UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Skriptovací jazyky pro tvorbu webových aplikací

Jan Voráček

Diplomová práce

2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Voráček**
Osobní číslo:	**I11422**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Skriptovací jazyky pro tvorbu webových aplikací**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem diplomové práce je představit nové a perspektivní skriptovací jazyky pro tvorbu webových aplikací (Dart, Typescript) a jejich porovnání s aktuálně používanými jazyky v této oblasti.
V praktické části diplomové práce bude provedena demonstrace použití vybraných skriptovacích jazyků na příkladu aplikace klient-server, možnosti testování programů vytvořených s využitím těchto jazyků (např. funkční testování, výkonové testování apod.).

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

MURPHY, M.,L. Android 2 - Průvodce programováním mobilních aplikací. Brno: Computer Press, 2011. 371 s. EAN 9788025131947.
LEWIS, H. R., DENENBERG, L. Data structures and their algorithms. Berkley, Adison-Wesley, 1997.
Android developers Homepage [online]. 2011 [cit. 2011-10-07]. Dostupné z WWW: http://developer.android.com/.

Vedoucí diplomové práce: ....
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2011**
Termín odevzdání diplomové práce: **18. května 2012**

prof. Ing. Simeon Karamazov, Dr.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2011

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 15. 5. 2013 Jan Voráček

Poděkování

Děkuji Ing. Michaelu Bažantovi, Ph.D. za vedení mé práce, jeho připomínky a rady. Děkuji také svým přátelům a rodině za poskytnutí výborných podmínek pro studium.

Anotace

Diplomová práce se zabývá problematikou skriptovacích jazyků v odvětví webových aplikací. V první části poskytuje obecný náhled do problematiky skriptovacích jazyků. Druhá část je pak věnována jazykům Dart a TypeScript, na které je práce primárně zaměřena.

Práce seznamuje čtenáře s různými skriptovacími jazyky, poskytuje drobný náhled do problematiky testování a představuje možnosti použití Dartu a TypeScriptu na serverové části webové aplikace a následně i na její klientské části.

Klíčová slova

Skriptovací jazyk, webová aplikace, testování aplikací, Dart, TypeScript, JavaScript.

Title

Scripting languages for web application development

Annotation

This thesis deals with scripting languages in relation with web applications. The first part provides general insight into scripting languages. The second part is dedicated to languages Dart and TypeScript, which are the main objective of the thesis.

The thesis introduces to reader various scripting languages, provides a small insight into software testing and introduces possibilities of usage Dart and TypeScript on server-side and then on client-side of web application.

Keywords

Scripting languages, web application, software testing, Dart, TypeScript, JavaScript

Obsah

Úvod **13**

Skriptovací jazyky **14**

1 Rozdělení programovacích jazyků 14

1.1 Jazyky se statickým / dynamickým typovým systémem 14

1.2 Systémové / skriptovací programovací jazyky 15

1.3 Dynamické programovací jazyky 15

1.4 Běžně používané skriptovací jazyky 17

1.4.1 PHP 19

1.4.2 Python 20

1.4.3 JavaScript 22

1.5 Transpilery 27

1.6 Dart 27

1.6.1 Vývoj jazyka 27

1.6.2 Základní rysy, paradigma 29

1.6.3 Přínosy 30

1.6.4 Vývojová prostředí 30

1.7 TypeScript 30

1.7.1 Vývoj jazyka 31

1.7.2 Základní rysy, paradigma 31

1.7.3 Přínosy 32

1.7.4 Vývojová prostředí 32

2 Testování softwaru 33

2.1 Funkční testování 33

2.1.1 Unit testy 34

2.1.2 Integrační testy 34

2.1.3 Systémové testy 35

2.1.4 Akceptační testy 35

2.2 Výkonové testování 35

Dart a TypeScript 37

3 Předmět praktické části ... 37
4 Použití na serveru ... 39
4.1 Webový server ... 39
4.2 Operace se systémovými prostředky ... 42
4.3 Práce se sítí ... 44
4.4 Používání sdílených prostředků ... 46
4.5 Možnosti automatizace ... 49
4.6 Používání externích knihoven ... 50
4.7 Testování ... 51
4.8 Test výkonu ... 53
5 Použití jako klientský skriptovací jazyk ... 55
5.1 Manipulace s DOMem ... 55
5.2 Práce s DOM událostmi ... 58
5.3 Komunikace se serverem ... 65
5.4 Používání externích knihoven ... 67
5.5 Testování ... 70

Závěr 72

Literatura 73

Příloha A – CD s aplikacemi 76

Příloha B – Testovaná aplikace v Dartu 77

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BDD	Behavior-Driven Development
CD	Compact Disc
CSS	Cascading Style Sheets
DB	Database
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
FB	Facebook
FTP	File Transfer Protocol
GTK	Gimp Toolkit
GWT	Google Web Toolkit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OS	Operační systém
PHP	PHP: Hypertext Preprocessor
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

Seznam obrázků

Obrázek 1 – Klasifikace jazyků dle počtu instrukcí na výraz a míry typovosti 15
Obrázek 2 – Oblíbenost programovacích jazyků 18
Obrázek 3 – Míra použití programovacích jazyků na GitHubu 18
Obrázek 4 – Porovnání výkonu .NET vs. Node.js z 12. 8. 2013 23
Obrázek 5 – Porovnání výkonu .NET a Node.js z 31. 3. 2013 24
Obrázek 6 – Diagram znázorňující prototypovou dědičnosti 25
Obrázek 7 – Graf výkonu Dartu, JavaScriptu ručně psaného a kompilovaného z Dartu 28
Obrázek 8 – Pyramida testování 33
Obrázek 9 – Náhled aplikace 37
Obrázek 10 – Společné API 38
Obrázek 11 – Zpracování požadavku na webovém serveru 39
Obrázek 12 – Test výkonu Dart a Node.js (TypeScript) 54
Obrázek 13 – Propagace události v DOMu 60
Obrázek 14 – Manuální instalace knihovny 68
Obrázek 15 – Instalace knihoven pomocí nástrojů Grunt, Bower a RequireJS 70

Seznam zdrojových kódů

Zdrojový kód 1 – Closure 16
Zdrojový kód 2 – Object runtime alternation 16
Zdrojový kód 3 – Funkce eval 17
Zdrojový kód 4 – Ukázka funkcionálního programování 17
Zdrojový kód 5 – Reflexe 17
Zdrojový kód 6 – Použití PHP v HTML 20
Zdrojový kód 7 – Objektově orientovaný přístup v PHP 20
Zdrojový kód 8 – Šablony v Djangu 21
Zdrojový kód 9 – Výstup z Pythonu do HTML bez šablonovacího jazyka 22
Zdrojový kód 10 – Ukázka kódu v Pythonu 22
Zdrojový kód 11 – Šablona v Jade 25
Zdrojový kód 12 – Plnění šablony ve frameworku express 26
Zdrojový kód 13 – Šablona v Knockoutu 26
Zdrojový kód 14 – Plnění šablony ve frameworku Knockout 27
Zdrojový kód 15 – Ukázka kódu v Dartu 29
Zdrojový kód 16 – Ukázka kódu v TypeScriptu 31
Zdrojový kód 17 – Unit test 34
Zdrojový kód 18 – Systémový test 35
Zdrojový kód 19 – Webový server v Dartu 40
Zdrojový kód 20 – Server ve frameworku start 40
Zdrojový kód 21 – Webový server v TypeScriptu 41
Zdrojový kód 22 – Webový server ve frameworku express 42
Zdrojový kód 23 – Informace o OS v Dartu 43

Zdrojový kód 24 – Informace o OS v TypeScriptu 44
Zdrojový kód 25 – TCP echo server v Dartu 44
Zdrojový kód 26 – TCP klient v Dartu 45
Zdrojový kód 27 – TCP echo server v TypeScriptu 45
Zdrojový kód 28 – TCP klient v TypeScriptu 46
Zdrojový kód 29 – Připojení k MongoDB v Dartu...46
Zdrojový kód 30 – Práce s MongoDB v Dartu...47
Zdrojový kód 31 – Autentifikace pomocí Facebook API v Dartu 47
Zdrojový kód 32 – Připojení k MongoDB v TypeScriptu 48
Zdrojový kód 33 – Práce s MongoDB v TypeScriptu 48
Zdrojový kód 34 – Autentifikace pomocí Facebook API v TypeScriptu 49
Zdrojový kód 35 – Automatizační skript v Gruntu 50
Zdrojový kód 36 – Testování v Dartu...52
Zdrojový kód 37 – Testování v tsUnit 52
Zdrojový kód 38 – Testování v Jasmine 53
Zdrojový kód 39 – Testování v Mocha 53
Zdrojový kód 40 – Vytváření DOM elementů v Dartu...55
Zdrojový kód 41 – Modifikace atributů v Dartu 56
Zdrojový kód 42 – Manipulace s umístěním elementů v dokumentu v Dartu....................56
Zdrojový kód 43 – Ovlivňování CSS vlastností v Dartu 56
Zdrojový kód 44 – Vytvoření elementů v TypeScriptu...57
Zdrojový kód 45 – Vytvoření elementů v TypeScriptu za pomoci jQuery 57
Zdrojový kód 46 – Modifikace atributů v TypeScriptu..57
Zdrojový kód 47 – Modifikace atributů v TypeScriptu pomocí jQuery 57
Zdrojový kód 48 – Manipulace s umístěním elementů v dokumentu v TypeScriptu..........58
Zdrojový kód 49 – Manipulace s umístěním elementů v dokumentu v TypeScriptu s využitím jQuery 58
Zdrojový kód 50 – Ovlivňování CSS vlastností v TypeScriptu 58
Zdrojový kód 51 – Ovlivňování CSS vlastností v TypeScriptu pomocí jQuery.................58
Zdrojový kód 52 – Nastavení posluchače v Dartu...59
Zdrojový kód 53 – Vlastní typ události v Dartu 59
Zdrojový kód 54 – Dočasné pozastavení a zrušení posluchače v Dartu............................59
Zdrojový kód 55 – Zastavení propagace události v Dartu 60
Zdrojový kód 56 – Zrušení výchozí reakce na událost v Dartu 61
Zdrojový kód 57 – Nastavení posluchače v TypeScriptu 61
Zdrojový kód 58 – Nastavení posluchače v TypeScriptu pomocí jQuery 62
Zdrojový kód 59 – Vlastní typ události v TypeScriptu..62
Zdrojový kód 60 – Vlastní typ události v TypeScriptu s využitím jQuery........................63
Zdrojový kód 61 – Dočasné zakázání a zrušení posluchače v TypeScriptu 63
Zdrojový kód 62 – Dočasné zakázání a zrušení posluchače v TypeScriptu s jQuery 64
Zdrojový kód 63 – Pozastavitelný posluchač v TypeScriptu 64
Zdrojový kód 64 – Zastavení propagace události v TypeScriptu......................................64
Zdrojový kód 65 – Zastavení propagace události v TypeScriptu s využitím jQuery 65

Zdrojový kód 66 – Zrušení výchozí reakce na událost v TypeScriptu 65
Zdrojový kód 67 – Zrušení výchozí reakce na událost v TypeScriptu za pomoci jQuery...65
Zdrojový kód 68 – Asynchronní komunikace se serverem v Dartu 65
Zdrojový kód 69 – Asynchronní komunikace se serverem v Dartu s knihovnou adaj 66
Zdrojový kód 70 – Asynchronní komunikace se serverem v TypeScriptu 67
Zdrojový kód 71 – Asynchronní komunikace se serverem v TypeScriptu pomocí jQuery.67
Zdrojový kód 72 – Používání externích knihoven v Dartu 68
Zdrojový kód 73 – Konfigurace RequireJS....69
Zdrojový kód 74 – Načtení knihovny pomocí RequireJS 69
Zdrojový kód 75 – Test v nástroji Selenium 71
Zdrojový kód 76 – Test v CasperJS....71

Úvod

V posledních době čím dál více získávají na důležitosti webové aplikace. Velké množství původně desktopových aplikací se přesouvá na internet a majoritní podíl nových aplikací cílí též na web. Důvody jsou jednoduché. Připojení k internetu má v dnešní době snad každý počítač, takže webové aplikace jsou dostupné pro každého a okamžitě. Webové aplikace jsou též dostupné na většině platforem. Už není potřeba psát verzi pro Windows, MacOS, Linux, různé mobilní verze a podobně. Stačí jedna aplikace dostupná skrze prohlížeč a je hotovo. Z pohledu vývoje aplikací je též jednodušší distribuce nových verzí. U webové aplikace není potřeba vytvářet aktualizační balíček, který si každý uživatel nainstaluje. Webovou aplikaci stačí aktualizovat pohodlně z kanceláře a nová verze je v mžiku dostupná všem jejím uživatelům.

S přesunem aplikací na web ale vyvstává drobný problém. Uživatelské rozhraní, které dříve bylo typicky psáno ve stejném jazyku jako zbytek aplikace, musí být nyní přetvořeno do webové stránky. Pokud navíc chceme aplikaci moderní a uživatelsky přívětivou, nevyhneme se implementaci alespoň minimální logiky právě v klientské části aplikace.

Především popularita webových aplikací dala za vznik jazykům, kterým se v mé práci budu věnovat – Dartu a TypeScriptu. Jedná se o jazyky poměrně nové a každý z nich je něčím specifický, zajímavý. Můžete tuto práci tedy využít k seznámení se s nimi.

Práce je rozdělena na dvě části – teoretickou a praktickou. V teoretické části se nejprve věnuji problematice skriptovacích jazyků jako celku, jejich rysům a možnostem. Zvolené jazyky – Dart a TypeScript – dávám do kontrastu s běžně používanými jazyky v tomto odvětví. Dále se v teoretické části zmiňuji o testování, a to především funkčním, s krátkou zmínkou o testování výkonovém.

Teoretická část je určena především těm, kteří chtějí nahlédnout do problematiky skriptovacích jazyků. Neuškodí však ani problému znalým, neboť poslouží jako stručné připomenutí a podá informaci o tom, jakým směrem se bude ubírat praktická část práce.

Praktická část už je zaměřena ryze na výše zmíněné jazyky – Dart a TypeScript. Dělí se na dvě větší podkapitoly, kde první z nich je zaměřena na možnosti použití jazyka na serveru. Druhá podkapitola se pak věnuje možnostem skriptování ve webovém prohlížeči.

Výstupem práce je pak zhodnocení způsobilosti jazyků Dart a TypeScript zaujmout místo mezi jejich běžně používanými alternativami.

Skriptovací jazyky

1 Rozdělení programovacích jazyků

Jak už to ve světě bývá, lze na každý problém nahlížet z vícera úhlů. Výjimkou není samozřejmě ani problematika programovacích jazyků.

Programovací jazyky lze dělit na základě velkého množství kritérií. Pro účely mé práce je vhodné zmínit především dělení na jazyky se statickým / dynamickým typovým systémem a dále dělení dle [1] na systémové a skriptovací programovací jazyky. Těmto dělením se budu věnovat v následujících podkapitolách.

Je slušností zmínit i další kritéria, na základě kterých můžeme programovací jazyky dělit. Hlouběji se jim však věnovat nebudu a pouze u každého uvedu odkaz na literaturu, kde je možné se dozvědět více. Programovací jazyky tedy můžeme dále dělit na:

- vyšší a nižší [2],
- kompilované a interpretované [3],
- procedurální a neprocedurální [4].

Jak můžeme vidět, způsobů dělení programovacích jazyků je opravdu dost. Nyní však už zpátky k výše zmíněným, pro mou práci nejdůležitějším, způsobům rozdělení jazyků.

1.1 Jazyky se statickým / dynamickým typovým systémem

Statičnost / dynamičnost typového systému spočívá v čase, kdy se vyhodnocují typy proměnných a hodnot do nich přiřazovaných (dále už jen typová kontrola). Statický typový systém se vyznačuje tím, že typová kontrola probíhá už při kompilaci zdrojového kódu. U jazyků s dynamickým typovým systémem probíhá toto vyhodnocení až za běhu programu.

Každý přístup má své klady i zápory, které se nyní pokusím shrnout. Hlavní výhoda statického typového systému vychází ze samotné jeho podstaty. Jelikož typová kontrola probíhá již při kompilaci (je to tzv. statická kontrola [5]), je poměrně velké procento nejčastějších chyb odchyceno ještě před spuštěním aplikace. S dynamickým typovým systémem odhalíme tyto chyby až za běhu programu, což vede k nutnosti důkladnějšího otestování aplikace.

Naopak velkou výhodou dynamického typového systému je, že se o typ proměnné nemusíme starat – můžeme do ní přiřadit cokoliv. Zároveň dynamický typový systém velice často přichází s automatickou konverzí typů[1], což programátorům na jednu stranu ulehčuje práci, na druhou přináší možnost vzniku poměrně špatně odhalitelných chyb.

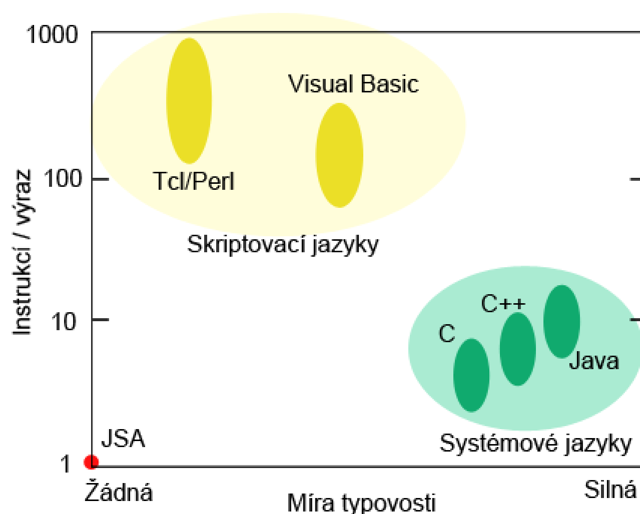
[1] Umožní programátorovi například zápis if (foo) { doSomething(); }, kde foo může být libovolného typu – vždy se interním mechanismem jazyka převede na booleovskou hodnotu. Například prázdný řetězec na false, neprázdný na true apod.

1.2 Systémové / skriptovací programovací jazyky

K vysvětlení rozdílu mezi systémovými a skriptovacími jazyky je vhodné zmínit, jak nejčastěji jazyky z těchto skupin vznikaly. Systémové programovací jazyky vznikly jako alternativa k jazykům symbolických adres (assembly languages). V jazycích symbolických adres odpovídá každý příkaz jedné instrukci procesoru a programátor si musí se vším poradit na této nízké úrovni. Jelikož je velice náročné jak tyto programy psát, tak je následně udržovat, začaly vznikat jazyky obsahující výrazy sestavené z posloupnosti jednotek až desítek instrukcí. To snížilo množství kódu k psaní a údržbě. Dále tyto jazyky typicky používají statický typový systém.

Skriptovací jazyky reprezentují zcela jiný přístup. Předpokládají existenci množství vestavěných, implementačně netriviálních „komponent“ (různé pokročilé kolekce, funkce pro práci s nimi, mnoho běžně používaných tříd – pro práci s datem, síťovými protokoly apod.), které se u systémových programovacích jazyků objevují až ve formě různých frameworků a knihoven. V porovnání se systémovými programovacími jazyky zde jeden výraz odpovídá přibližně 100–1000 instrukcím. Skriptovací jazyky, na rozdíl od systémových, využívají často dynamický typový systém a bývají interpretované.

