.listen(port, function() {
   console.log('listening on ' + port);
});
```

Če bi želeli zgornji primer implementirati zgolj z uporabo knjižnic, ki nam jih nudi NodeJS bi morali implementirati lastno pisanje HTTP glave (angl. head) za odgovore zahtevkom. Prav tako bi morali pripraviti celotno logiko usmerjanja. Poleg omenjenega nam ExpressJS olajša upravljanje z zahtevki in odzivi, persistiranje povezav, upravljanje s tokovi (angl. streaming) ipd.

Za potrebe naloge smo implementirali strežnike podobne zgornjemu. Vsaka končna točka je služila demonstraciji drugačnega primera. Končne točke, ki smo jih implementirali, so:

- `/` - začetna stran, ki se zgolj odzove s preprostim tekstom,
- `/simple-hash`, kjer poleg končne točke v URL dodamo še parameter »message«, ki vsebuje sporočilo, ki ga želimo poslati skozi sekljalno funkcijo,
- `/animal`, ki v implementaciji uporablja dedovanje in razrede,
- `/image`, ki vrača sliko.

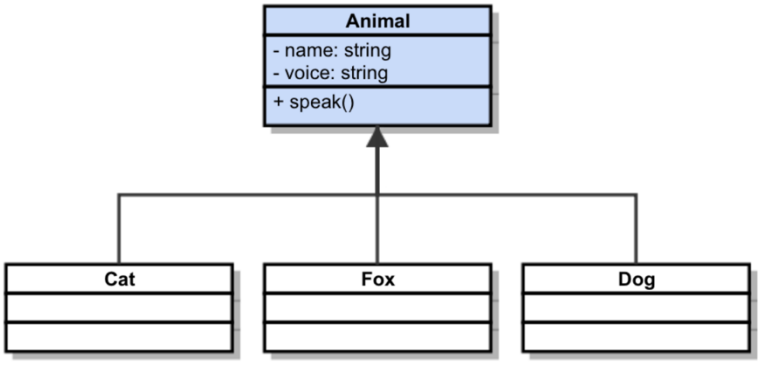
Končna točka `/` je namenjena zgolj temu, da lahko vidimo razliko med posameznim dialektom v načinu kodiranja.

Končna točka `/simple-hash` pošlje parameter `message` (podan v GET zahtevku) skozi sekljalno funkcijo Adler32. Ta je preprosta za implementacijo, vendar nam dopušča dovolj svobode, da smo jo lahko v vsakem dialektu implementirali malenkost drugače. Psevdokod je naslednji:

```
func adler32(string data)
{
    const MOD = 65521
    a = 1, b = 0
    foreach (character c in data)
    {
        a = (a + c) % MOD
        b = (b + a) % MOD
    }
    return (b << 16) | a
}
```

Najlažja za implementacijo je bila v dialektu CoffeeScript, saj napišemo najmanj kode. Sistem tipov TypeScript nam koristi pri sintaktični analizi in skrbi, da ne operiramo nad nekompatibilnimi tipi. Funkcionalnosti Babel nismo mogli posebej izpostaviti.

Končna točka `/animal` v ozadju uporabi razrede. V vsakem dialektu smo implementirali osnovni abstraktni razred Animal, ki ponazarja žival. Ta vključuje ime živali in njeno oglašanje. Animal je nato dedovan v konkretnem razredu, kot je npr. Fox. Vsak dialekt implementira drugo žival. Razredni diagram je prikazan na sliki 5.3.



Slika 5.3: Razredni diagram živali

Na tem primeru se lepo kaže ideologija CoffeeScript, da mora koda čim bolj kazati bistvo brez balasta. Tako smo datoteko animals.coffee in njene razrede uspeli implementirati v zgolj 9 vrsticah. Za enak rezultat smo pri Babel potrebovali 26 vrstic in pri TypeScript 23.

Končna točka `/image` se trudi ponazoriti problem, ki nastane, ko v naši kodi uporabimo veliko asinhronih funkcij s povratnimi klici (angl. callbacks). Najprej naredimo zahtevo na drug spletni strežnik, kjer iščemo podatke o določenem filmu (naš primer išče »The Matrix«). Ko dobimo odgovor iz podatkov, pridobimo URL-slike in naredimo novo zahtevo, ki sliko prenese v naš delovni pomnilnik. Sliko nato zmanjšamo in ob končanem opravilu pošljemo odjemalcu. Vsaka izmed omenjenih zahtev je asinhrona.
Zaradi uporabe bluebird paketa in pretvarjanja funkcij v obljube med izvajanjem kode nam vgrajeni sistem tipov dialekta TypeScript ni koristil. Babel in CoffeeScript implementaciji pa sta si bili precej podobni z razliko sintakse.

5.3.1 Babel

Ker je Babel najbolj podoben JavaScriptu, saj pišemo JavaScript, ki vključuje funkcionalnosti standarda ES6, smo implementacijo začeli z njim. Za vzpostavitev projekta ne potrebujemo ničesar posebnega, kar naredi proces razvoja enostavnejši.

Pri večini implementacije nam dodatne funkcionalnosti ES6 niso koristile in smo tako pisali navaden JavaScript. Moč Babel se pokaže pri delu z asinhronimi funkcijami in povratnimi klici (angl. callbacks). Tukaj lahko kodo glede na JavaScript naredimo bolj berljivo. Primer, ki smo ga uporabili, je osnoven, vendar se v produkcijskih okoljih pogosto gnezdi asinhrone funkcije tudi 5 ali več nivojev globoko. To privede do težko vzdrževalne kode, česar se z Babel lahko izognemo. Končna točka `/image` izgleda v navadnem JavaScript:

