



O frameworku

Je 12.6.2017. Už jsem začal pracovat na JKali frameworku. Mám rozpracovaný vývojový nástroj JKali-Rockets (JKR). Při rozmýšlení návrhu fungování frameworku jsem se posunul technologicky dále, než byl můj první návrh v předchozím díle Úvahy. Tento dokument bude sloužit jako takový návod na používání frameworku včetně ukázek. Až bude v použitelném stavu, tak bych ho chtěl zpřístupnit nějakou volnou formou (asi opensource), ale podmínkou bude, že budu mít zaplacenou práci, kterou jsem nad tím strávil. Buď to si budu muset sehnat nějakého sponzora a nebo to zkusím dobrovolnou sbírkou přes systém Eden (až bude funkční), jak je popsáno v článku „Charitativní sbírka“ v Úvahách na stránce 30. Teď rozeberu trochu stručně framework z pohledu uživatele a jaké to bude mít výhody.

Základní princip fungování:

Všecký vývoj se bude dělat pouze v JavaScriptu (i databázové záležitosti). JavaScript bude buďto společný pro web i server a nebo jen pro web či jen pro server. Zjednoduší se tím programování. Všechny JavaScript a i všechny ostatní soubory pro web, které prohlížeč vyžaduje budou uloženy v databázi a webový server je bude distribuovat z mezipaměti, do které se data dostanou buď z databáze při nastartování webového serveru a nebo nasazením pomocí nástroje JKR. Mezipaměť bude možné pro vývoj vypnout a přesměrovat přímo na souborový systém. Samotný webový server bude jen v Javě a ideálně se nabízí jednoduchý Tomcat. Není důvod přecházet na PHP. PHP je stejně tak jako Java multiplatformní a webový server se bude aktualizovat jen v případě aktualizace frameworku.

Databázová nezávislost:

Framework bude umožňovat univerzální PL/SQL jazyk psaný v JSON. Tento jazyk z počátku bude možné převádět do třech databází a to PostgreSQL, MySQL a Oracle, protože všechny tyto databáze mají podporu Procedural Language (PL). Může nastat situace, kdy se rozhodne přejít na jinou databázi a v takovém případě je komplikace přepisovat všechny, tabulky, pohledy, procedury, triggerů a jiné. Proto bude ideální zvolit nějaký střed, který bude překladatelný vždy alespoň do jedné z těchto databází a v tom programovat. Pokud se uživatel rozhodne využít výhody konkrétní databáze na úkor toho, že nebude program přenositelný, tak tu možnost mít bude.

Spouštění/lazení JavaScriptu:

Nejprve jsem si myslel, že JavaScript se bude spouštět v databázi PostgreSQL, která umožňuje PLV8, ale problém by byl na ostatních databázích, které JavaScript tolik nepodporují. Dobře umí ale ovládat JavaScript přímo Java, tak že bude ideální provádět interpretaci JavaScriptu na straně web serveru. Tím se alespoň trochu rozloží zátěž mezi web serverem a databází. Navíc vývojář bude moci u sebe na web serveru sledovat konzoli JS, což ledacos usnadní v případě, že by se na databázi mohl dostat jen vzdáleně. Java umí přidělit JavaScriptu kontext pro proměnné, tak že nebude problém kontextově oddělit více aplikací nasazených na stejném web serveru.

Verzování:

Nástroj JKR bude sledovat verze všech JavaScriptů, různých webových zdrojů a i databázových objektů a automaticky nahraje na server to, co je potřeba. Všechny změny v tabulkách a ostatních databázových objektech se budou evidovat a JKR umožní vývojáři vygenerovat SQL pro libovolnou databázi, které zařídí povýšení z verze X na verzi Y. Toto SQL může být odesláno příslušné osobě, která má oprávnění k spouštění SQL dotazů.

Aktualizace aplikace pomocí JKR:

Jedna možnost aktualizace aplikace je vygenerováním SQL skriptu. Další možnost je, že vývojář provede povýšení aplikace vzdáleně pomocí JKR, který se spojí s lokálním webovým serverem umožňující připojení pro JKR. Veřejné webové servery samozřejmě připojení přes JKR umožňovat nebudou z bezpečnostních důvodů.