Na obrázku 1 je pak graficky znázorněna klasifikace jazyků dle počtu instrukcí na výraz a míry typovosti jazyka.



Obrázek 1 – Klasifikace jazyků dle počtu instrukcí na výraz a míry typovosti
Zdroj: [1]

1.3 Dynamické programovací jazyky

Na první pohled by se mohlo zdát, že dynamické jazyky jsou vlastně jazyky s dynamickým typovým systémem. Nicméně pravda je někde jinde. Dynamický programovací jazyk vlastně vůbec nemusí mít dynamický typový systém. Ona dynamičnost jazyka je dána možnostmi,

které nabízí. Ačkoli je přesná definice dynamického jazyka pouze věcí názoru a spekulací, existují základní rysy, kterými se jazyky označované jako dynamické vyznačují. Jak říká [6], patří mezi ně:

- closures,
- object runtime alternation,
- funkce eval,
- podpora funkcionálního přístupu,
- reflexe a další.

Closures, neboli česky uzávěry, jsou speciální funkce vidící do kontextu, ve kterém jsou definované [7]. Využití pro ně najdeme například při práci s událostmi nebo jako prostý callback[1]. Ukázku takové uzávěry můžete vidět pod tímto odstavcem. Jedná se o samostatnou funkci, avšak používá i proměnné z nadřazeného kontextu.

```
function doSomeAction(number, onFinished){
  ...
  onFinished();
}

var number = 5;
doSomeAction(function() {
  console.log('action with number ' + number + ' is finished');
});
```

Zdrojový kód 1 – Closure

Object runtime alternation spočívá přesně v tom, co říká překlad tohoto termínu, a to možnost upravovat objekty za běhu programu. Úprava je zde myšlena ve smyslu přidávání a odebírání členských proměnných, přepisování metod a podobně, jak můžete vidět na ukázce pod odstavcem.

```
var obj = new Object();
obj.name = 'test';
obj.foo = 1;
delete obj.foo;
```

Zdrojový kód 2 – Object runtime alternation

Funkce eval se vyznačuje tím, že umožňuje vykonat kód, který je jí předán ve formě řetězce. Pro někoho je tato funkce vysvobozením, pro jiného kořen všeho zla[2]. Pravdou však je, že sráží čitelnost a předvídatelnost kódu na minimum. Posoudit můžete sami na ukázce na druhé straně.

[1] Funkce zavolaná jako výsledek jiné funkce.
[2] „Eval is evil" je známý výrok jednoho z nejslavnějších současných programátorů, Douglase Crockforda.

```
var functionName = 'test';
var code = 'function ' + functionName + '(){}';
eval(code);
test(); // funkce s dynamickým názvem
```

Zdrojový kód 3 – Funkce eval

Podporou funkcionálního programování se myslí především možnost používání higher-order funkcí[1], closures, currying[2] a další. Ukázka takového funkcionálního programování je opět pod odstavcem. Jedná se o součet všech sudých čísel od jedné do pěti.

```
numbers = [1 to 5] // [1, 2, 3, 4, 5]
numbers |> filter even |> sum // 6
```

Zdrojový kód 4 – Ukázka funkcionálního programování

Pojem reflexe znamená v kontextu programovacích jazyků především prostředek ke zkoumání různých jazykových konstruktů (tříd, funkcí, objektů) pomocí vestavěných jazykových prostředků. Například na ukázce níže přistupujeme k názvům všech členských proměnných objektu *o*.

```
var properties = MirrorHelper.getAllProperties(o);
for (var name in properties.keys) {
  print(name);
}
```

Zdrojový kód 5 – Reflexe

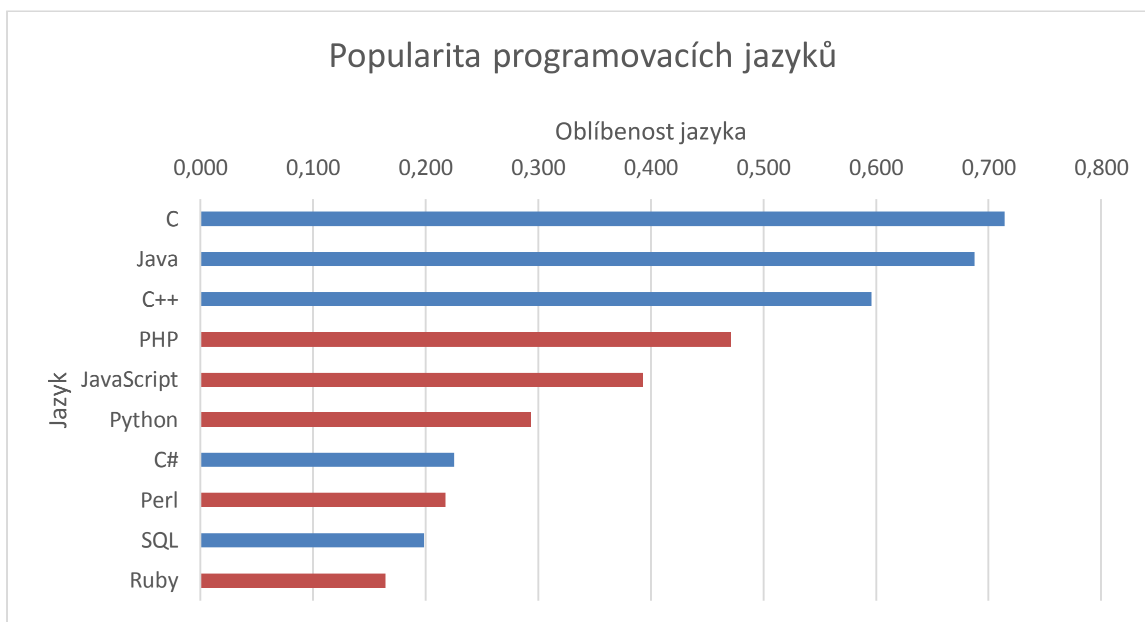
1.4 Běžně používané skriptovací jazyky

Říci, jaké jazyky jsou běžně používané a jaké ne, není nic jednoduchého. Jako příklad běžně používaných skriptovacích jazyků jsem pro svou práci zvolil PHP, Python a JavaScript. Vycházel jsem při tom z průzkumu LangPop.com, který pochází z roku 2011. Data tohoto průzkumu pocházejí z celkem sedmi zdrojů, a to z:

- vyhledávaní na Yahoo.com,
- nabídek práce Craiglist.com,
- knižní databáze Powell's Books,
- databáze open-source projektů a knihoven Freshmeat.net
- statistik projekt hostingu Google Code,
- aktivity uživatelů na del.icio.us,
- sociální sítě zaměřené na open-source – Ohloh.

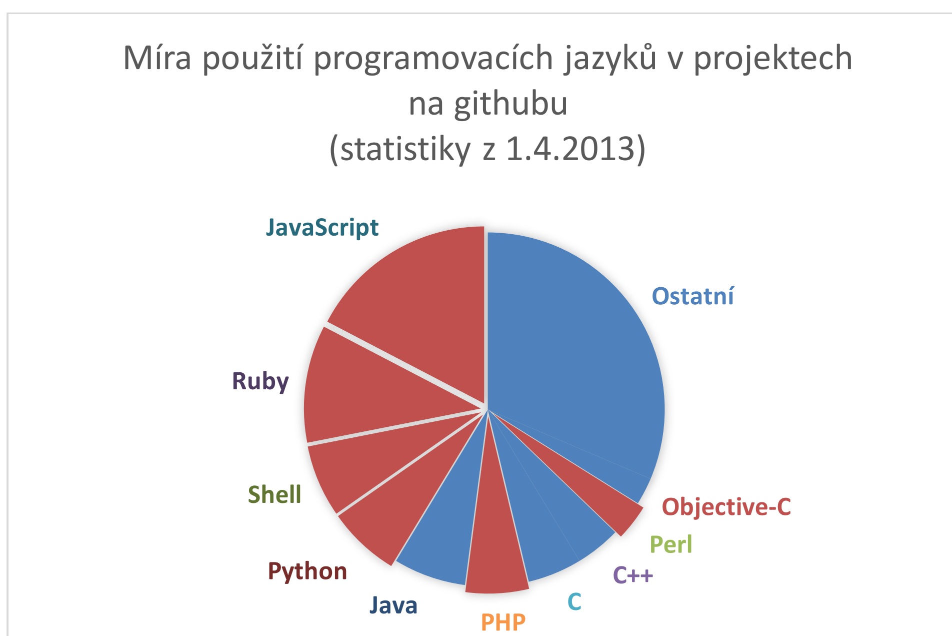
[1] Funkce přijímající na vstupu jednu či více funkcí.
[2] Možnost jednoduše převést volání metody s více parametry na postupné volání (partial evaluation) – např. místo add(1,2,3) lze zavolat add(1)(2)(3).

Konkrétní výstup je možné vidět na obrázku 2. Graf zachycuje normované hodnoty ze všech uvedených zdrojů, kdy každý z nich má stejnou váhu. Skriptovací jazyky jsou pak odlišeny červenou barvou.



Obrázek 2 – Oblíbenost programovacích jazyků
Zdroj: LangPop.com

V dnešní době by bylo vhodné do zdrojů průzkumu začlenit i statistiky z nejvýznamnější služby kombinující prvky sociální sítě a projekt hostingu, GitHubu. Tento zdroj však ve výše zmíněném průzkumu není a uvádím ho tedy samostatně jako obrázek 3. Skriptovací jazyky jsou opět odlišeny červenou barvou.



Obrázek 3 – Míra použití programovacích jazyků na GitHubu
Zdroj: GitHub.com

Z grafů vyplývá, že nelze úplně přesně určit, jaký konkrétní jazyk je nejpoužívanější (poměrně výrazně se liší i jednotlivé zdroje pro LangPop.com), nicméně lze vybrat jistou skupinu jazyků, které se ve statistikách umisťují na prvních příčkách. Jsou jimi PHP, Python, JavaScript a Ruby a Perl. Pro účely mé práce postačí, když se budu zabývat prvními třemi zmíněnými.

1.4.1 PHP

PHP je jazyk dle statistik LangPop.com nejoblíbenějším jazykem z rodiny skriptovacích jazyků. Jako hlavní důvody jeho oblíbenosti lze označit poměrně strmou křivku učení a dostupnost hostingových služeb.

Historie

PHP vzniklo v roce 1994 jako prostředek k obohacení webových stránek o dynamické prvky. Postupem času se z něj stal plnohodnotný jazyk, ve kterém se dají psát jak jednoduché skripty, tak celé aplikace. Prošlo zajímavým vývojem, kdy se z ryze imperativního, procedurálního jazyka stal jazyk hybridní[1]. Stalo se tak v roce 2004, kdy bylo představeno PHP 5 podporující objektově orientovaný přístup.

Rysy a paradigma

PHP je dynamický interpretovaný jazyk s dynamickým typovým systémem a syntaxí odvozenou od jazyka C. Jak už bylo řečeno v části o historii, jedná se multiparadigmatický jazyk. Konkrétně PHP podporuje imperativní a strukturovaný (objektově orientovaný) přístup.

PHP se využívá především na webových serverech, na které je i primárně zaměřeno. Jeho využití jako skriptovacího nástroje například v příkazové řádce je výjimečné. V prohlížečích nelze kód PHP spustit vůbec.

Ukázka kódu

Na následujících ukázkách můžeme vidět dva různé způsoby použití PHP v ukázce 6 je PHP použito jako šablonovací jazyk, kde přímo do HTML kódu vkládá dynamické útržky kódu.

[1] Hybridní neboli multiparadigmatický znamená, že jazyk dovoluje programátorovi používat více programovacích paradigmat – např. objektové, funkcionální, imperativní apod.

```
<!DOCTYPE html>
<html>
<head>
  <title><?php echo $template->title; ?></title>
</head>
<body>
  <h1><?php echo $template->heading; ?></h1>
  <nav>
    <ul>
      <?php foreach ($template->menuItems as $menuItem): ?>
      <li>
        <a href="<?php echo $menuItem->link; ?>">
          <?php echo $menuItem->text; ?>
        </a>
      </li>
      <?php endforeach; ?>
    </ul>
  </nav>
  <section>
    <?php echo $template->content; ?>
  </section>
</body>
</html>
```

Zdrojový kód 6 – Použití PHP v HTML

Druhá ukázka znázorňuje, jak by mohl vypadat kód, který připraví data pro šablonu v ukázce první. Ukázky zároveň znázorňují oddělení logiky a obsahu pomocí návrhového vzoru MVC.

```
<?php

class ExamplePresenter {
  private $exampleProvider;
  private $menuStructure;

  function __construct(ExampleProvider $exampleProvider,
                       MenuStructure $menuStructure) {
    $this->exampleProvider = $exampleProvider;
    $this->menuStructure = $menuStructure;
  }

  function renderDefault() {
    $example = $this->exampleProvider->getExample();

    $this->template->title = $example->title;
    $this->template->content = $example->content;
    $this->template->menuItems = $this->menuStructure;
  }
}
```

Zdrojový kód 7 – Objektově orientovaný přístup v PHP

1.4.2 Python

Dle LangPop.com je Python třetí neoblíbenější jazyk ze skupiny skriptovacích jazyků. Toto místo obsadil zřejmě díky své univerzálnosti. Jeho záběr je obrovský – lze v něm psát vše

od jednoduchých skriptů pro příkazovou řádku přes backendy[1] webových aplikací až po aplikace desktopové (především díky podpoře knihoven Qt a GTK+).

Historie

Počátky Pythonu se datují k roku 1991. Inspiroval se jazykem ABC, který sloužil spíše k výukovým účelům. Od jazyka ABC přebral Python hlavně způsob zápisu kódu a nízkou vstupní bariéru pro jeho naučení. Vývoj jazyka je spíš evoluční – spousta drobných změn – než jako v případě PHP, kdy při přechodu z jedné verze na druhou přibyl celý objektově orientovaný přístup. Jako poměrně velkou změnu můžeme označit rozdíl mezi verzemi 2.x a 3.x, které jsou v současné době vyvíjeny současně především kvůli vzájemné nekompatibilitě.

Rysy a paradigma

Python je, stejně jako PHP, dynamický interpretovaný jazyk s dynamickým typovým systémem. Podporuje objektový, procedurální a částečně i funkcionální paradigma. Jeho syntaxe je založená na odsazování – jednotlivé bloky se tedy nedefinují složenými závorkami, jako tomu je u C-like syntaxí, nýbrž právě odsazením.

Ukázka kódu

V Pythonu se zřídkakdy píšou webové aplikace bez použití jakéhokoliv frameworku. Nejznámějším z nich je pravděpodobně Django. Také z něj budou následující ukázky kódu. Ukázky funkčně odpovídají těm v PHP, takže jde jednoduše porovnat rozdíly.

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ title  }}</title>
</head>
<body>
  <h1>{{ heading }}</h1>
  <nav>
    <ul>
      {% for menu_item in menu_items %}
      <li>
        <a href="{{ menu_item.link }}">
          {{ menu_item.text }}
        </a>
      </li>
      {% endfor %}
    </ul>
  </nav>
  <section>
    {{ content | safe }}
  </section>
</body>
</html>
```

Zdrojový kód 8 – Šablony v Djangu

[1] Serverová část webové aplikace.

Nesmíme však zapomínat na to, že se zde jedná o šablonovací jazyk z Djanga. Bez něj by byl výstup z Pythonu do HTML poměrně kostrbatý. Jak by to zhruba vypadalo, je nastíněno v ukázce 9.

```
>>> template = "<html><body><h1>Hello %s!</h1></body></html>"
>>> print template % "World"
<html><body><h1>Hello World!</h1></body></html>

>>> from string import Template
>>> template = Template("<html><body><h1>Hello ${name}!</h1></body></html>")
>>> print template.substitute(dict(name='John Doe'))
<html><body><h1>Hello John Doe!</h1></body></html>
```

Zdrojový kód 9 – Výstup z Pythonu do HTML bez šablonovacího jazyka

Třetí ukázka opět ukazuje přípravu dat pro šablonu. Zde už vidíme, že se kód zásadně liší od PHP. Syntaxe je úspornější než v případě PHP.

```
from django.shortcuts import render_to_response

from example.models import examples
from example import menu_structure

def home(request):
    example = examples.objects.get(id=1)