```
var regular = function regular(title, resultCallback) {
  request.get("http://www.omdbapi.com/?t=" + encode(title), function
(err, response, raw) {
    if (err) {
      return resultCallback(err, null);
    }
    var body = JSON.parse(raw);
    var options = { url: body.Poster, encoding: null};
    request.get(options, function (err, response, body) {
      if (err) {
        return resultCallback(err, null);
      }
      sharp(body).resize(100).toBuffer(function (err, data) {
        resultCallback(err, data);
      });
    });
  });
};
```

Kot vidimo, se koda gnezdi globlje in globlje. Tako je težka za razumevanje in lahko privede do napak. Ob uporabi funkcionalnosti ES6, lahko zgornji primer preoblikujemo tako:

```
const shorterPromises = function*(title) {
  var res = yield
requestPromise.getAsync(`http://www.omdbapi.com/?t=${encode(title)}`);
  const body = JSON.parse(res.body);
  const options = { url: body.Poster, encoding: null };
  res = yield requestPromise.getAsync(options);
  return sharp(res.body).resize(100).toBuffer();
};
```

Koda je tako bistveno krajša in precej lažja za razumeti.

Priučitev dodatne funkcionalnosti je preprosta in nam lahko koristi tudi na drugih področjih, kjer se uporablja JavaScript. Podpora ES6 brez vmesnikov, kot je Babel, je le še vprašanje časa. Tako velja omeniti, da ko NodeJS podpre ES6 v svojem polnem naboru, lahko za nadaljnje delovanje le odstranimo korak prevajanja iz Babela v JavaScript.

5.3.2 CoffeeScript

Zaradi ideologije dialekta CoffeeScript, da mora naša koda čim bolj izražati svoje bistvo, je bila implementacija precej hitrejša glede na ostala dva dialekta.
CoffeeScript vsebuje precej uporabnih trikov, ki pohitrijo pisanje in tako tudi razvojni proces. Primer je prirejanje več spremenljivk naenkrat, ki ga storimo na naslednji način:

```
[a, b, c] = [1, 'a', 2.3]
```

Slabost raznoraznih okrajšav, ki nam jih CoffeeScript nudi, je težko zagotavljanje konsistence kode na večjih projektih. Funkcija v svoji najkrajši obliki zgleda le "`->`", kar nas lahko na določenih mestih zbega. Prav tako je veljaven zapis tudi "`() ->`", ki načeloma ne izgleda nič lepše. Težava nastopi tudi, ko želimo klicati funkcijo, kjer kot parameter podamo klic druge funkcije.