Vysoká výkonnost:

JavaScript bude využíván na prohlížeči i na serveru. Na serveru bude ale jen zajišťovat bezpečnostní kontrolu vstupních dat. Veškeré grafické operace a vše co s tím souvisí bude kompletně od serveru oddělené a bude to řešit jen prohlížeč. Navíc se bude využívat PL/SQL pro operace nad databází, což by mělo být efektivnější, než například využívání persistentních nástrojů jako třeba Hibernate.

Zachovávání relace přihlášeného uživatele:

Informace o přihlášeném uživateli si udržuje webový server. Když se na webový server nasazuje nová aplikace v Javě, tak to relaci ukončí. V tomto případě se ale bude měnit pouze JavaScript, tak že relace se při aktualizaci aplikace zachová.

Konfigurační položky:

Konfigurační položky se nebudou jednotlivě přenášet pokaždé, když bude uživatel potřebovat. Ani z databáze ani z případné relace. Ty se přenesou při přihlášení jednorázově na stranu prohlížeče, kde se uchovávají v paměti po dobu otevřeného okna. Nemusí se přenášet všechny, pokud jich bude hodně. Stačí jen jmenovitě určit které se přenesou na klienta.

Status 19.6.2017

Je 19.6.2017. Píši po sedmi dnech, protože se snažím věnovat energii hlavně programování a baví mě to, protože dělám den co den pokroky. Pamatuji si, že v týmu jsme mívali jeden čas pondělní statusy, kde se vždy probíralo, jak jdou práce na vývoji například přepisování do nového klienta. Asi by nebylo špatné něco podobného zavést k tomuto projektu. Vždy v pondělí dám vědět, jak probíhal vývoj v posledním týdnu. I kdyby to nikdo kromě mě nečetl, tak to budu mít alespoň jako vzpomínku na to, co jsme dělali.

Pokračuji tím způsobem, který jsme navrhoval a dělám pokroky, tak že jsem spokojený. Způsob, kterým vyvíjím je ten, že pokud mě napadne lepší řešení, než je to, nad kterým jsem už trávil čas, tak ho zvolím i když to třeba obnáší práci s přepisováním už napsaných kódů. Snažím se to brát opravdu od základů a důsledně rozvažovat, aby nemohlo pravděpodobně dojít k tomu, že až budu výše nad základy, tak mě napadne řešení, jak je vylepšit ale už bude více práce s předěláváním kódu, který je na nich postaven. Na konci eseje Úvahy jsem napsal zhruba plán, podle jakého budu postupovat. Prvně jsem měl v plánu udělat „rakety“, pak se zabývat webovou vrstvou a potom vzít celkově JavaScript klientsky i serverově. Základy, které teď řeší ale zahrnují všechny tyto tři části. Dá se říct, že tedy dělám na všem a nic není kompletně hotové. Framework se bude vyvíjet asi stále, ale v plánu označím modul jako hotový v době, kdy už bude použitelný pro nějaký základní vývoj. Něco bych do tohoto manuálu mohl už rozepsat, ale asi bych to nemohl vzít z toho správného konce pro nejlepší pochopitelnost, tak raději ještě s manuálem počkám.

Co tedy mám už funkční a celkem pěkně vyřešené? Mluvil jsem o tom, že je potřeba psát databázově nezávislé „SQL“ specifikace. Práci s databází potřebuji mít ideální hned ze začátku hotovou, protože framework bude generovat a pracovat s řídicími tabulkami, ve kterých budou zdrojové kódy, různá data a verzování. Bylo potřeba vytvořit generátory SQL, které budou umět hezky čitelně formátovat výstup a to se mi podařilo i když zatím umí vytvořit jen tabulky s pár druhy sloupci, ale základní cesta je prošlápnutá. Dokonce umí i upozornit na různé chyby ve vstupních datech. K jejich tvorbě jsem ale potřeboval (idelně) zavést OOP (Objektově Orientované Programování) a celkově správce tříd funkční pod NodeJS a to se mi povedlo. Dokonce mám OOP rozběhané i přímo pod Java8, která disponuje JavaScript interpretem. Dalo to trochu práce, ale

podářilo se. Generátory umí informovat o validitě vstupů a tyto informace je potřeba do budoucna umět lokalizovat (mechanismus překládání zpráv do libovolných jazyků). Lokalizaci mám také celkem pěkně vytvořenou v rámci JavaScriptu včetně tvorby výjimek (výjimky jsou speciální způsob zachytávání chyb v programování) obsahující informace pro překlad. Na straně serveru v Javě jsem si připravil zatím možnost připojování do jednotlivých databází (PostgreSQL, MySQL a Oracle), práci se zdroji dat pomocí CP (Connection Pool) a seznámil jsem se i trochu blíže se zabezpečením a práci se schémata/databázemi/uživateli na jednotlivých databázích. Vše mi funguje, jak má, což je fajn.