    response = {
         'title': example.title,
         'heading': example.heading,
         'content': example.content,
         'menu_items': menu_structure
         }
    return render_to_response('index.html', response)
```

Zdrojový kód 10 – Ukázka kódu v Pythonu

1.4.3 JavaScript

JavaScriptu se budu věnovat trochu víc než předešlým jazykům především kvůli tomu, že je to právě JavaScript, který otevřel dveře skriptování ve webovém prohlížeči. A dosud se žádnému jinému jazyku nepodařilo na tomto poli prosadit. V praxi to tedy znamená, že pokud chcete tvořit dynamické HTML stránky, nezbývá vám, než použít JavaScript nebo jazyk, který do něj jde zkompilovat (viz kapitola 1.5 teoretické části).

Historie

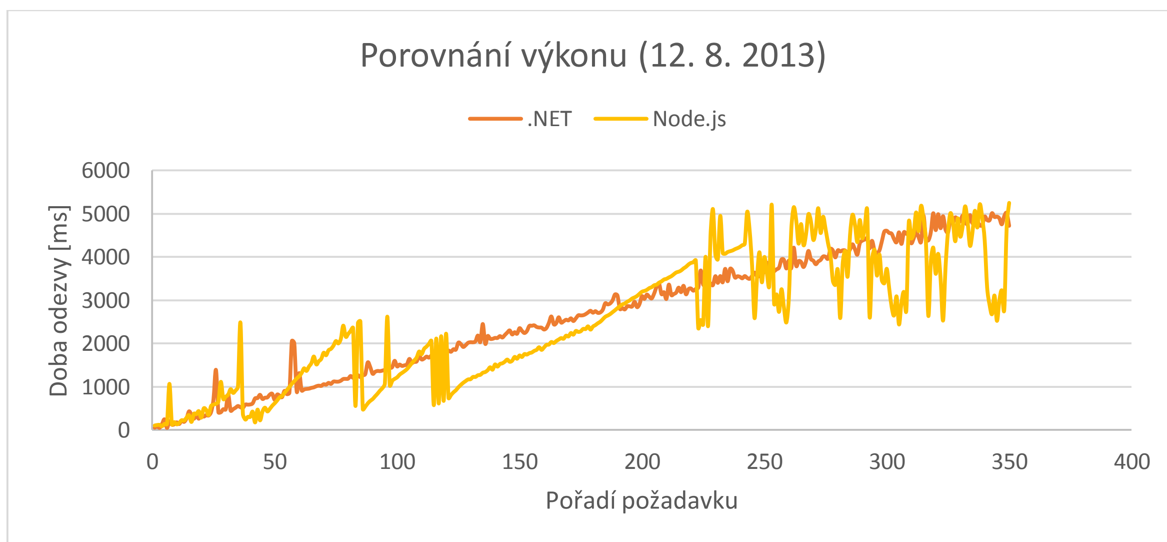
Jak říká [8], počátky JavaScriptu se datují k roku 1995, kdy byl firmou Netscape začleněn do jejich webového prohlížeče. V roce předchozím firma Sun představila jazyk Java, od které JavaScript (původně pojmenovaný LiveScript) přebral jak syntaxi, tak část jména. V roce 1997 byl standardizován asociací ECMA a od té doby má své místo ve všech prohlížečích.

Další zlom v historii JavaScriptu nastal v roce 2009, kdy byl uveden Node.js. Node.js je JavaScript engine V8 vyvíjený společností Google obohacený o sadu standardních knihoven spustitelný mimo prohlížeč. V praxi to znamená, že si JavaScript našel cestu z prohlížečů do příkazové řádky, kde díky standardním knihovnám může fungovat jako například jako webserver.

Rysy a paradigma

JavaScript je dynamický, objektově orientovaný jazyk s dynamickým typovým systémem založený na prototypové dědičnosti. Podporuje též procedurální a funkcionální přístup.

Původně byl určený pro oživení webových stránek – validace formulářů, modifikace webových stránek přímo v prohlížeči na základě uživatelských podnětů apod. Ačkoli se jedná o jazyk poměrně starý, zažívá teď skutečný „boom". Díky neustálým optimalizacím interpretů JavaScriptu se jeho výkon, jak ukazuje graf na obrázku 4, v některých situacích značně přibližuje výkonu běžně používaných platforem. Graf znázorňuje schopnost platformy rychle odbavit příchozí požadavek na webserver. Porovnávané platformy jsou .NET a Node.js.



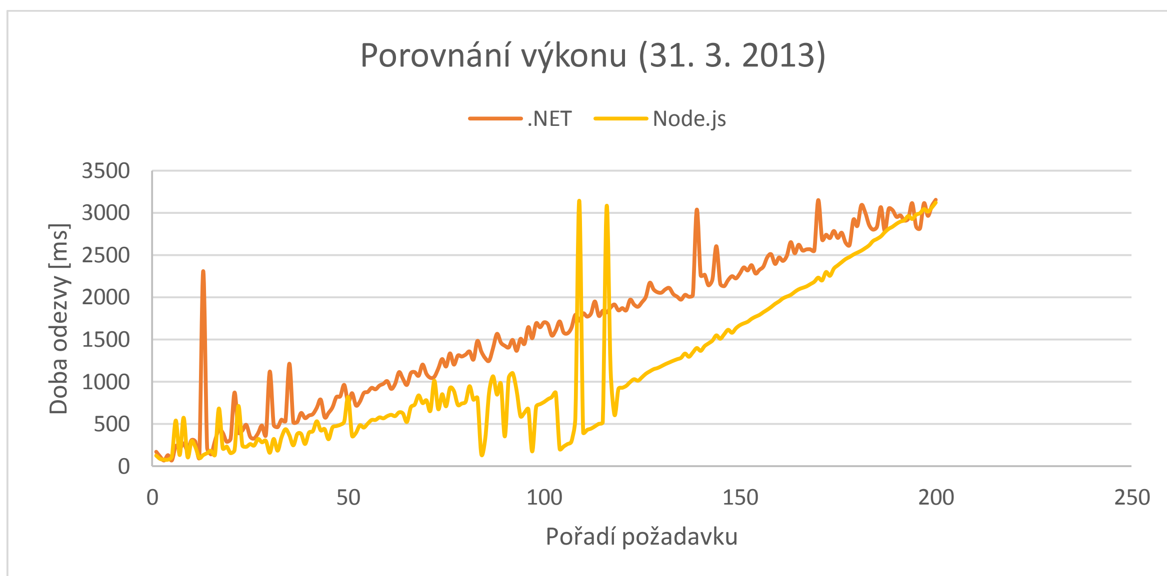
Obrázek 4 – Porovnání výkonu .NET vs. Node.js z 12. 8. 2013
Zdroj: vlastní zpracování

Jedná se o test sestavený v [9] s následujícími parametry:

- verze .NET: 4.5,
- verze Node.js: 0.10.13,
- počet paralelních požadavků: 350.

Test byl vyhotoven 12. 8. 2013. Test jsem prováděl znovu, jelikož autorův test končil přesně v okamžiku, kdy se Node.js dostal na stejnou úroveň jako .NET (viz Obrázek 5) a vypadalo, že byl test cíleně ukončen kvůli tomu, aby by Node.js dopadl v testu dobře. Tato hypotéza

se nepotvrdila a obě platformy byly i nadále vyrovnané. Velké výkyvy u Node.js jsou způsobeny odbavováním požadavků v jiném pořadí, než v jakém byly odeslány.



Obrázek 5 – Porovnání výkonu .NET a Node.js z 31. 3. 2013
Zdroj: vlastní zpracování na základě dat z [9]

Prototypová dědičnost

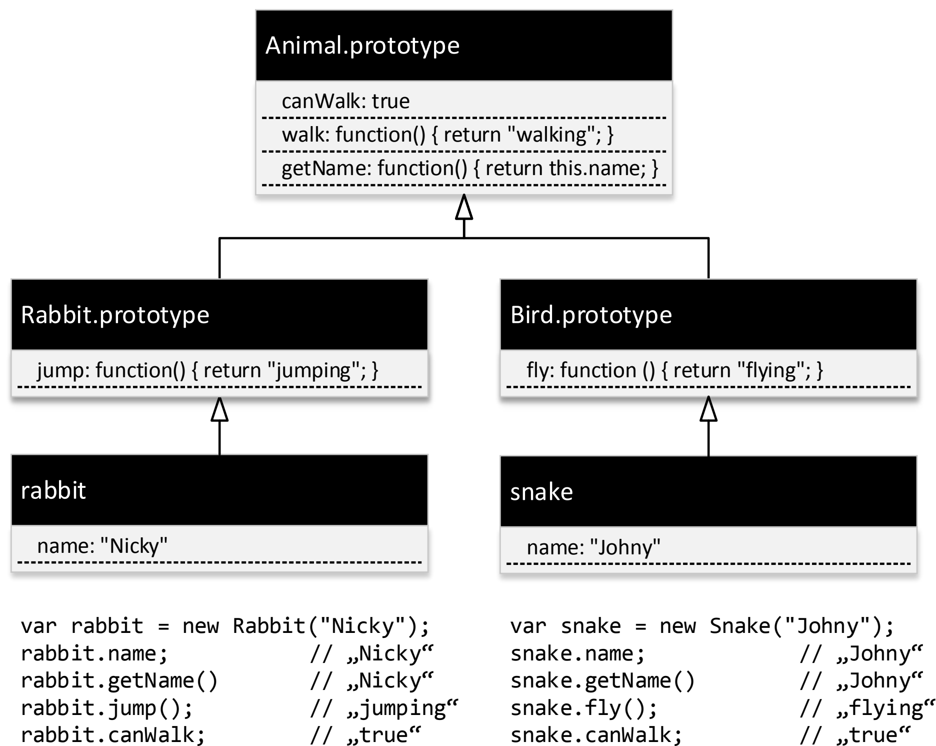
V rysech programovacího jazyka jsem zmínil, že je založen na prototypové dědičnosti. Abych se mohl vymezit oproti klasické, třídní dědičnosti, pokusím se ji nejprve v kostce shrnout. V třídní dědičnosti mohou třídy zdědit atributy a chování existujících tříd, které se nazývají rodičovské třídy. Výsledné třídy jsou pak potomky (odvozené třídy) těchto tříd.

Oproti tomu v prototypové dědičnosti je tzv. prototyp šablonou pro konstrukci třídy. Tento objekt je pak navzájem sdílen všemi objekty se stejným prototypem. V JavaScriptu se prototyp nastavuje pomocí vlastnosti *prototype* a ve výchozím stavu je to prázdný objekt.

V praxi to funguje zhruba takto. Pokud se snažíme přistoupit například k vlastnosti objektu, provede se následující:

1. Interpret se podívá, zda má daný objekt definovanou požadovanou vlastnost.
2. Pokud ano, danou vlastnost zpřístupní.
3. Pokud ne, podívá se do prototypu tohoto objektu, existuje-li. Tj. pokračuje jako v kroku 1, jen s prototypem. Pokud neexistuje, vrací hodnotu *undefined*.

Toto chování je znázorněno na obrázku 6.



Obrázek 6 – Diagram znázorňující prototypovou dědičnosti
Zdroj: vlastní zpracování

Problematika prototypové dědičnosti však dalece přesahuje rámec této diplomové práce. Odkáži proto na třídílný seriál o třídách a dědičnost v JavaScriptu [10], ve kterém je vše důsledně vysvětleno.

Ukázka kódu

Co se týče příkladů kódu, zvolil jsem ukázku dvou různých přístupů. První je ukázka renderování HTML na serveru pomocí šablonovacího jazyku *Jade*. HTML struktura se zde skládá pomocí odsazování (podobně jako bloky v Pythonu). Šablonu můžete vidět na ukázce 11.

```
!!! 5
html
  head
    title=title
  body
    h1=heading
    nav
      ul
        - each menuItem in menuItems
          li: a(href=menuItem.link)=menuItem.text
    section!=content
```

Zdrojový kód 11 – Šablona v Jade

Syntaxe je poměrně úsporná, avšak z vlastní zkušenosti musím říct, že vytváření bloků pomocí odsazování je pro tak složité struktury, jako je HTML, celkem nepříjemné. Pro čtenáře takového kódu je poměrně složitě odhadnutelné, kde blok končí, a často dojde na

ruční počítání jednotlivých mezer. Na následující ukázce je pak znázorněno plnění takové šablony daty pomocí frameworku *express*, ve spojení s kterým se *Jade* často používá.

```
var exampleProvider = require('../providers/ExampleProvider'),
    menuStructure = require('../MenuStructure');

module.exports = function(app) {
    app.get('/example', function(req, res) {
        var example = exampleProvider.getExample();
        var templateData = {
            title: example.title,
            heading: example.heading,
            content: example.content,
            menuStructure: menuStructure
        };

        res.render('example', templateData);
    });
}
```

Zdrojový kód 12 – Plnění šablony ve frameworku express

Druhým přístupem je vytváření HTML až na straně klienta – v prohlížeči. Zde jsem pro ukázku použil framework *Knockout*. Výsledné HTML se vytváří pomocí HTML5 data atributů, na základě kterých framework šablonu modifikuje a plní daty.

```
<!DOCTYPE html>
<html>
<head>
  <title data-bind="text: title"></title>
</head>
<body>
  <h1 data-bind="text: heading"></h1>
  <nav>
    <ul data-bind="foreach: menuItems">
      <li><a data-bind="href: link, text: text"></a></li>
    </ul>
  </nav>
  <section data-bind="html: content">
  </section>
</body>
</html>
```

Zdrojový kód 13 – Šablona v Knockoutu

Plnění takové šablony ve frameworku *Knockout* probíhá pomocí takzvaného *ViewModelu*, což je zjednodušeně objekt obsahující zobrazovaná data. Jak vypadá takový *ViewModel* můžete vidět na ukázce zdrojového kódu č. 14.

```
function ExampleViewModel() {
    this.title = ko.observable();
    this.heading = ko.observable();
    this.content = ko.observable();
    this.menuStructure = ko.observablArray();
}

var exampleViewModel = new ExampleViewModel();
fillExampleViewModel(exampleViewModel);

ko.applyBindings(exampleViewModel);
```

Zdrojový kód 14 – Plnění šablony ve frameworku Knockout

1.5 Transpilery

Transpiler je označení pro source-to-source kompilátor, neboli kompilátor, který vezme zdrojový kód v jednom jazyce a přetvoří ho do zdrojového kódu v jiném jazyce. Například transpilerů do JavaScriptu existuje velké množství. Z nejpoužívanějších mohu zmínit například CoffeeScript, LiveScript, GWT, Script#, Haxe a další. Existují samozřejmě i transpilery do jiných jazyků. Například cfront překládající kód v C++ do jazyka C nebo HipHop for PHP sloužící k překladu PHP do C++, případně J2ObjC, který dokáže kód napsaný v Javě přeložit do Objective-C, a spousta dalších. Existuje i transpiler pro překlad kódu z Pythonu verze 2.x do verze 3.x – pro překlenutí rozdílů, které jsem zmiňoval v kapitole 1.4.2.

Transpiler typicky kompiluje vyšší programovací jazyk do nižšího, případně převádí kód do jazyka na stejné úrovni. Opačný směr není v obecné rovině možný[1].

Jeden z jazyků, na které se má práce primárně zaměřuje, TypeScript, je též kompilován do JavaScriptu. Druhý, Dart, má tuto možnost také, nicméně může být spuštěn i ve svém VM[2].

1.6 Dart

Dart je poměrně nový jazyk pocházející od společnosti Google. Vznikl jako jazyk, který má nahradit JavaScript na poli webových aplikací. Hlavní motivací bylo blížící se dosažení pomyslného stropu výkonnosti interpretů JavaScriptu a roztříštěnost JavaScriptového světa (různé postupy pro vytváření tříd, modulů apod.).

1.6.1 Vývoj jazyka

Dart byl poprvé představen na GOTO konferenci v říjnu roku 2010 s již zmíněným cílem nahradit JavaScript. Jazyk však nebyl přijat s takovým nadšením, jaké si Google sliboval, a bylo poukazováno na spoustu nedostatků. Od té doby se Dart neustále vyvíjí a prochází spoustou změn. Největší množství změn v jazyce jako takovém bylo v milníku M1 následovaný milníky M1 až M5, který byl označen jako beta.

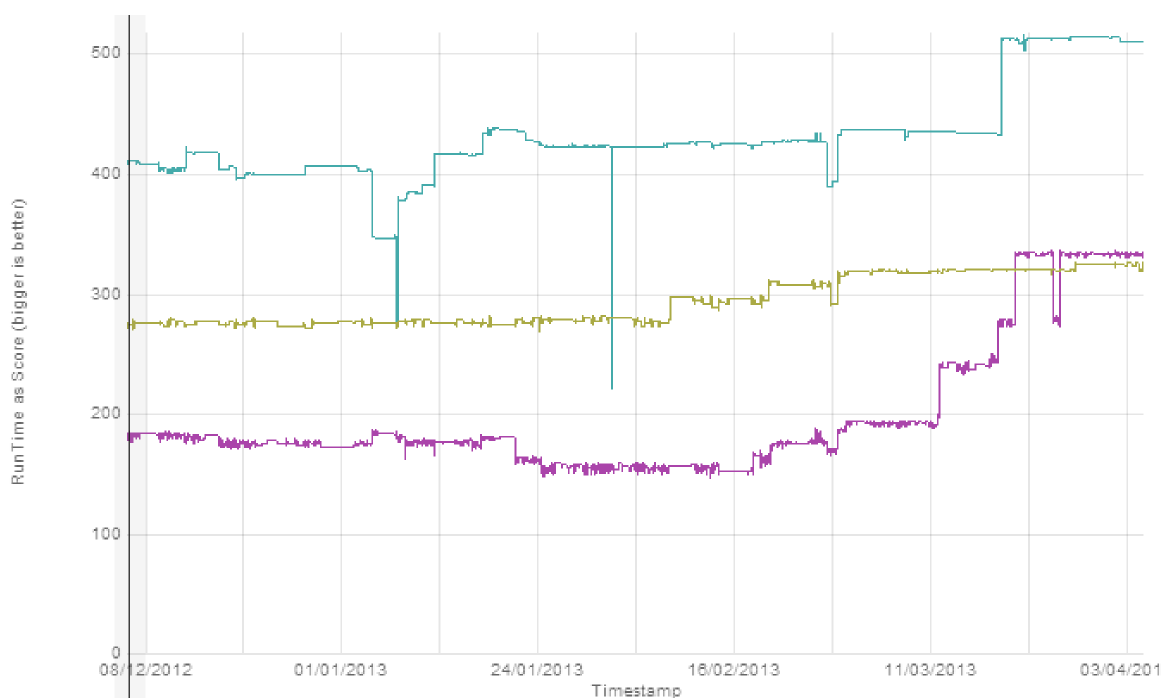
[1] Vyžadoval by extrémně hlubokou analýzu kódu a tvůrčí myšlení, kterým oplývá jen člověk.
[2] Virtual Machine – sada programů a struktur sloužící ke spouštění kódu programu.

Milník M1 přišel v červenci 2012 a přinesl, jak už bylo naznačeno, obrovské množství změn [11]. Hlavními změnami byly

- first-class typy – přineslo možnost odkazovat se na třídu pomocí jejího názvu,
- implicitní rozhraní – odstranilo nutnost psát datové typy tam, kde to kompilátor dokáže sám odvodit,
- kaskádový operátor – slouží pro jednoduché vytvoření fluent interface[1],
- operátor ekvivalence nahrazen top-level funkcí – z Dartu zmizel operátor složený ze tří znamének „rovná se" a byl nahrazen funkcí *identical*.

Milníku M2 bylo dosaženo o poznání rychlení než M1, a to už v prosinci 2012. Byly do něj začleněny především kosmetické změny – drobný refaktoring, vylepšená kompilace do JavaScriptu, vylepšení dokumentace a další [12].

Zajímavým okamžikem v historii Dartu je výrazně vylepšená kompilace do JavaScriptu, kdy kód kompilovaný z Dartu předčil ve výkonu ručně psaný JavaScript (srovnání zobrazeno na obrázku 7). Tato situace nastává samozřejmě jen v některých případech, nicméně lze to označit za úspěch. Tato optimalizace byla do Dartu začleněna 28. března 2013.



Obrázek 7 – Graf výkonu Dartu, JavaScriptu ručně psaného a kompilovaného z Dartu
Zdroj: http://www.dartlang.org/performance/ Dne: 5. dubna 2013

[1] Zápis metod ve tvaru object.doThis().doThat()

V následujících milnících docházelo především k drobnému rozšiřování a hlavně k ustalování API. Krátce shrnuto probíhal vývoj zhruba takto:

- milník M3 – nové API práci se streamy, s každou asynchronní akcí lze nyní pracovat jako se streamem [13],
- milník M4 – ustálení API v knihovnách *core*, *collection* a *async*, možnost používat třídy jako mixiny[1] [14],
- milník M5 (beta) – výkonnostní optimalizace, opravy chyb, drobné rozšíření stávajících knihoven [15].

1.6.2 Základní rysy, paradigma

Dart je jazykem dvou tváří. Může běžet jak ve svém vlastním VM, tak být zkompilován do JavaScriptu. Nevýhodou kompilace do JavaScriptu je mnohem nižší výkon. Výhodou naopak rozšířenost interpretů JavaScriptu – hlavně na poli webových prohlížečů.

Dart je objektově orientovaný jazyk s funkcionálními prvky. Používá syntax odvozenou od jazyka C (bloky ohraničené složenými závorkami, parametry metod v kulatých závorkách apod.). Snaží se inspirovat v mnoha různých jazycích, ale zároveň přinést něco nového. Inspirací mohou být například lambda funkce, settery a gettery a další. Novinkou je například oproštění se od specifikace viditelnosti členských proměnných na úrovni třídy[2]. Místo toho je v Dartu použita viditelnost na úrovni knihovny. Uvnitř knihovny jsou viditelné všechny třídy, metody a atributy. Zvenku pak pouze ty, které nezačínají podtržítkem (dalo by se přirovnat k package private z Javy nebo modifikátoru internal z C#). Knihovny mohou být rozprostřené přes vícero souborů. Hlavní soubor knihovny je pak identifikován spojením *library <název knihovny>*. Další soubory poté *part of <název knihovny>*. Jak takový kód v Dartu vypadá, můžete vidět na ukázce níže.