```
item = someFunction argA, someOtherFunction argX, argWhat
```

V zgornjem primeru je zelo težko vedeti, v katero funkcijo je bil podan argument `argWhat`. Tovrstne napake so včasih lahko zelo zapletene za identifikacijo, saj CoffeeScript ne vidi nobenih težav. Zgornji primer preveden v JavaScript:

```
var item;
item = someFunction(argA, someOtherFunction(argX, argWhat));
```

Komaj tukaj lahko vidimo, da je `argWhat` podan kot argument notranje funkcije. Tovrstnega dvoumja se lahko izognemo z eksplicitno uporabo oklepajev, kot bi to storili pri JavaScriptu.

```
item = someFunction argA, someOtherFunction(argX), argWhat
```

Tako smo eksplicitno povedali, da želimo, da se `argWhat` poda kot parameter zunanje funkcije `someFunction`. Podobna nekonsistentnost se lahko pojavi pri uporabi objektov.

Torej metodologija, da z manj kode ustvarimo manj napak, drži do določene mere. Če nam uspe zagotoviti, da vsi razvijalci pišejo enak stil kode, se lahko tovrstno dvoumje skoraj izniči. Sodobna orodja za zaganjanje testov nam omogočajo tudi sintaktično analizo kode, kar poskrbi, da se projekt drži določenega standarda.
Učenje CoffeeScripta se sprva izkaže za malenkost naporno, saj jezik izgleda precej drugače kot JavaScript. Ko se uspemo priučiti osnove, pa lahko naše delo poteka res hitro, saj je za podobne rezultate pri drugih dialektih potrebno spisati precej več kode.

5.3.3 TypeScript

Vzpostavitev TypeScript razvoja se je izkazala za najzahtevnejšo izmed trojice. Težave gre pripisati predvsem slovarjem tipov, ki jih TypeScript potrebuje za uporabo knjižnic. Kot smo že omenili, smo slovarje dodali v `devDependencies`, ker jih potrebujemo le za razvoj. Preverjanje tipov se v TypeScript izvaja ob prevajanju v JavaScript, in ko je proces zaključen, slovarjev ne potrebujemo več. Tukaj nam na pomoč vskoči tudi Atom, saj preverja napake sproti in nas na njih opozarja. Po nameščanju slovarjev je treba TypeScript prevajalnik tudi primerno nastaviti. Za to skrbi datoteka tsconfig.json, ki je v našem primeru naslednja:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ],
  "typeRoots": [ "node_modules/@types" ],
  "compileOnSave": false
}
```

Nastavili smo, da želimo uporabljati funkcionalnosti ES6, povedali, kje so naše izvorne datoteke, izključili prevajanje `node_modules`, ki so v odvisnosti nameščene preko npm, in povedali, da ne želimo ponovnega prevajanja, ko shranimo katero izmed Typescriptovih

datotek. Pomembna vrstica je `typeRoots`. Ta namreč prevajalniku pove, kje lahko najde slovarje s tipi. Proces je bil zelo poenostavljen z uvedbo TypeScript verzije 2. Pred tem je bilo treba prevajalniku puščati pragmatične komentarje v posameznih datotekah.

TypeScript nas sili v uporabo razredov kot primarni način programiranja, zaradi česar je treba ponekod malenkost spremeniti koncept razmišljanja. Večina knjižnic za NodeJS sloni na funkcijskem načinu programiranja, zaradi česar moramo na določenih mestih prilagajati povratne klice.

Dodatna težava je nastopila, ko smo z uporabo `bluebird` paketa prilagodili knjižnico `request` tako, da je podpirala uporabo obljub (angl. promises). Ker se tovrstna prilagoditev zgodi med izvajanjem kode (angl. runtime), TypeScript ob prevajanju ne ve, kakšni tipi so prisotni. Tako smo morali na več mestih uporabiti generične tipe, da se je naša koda sploh prevedla. S tem pa izgubimo največjo prednost dialekta. Uporaba bluebird je namreč naslednja:

```
const requestPromise: any = bluebird.promisifyAll(request);
```

Tako ima spremenljivka `requestPromise` še vedno enake tipe kot `request`, le da vrača obljube (angl. promises).