Status 26.6.2017

Je 26.6.2017. Jsem celkem „ponořen“ v programování. Mám toho víc na psaní. Chtěl bych ale některé věci dodělat, než je začnu popisovat. Status odložím na zítřek. Zítra se budu věnovat asi hlavně psaní. Chtěl bych framework popsat tak, že to bude srozumitelné i pro méně zasvěcené lidi.

...

Je 28.6.2017. Stále jsem rozjetý v programování a nechce se mi pouštět do většího psaní. Není dobré moc často střídat aktivity, když člověk chce dělat na něčem intenzivně. Už se mi ale uzavírá po necelých třech týdnech jedna velká kapitola. S aktuálním stavem jsem velmi spokojený, protože základy jsou udělané kvalitně. Details rozepíšu později, až se budu věnovat více psaní a vše co budu popisovat bude plně funkční. Možná během pár dní označím první a druhou část frameworku jako vyřešenou, protože se už přesunu k třetí a poslední části, která se týká hlavně klienta (prohlížeč / zobrazování). Teď se hodně věnuji rockets a serverové části. Až budu mít první dvě části v uspokojivém stavu, tak asi pomalu začnu připravovat serverovou část systému a vývoj třetí části frameworku budu rovnou přizpůsobovat potřebám pro vývoj systému. Samotný popis prvních dvou částí bych chtěl začít až budou mít tyto části v uspokojivém stavu. Popis zabere asi pár hodin a doladění rockets a serveru asi nějaký ten den. Příští týden si asi vezmu volno, tak bude ideální to do té doby stihnout. Po volnu se asi pustím do té třetí části. S tou mám mnohem větší zkušenosti z vývoje frameworku EJS a nebude to pro mě tak průkopnická činnost jako natahat JavaScript na server a nodejs, což řeším teď. Pak už jenom navrhnout a udělat nějaké šikovné skloubení kódů prohlížeče a serveru a je vyhráno.

Rychlokurz JavaScriptu



Tak pomalu začíná červenec a já se zase pouštím do psaní, protože jsem poslední dobou odvedl kus práce na programování. Chystám se popsat to, co jsem udělal, ale nejprve tu pro lidi, které to zajímá a nemají moc znalost JavaScriptu a celkově programování, tak tu mám takový rychlokurz, který si může funkčně každý vyzkoušet na praktických ukázkách. Stačí si otevřít prohlížeč a najet na mé stránky <http://jkali.cz/>, kde budeme trochu s nimi experimentovat. Zkoušel jsem to na prohlížeči Firefox, ale mělo by to jít podle mě ve všech

stejně. Po najetí stránky otevřete vývojovou konzoli stiskem klávesy F12. Ve spodní části okna se otevře vývojový panel a přepnete do záložky „Konzole“. Nalevo je vidět výstup z konzole JavaScriptu a vpravo je editor kódu. Vždycky stačí jen překopírovat zvýrazněnou ukázkou kódu do editoru, spustit skript tlačítkem „Spustit“ a sledovat, co se děje. U prohlížeče Chrome je po stisku F12 konzola vidět úplně dole a editor je jen jednořádkový. Dá se ale do něj zkopírovat víceřádkový text, tak že nás nic neomezuje v spouštěné ukázkách. V Chromu se spustí skript klávesou enter. Pokud ale budete mít otevřenou konzoli třeba nad panelem s PDF dokumentem namísto stránky, tak asi nebude fungovat.