```
// Computes the nth Fibonacci number.
int fibonacci(int n) {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// Prints a Fibonacci number.
void main() {
  int i = 20;
  String message = "fibonacci($i) = ${fibonacci(i)}";
  // Print the result in the "Console" box.
  print(message);
}
```

Zdrojový kód 15 – Ukázka kódu v Dartu

[1] Mixin – třída obsahující kombinaci vybraných metod z jiných tříd
[2] Dart neobsahuje modifikátory přístupu – private, protected, public.

1.6.3 Přínosy

Jelikož je Dart míněn jako přímá konkurence JavaScriptu, zmíním především přínosy, které má oproti němu.

Velkým benefitem je podpora tříd. Jak bylo řečeno, JavaScript je objektově orientovaný jazyk založený na prototypech, což nemusí všem vývojářům vyhovovat (a směr vývoje JS transpilerů tomu přinejmenším nasvědčuje). Dart je tedy založen na třídách a třídní dědičnosti známé z velkého množství objektově orientovaných jazyků.

Jako další plus vidím volitelné statické typování s pokročilým odvozováním typů. Programátorovi tak stačí psát typy pouze tam, kde to má smysl – kompilátor si zbytek jednoduše odvodí podle prováděných operací.

Obrovským přínosem je vestavěná podpora pro načítání knihoven. V JavaScriptových aplikacích (míněno v prohlížeči) je nutné pro každou knihovnu psát extra *<script>* tag a tyto tagy udržovat ve správném pořadí. Dart toto řeší naprosto transparentně na pozadí. Naopak při kompilaci z knihoven vytahuje pouze ten kód, který se nakonec použije, takže ještě uspoří přenesená data.

Ještě bych chtěl vypíchnout přímou podporu pro asynchronní zpracování ve formě takzvaných *Futures* založených na asynchronním návrhovém vzoru pro *Promise*.

1.6.4 Vývojová prostředí

Google poskytuje pro vývoj aplikací v Dartu vývojové prostředí nazvané Dart Editor vycházející ze známého vývojového prostředí Eclipse. Podpora jazyka je v současné verzi (0.6.14) bohužel pouze základní. Samozřejmostí je zvýrazňování kódu, označování syntaktických chyb. Z nástrojů pro refaktoring jsou dostupné: přejmenování, extrakce do metody a vytvoření getteru z metody. Nástroje pro generování kódu obsaženy bohužel nejsou – například pokud u třídy uvedeme, že implementuje nějaké rozhraní, vývojové prostředí oznámí, že metody z uvedeného rozhraní ve třídě nejsou obsaženy, ale už nenabídne možnost jejich předgenerování. To výrazně snižuje rychlost vývoje.

Dalším dostupným vývojovým prostředím je WebStorm od JetBrains s nainstalovaným rozšířením pro Dart. To je na tom kupodivu o něco lépe než oficiální prostředí od Googlu. Nabízí víceméně to samé, co Dart Editor, ale rozšiřuje funkcionalitu právě o možnost generování kódu.

Psát aplikace v Dartu lze i v jednodušších editorech typu Sublime Text, kde podpora daného jazyka většinou znamená jen zvýraznění syntaxe. Nicméně na kratší skripty též postačí.

1.7 TypeScript

TypeScript je jazyk vydaný společností Microsoft. Jedná se o jazyk kompilovaný do JavaScriptu (nemá vlastní interpreter). Sám Microsoft ho označuje jako nadmnožinu

JavaScriptu[1]. Jeho celá myšlenka tkví v transformaci JavaScriptu na takový jazyk, v kterém jsou velké projekty dobře strukturované a udržovatelné.

1.7.1 Vývoj jazyka

TypeScript byl uveden na začátku října 2012 a během krátké doby získal obrovskou popularitu. Podařilo se mu totiž jednoduše spojit svět JavaScriptu a statického typování s ponecháním všech výhod JavaScriptu.

Nyní je TypeScript stále ve fázi vývoje. Velký skok ve vývoji jazyka nastal s verzí 0.9, kdy byla zavedena generika. Kromě Dartu je tak zřejmě jediným transpilerem do JavaScriptu obsahujícím generika. V nadcházejících verzích je v plánu, doplnit například *protected* přístup k členským proměnným a metodám, syntaktické zkratky pro asynchronní programování známé z jazyka C# (async a await) nebo například mixiny.

Vzhledem k tomu, že se jedná o velice mladý jazyk, neudálo se toho za dobu jeho existence mnoho. Vzhledem k nadšení komunity mu ale předpovídám slibnou budoucnost.

1.7.2 Základní rysy, paradigma

Základní myšlenkou, jak už bylo řečeno, je přidat statickou kontrolu datových typů do JavaScriptu, sjednocení způsobu zápisu tříd a modulů a přinesení komfortu pro programátora v podobě pluginu do vývojového prostředí Visual Studio, díky kterému je například dostupné našeptávání, jaké je běžné například při vývoji v Javě či C#, nebo různé zvýrazňování kódu – například úseku se syntaktickou chybou.

```
// Computes the nth Fibonacci number.
var fibonacci = (n: number):number => {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// Prints a Fibonacci number.
var i = 20;
var message = 'fibonacci(' + i + ') = ' + fibonacci(i);
console.log(message);
```

Zdrojový kód 16 – Ukázka kódu v TypeScriptu

Zajímavým faktem o TypeScriptu je, že se snaží držet dopřednou kompatibilitu se specifikací ECMAScript 6, která je zatím ve fázi konceptu. Díky tomu by se mohl v budoucnu stát její první implementací, což by mohlo zajistit výhody pro aplikace v něm napsané.

Jako excelentní nápad vidím možnost přidávat externí definiční soubory k JavaScriptovým knihovnám, které tímto obohatí onu knihovnu o typy, které může kompilátor TypeScriptu

[1] „TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.“ [21]

používat. V praxi to vypadá tak, že se k souboru *knihovna.js* přidá soubor *knihovna.d.ts*, který typicky obsahuje rozhraní všech tříd a objektů v dané knihovně.

1.7.3 Přínosy

Hlavními přínosy TypeScriptu jsou konstrukty pro členění kódu a volitelné statické typování. Díky tomu mohou automatizované nástroje dobře rozumět kódu.

TypeScript je také pomyslný most z prostředí .NET do prostředí JavaScriptu. Programátoři, kteří pracují například v C#, tedy budou v TypeScriptu „jako doma“.

Jak už jsem zmínil v základních rysech, jako velké plus vidím externí definiční soubory. To umožňuje používat v TypeScriptu jakoukoli JavaScriptovou knihovnu tak, jako by byla napsána v TypeScriptu a hlavně může kompilátor použít statickou analýzu i nad kódem využívajícím libovolnou JavaScriptovou knihovnu.

1.7.4 Vývojová prostředí

Pro vývoj aplikací v TypeScriptu poskytuje Microsoft plugin do svého vývojového prostředí, Visual Studia. Práce s TypeScriptem je ve Visual Studio poměrně komfortní a dalo by se říci, že se kvalitou blíží prací s C#. Pokud navíc přidáme plugin Web Essentials, získáme plnohodnotný nástroj pro tvorbu webových aplikací.

Alternativou k Visual Studiu je, stejně jako u Dartu, vývojové prostředí WebStorm od JetBrains. S nainstalovanými patřičnými doplňky je ve srovnání s Visual Studiem na stejné, ne-li lepší, úrovni.

Stejně jako Dart, lze i TypeScript psát v jednodušších textových editorech, které v současné době TypeScript poměrně dobře podporují.

2 Testování softwaru

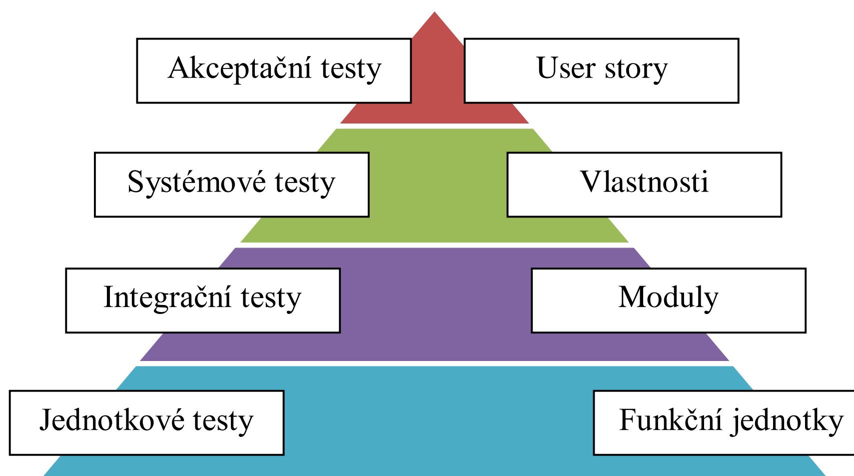
Když se (především mezi vývojáři) mluví o testování, je tím často myšleno psaní unit testů a jejich spouštění v nějakém unit-testing frameworku. Problematika testování je ale mnohem širší. Jak říká [16], testování lze rozdělit na funkční a nefunkční. Každé z této kategorií je věnována samostatná podkapitola. Testování lze dále dělit na ruční a automatizované. Ve své práci budu předpokládat plnou automatizaci testování.

Testování jako celek je často velmi podceňovanou součástí vývoje aplikací. Vývojáři se často spoléhají na svou neomylnost a dokonalé myšlení, což je kruciální chyba, za kterou někdo téměř vždy zaplatí. Čím později je chyba v aplikaci odhalena, tím dražší je její oprava. Tento rostoucí trend může být, jak říká [17], až exponenciální.

2.1 Funkční testování

Funkčním testováním se rozumí testování funkcí softwaru. Na testování funkčních prvků aplikace lze pohlížet různou optikou. Můžeme se zaměřit jak na testování malých funkčních bloků, ze kterých se aplikace skládá, tak testování celkové funkcionality zákazníkem [18].

Oba pohledy jsou extrémy funkčního testování, mezi kterými leží několik úrovní. Konkrétní úrovně testování je možné vidět na obrázku 8. Čím výše se v pyramidě nacházíme, tím komplexnější testované úlohy jsou a tím obtížnější je odhalit konkrétní příčinu chyby. Nižší úrovně ale naopak nemusí odhalit všechny chyby, které se nakonec v aplikaci projeví.



Obrázek 8 – Pyramida testování
Zdroj: vlastní zpracování na základě [19]

Při vývoji není potřeba implementovat všechny úrovně funkčního testování. Nicméně pokud chceme mít jistotu, že naše aplikace obsahuje minimum chyb, je záhodno se tomu nevyhýbat.

2.1.1 Unit testy

Unit testing, neboli testování jednotek, spočívá u objektově orientovaného programování v testování tříd a jejich metod. Testovanou jednotkou je pak samostatně testovatelná část aplikačního programu odstíněná od všech závislostí. V případě funkcionálního kódu se jedná o testování jednotlivých funkcí [20].

Unit testy jsou zpravidla zapsány formou zdrojového kódu a pro jejich vytváření a spouštění se využívají různé frameworky.

Jednotkové testování se poměrně špatně zavádí u již zaběhlých projektů. Vývojáři se pak nevyhnou rozsáhlému refaktoringu[1] kódu, případně často i velkým změnám v architektuře aplikace. Ideální je začít psát unit testy hned od začátku vývoje aplikace. Jak takový unit test vypadá, je možné vidět na následující ukázce. Konkrétně se jedná o unit test napsaný pomocí TypeScriptu v testovacím frameworku *mocha* s využitím assertovací[2] knihovny *chai*.

```
describe('urlUtils', () => {
    it('should be similar if the last slash is missing', () => {
        var dbUrl = 'http://example.com/',
            usersUrl = 'http://example.com';

        urlUtils.areSimilar(dbUrl, usersUrl).should.be.true;
    });
});
```

Zdrojový kód 17 – Unit test

2.1.2 Integrační testy

Integrační testy, často označované jako testy vnitřní integrity, slouží k ověření správné komunikace mezi jednotlivými komponentami uvnitř aplikace a k ověření komunikace komponent s operačním systémem, hardwarem a rozhraními externích systémů (například databáze apod.) [20].

U komponent se testuje, zda si správným způsobem a ve správný čas předají zprávy se správným obsahem i formátem. Integrační testy jsou často zahrnuty v systémových testech a zároveň se tyto dva termíny i často zaměňují, jelikož se integrita komponent často testuje v podobě, v jaké ji bude používat zákazník, což už spadá do systémových testů (viz kapitola 2.1.3 teoretické části).

Drobnou souvislost s integračními testy mají i takzvané integrační servery. Integrační server je typicky server se specializovaným softwarem, který si při každé změně v kódu (případně v pravidelných intervalech) stáhne zdrojové kód aplikace, pokusí se ji zkompilovat (případně provést jiné přípravné úkoly – spustí tzv. build skript) a následně spustí všechny automatické testy, které jsou pro aplikaci připraveny. Typicky jsou to unit testy, integrační testy a

[1] Přeskládání a přepsání částí kódu při zachování původní funkčnosti.
[2] Knihovna sloužící k ověřování správnosti hodnot – například, že jsou dvě hodnoty stejné apod.

systémové testy. V případě chyby je tak jednoduše dohledatelné, při jaké změně v kódu se stala a kdo za ni může.

2.1.3 Systémové testy

Systémové testy nastupují hned po provedení integračních. Prověřují aplikaci jako celek v podobě, v jaké by ji měl používat zákazník [20]. Ověřuje se soulad reálného chování s chováním očekávaným. Provádí se testování i všech možných negativních průběhů, validují se výstupy a podobně. Systémové testy by u aplikace měly být samozřejmostí – v opačném případě se zákazníkovi předává produkt, o kterém nevíme, zdali vůbec funguje.

Ukázku takového systémového testu můžeme vidět pod tímto odstavcem. Jedná se o ověření toho, že se na hlavní stránce zobrazí přesně 5 příspěvků – ověří se tak funkčnost napříč celou aplikací. U systémových testů se také často kladou různé předpoklady pro jejich pozitivní vyhodnocení. Například tento test předpokládá, že je vždy k dispozici alespoň 5 příspěvků, které může aplikace zobrazit.

```
test('number of posts on root page, () => {
    visit('/').then(() => {
        equal(find('.post').length, 5, 'The first page should have 5 posts');
  });
});
```

Zdrojový kód 18 – Systémový test

2.1.4 Akceptační testy

Akceptační testy jsou obdobou systémových testů, pouze probíhají na straně zákazníka, který s jejich pomocí validuje odvedenou práci [20]. Tyto testy bývají typicky prováděny ručně podle scénářů shodných se scénáři systémových testů.

Testy probíhají v testovacím prostředí u zákazníka. Nalezené nesrovnalosti mezi aplikací a specifikací jsou reportovány zpět vývojovému týmu.

2.2 Výkonové testování

Úplně jiným pohledem na testování aplikací je pak testování výkonové. Jedná se o testy, které nám ověří chování aplikace pod zátěží. Je velice důležité, v jakém prostředí se tyto testy provádějí. Typicky nemá smysl je provádět na testovacím prostředí, pokud není do detailu shodné s produkčním.

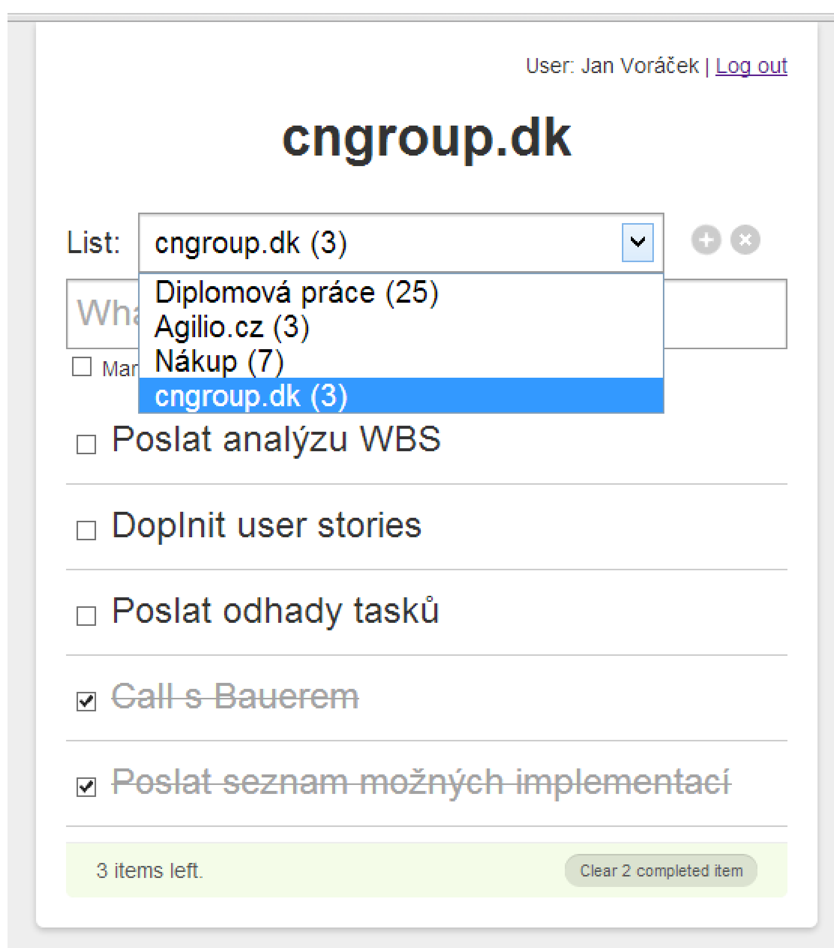
Výkonové testy lze také dělit na několik kategorií podle toho, jaký je cíl [21]. Zátěžové testy mohou být prováděny pro:

- testování výkonu – cílem není nalézt chyby v aplikaci, nýbrž identifikovat a eliminovat úzká hrdla,

- testování zátěže – cílem je ověřit, zda aplikace splňuje požadovaná kritéria zátěže,
- testování odolnosti – cílem je zjistit chování aplikace ve výjimečných stavech (přetížení, nedostatek zdrojů, násilné odebrání zdrojů – např. databáze)
- testování nárazových špiček – vhodné pro ověření chování systému při náhlé, dočasně zvednuté zátěži.

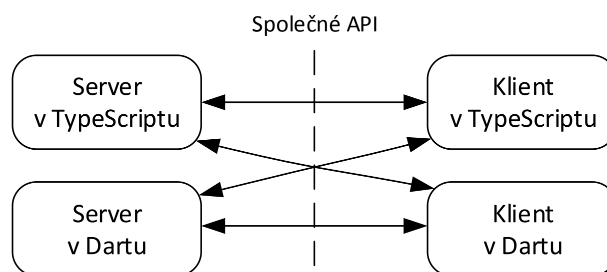
3 Předmět praktické části

Předmětem praktické části je, jak už bylo zmíněno v úvodu, představní možností jazyků Dart a TypeScript při implementaci webové aplikace. Pro ilustraci jsem si zvolil jednoduchou aplikaci pro zapisování úkolů. V aplikaci je možné vytvářet libovolný počet seznamů úkolů a v každém z nich libovolný počet úkolů, které se skládají z textu úkolu a stavu dokončeno / nedokončeno. Aplikace podporuje víceuživatelský přístup. Autentizace je realizována pomocí uživatelského jména a hesla a prostřednictvím sociální sítě Facebook. Jak aplikace vypadá, můžete vidět na obrázku 9.



Obrázek 9 – Náhled aplikace
Zdroj: vlastní zpracování

Aplikaci jsem implementoval v obou jazycích, na které je má diplomová práce zaměřena, v Dartu a TypeScriptu. Obě implementace vypadají shodně a mají shodné chování. Díky společnému API je dokonce možné používat serverovou část napsanou v Dartu a klienta v TypeScriptu či naopak, jak ukazuje obrázek 10.



Obrázek 10 – Společné API
Zdroj: vlastní zpracování

Při psaní návrhu aplikace jsem se soustředil na to, aby vznikla ač jednoduchá, tak použitelná a moderní single-page[1] aplikace.

Aplikace psaná v Dartu je na straně serveru spouštěna v Dart VM na straně klienta jsem v průběhu vývoje používal speciální verzi prohlížeče Chrome zvanou Dartium. Tato verze obsahuje interpret Dartu a není ho tedy potřeba překládat do JavaScriptu. Nicméně překlad do JavaScriptu je funkční a aplikaci lze spustit v libovolném moderním prohlížeči.

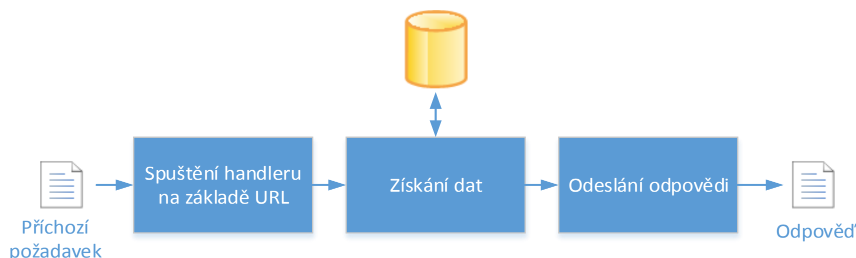
Verze psaná v TypeScriptu je na serveru překládána do JavaScriptu a spouštěna v Node.js. V praktické části budu tedy často popisovat právě Node.js. Na straně klienta je aplikace též překládána do JavaScriptu, který funguje ve všech moderních prohlížečích.

Součástí praktické části budou též kapitoly, jejichž obsah jsem v aplikaci nevyužil, nicméně jejich zmínění je důležitou součástí porovnání těchto jazyků.

[1] Webová aplikace nezpůsobující neustálé obnovování stránky v prohlížeči po vykonání akce – po jejím spuštění probíhají už všechny dotazy asynchronně na pozadí.

4 Použití na serveru

V případě implementace webové aplikace spočívá použití jazyka na serveru především ve spuštění webového serveru, obsloužení jednotlivých URL odpovídajícími handlery[1], získání dat a jejich zaslání zpět klientu (Obrázek 11). Spadá sem například i používání sdílených prostředků, jako jsou soubory, databáze, používání API jiných služeb apod.



Obrázek 11 – Zpracování požadavku na webovém serveru
Zdroj: vlastní zpracování

Nezbytnou součástí vývoje je i psaní různých skriptů pro automatizaci rutinních úkolů. I to zahrnu do této části práce.

4.1 Webový server

Dart i Node.js obsahují v základní sadě knihoven jednoduchý webový server. Jedná se o poměrně nízkoúrovňové API, které umožní poslouchat na vybraném TCP portu a v případě příchozího HTTP požadavku dostaneme objekt reprezentující daný požadavek. Jak na něj zareagujeme, je už na nás. V případě implementace mé aplikace jsem se rozhodl toto nízkoúrovňové API obejít a použil jsem v obou případech balíčky (více o nich v kapitole 4.6 praktické části) ulehčující práci s webovým serverem.

Dart

V případě Dartu se třídy pro práci s HTTP serverem nachází v knihovně *dart:io*. Jak se takový server spustí, je možné vidět na ukázce 19. Po zavolání `HttpServer.bind` začne naslouchání na dané IP a portu. Poté se pomocí metody `listen` pověsí handler na událost příchozího požadavku, na který se následně odpoví „Hello, world“.

[1] Obslužnými rutinami.