Učenje TypeScripta se izkaže za sorazmerno preprosto, saj navadnemu JavaScriptu dodamo le manjše spremembe. Dodane so anotacije tipov, ki nam z izjemo malenkost dodatnega tipkanja močno olajšajo delo pri iskanju napak. Statistično je v JavaScriptu pogosta napaka povezana s primerjavami spremenljivk različnih tipov, ki v TypeScriptu več niso možne. Če želimo, lahko sami namerno vsilimo netipizirane spremenljivke, vendar s tem izgubimo glavno prednost dialekta. Večja težava nastopi, ko uporabljamo knjižice, ki ne ponujajo slovarjev s tipi. Takrat nam je v največjo pomoč priprava vmesnega razreda, ki enkapsulira funkcionalnosti, ki jih potrebujemo iz knjižnice (angl. wrapper class). Tovrsten pristop zahteva malenkost več dela, vendar nam povrne delo s tipi.

5.4 Zmogljivostni testi

Pričakovali smo, da bodo razlike med hitrostjo izvajanja dialektov minimalne. Kot rečeno, smo teste opravljali z orodjem httperf. Ukaz za testiranje je naslednji:

```
httperf --hog --server localhost --port 1337 --uri /simple-
hash?message=derp --num-conn 10000 --rate 100 --timeout 5
```

Podane zastavice pomenijo naslednje:

- `hog`; za pošiljanje zahtevkov uporabimo kar se da veliko število TCP-povezav;
- `server`; povemo, kje se nahaja naš strežnik;
- `port`; povemo vrata, na katerih naš strežnik posluša;
- `uri`; za naše potrebe smo klicali končno točko `simple-hash`, ki smo ji kot argument podali sporočilo »derp« v spremenljivko, poimenovano `message`;
- `num-conn`; uporabili smo tisoč povezav;
- `rate`; opravljali smo sto zahtevkov na sekundo;
- `timeout`; najdaljši dovoljeni življenjski čas klica, v našem primeru je 5 sekund.

Rezultati so vidni v tabeli 5.1 (časi so v milisekundah).

Tabela 5.1: Rezultati zmogljivostnega testiranja po dialektih

Metrika / Dialekt	Min. čas	Povp. čas	Max. čas	Mediana	Standardni odklon
CoffeeScript	1.0	1.5	13.8	1.5	0.7
TypeScript	0.9	1.5	43.2	1.5	1.0
Babel	0.9	1.4	21.8	1.5	0.7

Zgornja preglednica kaže na minimalne razlike v izvajanju kode. Tovrstni rezultati so bili pričakovani, saj so prevajalniki pri prevajanju v JavaScript že precej »pametni« in kodo dobro optimizirajo. Bolj opazna razlika bi se verjetno pojavila ob funkciji, ki bi obsegala več sto vrstic.

V primeru, da bi potrebovali zelo hitro obnašanje posameznih delov kode, se bolj izplača uporabiti kombinacijo NodeJs + C++. Ker je tolmač za JavaScript NodeJS (V8) kodiran v jeziku C++, se da preko posebnega dodatka uporabljati C++ kodo znotraj NodeJS. Tako lahko kritične odseke kode spremenimo na zgoraj opisan način.

6 KONČNI VTISI

Izkaže se, da je vsak izmed dialektov želel rešiti določen problem oz. pomanjkljivost JavaScripta na NodeJS in vsak je to dosegel drugače. Razlike pri hitrostih so zelo majhne in jih lahko zanemarimo.

Babel se nam splača uporabiti, ko želimo kodno bazo (angl. codebase) ohraniti na JavaScript, vendar želimo izrabiti funkcionalnosti prisotne v ES6. Ohranjati kodno bazo je pogosto precej smiselno, saj je za JavaScript bistveno večja podpora in več kadra kot za ostale dialekte. Prav tako ima Babel namen vključevati novejše ECMAScript standarde, ko se pojavijo. Za potrebe razhroščevanja je Babel najlažji dialekt, saj je izhodna koda najbolj podobna izvorni.