Jenom krátce něco o JavaScriptu. JavaScript vznikl ve společnosti Netscape, která vyvíjela

prohlížeč Netscape Navigator. Prvotní posláním bylo dát vývojářům webu možnost programovat obsah webových stránek bez potřeby dotazování serveru na jejich obětovně vygenerování. Samotná firma Netscape nebyla příliš úspěšná a ani nevím, zda ten prohlížeč stále vyvíjí, ale JavaScript se stal velmi populární, asi díky své jednoduchosti. Je inspirován jazykem Java (proto ten název), ale je to skriptovací typ jazyka na rozdíl od Javy a tak nevyžaduje překlad do žádných spustitelných souborů, ale rovnou se spustí v textové formě (skript), která je čitelná pro vývojáře. Proto také nemusíte instalovat žádné překladače a rovnou lze JS spustit v konzoli prohlížeče, který má vlastně každý nainstalovaný. Jako skriptovací jazyk umožňuje vývojářům velkou svobodu. Dnes už se JS přidává k různým aplikacím (mimo prohlížeče), aby umožnili uživatelům automatizovat práci s aplikací. Jako samostatný modul JS, se používá aplikace NodeJS, která umožní uživateli v podstatě stejný vývoj jako u klasických aplikací, tak že si může naprogramovat třeba server, prohlížet a zasahovat do souborového systému na počítači a ovládat všechno možné. NodeJS je dostupný i na mini počítače Raspberry, které se dají v něm programovat. JS už se dostává i na servery, kde se mi zdá, že začíná pomalu vytlačovat původní programovací jazyky a já se tímto frameworkem k tomuto dílu přičiňuji, ale nejsem jediný, koho napadlo zabývat se tímto novým trendem.

Výrazy

Začneme vyhodnocováním výrazů. Zkopírujte modrý text do editoru a spusťte. Výsledek bude vidět v konzoli. Výsledek by měl být.

Skript:

```
1 + 2 + 3
```

Výsledek:

6

Výrazy mohou být i textové. Textovým hodnotám se říká řetězce a jsou v uvozovkách a nebo apostrofech. Dají se spojovat operandem (znaménkem) +.

Skript:

```
"text" + 1 + "....." + "text" + 2
```

Výsledek:

```
"text1.....text2"
```

Posloupnost příkazů

Programování je hlavně o psaní programu a ne jen výrazů. Program je posloupnost příkazů, které se provádí jeden po druhém jak jdou za sebou. Pořadí se bere zleva doprava ze zhora dolů a jednotlivé příkazy se správně mají oddělovat středníkem, ale JS středník odpouští. Teď ukážu operand = pro přiřazení hodnoty do proměnné. Nejprve nastavíme proměnnou x, potom y a pak je spojíme do výsledného výrazu. Operand přiřazení = hodnotu výrazu na pravé straně přidělí do proměnné a nebo výrazu na levé straně. Pokud proměnná neexistuje, tak se vytvoří. Toto je hlavní vlastnost skriptovacích jazyků. Nikde není potřeba proměnnou předem registrovat, ale rovnou se používá.

Skript:

```
x = 1 + 2 + 3;  
y = "...abc" + "def";  
"text..." + x + y;
```

Výsledek:

```
"text...6...abcdef"
```

Podmíněný blok příkazů

Někdy je potřeba vymezit určitý blok kódu, který se provede jen za určitých podmínek. K tomu se používá klíčové slovo **if** a **else**. Za slovem **if** je v klasických závorkách uveden logický výraz (výrok), který platí a nebo neplatí a na základě toho se provede nebo neprovede blok ohraničený ve složených závorkách. Pro lepší přehlednost se příkazy uvnitř bloku trochu odsadí doprava.

Vytvoříme tedy proměnnou `out` s prázdným řetězcem, který budeme postupně v blocích plnit a na konci se uvidí, které bloky se provedly a které ne. Za slovem **else**, které je nepovinné se uvádí blok, který se provede v opačném případě, kdy výrok neplatí. Připojování řetězce se dělá operandem `+=` (`out = out + „1“` je to samé jako `out += „1“`). Operand `==` slouží jako porovnání dvou hodnot. Pokud jsou hodnoty rovné, tak je výrok pravdivý. Musí to být `==`, protože samotné `=` je už použité pro přiřazování hodnot. Přiřazování hodnot se používá častěji, tak proto asi dali přednost jednoduchému `=` pro tuto operaci. Opačně funguje operand `!=`, který platí jen tehdy, pokud porovnávané hodnoty jsou jiné. Operand `||` (nebo) platí pokud alespoň jedna z porovnávaných hodnot je pravdivá a operand `&&` (a zároveň) platí jen tehdy, pokud oba výrazy jsou pravdivé.