```
import 'dart:io';

main() {
  HttpServer.bind('127.0.0.1', 3000).then((server) {
    server.listen((HttpRequest request) {
      request.response.write('Hello, world');
      request.response.close();
    });
  });
}
```

Zdrojový kód 19 – Webový server v Dartu

Princip je to jednoduchý, avšak pokud bychom chtěli různá URL obsluhovat různě, museli bychom příchozí požadavek ručně zpracovat. Proto jsem se při implementaci aplikace rozhodl použít knihovnu, která tuto práci udělá za mě. Po prozkoumání a vyzkoušení různých řešení jsem dospěl k frameworku *start*. Ten je inspirovaný známým Ruby frameworkem *Sinatra*. Ukázku přímo z mé aplikace je možné vidět na zdrojovém kódu č. 20.

Framework využívá takzvaný routing, kdy porovnává příchozí HTTP požadavek se vzory, které nadefinujeme, a následně spustí handler odpovídající danému vzoru. Vzorem je myšlena především HTTP metoda a maska pro URL. Pokud žádný vzor neodpovídá, pokusí se server odeslat statický obsah z definovaného adresáře. Tím zajistí například zaslání správného obrázku po zadání jeho URL a podobně.

```
import 'dart:io';
import 'dart:async';
import 'dart:json' as Json;

import 'package:start/start.dart';

start(public: '../client/web/out', host: '127.0.0.1', port: 3000).then((app) {
  app.get('/todo-lists').listen(authorizedRoute((request) {
    var user = getUserFromRequest(request);
    request.response.json({'user': user.login, 'todoLists': user.todoLists});
  }));

  app.post('/todo-lists').listen(authorizedRoute(withPostParams((request, params) {
    var user = getUserFromRequest(request);
    user.todoLists = Json.parse(params["todoLists"]);
    saveUser(user);
    request.response.json({'success': true});
  })));

  app.post('/login').listen(withPostParams((request, params) { ... }));

  app.get(config['facebook']['url']).listen((request) { ... });

  app.get('/logout').listen((request) { ... });

  print('Listening...');
```

Zdrojový kód 20 – Server ve frameworku start

TypeScript

V Node.js se nachází třídy pro práci s HTTP serverem v balíčku *http*. Spuštění serveru je možné vidět na ukázce zdrojového kódu č. 21. Všimněte si především prvního řádku, kde se načítá definiční soubor TypeScriptu pro celý Node.js. Od tohoto okamžiku jsou všechny třídy v Node.js staticky typované.

Instance webového serveru se vytvoří pomocí metody `createServer`, kterou následně metodou `listen` spustíme na daném portu. Metoda `createServer` přijímá jako parametr funkci, kterou vykoná při každém příchozím požadavku. Ta se volá s parametry představující požadavek a odpověď.

```
/// <reference path="d.ts/DefinitelyTyped/node/node.d.ts" />

import http = require('http');

http.createServer((req, res) => {
   res.end('Hello, world');
}).listen(8080);
```

Zdrojový kód 21 – Webový server v TypeScriptu

Stejně jako v případě Dartu bychom si dále museli doplnit logiku pro obsluhu různých URL a HTTP metod. Proto jsem se také rozhodl využít již existující framework, který má tuto logiku již implementovanou. Vzhledem k mým dřívějším zkušenostem jsem zvolil framework *express*, který je taktéž inspirován Ruby frameworkem *Sinatra*. Jak aplikace napsaná pomocí frameworku *express* vypadá, můžete vidět na ukázce 22.

Framework *express* využívá stejný routing jako *start*. Má však jiný přístup k vytváření webového serveru. *Start* obsahuje vlastní API pro spuštění webového serveru, zatímco *express* vytváří pouze „aplikaci“, která se nakonec předá standardnímu HTTP serveru z knihovny Node.js.

Tato aplikace pak obsahuje navíc podporu takzvaných middlewarů. Což jsou, zjednodušeně řečeno, funkce, které se volají v pořadí, v jakém byly do aplikace přidány. Každý middleware má k dispozici objekt požadavku, odpovědi a také funkci, která zajistí spuštění dalšího middlewaru. Pomocí middlewaru lze tedy libovolně modifikovat požadavku i odpovědi, případně provádět libovolné akce. Například sledování uživatelské aktivity (jak se uživatel po webu pohybuje) je otázkou jednoho middlewaru, který například do databáze zapíše identifikátor uživatele a URL daného požadavku. Zbytek aplikace pak zůstane naprosto neovlivněn.

```
/// <reference path="d.ts/DefinitelyTyped/node/node.d.ts" />
/// <reference path="d.ts/DefinitelyTyped/express/express.d.ts" />

import express = require('express');
import http = require('http');
import path = require('path');

var app = express();
app.set('port', process.env.PORT || 3000);
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.session());
app.use('/client', express.static(path.join(__dirname, '../client')));
app.use(app.router);

app.get('/todo-lists', authorizedRoute((req, res) => {
    getUser(req).then(user => {
        res.json({user: user.login, todoLists: user.todoLists});
    });
}));

app.post('/todo-lists', authorizedRoute((req, res) => {
    getUser(req).then(user => {
        user.todoLists = JSON.parse(req.body.todoLists);
        userRepository.saveUser(user);
        res.json({success: true});
    });
}));

http.createServer(app).listen(app.get('port'), () => {
    console.log('Express server listening on port ' + app.get('port'));
});
```

Zdrojový kód 22 – Webový server ve frameworku express

4.2 Operace se systémovými prostředky

Operacemi se systémovými prostředky je myšleno především získávání informací o operačním systému o počítači jako takovém. Užitečné jsou například informace o typu operačního systému (Windows vs. Linux vs. MacOS a další), případně o počtu procesorů, volné paměti a podobně.

Úlohy, v kterých hodlám jazyky porovnat jsou následující:

- zjištění platformy (operačního systému) – vhodné pro účely skriptování,
- zjištění počtu procesorů – hodí se zjištění vhodného počtu paralelně spuštěných aplikací[1],
- množství volné paměti – užitečné pro měření paměťové náročnosti skriptu.

[1] Jelikož obě technologie pracují v jednom vlákně, je v některých situacích vhodné spustit proces vícekrát a využít tak více jader procesoru.

Dart

Dart poskytuje pro získávání informací o operačním systému především prostřednictvím třídy *Platform* v knihovně *dart:io*.

Typ operačního systému se v Dartu zjistí z vlastnosti *operatingSystem* třídy *Platform*. Tato vlastnost může nabývat hodnot „linux“, „macos“, „windows“ nebo „android“. Další informace o operačním systému je pak nutné „vytáhnout“ z *Platform.environment*, prostřednictvím kterého máme přístup ke všem proměnným prostředí daného operačního systému.

Počet procesorů je v Dartu zjistitelný skrze vlastnost *numberOfProcessors* třídy *Platform*. Třída o daných procesorech neposkytuje další informace.

Pro zjištění množství volné paměti jsem v Dartu nenalezl žádné řešení. Je pravděpodobné, že se tam tato informace objeví v některé z budoucích verzí.

```
import 'dart:io';

main() {
  print(Platform.operatingSystem);
  // "windows"
  print(Platform.numberOfProcessors);
  // 4
}
```

Zdrojový kód 23 – Informace o OS v Dartu

TypeScript

V Node.js se k těmto informacím přistupuje prostřednictvím balíčku *os*, který obsahuje funkce poskytující všechny požadované informace.

Typ operačního systému zjistíme zavoláním funkce *os.platform*. Funkce *platform* vrací hodnoty například „win32“, „linux“, „darwin“ atd. Pokud chceme přesnější informaci o operačním systému, můžeme použít například funkci *os.release*, která vrátí konkrétní verzi OS.

Node.js poskytuje ucelené informace o procesorech prostřednictvím funkce *os.cpus*. Dozvíme se tak nejen počet procesorů, ale i jejich typ, frekvenci a další.

Volnou paměť zjistíme v Node.js zavoláním funkce *os.freemem*, celkové množství paměti pak pomocí *os.totalmem*.

```
/// <reference path="d.ts/DefinitelyTyped/node/node.d.ts" />

import os = require('os');
console.log(os.platform());
// "win32"
console.log(os.cpus().length);
// 4
console.log(os.freemem() + ' / ' + os.totalmem());
// 947232768 / 4174974976
```

Zdrojový kód 24 – Informace o OS v TypeScriptu

4.3 Práce se sítí

Stejně jako potřebujeme spouštět webové servery, může se nám občas hodit čisté TCP spojení. Ani to není v Dartu a TypeScriptu problém. Použití TCP serveru budu demonstrovat na jednoduchém echo serveru s klientem, který se na tento server připojí.

Dart

V Dartu jsou třídy pro práci s TCP spojením součástí knihovny *dart:io*. Jak je vidět na ukázce 25, pomocí třídy *ServerSocket* spustíme server na zvoleném portu a čekáme na příchozí spojení. Po jeho otevření se zavolá funkce, jejímž parametrem je instance třídy *Socket*. Bezprostředně po navázání pak začneme poslouchat, jaká data klient serveru posílá, a po jejich přijetí mu odešleme zprávu zpět.

```
import 'dart:io';

main() {
  ServerSocket.bind('127.0.0.1', 8124).then((ServerSocket server) {
    server.listen((Socket socket) {
      socket.listen((List<int> data) {
        socket.add("\r\nYou wrote: ".runes.toList());
        socket.add(data);
        socket.add("\r\n".runes.toList());
      });
    });
  });
}
```

Zdrojový kód 25 – TCP echo server v Dartu

Jak můžete vidět na ukázce 26, klient vypadá velice podobně. Pomocí třídy *Socket* navážeme spojení, po jehož otevření

1. začneme poslouchat příchozí data, abychom je mohli vypsat na konzoli a
2. spustíme časovač, který v pravidelných interval (3 sekundy) pošle na server data ve formě řetězce.

```
import 'dart:io';
import 'dart:core';
import 'dart:async';

main() {
  Socket.connect('127.0.0.1', 8124).then((Socket socket) {
    socket.listen((List<int> data){
      print(new String.fromCharCodes(data));
    });

    var data = 'some data';
    var timer = new Timer.periodic(new Duration(seconds: 3), (Timer t) {
      print('Sending: ' + data);
      socket.add(data.runes.toList());
    });
    socket.done.whenComplete(timer.cancel);
  });
}
```

Zdrojový kód 26 – TCP klient v Dartu

TypeScript

V případě TypeScriptu se třídy pro manipulaci s TCP spojeními nachází v balíčku *net*. Server se vytvoří podobně jako v Dartu. Vytvoří se instance TCP serveru, který se spustí na zvoleném portu. V případě příchozího spojení se zavolá funkce předaná metodě vytvářející server. V té se navážeme na událost příchozích dat a patřičně na ni zareagujeme.

```
/// <reference path="d.ts/DefinitelyTyped/node/node.d.ts" />

import net = require('net');

net.createServer((socket:net.NodeSocket) => {
    socket.on('data', (data) => {
        socket.write('\r\nYou wrote: ');
        socket.write(data);
        socket.write('\r\n');
    });
}).listen(8124);
```

Zdrojový kód 27 – TCP echo server v TypeScriptu

Klient vypadá též dost podobně jako v Dartu. Připojíme se na server a po připojení opět spustíme časovač, který v pravidelných intervalech zašle data na server. V případě příchozích dat tato vypíšeme na konzoli.

```
/// <reference path="d.ts/DefinitelyTyped/node/node.d.ts" />

import net = require('net');

var client = net.connect({port: 8124}, () => {
    setInterval(sendData, 3000);
});

client.on('data', (data)=> {
    console.log(data.toString());
});

var sendData = () => {
    var data = 'some data';
    console.log('Sending:', data);
    client.write(data);
};
```

Zdrojový kód 28 – TCP klient v TypeScriptu

4.4 Používání sdílených prostředků

Za sdílený prostředek můžeme označit jakoukoli službu či hardware dostupný skrze počítačovou síť [22]. Co se týče sdílených prostředků, použil jsem ve své diplomové práci dokumentovou databázi MongoDB pro ukládání dat a Facebook API pro autentifikaci.

Dart

Pro připojení k MongoDB z Dartu jsem využil knihovnu *mongo_dart*. Ta poskytuje kompletní API pro práci s databází. Veškeré kritické akce jsou v knihovně psány asynchronně pomocí *Futures*. Práce s databází samotnou celkem příjemná. Jak můžete vidět na ukázce 29, stačí v podstatě vytvořit objekt představující připojení k databázi, toto spojení otevřít a vyžádat si kolekci s požadovaným názvem.

```
Future<DbCollection> _getUserCollection() {
  var completer = new Completer();
  Db db = new Db('mongodb://127.0.0.1/thesis-todo');
  var collection = db.collection('users');
  db.open()
    .then((_) {
      completer.complete(collection);
    })
    .catchError((error) => completer.completeError(error));
  return completer.future;
}
```

Zdrojový kód 29 – Připojení k MongoDB v Dartu

Práce s takto získanou kolekcí je poté celkem elegantní, jak můžete vidět na ukázce 30. Celé API je, jak jsem zmiňoval, tvořeno pomocí *Futures*, takže je výsledný kód dobře čitelný. Jedinou „vadou na kráse“ je nutnost implementovat vlastní mapování z objektů uložených v databázi na doménové objekty aplikace a nazpátek. API totiž vrací obyčejné *HashMapy*.

```
Future<User> findUser(String login) {
  return _getUserCollection()
    .then((collection) => collection.findOne(where.eq('login', login)))
    .then(_transformToUser);
}

createUser(User user) {
  _getUserCollection().then((userCollection) {
    userCollection.insert(Json.parse(Json.stringify(user))).then((plainUser) {
      _userMap[user] = plainUser['_id'];
    });
  });
}

saveUser(User user) {
  var plainUser = Json.parse(Json.stringify(user));
  plainUser['_id'] = _userMap[user];
  _getUserCollection().then((userCollection){
    userCollection.save(plainUser);
  });
}
```

Zdrojový kód 30 – Práce s MongoDB v Dartu

Dalším sdíleným prostředkem v mé aplikaci je zmiňované Facebook API pro autentifikaci. Pro Dart bohužel v současné době neexistuje žádná knihovna, která by práci s FB API nějakým způsobem zjednodušovala. Musel jsem tedy sáhnout k manuálnímu provádění jednotlivých HTTP dotazů na Facebook. Každý asynchronní kód v Dartu typicky používá *Futures*, takže i zde je kód celkem dobře čitelný (Zdrojový kód 31).

```
app.get(config['facebook']['url']).listen((Request request) {
  var code = Uri.encodeComponent(request.param('code'));

  http.read(buildFacebookAccessTokenUrl(code)).then((contents) {
    var params = QueryString.parse('?$contents');
    var accessToken = params['access_token'];

    http.read(buildFacebookAccessTokenUrl(accessToken)).then((contents) {
      var userInfo = Json.parse(contents);
      getOrCreateUser(userInfo['username'], '').then((user) {
        loggedUsers.login(request.input.session.id, user);
        request.response.redirect('/index.html');
      });
    });
  });
});
```

Zdrojový kód 31 – Autentifikace pomocí Facebook API v Dartu

TypeScript

Při implementaci perzistentní vrstvy v TypeScriptu jsem použil balíček nazvaný *mongodb*. Ačkoli se jedná o nejpoužívanější balíček pro práci s MongoDB z Node.js, zklamal mě svým API. Nepoužívá totiž asynchronní zpracování založené na vzoru *Promise*, nýbrž obyčejné callbacky, které jsou volány vždy se dvěma parametry – chybovým objektem a výsledkem volání. V každém takovém callbacku pak musí následovat větvení kódu na základě toho,

jestli asynchronní akce skončila úspěšně či neúspěšně. Toto chování jsem se na úrovni aplikace snažil odstínit vlastním API, které návrhový vzor *Promise* používá. K jeho implementaci jsem využil knihovnu *q*.