CoffeeScript nam drastično pohitri proces kodiranja, vendar se ga je treba predhodno dobro naučiti. Ker je na voljo že od leta 2009, ima za sabo veliko skupnost, ki nam lahko koristi, če potrebujemo pomoč. Jezik izpostavlja funkcionalnost in delovanje napisane kode. Težava nastopi pri konsistentnosti večjih kodnih baz. Dokler ne vemo točno, v kaj se dialekt prevede, nam je oteženo tudi razhroščevanje. Ampak kot kaže, je razhroščevalnik neposredno za CoffeeScript blizu zaključka.

TypeScript nam na večjih projektih bistveno zmanjša možnost napak zaradi uporabe tipov in njihovega preverjanja ob prevajanju. Izhodna koda je sorazmerno podobna generirani, zato ni pretežka za razhroščevanje. V primeru, da delamo na operacijskem sistemu Windows, lahko razhroščujemo TypeScript neposredno z uporabo programa Visual Studio. Težave lahko nastopijo pri uporabi bolj eksotičnih knjižnic, ki ne ponujajo vnaprej pripravljenih slovarjev s tipi.

7 SKLEP

V diplomskem delu smo predstavili kompletno rešitev za razvoj na strežniški strani z uporabo NodeJS. Na večje težave pri pripravi nismo naleteli, največ dela pa je bilo z vzpostavitvijo razvojnega okolja in podporo TypeScriptu.

Najprej smo si ogledali, kako je JavaScript skozi zgodovino prešel iz preprostega skriptnega jezika za odjemalce v bogat jezik, ki se je preselil tudi na druge ekosisteme. Konkretneje smo si ogledali, kako je nastalo okolje NodeJS in kakšna problematika se je pojavila s selitvijo JavaScripta na strežniško stran. Predstavili smo tri različne dialekte JavaScripta, ki so nastali z namenom, da izboljšajo in olajšajo razvojni proces.

Za potrebe implementacije strežnikov v različnih dialektih smo najprej pogledali ponudbo razvojnih okolij. Po izbiri najprimernejšega smo ga ustrezno nastavili in pripravili, da je ustrezal našim potrebam za razvoj. Pozornost smo namenili razhroščevanju, saj je pri večjih projektih ključnega pomena za odkrivanje in odpravo napak.

Osredotočili smo se na tri največje dialekte JavaScripta, ki so: Babel, CoffeeScript in TypeScript. V vsakem izmed omenjenih smo implementirali preprost spletni strežnik, čigar namen je bil čim bolje izrabiti funkcionalnosti posameznega dialekta. Izvorna koda dialektov se je nato prevedla v JavaScript, ki ga je NodeJS izvajal. Vsak strežnik je bil izoliran v svoj vsebnik preko virtualizacijskega orodja Docker. Tako smo lahko dosegli popolno neodvisnost med strežniki in jih ustrezno pripravili za test. Teste smo izvajali z uporabo orodja httperf, s pomočjo katerega smo dobili zmogljivostno statistiko za posamezen strežnik.

Uspešno nam je uspelo opraviti teste na vseh treh strežnikih in identificirati ključne prednosti ter pomanjkljivosti posameznih dialektov. Predstavljeno znanje je dobro izhodišče za izbor določenega dialekta v produkcijske namene. Prav tako nam lahko koristi pri vzpostavitvi razvojnega okolja in osebnega razvojnega procesa.

8 VIRI

[1] Atom dokumentacija, 12 April 2017. [Online]. Available: https://atom.io/docs.

[2] Electron, 15 April 2017. [Online]. Available: https://electron.atom.io.

[3] Atom paketi, 12 April 2017. [Online]. Available: https://atom.io/packages.

[4] ECMAScript, 14 April 2017. [Online]. Available: https://www.ecma-international.org/publications/standards/Standard.htm.

[5] Babel, 30 Marec 2017. [Online]. Available: https://babeljs.io.

[6] Babel funkcionalnosti, 29 Marec 2017. [Online]. Available: https://babeljs.io/learn-es2015/.

[7] CoffeeScript, 21 April 2017. [Online]. Available: http://coffeescript.org.

[8] TypeScript, 10 April 2017. [Online]. Available: https://www.typescriptlang.org.

[9] R. J. Peterson, 15 April 2017. [Online]. Available: https://www.toptal.com/javascript/10-most-common-javascript-mistakes .