Skript:

```
out = "";
```

```
if (2 > 1) {  
    out += "1 ";  
}
```

```
if (2 <= 1) {  
    out += "2 ";  
}
```

```
if ("aa" == "aa") {  
    out += "3 ";  
}
```

```
if ("aa" == "aa") {  
    out += "4 ";  
} else {  
    out += "5 ";  
}
```

```
if ("aa" == "bb") {  
    out += "6 ";  
} else {  
    out += "7 ";  
}
```

```
if (1 > 2 || 2 > 1) {  
    out += "8 ";  
}
```

```
if ((1 > 2 || 2 > 1) && 10 != 10) {  
    out += "9 ";  
}
```

```
out;
```

Výsledek:

"1 3 4 7 8 "

Cykly

Někdy je potřeba určité bloky kódu ne jen podmínit, ale v případě, že se provedou, tak se podmínka kontroluje znova a případně se opakovaně blok provádí do doby, kdy podmínka neplatí. Klíčové slovo **while** se používá jako if, ale nemá žádný přídavek else. V ukázce zase budeme řetězit text do proměnné out, který na konci vypíšeme do konzole. První cyklus while kontroluje zda proměnné i je ≤ 9 a pokud ano tak se stále dokola provádí a tím přidává do out text. Text je zakončen symbolem **\n**, což znamená, že ve viditelném výsledku v konzoli se v tomto místě přechází na nový řádek. Druhý cyklus není while, ale for. Funguje úplně stejně jako while, ale prvotní nastavení proměnné i, podmínka a přičtení hodnoty do proměnné i se uvádí pro lepší přehlednost vše do závorek za slovem **for** jako samostatné příkazy.

Skript:

```
out = "";
```

```
i = 0;
```

```
while (i <= 9) {  
    out += "line" + i + "\n";  
    i += 1;  
}
```

```
out += "-----\n";
```

```
for (i = 0; i <= 9; i += 1) {  
    out += "line" + i + "\n";  
}
```

```
out;
```

Výsledek:

```
"line0  
line1  
line2  
line3  
line4  
line5  
line6  
line7  
line8  
line9  
-----  
line0  
line1  
line2  
line3  
line4  
line5  
line6  
line7  
line8  
line9  
"
```

Funkce (podprogramy)

Často je potřeba nějak si označit či pojmenovat určitý blok příkazů a provádět ho podle potřeby. To

řeší funkce (podprogramy). Klíčovým slovem **function** vytvoříme funkci. Do klasických závorek za slovem uvedeme vstupní parametry, pokud nějaké jsou. Ve složených závorkách je blok příkazů. Aby byl kód přehlednější a jednodušší, tak si uložíme do proměnné print funkci s jedním parametrem. Funkce bude mít za úkol přidat do výsledného řetězce out text na samostatný řádek. Text je v parametru funkce. Hned pod definicí funkce jí zavoláme pro vytištění odrážky FUNKCE —... Když chceme funkci provést, tak se tomu také říká zavolat jí. Volání se dělá tak, že se za proměnnou s funkcí napíšeme klasické závorky ať už s parametrem nebo bez parametru. Další dvě použití funkce neobsahují nic nového. U tisku matice se jen používá vnořený cyklus. Druhá ukázka funkce (mocnina - pow) má dva parametry a k tomu navíc používá návratovou hodnotu, tak že je možné ji volat uvnitř nějakého výrazu, jehož součástí je návratová hodnota funkce. Uvnitř funkce se používá klíčové slovo **return**, kterým řekneme, že zde chceme ukončit chod funkce a navrátit hodnotu výrazu napravo od return, což je proměnná x. Hned pod definicí funkce je její volání uvnitř parametrů funkce print.

Skript:

```
out = "\n";
print = function(text) {
    out += text + "\n";
}

print("FUNKCE -----");

for (i = 0; i < 10; i++) {
    print("line " + i);
}

print("\nMATICE -----");

for (i = 0; i < 10; i++) {
    line = "";
    for (n = 0; n < 10; n++) {
        line += n + "x" + i + " ";
    }
    print(line);
}

print("\nMOCNINA -----");

pow = function(x, m) {
    _x = x;
    for (i = 1; i < m; i++) {
        x *= _x;
    }
    return x;
}

print("6^2 = " + pow(6, 2));
print("3^3 = " + pow(3, 3));
print("2^4 = " + pow(2, 4));

out;
```