```
function getUserCollection() {
  var deferred = q.defer<mongo.Collection>();
  var server = new Server('127.0.0.1', 27017);
  new Db('thesis-todo', server, {native_parser: true, safe: false})
    .open((err, db) => {
      if (err) deferred.reject(err);
      else db.collection('users', (err, collection) => {
          if (err) deferred.reject(err);
          else deferred.resolve(collection);
      });
    });
  return deferred.promise;
}
```

Zdrojový kód 32 – Připojení k MongoDB v TypeScriptu

Callbacky je „prošpikované" opravdu celé API, takže výsledný kód pak nevypadá tak elegantně jako v případě Dartu.

```
export function findUser(login: String) {
  var deferred = q.defer<User>();
  getUserCollection().then(userCollection => {
    userCollection.findOne({login: login}, (err, user) => {
      if (err) deferred.reject(err);
      else deferred.resolve(user);
    });
  }, err => deferred.reject(err));
  return deferred.promise;
}

export function createUser(login: String, password: String) {
  var deferred = q.defer<User>();
  getUserCollection().then(userCollection => {
    userCollection.insert(new User(login, password), (err, result) => {
      if(result.length > 0)
        deferred.resolve(result[0]);
      else
        deferred.reject(err);
    });
  });
  return deferred.promise;
}

export function saveUser(user: User) {
  getUserCollection()
    .then(collection => collection.save(user, () => {}));
}
```

Zdrojový kód 33 – Práce s MongoDB v TypeScriptu

Oproti tomu autentifikace pomocí Facebooku je v TypeScriptu poměrně jednoduchá. Díky knihovně *everyauth*, která podporuje přihlašování prostřednictvím celkem 36 různých služeb a protokolů, je přidání nového způsobu autentifikace otázkou chvilky. Jak můžete vidět na

ukázce 34, je to vše (až na logiku spojení Facebook uživatele s uživatelem v mé aplikaci) jen otázkou konfigurace.

```
everyauth.facebook
  .appId('585883654788924')
  .appSecret('373915d4bb842fdc2388ac54c23912ff')
  .entryPath('/login/facebook')
  .findOrCreateUser(function(session, accessToken, _, fbUserMeta) => {
    var promise = this.Promise();
    userRepository.findOrCreateUser(fbUserMeta.username, '').then(user => {
        promise.fulfill(user);
    });
    return promise;
  })
  .redirectPath('/');
```

Zdrojový kód 34 – Autentifikace pomocí Facebook API v TypeScriptu

4.5 Možnosti automatizace

Automatizace rutinních úloh je nedílnou součástí vývoje aplikací. V případě webových aplikací se jedná o různé úpravy zdrojových kódů – spojování více souborů dohromady, minifikace[1], spouštění transpilerů a podobně.

Dart

Pro Dart bohužel neexistuje žádný nástroj, který by spouštění podobných úloh umožňoval. Pokud například programátor potřebuje spustit skript, který bude sledovat jednotlivé soubory projektu a po jejich změně například kompilovat Dart do JavaScriptu, má poměrně svázané ruce. Dart zatím neobsahuje například knihovnu pro sledování změn souboru (už je ale vznesen požadavek[2] na jeho implementaci). U Dartu se v tomto ohledu tedy poměrně projevuje jeho nedospělost.

TypeScript

Oproti tomu pro Node.js existuje například nástroj *Grunt* sloužící právě ke spouštění různých automatizovaných úloh. Jen pro tento nástroj existuje přes 1200 pluginů instalovatelných skrze *npm*. Většina standardních úloh, které potřebujeme automatizovat, pak spočívá už jen v konfiguraci *Gruntu*. Jak taková konfigurace vypadá, můžete vidět na ukázce 35.

Uvedený skript je trochu zjednodušený, avšak dobře demonstruje sílu automatizačního nástroje. Pochází z aplikace, která je zároveň jak webovou aplikací dostupnou skrz prohlížeč, tak mobilní aplikací pro Android. Uvedená konfigurace dokáže po spuštění překopírovat statický obsah (obrázky, styly apod.), zkompilovat aplikaci z TypeScriptu do JavaScriptu, nahrát hotovou aplikaci pomocí FTP na vzdálený server a nakonec prostřednictvím služby PhoneGap Build vytvořit instalační balíček Android aplikace.

[1] Zmenšení velikosti souboru například pomocí odstranění mezer pro minimalizaci přeneseného objemu dat po síti.
[2] https://code.google.com/p/dart/issues/detail?id=3310

```
module.exports = function (grunt) {

    var buildDir = 'build/',
        htmlBuild = buildDir + 'html/';

    grunt.initConfig({
        copy: {
            htmlBuild: { files: [ { src: '**', dest: htmlBuild } ] }
        },
        typescript: {
            livechess: { src: ['js/app.ts'], dest: htmlBuild + 'js/livechess.js' }
        },
        compress: {
            android: { files: [ { cwd: htmlBuild, src: '**', expand: true } ] }
        },
        phonegap_build: {
            options: { ... }
        },
        ftp_deploy: {
            auth: { ... }, src: htmlBuild, dest: 'site/wwwroot'
        }
    });

    grunt.loadNpmTasks('grunt-contrib-copy'); // pro každý plugin

    grunt.registerTask('default', ['copy', 'typescript']);
    grunt.registerTask('deploy', ['default', 'ftp_deploy', 'compress',
'phonegap_build']);

};
```

Zdrojový kód 35 – Automatizační skript v Gruntu

4.6 Používání externích knihoven

V posledních letech je trendem, aby součástí ekosystému okolo jazyka byl i centrální repozitář různých frameworků a knihoven. Pro Javu existuje například Maven Central Repository[1], pro .NET balíčkovací systém Nuget[2], pro PHP repozitář Packagist[3] s balíčkovacím systémem Composer[4] atd. Nejinak je tomu i v případě Dartu a Node.js.

Dart

Balíčkovací systém s repozitářem pro Dart se jmenuje *pub*. V současné době obsahuje 426 balíčků. Balíčky instalované k projektu jsou evidované v souboru *pubspec.yaml*. Instalace nového balíčku se provádí příkazem *pub install <název balíčku>*.

V projektu jsou tyto balíčky dostupné pomocí klauzule *import*. Ta se nachází vždy v hlavním souboru knihovny ve tvaru *import „package:<cesta k hlavnímu souboru balíčku>“*. Pokud by se neuvedl prefix *package*, pokusil by se Dart hledat v souborech relativně k aktuálnímu a případně i ve standardních knihovnách.

[1] http://search.maven.org/
[2] http://www.nuget.org/
[3] http://packagist.org/
[4] http://getcomposer.org/

TypeScript

Pro Node.js existuje balíčkovací systém *npm* s repozitářem obsahujícím neuvěřitelných cca 38 tisíc balíčků. Při tomto množství už se dá říci, že existuje balíček téměř na vše. Bohužel je mnoho z nich ve verzi 0.0.1 a podobně, takže se na ně nedá moc spolehnout a programátor si musí pečlivě vybírat, který balíček na daný problém použije.

Balíčky instalované v projektu se evidují v souboru *package.json*. Nový balíček se do projektu instaluje pomocí příkazu *npm install <název balíčku> --save*. Pokud by se neuvedl přepínač *save*, balíček se nainstaluje, avšak nebude uložen v *package.json*. To se hodí například pro experimentování s různými balíčky.

4.7 Testování

V této kapitole se budu zabývat především jednotkovým testováním, se kterým mám z oblasti testování aplikací největší zkušenosti. Z hlediska teorie bylo jednotkové testování vysvětleno v kapitole 2.1.1 praktické části.

Dart

Nejpoužívanějším testovacím frameworkem pro Dart je zřejmě *unittest*. Používají ho snad všechny knihovny, na které jsem při studiu Dartu a práci na diplomové práce narazil. Jedná se o testovací framework vytvořený týmem programátorů stojícím za vývojem Dartu samotného.

Jednotlivé testy se zde vyjadřují pomocí top-level funkcí. Tyto funkce se navíc dají shlukovat do skupin. Jak takové testy vypadají, můžete vidět na ukázce 36. V první části kódu je test bez skupiny, následují testy rozčleněné do skupin. Tyto skupiny lze dále zanořovat do sebe a vytvářet tak sémantický strom jednotlivých testů.

```
import 'package:unittest/unittest.dart';

main() {
  test('test 1', () {
    int x = 2 + 3;
    expect(x, equals(5));
  });
  group('group A', () {
    test('test A.1', () {
      int x = 2 + 3;
      expect(x, equals(5));
    });
    test('test A.2', () {
      int x = 2 + 3;
      expect(x, equals(5));
    });
  });
  group('group B', () {
    test('this B.1', () {
      int x = 2 + 3;
      expect(x, equals(5));
    });
  });
}
```

Zdrojový kód 36 – Testování v Dartu

TypeScript

Přímo pro TypeScript jsem našel pouze jeden testovací framework – tsUnit. Svým přístupem připomíná například *JUnit* nebo *NUnit* (Zdrojový kód 37). Nenašel jsem však žádný projekt, který by ho používal (čemuž odpovídá i jeho celkových 66 stažení z codeplex.com). Doporučuji také porovnat syntax *tsUnit* a ostatních představovaných frameworků. Je vidět, jak jsou testy založené na třídách a metodách „těžkopádné" a nevýřečné.

```
class CalculatorTest extends tsUnit.TestClass {

  addWith1And2Expect3() {
    var calculator = new Calculator();
    var result = 1 + 2;
    this.areIdentical(3, result);
  }
}

var test = new tsUnit.Test();
test.addTestClass(new CalculationsTests.SimpleMathTests());
test.showResults(document.getElementById('results'), test.run());
```

Zdrojový kód 37 – Testování v tsUnit

Mnohem častěji se ve spojení s TypeScriptem používají běžné JavaScriptové testovací frameworky s připojenými definičními soubory TypeScriptu. Mezi nejpoužívanější a nejznámější patří frameworky *Jasmine* a *Mocha*. Jedná se o frameworky zaměřené na Behavior-driven development , neboli BDD. Tento přístup spočívá v tom, že se pomocí testů popisuje, jak se má aplikace chovat, místo slepého testování vstupů a výstupů. Také je specifický tím, že se jednotlivé testy dají často číst jako věty.

Jasmine i *Mocha* mají v podstatě stejné rozhraní pro zápis testů jako takových. Liší se pouze v názvech assertovacích metod (viz ukázky 38 a 39). Nicméně *Mocha* přímo na svém webu nabízí hned několik alternativ těchto metod. Mně sympatická je knihovnou je *chai*, kterou taktéž demonstruji v ukázce testů ve frameworku *Mocha*.

```
describe('Calculator', () => {
  var calculator = new Calculator();

  describe('#add' () => {
    it('should add two numbers', () => {
      expect(calculator.add(1, 2)).toBe(3);
    })
  });

});
```

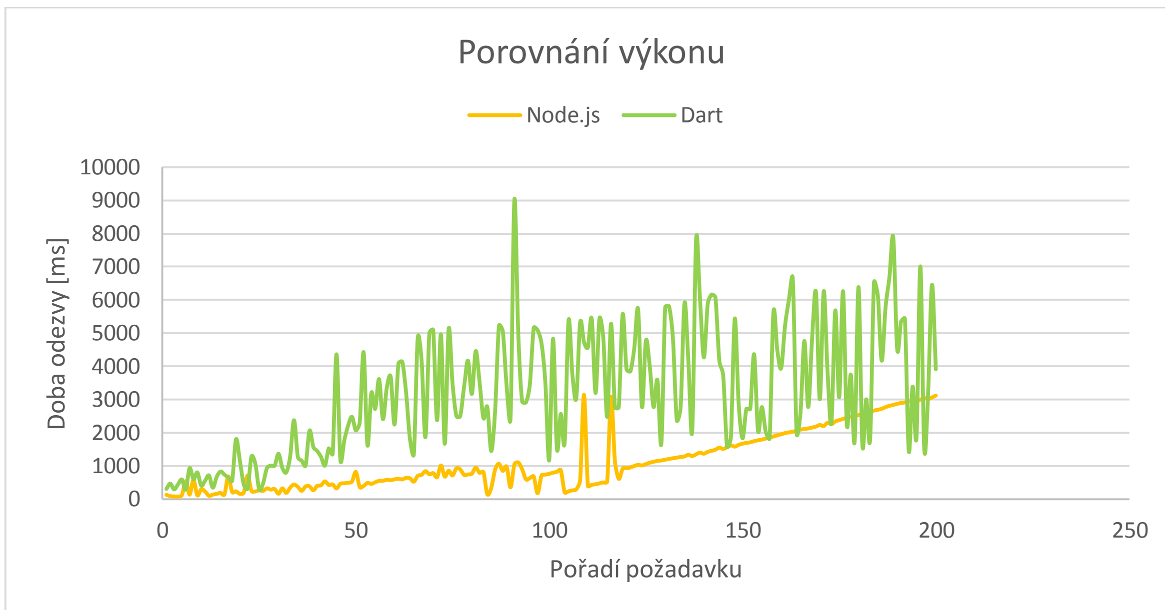
Zdrojový kód 38 – Testování v Jasmine

```
describe('Calculator', () => {
  var calculator = new Calculator();

  // Výchozí assertovací metody
  describe('#add' () => {
    it('should add two numbers', () => {
      assert.equal(calculator.add(1, 2), 3);
    })
  });

  // Assertovací metody z knihovny chai
  describe('#add' () => {
    it('should add two numbers', () => {
      calculator.add(1, 2).should.be(3);
    })
  });


});
```

Zdrojový kód 39 – Testování v Mocha

4.8 Test výkonu

Pro porovnání výkonu jsem zvolil stejný test, jaký byl uveden v kapitole 1.4.3. Jako výsledky pro TypeScript je možné použít původní výsledky JavaScriptu. V Dartu jsem naprogramoval webový server odpovídající tomu v JavaScriptu. Pro vícevláknové zpracování jsem v Dartu využil Isolates.



Obrázek 12 – Test výkonu Dart a Node.js (TypeScript)
Zdroj: vlastní zpracování

Jak je z grafu patrno, je na tom Dart v tomto směru ještě poměrně špatně a potřebuje v průměru zhruba trojnásobný čas na odbavení HTTP požadavku. Nutno však podotknout, že Node.js je v tomto směru velice dobře optimalizovaný a jeho vývojáři se při optimalizaci zaměřují právě na rychlost odbavování paralelních požadavků.

5 Použití jako klientský skriptovací jazyk

Jak jsem zmiňoval v kapitolách o programovacích jazycích (kapitoly 1.4, 1.6 a 1.7 teoretické části), jediným jazykem, který je v dnešní době spustitelný na straně klienta – v prohlížeči – je JavaScript. Proto každý jazyk, který usiluje o to, aby se v něm psali i klientské aplikace, musí být do JavaScriptu zkompilovatelný. To Dart i TypeScript splňují. Pro Dart existuje navíc speciální verze prohlížeče Chrome zvaná Dartium (zmíněná i v kapitole 3 praktické části), která je schopna spouštět Dart přímo. Jedná se však pouze o nástroj pro vývojáře.

Každý z těchto jazyků musí být schopen pracovat s *Document Object Modelem* – DOMem. Především musí mít nástroje pro manipulaci s jeho uzly a být schopen navázat se na jeho události. Co se týče TypeScriptu, můžeme se spolehnout na funkčnost, kterou přebírá z JavaScriptu (jelikož je jeho nadmnožinou). U Dartu je situace o poznání zajímavější – zde si architekti jazyka mohli vymyslet vlastní abstrakci, teoreticky lepší než v JavaScriptu.

Dalším požadavkem kladeným na jazyk spouštěný v prohlížeči je schopnost komunikovat se serverem prostřednictvím asynchronních požadavků a v neposlední řadě také podpora externích knihoven.

5.1 Manipulace s DOMem

Pod manipulací s DOMem se rozumí především následujících několik úloh:

- přidání, odebrání a přesunutí uzlu,
- přidání, odebrání a změna atributů,
- ovlivňování CSS vlastností.

Dart

V Dartu potřebujeme pro manipulaci s DOMem importovat knihovnu *dart:html*, která obsahuje všechny potřebné objekty. Jednotlivé elementy[1] jsou reprezentovány pomocí tříd. Jejich vytvoření pak spočívá ve vytvoření nové instance dané třídy, jak je vidět na ukázce 40.

```
var loginForm = new FormElement();
var loginInput = new TextInputElement()..name = 'login';
var passwordInput = new PasswordInputElement()..name = 'password';
var submitButton = new SubmitButtonInputElement()..value = 'Přihlásit';
```

Zdrojový kód 40 – Vytváření DOM elementů v Dartu

Pozn.: *Všimnout si zde můžete i nového operátoru, který Dart přináší. Jsou jím dvě tečky. Operátor má stejný význam jako standardní tečkový operátor (zpřístupňuje členské proměnné a metody), avšak po provedení akce na pravé straně operátoru – v případě ukázky přiřazení hodnoty do členské proměnné – se vrátí hodnota na levé straně operátoru –*

[1] Uzly v DOMu.

v ukázce výsledek konstruktoru, tj. onen vytvořený objekt. Klasický tečkový operátor vrací výsledek operace umístěné napravo od něj.

Nyní můžeme vytvořeným elementům nastavit atributy. Ty jsou přístupné prostřednictvím hashmapy, se kterou můžeme libovolně manipulovat, jak je znázorněno na ukázce 41.

```
loginInput.attributes['placeholder'] = 'Uživatelské jméno';
passwordInput.attributes['placeholder'] = 'Heslo';
passwordInput.attributes.remove('placeholder');
passwordInput.dataset['foo'] = 'bar';
```

Zdrojový kód 41 – Modifikace atributů v Dartu

Můžeme si zde také všimnout práce s HTML5 data atributy pomocí samostatné hashmapy *dataset* (tento zápis se převede na data atribut ve formátu `data-foo="bar"`).

Takto vytvořené elementy obohacené o atributy můžeme připojit k dokumentu. K dispozici máme metodu *append*, případně seznam potomků daného elementu, se kterým můžeme libovolně manipulovat (Zdrojový kód 42). Místo, na které výsledný element připojíme, můžeme najít například pomocí metody *query* akceptující libovolný CSS selektor.

```
loginForm
  ..append(loginInput)
  ..append(passwordInput)
  ..append(submitButton);

var root = query('#root');
root.append(loginForm);

loginForm.children.insert(0, submitButton);
loginForm.children.remove(loginInput);
```

Zdrojový kód 42 – Manipulace s umístěním elementů v dokumentu v Dartu

Poslední úlohou je ovlivňování CSS vlastností, které se v Dartu provádí stejně jako v JavaScriptu – pomocí atributu style a jeho vlastností (Zdrojový kód 43).