[10] TypeScript primeri, 11 April 2017. [Online]. Available: https://www.typescriptlang.org/samples/index.html.

[11] NodeJs, "NodeJs," 20 April 2017. [Online]. Available: https://nodejs.org/en/docs/.

[12] Homebrew, 11 April 2017. [Online]. Available: https://brew.sh.

[13] Docker, 14 April 2017. [Online]. Available: https://www.docker.com.

ZA ŠTUDENTE E, ITK, MK, RI-T in TK

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O USTREZNOSTI ZAKLJUČNEGA DELA

Podpisani mentor/-ica : Tomaž Kosar
(ime in priimek mentor-ja/-ice)

in somentor/-ica (eden ali več, če obstajajo): _____
(ime in priimek somentor-ja/-ice)

Izjavlja-m/-va/-mo, da je študent/-ka

Ime in priimek: Jan Haložan, *ID številka:* 1002079659,

vpisna številka: E1064342, *na študijskem programu:*

Računalništvo in Informacijske Tehnologije

izdelal/-a zaključno delo z naslovom:

Primerjava dialektov jezika JavaScript

(naslov zaključnega dela v slovenskem jeziku)

v skladu z odobreno temo zaključnega dela, navodili o pripravi zaključnih del in mojimi (najinimi/našimi) navodili.

Preveril/-a/-i in pregledal/-a/-i sem/sva/smo poročilo o preverjanju podobnosti vsebin z drugimi deli (priloga) in potrjujem/potrjujeva/potrjujemo, da je zaključno delo ustrezno.

Datum in kraj:
3. 8. 2017, Maribor

Podpis mentor-ja/-ice:

Datum in kraj:

Podpis somentor-ja/-ice (če obstaja):

Priloga:
- Poročilo o preverjanju podobnosti vsebin z drugimi deli.

ZA ŠTUDENTE E, ITK, MK, RI-T in TK

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

FERI

IZJAVA O AVTORSTVU IN ISTOVETNOSTI TISKANE IN ELEKTRONSKE OBLIKE ZAKLJUČNEGA DELA

Ime in priimek študent-a/-ke: Jan Haložan
Študijski program: Računalništvo in Informacijske Tehnologije
Naslov zaključnega dela: Primerjava dialektov jezika JavaScript

Mentor: Tomaž Kosar
Somentor: _____

Podpisan-i/-a študent/-ka Jan Haložan

- izjavljam, da je zaključno delo rezultat mojega samostojnega dela, ki sem ga izdelal/-a ob pomoči mentor-ja/-ice oz. somentor-ja/-ice;
- izjavljam, da sem pridobil/-a vsa potrebna soglasja za uporabo podatkov in avtorskih del v zaključnem delu in jih v zaključnem delu jasno in ustrezno označil/-a;
- na Univerzo v Mariboru neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico ponuditi zaključno delo javnosti na svetovnem spletu preko DKUM; sem seznanjen/-a, da bodo dela deponirana/objavljena v DKUM dostopna široki javnosti pod pogoji licence Creative Commons BY-NC-ND, kar vključuje tudi avtomatizirano indeksiranje preko spleta in obdelavo besedil za potrebe tekstovnega in podatkovnega rudarjenja in ekstrakcije znanja iz vsebin; uporabnikom se dovoli reproduciranje brez predelave avtorskega dela, distribuiranje, dajanje v najem in priobčitev javnosti samega izvirnega avtorskega dela, in sicer pod pogojem, da navedejo avtorja in da ne gre za komercialno uporabo;
- dovoljujem objavo svojih osebnih podatkov, ki so navedeni v zaključnem delu in tej izjavi, skupaj z objavo zaključnega dela;
- izjavljam, da je tiskana oblika zaključnega dela istovetna elektronski obliki zaključnega dela, ki sem jo oddal/-a za objavo v DKUM.

Datum in kraj:
3. 8. 2017, Maribor

Podpis študent-a/-ke:

Podpis mentor-ja/-ice: _____
(samo v primeru, če delo ne me biti javno dostopno)

Ime in priimek ter podpis odgovorne osebe naročnika in žig:

_____
(samo v primeru, če delo ne sme biti javno dostopno)