Výsledek:

```
"
FUNKCE -----
line 0
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9

MATICE -----
0x0 1x0 2x0 3x0 4x0 5x0 6x0 7x0 8x0 9x0
0x1 1x1 2x1 3x1 4x1 5x1 6x1 7x1 8x1 9x1
0x2 1x2 2x2 3x2 4x2 5x2 6x2 7x2 8x2 9x2
0x3 1x3 2x3 3x3 4x3 5x3 6x3 7x3 8x3 9x3
0x4 1x4 2x4 3x4 4x4 5x4 6x4 7x4 8x4 9x4
0x5 1x5 2x5 3x5 4x5 5x5 6x5 7x5 8x5 9x5
0x6 1x6 2x6 3x6 4x6 5x6 6x6 7x6 8x6 9x6
0x7 1x7 2x7 3x7 4x7 5x7 6x7 7x7 8x7 9x7
0x8 1x8 2x8 3x8 4x8 5x8 6x8 7x8 8x8 9x8
0x9 1x9 2x9 3x9 4x9 5x9 6x9 7x9 8x9 9x9

MOCNINA -----
6^2 = 36
3^3 = 27
2^4 = 16
"
```

Výjimky (chyby)

Někdy nastane závažný problém, který je potřeba řešit zastavení vykonávání funkce a nebo celého programu, protože v důsledku tohoto problému by se mohl program chovat špatně. Příkazem **throw** se vyhodí výjimka, která může být třeba text či objekt a ta zastavuje vykonávání programu tím, že zastavuje funkci po funkci v místě, kde se program zrovna vykonává. Výjimka ale může být zachycena ošetřena v **try-catch** blocích. V ukázce jsou tři funkce. V té třetí, která je nejvíce vnořená, tak nastane výjimka. Ta provádění třetí a druhé funkce zastaví, ale v první funkci je výjimka zachycena try blokem a v catch bloku se zpracuje.

Skript:

```
out = "\n";
print = function(text) {
    out += text + "\n";
}

fn1 = function() {
    print("fn1 start");
    try {
        fn2();
    } catch (e) {
        print("Nastala chyba: " + e + "", ale problém byl vyřešen a program může pokračovat");
    }
    print("fn1 end");
};

fn2 = function() {
```

```

print("fn2 start");
fn3();
print("fn2 end");
};

fn3 = function() {
  print("fn3 start");
  throw "Máme tu problém ve funkci fn3. Nelze funkci dokončit";
  print("fn3 end");
};

fn1();

out;

```

Výsledek:

```

"
fn1 start
fn2 start
fn3 start
Nastala chyba: 'Máme tu problém ve funkci fn3. Nelze funkci dokončit', ale
problém byl vyřešen a program může pokračovat
fn1 end
"

```

Životnost proměnných

Některé proměnné se nám hodí uchovávat v paměti jen nějakou dobu ve které je potřebujeme používat. To nám zajistí, že se navzájem neovlivňují stejně pojmenované proměnné v různých funkcích nebo blocích. Slouží k tomu klíčová slova **var** a **let**, která se mohou psát před proměnnou v okamžiku přiřazení její hodnoty, ale to přiřazení tam být nemusí, tak že například (var x = 1; a nebo jen var x;). Rozdíl mezi var a let je v tom, že var určuje životnost proměnné na celou funkci od momentu jejího vytvoření, ale let určuje životnost pouze na rozmezí bloku (například podmíněný blok if-else, nebo cyklus for) od jejího vzniku. Pokud ale chci číst z proměnné, která zatím neexistuje, tak se tím vygeneruje výjimka, která může ukončí program. V další ukázce testujeme dostupnost proměnných a odchyťujeme výjimky, aby program mohl pokračovat. Je použit podmíněný blok, jehož podmínka je hodnota **true** (pravda). Tato podmínka bude vždy platit a program se vždy vykoná. Další možnost by byla použít místo true třeba výraz 1 == 1, ale toto je hezčí. Opakem true je **false** (lež) a taková podmínka by se nikdy neprovedla.