```
submitButton.style.color = 'blue';
submitButton.style.border = '1px solid blue';
```

Zdrojový kód 43 – Ovlivňování CSS vlastností v Dartu

TypeScript

TypeScript jako takový nepřináší v tomto ohledu nic nového. Můžeme v něm použít základní JavaScriptové API nebo využít knihoven, které manipulaci s DOMem ulehčují. Neznámější knihovnou je bezesporu jQuery, ke které sám Microsoft dodává definiční soubor TypeScriptu. V případě TypeScriptu budu v ukázkách tedy uvádět dvě různá řešení – jedno čistě JavaScriptové a druhé za pomoci jQuery.

Pro vytvoření DOM elementu slouží metoda *createElement* globálního objektu *document*. Ta jako parametr přebírá název HTML tagu daného elementu. V jQuery stačí pro vytvoření

elementu zavolat jQuery funkci[1], které se jako parametr předá HTML tag elementu (včetně špičatých závorek). Takto je možné vytvářet i komplexnější strukturu, kdy se jQuery předá útržek HTML kódu a ono jej samo převede na odpovídající objektovou strukturu. Konkrétní funkčnost je ilustrována ukázkami 44 a 45.

```
var loginForm = document.createElement('form');
var loginInput = document.createElement('input');
var passwordInput = document.createElement('input');
var submitButton = document.createElement('input');
```

Zdrojový kód 44 – Vytvoření elementů v TypeScriptu

```
var loginForm = $('<form>');
var loginInput = $('<input name="login">');
var passwordInput = $('<input name="password">');
var submitButton = $('<input type="submit">');
```

Zdrojový kód 45 – Vytvoření elementů v TypeScriptu za pomoci jQuery

Jelikož v čistě JavaScriptovém voláním `document.createElement` vytváříme elementy bez jakýchkoliv atributů, je jejich nastavování o něco zdlouhavější (Zdrojový kód 46). Oproti tomu při použití jQuery vytvoříme už „předpřipravené" elementy a definování atributů je pak kratší (Zdrojový kód 47). Všechny předem známé atributy bychom mohli přesunout už do HTML tagu použitého při konstrukci elementu a výsledný kód by se ještě zkrátil. Nicméně pro účely demonstrace je definuji takto dodatečně.

```
loginInput.name = 'login';
loginInput.setAttribute('placeholder', 'Uživatelské jméno');
passwordInput.type = 'password';
passwordInput.name = 'password';
passwordInput.setAttribute('placeholder', 'Heslo');
passwordInput.removeAttribute('placeholder');
submitButton.type = 'submit';
submitButton.value = 'Přihlásit';
passwordInput.dataset.foo = 'bar';
```

Zdrojový kód 46 – Modifikace atributů v TypeScriptu

```
loginInput.attr('placeholder', 'Uživatelské jméno');
passwordInput.attr('placeholder', 'Heslo');
passwordInput.removeAttr('placeholder');
submitButton.val('Přihlásit');
passwordInput.data('foo', 'bar');
```

Zdrojový kód 47 – Modifikace atributů v TypeScriptu pomocí jQuery

Připojení jednotlivých elementů je velice podobné řešení v Dartu. Pouze zde nemáme operátor dvou teček a jednoduše modifikovatelnou kolekci potomků elementu. Element, do kterého se vytvořené objekty připojí, zde hledám pomocí volání `document.querySelector`, které bohužel není dostupné ve všech majoritních verzích prohlížečů (v Internet Exploreru

[1] Globální funkce *jQuery*, případně zkratka reprezentovaná znakem *$*.

je podporováno až od verze 8). Pokud bychom chtěli zajistit kompatibilitu se staršími prohlížeči, museli bychom využít jiných, ne tak elegantních, metod.

```
loginForm.appendChild(loginInput);
loginForm.appendChild(passwordInput);
loginForm.appendChild(submitButton);

var root = document.querySelector('#root');
root.appendChild(loginForm);

loginForm.insertBefore(submitButton, loginInput);
loginInput.remove();
```

Zdrojový kód 48 – Manipulace s umístěním elementů v dokumentu v TypeScriptu

U jQuery (Zdrojový kód 49) opět vidíme poměrně krátký zdrojový kód se stejným výsledkem jako u kódu předchozího. Připojování elementů má trochu jiné API, takže můžeme zřetězeně volat připojení jednotlivých částí formuláře. Cílový element se, stejně jako u Dartu, získá použitím CSS selektoru.

```
loginForm
  .append(loginInput)
  .append(passwordInput)
  .append(submitButton);

var root = $('#root');
root.append(loginForm);

loginForm.prepend(submitButton);
loginInput.remove();
```

Zdrojový kód 49 – Manipulace s umístěním elementů v dokumentu v TypeScriptu s využitím jQuery

Co se týče nastavování CSS vlastností, vypadá čistá JavaScriptová verze naprosto stejně jako řešení v Dartu. Zdrojový kód využívající jQuery pak nabízí zajímavou alternativu – nastavení CSS vlastností pomocí objektu, což se hodí například pro znovupoužití jednotlivých stylů.

```
submitButton.style.color = 'blue';
submitButton.style.border = '1px solid blue';
```

Zdrojový kód 50 – Ovlivňování CSS vlastností v TypeScriptu

```
submitButton.css({color: 'blue', border: '1px solid blue'});
```

Zdrojový kód 51 – Ovlivňování CSS vlastností v TypeScriptu pomocí jQuery

5.2 Práce s DOM událostmi

DOM události jsou neodmyslitelnou částí vývoje klientské části webové aplikace. Práce s nimi by tedy měla být jednoduchá a intuitivní. Hlavními potřebnými úkony jsou:

- navázání posluchače na událost,

- zrušení posluchače,
- ruční vyvolání události,
- zastavení propagace události,
- potlačení výchozí akce.

Dart

Dart poskytuje pro navazování posluchačů na události celkem pěkné API. Všechny standardní události jsou dostupné skrze *on** metody, kde * odpovídá názvu události. Nalezneme zde tedy metody jako *onClick*, *onMouseOver*, *onKeyDown* apod. Konkrétní navázání se na událost najetí myši můžete vidět na následující ukázce.

```
var link = query('a#link');
var mouseOverListener = link.onMouseOver.listen((event) => print('hover'));
```

Zdrojový kód 52 – Nastavení posluchače v Dartu

Na všechny události, ať už ty standardní, z různých knihoven či vlastní, se lze navázat pomocí hashmapy *on* (Zdrojový kód 53). Zároveň je na této ukázce znázorněno vyvolání vlastní události, které spočívá ve vytvoření instance třídy *Event* a její odeslání danému elementu.

```
link.on['custom-event-type'].listen((event) => print(event.type));

Event e = new Event('custom-event-type');
link.dispatchEvent(e);
```

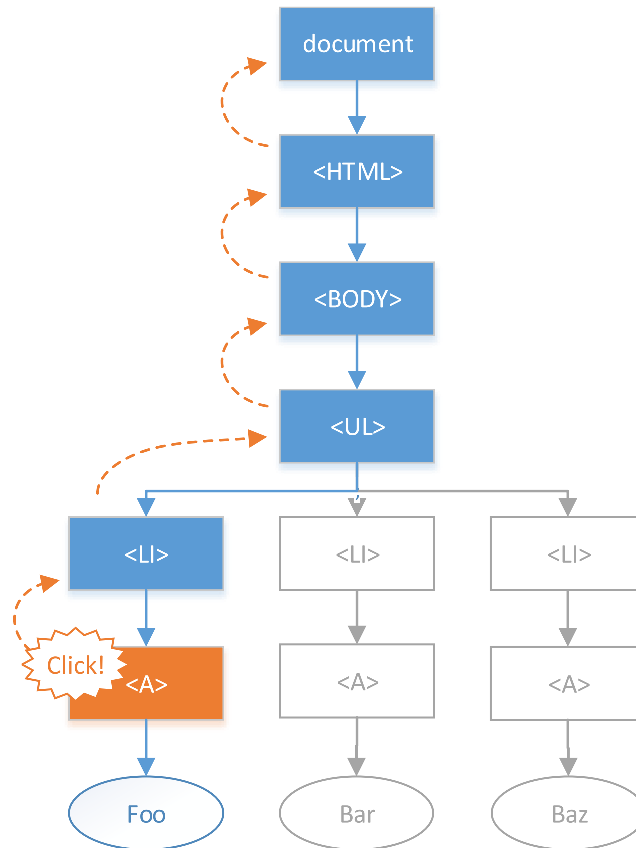
Zdrojový kód 53 – Vlastní typ události v Dartu

Zajímavou funkčností, kterou Dart obsahuje navíc oproti JavaScriptu, je možnost ovládat vytvořené posluchače – konkrétně je pozastavit (zůstanou navěšeni na událost, ale jsou neaktivní), obnovit (reaktivace po jejich pozastavení) nebo je úplně zastavit (odstranit). Na ukázce zdrojového kódu č. 54 jsou pak vidět konkrétní použité metody.

```
query('#pause').onClick.listen((event) => mouseOverListener.pause());
query('#resume').onClick.listen((event) => mouseOverListener.resume());
query('#stop').onClick.listen((event) => mouseOverListener.cancel());
```

Zdrojový kód 54 – Dočasné pozastavení a zrušení posluchače v Dartu

Ve výchozím nastavení všechny události „bublají“ DOMem od elementu, na kterém se událost vyvolá až po kořen (objekt *document*), jak je ilustrováno na obrázku 13.



Obrázek 13 – Propagace události v DOMu
Zdroj: vlastní zpracování

Událost kliknutí, vyvolaná na oranžově zvýrazněném elementu *a*, postupuje po předcích až ke kořenu dokumentu. Takové chování může být někdy nevítané. Jak se taková propagace události zastavení, je znázorněno na ukázce zdrojového kódu níže.

```
query('#root').onClick.listen((event) => print('#root click'));
query('#pause').onClick.listen((event) => event.stopPropagation());
```

Zdrojový kód 55 – Zastavení propagace události v Dartu

Dalším občas nevítaným chováním jsou výchozí akce prohlížečů navázané na danou událost. Jedná se například o navigaci na jinou stránku v případě kliknutí na odkaz, odeslání HTTP požadavku po odeslání formuláře a podobně. Pokud chceme například načíst obsah stránky pomocí AJAXu, musíme po kliknutí na odkaz zrušit výchozí chování prohlížeče. Způsob, jakým toho dosáhnout můžete opět vidět na ukázce pod odstavcem.

```
link.onClick.listen((event) => event.preventDefault());
```

Zdrojový kód 56 – Zrušení výchozí reakce na událost v Dartu

TypeScript

Stejně jako u manipulace s DOMem, i u práce s událostmi můžeme využít základních funkcí JavaScriptu. Jejich syntaxe není však moc elegantní a zápis je zbytečně složitý. Proto jsem se opět rozhodl zmínit i ukázky v jQuery, které má API daleko příjemnější.

Navázání na událost probíhá s využitím JavaScriptového API pomocí metody *addEventListener*, které se jako první parametr předá název události a jako druhý parametr handler.

```
var link = document.querySelector('a#link');
link.addEventListener('mouseover', () => console.log('hover'));
```

Zdrojový kód 57 – Nastavení posluchače v TypeScriptu

Verze v jQuery je o něco úspornější. Pro základní DOM události má definované metody se stejným názvem jako událost. Můžeme tak používat metody *click*, *mouseover*, *keydown* a další. Tyto metody jsou zkratkami k metodě *on* popsané níže.

```
var link = $('a#link');
link.mouseover(() => console.log('hover'));
```

Zdrojový kód 58 – Nastavení posluchače v TypeScriptu pomocí jQuery

Navázání na nestandardní typ události vypadá při použití JavaScript API shodně jako u standardních. Vyvolání takové události pak spočívá ve vytvoření instance třídy *CustomEvent* a její odeslání zvolenému elementu.

```
link.addEventListener('custom-event-type', event => console.log(event.type));

var e = new CustomEvent('custom-event-type');
link.dispatchEvent(e);
```

Zdrojový kód 59 – Vlastní typ události v TypeScriptu

Stejně jako v předchozích příkladech, i nyní jQuery poskytuje API, díky kterému je výsledný kód kratší a přehlednější. Na libovolnou událost je možné se navázat pomocí metody *on*, jak je vidět na ukázce pod odstavcem. Vyvolání události se pak děje skrze metodu *trigger*.

```
link.on('custom-event-type', event => console.log(event.type));

var e = $.Event('custom-event-type');
link.trigger(e);
```

Zdrojový kód 60 – Vlastní typ události v TypeScriptu s využitím jQuery

U Dartu jsem zmiňoval, že umožňuje ovládat jednotlivé posluchače – konkrétně je dočasně či trvale deaktivovat a obnovit po dočasné deaktivaci. JavaScript API ani jQuery nic takového bohužel neposkytují a tato funkčnost se musí obcházet odebíráním a novým

přidáváním posluchače. To je však poměrně nepohodlné a čistém JavaScriptu i značně nepřehledné (Zdrojový kód 61).

```
var listener = () => console.log('hover');
link.addEventListener('mouseover', listener);
document.querySelector('#pause').addEventListener('click', () =>
  link.removeEventListener('mouseover', listener));
document.querySelector('#resume').addEventListener('click', () => {
  link.removeEventListener('mouseover', listener);
  link.addEventListener('mouseover', listener);
});
document.querySelector('#stop').addEventListener('click', () =>
  link.removeEventListener('mouseover', listener));
```

Zdrojový kód 61 – Dočasné zakázání a zrušení posluchače v TypeScriptu

Při použití jQuery sice dosáhneme kratšího zápisu, nicméně operace jsou identické (Zdrojový kód 62).

```
var listener = () => console.log('hover');
link.mouseover(listener);

$('#pause').on('click', () => link.off('mouseover', listener));
$('#resume').on('click', () => link.off('mouseover', listener).mouseover(listener));
$('#stop').on('click', () => link.off('mouseover', listener));
```

Zdrojový kód 62 – Dočasné zakázání a zrušení posluchače v TypeScriptu s jQuery

Alternativou by bylo použití „chytrých“ posluchačů, které tuto funkčnost implementují. Příklad takové implementace můžete vidět na ukázce 63. Jedná se využití first-class funkcí, kdy funkce *pausableListener* vrací jinou funkci. Tato funkce (*smartListener*) po svém zavolání rozhodne na základě atributu *active*, zda se vykoná původní posluchač. Tento atribut je možné ovlivnit pomocí metod *pause* a *resume*.

```
var pausableListener = function pausableListener(listener) {
  var smartListener = () => smartListener.active ? smartListener.cb() : undefined;

  smartListener.active = true;
  smartListener.cb = listener;
  smartListener.pause = () => smartListener.active = false;
  smartListener.resume = () => smartListener.active = true;
  return smartListener;
};

var listener = pausableListener(() => console.log('hover'));
link.mouseover(listener);
$('#pause').on('click', () => listener.pause());
$('#resume').on('click', () => listener.resume());
$('#stop').on('click', () => link.off(listener));
```

Zdrojový kód 63 – Pozastavitelný posluchač v TypeScriptu

Nyní ale zpět k DOM událostem. Dalším zmiňovaným požadavkem je zastavení propagace události dokumentovým stromem. Toho se v TypeScriptu dosáhne identicky jako v Dartu, tj. zavoláním metody *stopPropagation* na objektu reprezentujícím událost. Jak můžete vidět

na ukázkách 64 a 65, zápis pomocí čistého JavaScript API a s využitím jQuery je velice podobný.

```
document.querySelector('#root')
  .addEventListener('click', () => console.log('#root click'));
document.querySelector('#pause')
  .addEventListener('click', event => event.stopPropagation());
```

Zdrojový kód 64 – Zastavení propagace události v TypeScriptu

```
$('#root').click(() => console.log('#root click'));
$('#pause').click(event => event.stopPropagation());

link.click(event => event.preventDefault());
```

Zdrojový kód 65 – Zastavení propagace události v TypeScriptu s využitím jQuery

Zrušení výchozí akce prohlížeče spočívá v zavolání metody *preventDefault* na objektu reprezentující událost. Řešení je shodné s tím v Dartu – a to jak v čistém JavaScript API, tak v jQuery. Je zde patrné, že Dart i jQuery toto API převzaly.

```
link.addEventListener('click', event => event.preventDefault());
```

Zdrojový kód 66 – Zrušení výchozí reakce na událost v TypeScriptu

```
link.click(event => event.preventDefault());
```

Zdrojový kód 67 – Zrušení výchozí reakce na událost v TypeScriptu za pomoci jQuery

5.3 Komunikace se serverem

Při implementaci ať už moderní, dynamické aplikace, či čistě single-page aplikace, je nutné zajistit komunikaci se serverem prováděnou asynchronně na pozadí. V případě mé aplikace se jednalo například o autentifikační proces a samozřejmě o synchronizaci datových struktur obsahujících uživatelská data.

Tato komunikace na pozadí je často nazývána jako AJAX. V prvopočátcích AJAXu sloužil k výměně dat formát XML. Ten je v poslední době často nahrazován formátem JSON – především kvůli menšímu datovému objemu. Má aplikace taktéž využívá formát JSON.

Dart

V Dartu je možné pokládat asynchronní dotazy na pozadí pomocí třídy *HttpRequest* z knihovny *dart:html*. Knihovna je však napsána obecně a pracuje s odpovědí pouze jako s řetězcem. Pokud chceme využít například výše zmiňovaný formát JSON, musíme odpověď ručně zpracovat funkcí *parse* z knihovny *dart:json* (Zdrojový kód 68).

```
HttpRequest.request('/todo-lists').then((request) {
  var jsonResponse = Json.parse(request.responseText);
  print(jsonResponse);
});

var serializedTodoLists = Json.stringify({'todoLists': model.todoLists});
HttpRequest.request('/todo-lists', method:'POST', method:'application/json',
                    sendData: serializedTodoLists);
```

Zdrojový kód 68 – Asynchronní komunikace se serverem v Dartu

Také při odesílání je nutné objekt převést na řetězec – tentokrát funkcí *stringify* ze stejné knihovny. Celkově odesílání dat pomocí třídy *HttpRequest* není z důvodu její univerzálnosti moc pohodlné. Proto jsem k tomuto úkolu zvolil knihovnu *adaj*[1], která je vytvořena přímo pro komunikaci pomocí formátu JSON.

Knihovna má příjemnější API než základní řešení. HTTP metoda se zde definuje použitou funkcí, kde název funkce odpovídá požadované HTTP metodě. Automaticky také probíhá serializace a deserializace objektů do formátu JSON. Příklad použití můžete vidět na ukázce 69.

```
adaj.get('/todo-lists').done((response) {
  print(response);
}).go();

adaj.post('/todo-lists', {'todoLists': model.todoLists }).go();
```

Zdrojový kód 69 – Asynchronní komunikace se serverem v Dartu s knihovnou adaj

TypeScript

Použití základního API v TypeScriptu je značně nepříjemné. Jak ukazuje útržek zdrojového kódu níže, je potřeba vytvořit instanci třídy *XMLHttpRequest*, navázat se na událost změny stavu daného požadavku, otevřít spojení a požadavek odeslat. Každý úspěšný požadavek prochází postupně pěti očíslovanými stavy [23]:

- 0 – požadavek není inicializován,
- 1 – navázáno připojení k serveru,
- 2 – požadavek doručen,
- 3 – zpracování požadavku,
- 4 – požadavek dokončen, odpověď je k dispozici.

Při změně stavu tedy musíme kontrolovat, zda se jedná poslední stav (zbytek nás typicky nezajímá) a také HTTP kód odpovědi. Samotná odpověď je stejně jako v případě Dartu ve

[1] Název adaj je zkratkou spojení Asynchronous Dart And JSON.

formě řetězce. Je nutné ji tedy převést na objekt pomocí volání *Json.parse* (Zdrojový kód 70).

```
var request = new XMLHttpRequest();
request.onreadystatechange = () => {
  if (request.readyState == 4 && request.status == 200) {
    var jsonResponse = Json.parse(request.responseText);
    console.log(jsonResponse);
  }
};
request.open('GET', '/todo-lists', true);
request.send();

var request = new XMLHttpRequest();
request.open('POST', '/todo-lists', true);
request.setRequestHeader('Content-Type','application/json');
request.send(Json.stringify({todoLists: model.todoLists}));
```

Zdrojový kód 70 – Asynchronní komunikace se serverem v TypeScriptu

Odesílání dat pro programátora také není přívětivé, ačkoliv je jednodušší než jejich přijímání. Proto jsem se i zde rozhodl využít jQuery, které volání z předchozí ukázky zabalí do přívětivého API v ukázce následující.

```
$.get('/todo-lists').then(response => console.log(response));
$.post('/todo-lists', { todoLists: model.todoLists });
```

Zdrojový kód 71 – Asynchronní komunikace se serverem v TypeScriptu pomocí jQuery

5.4 Používání externích knihoven

Také v klientské části aplikace potřebujeme často používat externí knihovny. Ať už se jedná o v předchozích ukázkách zdrojových kódu zmiňované knihovny adaj a jQuery, či libovolné jiné. Jejich jednoduché připojování a správa bývá důležitou součástí pohodlného vývoje webových aplikací.

Dart

Velkou výhodou Dartu je možnost používat klauzuli *import* a s ní většinu knihoven použitelných na serveru i v klientské části aplikace. Dart VM se pak sám stará o jejich načítání. O správu verzí a závislostí je postaráno díky nástroji *pub*.

V Dartu je dokonce možné používat i JavaScriptové knihovny. Zde se však o jejich načítání a správu musíme starat sami (případně pomocí nástrojů zmíněných u TypeScriptu). Přístup k JavaScriptovým knihovnám zajišťuje knihovna *js*. Používání jak Dart knihoven, tak JavaScriptových knihoven je nastíněno na ukázce 72.

```
import 'package:js/js.dart' as js;
import 'package:adaj/adaj.dart' as adaj;

main() {
  adaj.get('/todo-lists');

  var $ = js.context.$;
  print($('#root').attr('id'));
}
```

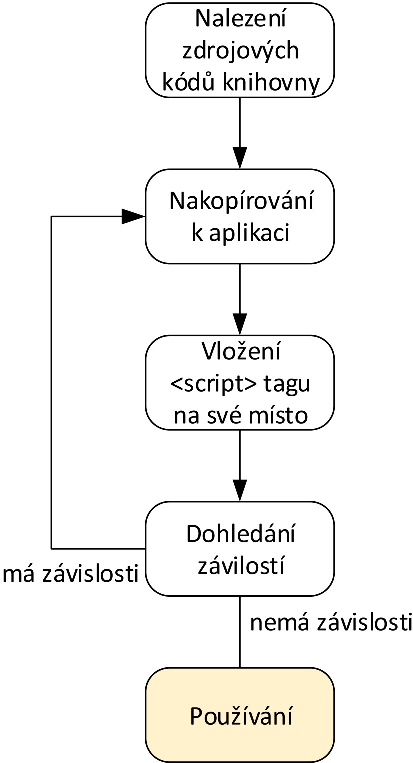
Zdrojový kód 72 – Používání externích knihoven v Dartu

TypeScript

Situace v TypeScriptu je oproti Dartu o poznání složitější. V základu není k dispozici žádný prostředek k načítání a správě různých knihoven. Často se tedy tato situace řeší ručním vkládáním *script* tagů do HTML, což přináší několik úskalí.

1. Musíme ručně přiložit i všechny knihovny, na kterých ty námi zvolené závisí.
2. Jednotlivé *script* tagy musíme udržovat ve správném pořadí (aby nebyla dříve vložena knihovna se závislostmi než její závislosti).
3. Ručně aktualizovat v případě nových verzí, sledovat nové závislosti apod.

Jak takový proces probíhá, je nastíněno na obrázku 14. Jedná se o poměrně zdlouhavý proces, který se s aktualizací knihoven vždy opakuje.



Obrázek 14 – Manuální instalace knihovny
Zdroj: vlastní zpracování

Abychom dokázali tohle všechno automatizovat, musíme použít hned několik nástrojů. Jako vhodná se ukázala kombinace nástrojů *Grunt*, *bower* a *RequireJS*. O Gruntu jsem se zmiňoval v kapitole 4.5. Jedná se o automatizační nástroj sloužící ke spouštění skriptů s širokou paletou skriptů již hotových.

Bower je alternativou *npm* určenou pro front-end[1] vývoj. Jedná se tedy o nástroj pro správu balíčků. Ovládá se podobně jako *npm* – pomocí příkazového řádku. Pro instalaci například *jQuery* stačí napsat `bower install jquery`. *Bower* nám tedy zajistí instalaci knihoven, jejich závislostí atd. Do výsledné aplikace je ale stále musíme přidat ručně.

O dynamické načítání knihoven v aplikaci se může postarat například *RequireJS*. Na základě konfiguračního souboru (Zdrojový kód 73) je pak možné v aplikaci dynamicky načítat knihovny (Zdrojový kód 74).

```
requirejs.config({
  baseUrl: './',
  paths: {
    'jquery-ui': 'bower_components/jquery-ui/ui/jquery-ui',
    jquery: 'bower_components/jquery/jquery'
  },
  shim: {
    'jquery-ui': {
      deps: ['jquery']
    }
  }
});
```

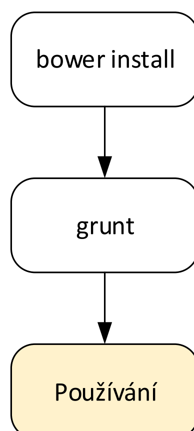
Zdrojový kód 73 – Konfigurace RequireJS

```
require(['jquery'], ($) => {
  console.log($);
});
```

Zdrojový kód 74 – Načtení knihovny pomocí RequireJS

Zbývá poslední část, kterou je nutno provádět manuálně, a to psaní konfiguračního souboru pro *RequireJS*. O to se postará právě výše zmiňovaný *Grunt* ve spojení s pluginem *grunt-bower-requirejs*, který po spuštění projde všechny knihovny nainstalované skrze *Bower* a vytvoří z nich konfigurační soubor pro *RequireJS*. Celý postup s využitím těchto nástrojů je nastíněn na obrázku 15.

[1] Klientská část webové aplikace.



Obrázek 15 – Instalace knihoven pomocí nástrojů Grunt, Bower a RequireJS
Zdroj: vlastní zpracování

5.5 Testování

Testování klientských aplikací se oproti serverové části trochu liší. Mnohem více se zde uplatňuje systémové testování a testy uživatelského rozhraní. Testuje se tedy celkové chování aplikace. Například že po zobrazení úvodní stránky má být zobrazen určitý nadpis a dané množství článků. Nebo že po přihlášení se uživateli zpřístupní uživatelské menu a podobně.

K tomuto testování nejsou obecně potřeba žádné vestavěné prostředky daného jazyka. Aplikace se může testovat i externím nástrojem, jakým je například *Selenium*. V něm se testy dají psát například v C#, Javě, Pythonu či Ruby. Jak takový test vypadá, můžete vidět například na ukázce 75.

```
import junit.framework.TestCase;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.*;
import java.net.URL;
import java.util.concurrent.TimeUnit;

public class SimpleTest extends TestCase {
  private WebDriver driver;

  public void setUp() throws Exception {
    DesiredCapabilities capabilities = DesiredCapabilities.firefox();
    capabilities.setCapability("name", "Testing Selenium 2");
    this.driver = new RemoteWebDriver(new URL("http://..."), capabilities);
  }

  public void testSimple() throws Exception {
    this.driver.get("http://www.google.com");
    assertEquals("Google", this.driver.getTitle());
  }

  public void tearDown() throws Exception {
    this.driver.quit();
  }
}
```

Zdrojový kód 75 – Test v nástroji Selenium
Zdroj: http://testingbot.com/support/getting-started/java.html

Alternativou může být například *CasperJS*, který nevyužívá standardních prohlížečů jako *Selenium*, nýbrž tzv. *headless* prohlížeč[1] *PhantomJS* založený na jádře WebKit (stejném jako například Chrome, Safari a Opera). *CasperJS* nativně podporuje testy psané v JavaScriptu a CoffeeScriptu. Pokud bychom chtěli zapojit TypeScript, je to možné, avšak musíme ho před spuštěním testů přeložit do JavaScriptu. Jak takový test napsaný v CofeeScriptu vypadá, můžete vidět na ukázce pod odstavcem. Funkčně je shodný s předchozím testem.

```
casper.test.begin "Google title is Google", 1, (test) ->
  casper.start "http://www.google.com/", ->
    test.assertTitle "Google", "google homepage title is the one expected"

  casper.run ->
    test.done()
```

Zdrojový kód 76 – Test v CasperJS

[1] Prohlížeč bez grafického výstupu – k ověření funkčnosti aplikace dostačující.

Závěr

Primárním cílem této práce bylo představit nové, perspektivní skriptovací jazyky Dart a TypeScript a ukázat jejich použití při implementaci webové aplikace typu klient-server. Jako svůj soukromý cíl jsem si navíc stanovil zhodnocení, zda jsou vybrané jazyky způsobilé zaujmout místo mezi svými běžněji používanými alternativami. Hlavními metrikami byly podpora vývojových prostředí, množství dostupných knihoven a subjektivní pocity při vývoji v daném jazyce.

V teoretické části práce nejprve nastiňuji rozdělení programovacích jazyků, jehož součástí je i definice jazyků skriptovacích, na které se má práce zaměřuje. Následuje představení v současné době nejpoužívanějších skriptovacích jazyků a jejich postavení do kontrastu s jazyky Dart a TypeScript. Součástí teoretické části je i náhled do problematiky testování softwaru.

Praktická část se pak věnuje ryze jazykům Dart a TypeScript. Jako prostředek k jejich představení jsem zvolil webovou aplikaci typu „úkolníček“. Jelikož však použití některých prostředků daných jazyků nebylo možné v rámci zvolené aplikace jednoduše použít, vytvořil jsem pro jejich demonstraci další, umělé příklady. Jazyky představuji především formou fragmentů zdrojových kódů s krátkým komentářem.

Představení jazyků je rozděleno do dvou částí, a to do ukázek použití jazyka v serverové části aplikace a do ukázek použití jazyka v klientské části aplikace. V každé z těchto částí práce jsou demonstrovány typické situace, se kterými se bude programátor implementující webovou aplikaci potýkat, a jejich řešení v obou zvolených jazycích. Tím považuji hlavní cíl práce za splněný.

Co se týče mého soukromého cíle, zvolené jazyky mě poměrně potěšily. Na Dartu je vidět, že se jedná o nově navržený jazyk, který ještě netrpí žádnými neduhy zanesenými ve starých verzích. API jeho základních knihoven je homogenní a intuitivní. Jeho hlavní nevýhodu vidím v tom, že se jedná o velice málo rozšířený jazyk, a tudíž ani množství dostupných knihoven pro něj není velké. Také podporu ve vývojových prostředích považuji za podprůměrnou.

TypeScript na druhou stranu pouze přináší nové koncepty do světa JavaScriptu. Tuto nadstavbu považuji za výbornou, nicméně do TypeScriptu prosakuje technologický dluh a neduhy JavaScriptu. Množství knihoven je zde naopak obrovské – právě díky kompatibilitě s JavaScriptem. Podpora ve vývojových prostředích je lepší než u Dartu, nicméně stále to není to pravé.

Celkově považuji oba jazyky za konkurenceschopné pro své běžně používané alternativy. TypeScript bych i doporučil nasadit všude tam, kde se nyní používá JavaScript. U Dartu je situace složitější. Jako jazyk se mi líbí víc, avšak jeho použití například ve firmách ještě podle mě možné není.

Literatura

1. **OUSTERHOUT, John K.** Scripting: Higher Level Programming for the 21st Century. *IEEE Computer magazine.* Březen 1998, Sv. 31, 3. ISSN 0018-9162.

2. **HYDE, Randall.** *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level.* San Francisco : No Starch Press, Inc., 2008. ISBN 1593270658.

3. **RILEY, John S.** Interpreted vs. Compiled Languages. [autor knihy] John S. Riley. *Getting Started in Programming.*

4. **STANNARD, Kevan.** Procedural vs Object Oriented Programming. *Object Oriented ColdFusion.* [Online] [Citace: 26. březen 2013.] http://objectorientedcoldfusion.org/procedural-vs-object-oriented.html.

5. **KŘIVÁNEK, Pavel.** Statická vs. dynamická typová kontrola. *Root.cz.* [Online] 24. červen 2004. [Citace: 26. březen 2013.] http://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/. ISSN 1212-8309.

6. **PECH, Václav.** Dynamické programovací jazyky. [Online] 8. říjen 2008. [Citace: 1. duben 2013.] http://www.europen.cz/Proceedings/33/Dynamicke_jazyky.pdf.

7. **SKEET, Jon.** The Beauty of Closures. *Developer Fusion.* [Online] 8. říjen 2008. [Citace: 26. březen 2013.] http://www.developerfusion.com/article/8251/the-beauty-of-closures/.

8. **PICHLÍK, Roman.** Exkurze do historie JavaScriptu aneb jak to všechno začalo. *Sova v síti.* [Online] Říjen 2002. [Citace: 11. Srpen 2013.] http://www.sovavsiti.cz/download/sova0210.txt. ISSN 1213-9076.

9. **QUAZI, Salman.** .NET and Node.JS – Performance Comparison. *Salman Quazi.* [Online] 26. Březen 2013. [Citace: 12. Srpen 2013.] http://www.salmanq.com/blog/net-and-node-js-performance-comparison/2013/03/.

10. **STEIGERWALD, Daniel.** Seriál: OOP v Javascriptu. *Zdroják.cz.* [Online] Březen 2010. [Citace: 13. Srpen 2013.] http://www.zdrojak.cz/serialy/oop-v-javascriptu/. ISSN 1803-5620 .

11. **NYSTROM, Bob.** Milestone 1 Language Changes. *Dart: Structured web apps.* [Online] červenec 2012. [Citace: 5. srpen 2013.] https://www.dartlang.org/articles/m1-language-changes/.

12. **LADD, Seth.** What's New with Dart's M2 Release. *Dart: Structured web apps.* [Online] prosinec 2012. [Citace: 5. srpen 2013.] https://www.dartlang.org/articles/m2-whats-new/.

13. **LOITSCH, Florian.** New Streams API with Dart Milestone 3. *Dart News & Updates.* [Online] 20. únor 2013. [Citace: 5. srpen 2013.] http://news.dartlang.org/2013/02/new-streams-api-with-dart-milestone-3.html.

14. **BAK, Lars.** Core libraries stabilize with Dart's new M4 release. *Dart News & Updates.* [Online] 16. duben 2013. [Citace: 5. srpen 2013.] http://news.dartlang.org/2013/04/core-libraries-stabilize-with-darts-new.html.

15. **LADD, Seth.** Release Notes for Dart's Beta Release. *Dart News & Updates.* [Online] 19. červen 2013. [Citace: 5. srpen 2013.] http://news.dartlang.org/2013/06/release-notes-for-darts-beta-release.html.

16. **JORGENSEN, Paul.** *Software testing.* Auerbach : Boca Raton, 2008. ISBN 0-8493-7475-8.

17. *Zpráva o stavu a potřebách v oblasti testování informačních systémů.* **VRBKA, Zdeněk.** Brno : Fakulta informatiky Masarykova univerzita, 2008.

18. **TutorialsPoint.** Software Testing Quick Guide. [Online] [Citace: 15. březen 2013.] http://www.tutorialspoint.com/software_testing/software_testing_quick_guide.htm.

19. **HAMMER, Derek.** Programming Practice :: Inverted Pyramid. [Online] 25. květen 2010. [Citace: 20. spen 2013.] http://www.derekhammer.com/2010/03/25/programming-practice-inverted-pyramid.html.

20. **HLAVA, Tomáš.** Fáze a úrovně provádění testů. *Testování softwaru.* [Online] 21. spren 2011. [Citace: 16. březen 2013.] http://testovanisoftwaru.cz/tag/unit-testing/.

21. **GHEORGHIU, Grig.** Performance vs. load vs. stress testing. *Agile Testing.* [Online] 28. únor 2005. [Citace: 16. březen 2013.] http://agiletesting.blogspot.cz/2005/02/performance-vs-load-vs-stress-testing.html.

22. **Janssen, Cory.** Shared Resources. *Techopedia.* [Online] [Citace: 15. srpen 2013.] http://www.techopedia.com/definition/24796/shared-resources.

23. **Refsnes Data.** AJAX - The onreadystatechange Event. *W3Schools.* [Online] [Citace: 20. srpen 2013.] http://www.w3schools.com/ajax/ajax_xmlhttprequest_onreadystatechange.asp.

24. **Microsoft.** TypeScript. [Online] 2012. [Citace: 13. duben 2012.] http://www.typescriptlang.org/.

25. **ŽÁRA, Ondřej.** DOM události: co o nich možná nevíte. *Zdroják.cz.* [Online] 24. červenec 2013. [Citace: 19. srpen 2013.] http://www.zdrojak.cz/clanky/dom-udalosti-co-o-nich-mozna-nevite/. ISSN 1803-5620.

26. **NYSTROM, Bob.** Improving the DOM. *Dart: Structured web apps.* [Online] březen 2013. [Citace: 3. srpen 2013.] https://www.dartlang.org/articles/improving-the-dom/.

27. **HRÁČEK, Filip.** Filip Hráček o Google, Dartu a Glass. [podcast] Praha : devminutes.cz, 6. červenec 2013.

Příloha A – CD se zdrojovými kódy

Přiložené CD obsahuje:

- tuto práci v elektronické podobě,
- implementované aplikace,
- zdrojové kódy aplikací použitých pro testování výkonu.

Příloha B – Testovaná aplikace v Dartu

```
import 'dart:io';
import 'dart:isolate';

handleRequest() {
  port.receive((fileId, reply) {
    var inputNumber;
    try {
      inputNumber = int.parse(fileId);
    } catch (e) {
      reply.send(null);
      return;
    }

    var fileNumber = inputNumber < 10 ? '00$inputNumber' :
                     inputNumber < 100 ? '0$inputNumber' : '$inputNumber';

    var filename = 'input$fileNumber.txt';

    new File('C:/Users/Jan/Desktop/nodenetperf/data/$filename')
    .readAsString().then((fileContent) {
      var listOfNumbers = fileContent.split("\r\n");
      listOfNumbers.sort();

      reply.send('$filename\t' + listOfNumbers[(listOfNumbers.length/2).toInt()]);
    });
  });
}

main() {
  List<SendPort> requestHandlers = new List();
  for(int i = 0; i < 8;i++){
    requestHandlers.add(spawnFunction(handleRequest));
  }

  HttpServer.bind('127.0.0.1', 8080).then((server) {
    var requestHandlerNumber = 0;
    server.listen((HttpRequest request) {

      if(requestHandlerNumber >= requestHandlers.length) requestHandlerNumber = 0;

      requestHandlers[requestHandlerNumber++].call(request.uri.path.substring(1))
      .then((result) {
        if(result == null) {
          request.response.statusCode = 400;
        } else {
          request.response.write(result);
        }
        request.response.close();
      });
    });
  });

  var receiver = new ReceivePort();
  receiver.receive((stay, alive){});
}
```