Skript:

```

out = "\n";
print = function(text) {
  out += text + "\n";
};

test = function() {
  if (true) {
    x = 1;
    var y = 2;
    let z = 3;

    print("\n----- v bloku:");
    try { print("x = " + x); } catch(e) { print("problém: " + e); }
    try { print("y = " + y); } catch(e) { print("problém: " + e); }
  }
}

```

```

    try { print("z = " + z); } catch(e) { print("problém: " + e); }
  }

  print("\n----- ve funkci:");
  try { print("x = " + x); } catch(e) { print("problém: " + e); }
  try { print("y = " + y); } catch(e) { print("problém: " + e); }
  try { print("z = " + z); } catch(e) { print("problém: " + e); }
};

print("---- před funkcí:");
try { print("x = " + x); } catch(e) { print("problém: " + e); }
try { print("y = " + y); } catch(e) { print("problém: " + e); }
try { print("z = " + z); } catch(e) { print("problém: " + e); }

test();

print("\n----- po funkci:");
try { print("x = " + x); } catch(e) { print("problém: " + e); }
try { print("y = " + y); } catch(e) { print("problém: " + e); }
try { print("z = " + z); } catch(e) { print("problém: " + e); }

out;

```

Výsledek:

```

"
před funkcí ----
x = 1
problém: ReferenceError: y is not defined
problém: ReferenceError: z is not defined

v bloku -----
x = 1
y = 2
z = 3

ve funkci -----
x = 1
y = 2
problém: ReferenceError: z is not defined

po funkci -----
x = 1
problém: ReferenceError: y is not defined
problém: ReferenceError: z is not defined
"

```

Objekty a pole

Kromě typů hodnot, jako jsou čísla a řetězce a nebo logické hodnoty (true/false), tak existují i objekty, které jsou jakési složky, které mohou obsahovat proměnné a i další objekty. V další ukázce je vidět, že vytváříme proměnnou osoba, do které přiřazujeme něco ve složených závorkách. Jak je za operandem = složená závorka, tak se nejedná o blok příkazů, ale o definici objektu, který obsahuje své členské proměnné (vlastnosti). Ty se v definici oddělují čárkou a místo přiřazování jejich hodnot se namísto operandu = používá :: Vytvoříme objekt osoba s vlastnostmi jmeno, prijmeni a vek, které nastavím své, ale vek nastavím místo na 32 na 23. Hned pod definicí objektu se rozhodnu ale vek opravit. Udělám to zápisem osoba.vek = 32;. Kdybych napsal jen vek = 32, tak se vlastnost objektu neopraví a místo toho se vytvoří nová proměnná vek mimo objekt. Následně zkusíme objekt vypsat, zda má správné hodnoty, ale pouze příkazem print(osoba); se nám vypíše jen „[object Object]“, což nám nic neřekne. Můžeme ale použít funkci **stringify** v předpřipraveném objektu **JSON**, která převede objekt do řetězce (stringu) ve formátu zvaném JSON (JavaScript Object Notation). Po vypsání vidíme toto „{„jmeno“:„Jiří“,„prijmeni“:„Kalina“,„vek“:32}“. Je vidět, že výsledný formát JSON je v podstatě JavaScriptový zápis objektu bez mezer a názvy vlastností jsou v uvozovkách, což mohu napsat i přímo v JS, kdybych chtěl mít například mezeru v názvu vlastnosti. Nastavení vlastnosti (atributu) vek ale proběhlo správně.

V další ukázce je pole (array), které je objektu celkem podobné, ale uživatel se při jeho vytváření obvykle nemusí starat o pojmenování vlastností, protože pole nemá žádné názvy vlastností ale jen číslování. Jednotlivé prvky pole se číslují od 0 do n a nejsou tam mezery. Při definici pole se používají namísto složených závorek závorky hranaté a příklad může být třeba tento „**jmena** = [„Jan“, „Petr“, „Michal“];“. Já v ukázce ale použil pole osoby obsahující objekty osob. Po definici pole ještě zavolám vestavěnou funkci (vlastnost) pole **push**, kterou přidám na konec pole ještě jednu osobu navíc. U druhé osoby (index 1) pomocí hranatých závorek nastavím vek na 35. Vypíšeme funkcí JSON.stringify a výsledek je správný. Kousek níže pole vypíšeme ještě pomocí cyklu for, kde si pomůžeme vlastností pole **length** (velikost pole), která je v této situaci s hodnotou 4. Jednodušší zápis cyklu for speciálně pro pole a objekty je ale tento „**for (i in osoby) {...**“, což je ekvivalent k „**for (i = 0; i < osoby.length; i+= 1) {...**“.

Skript:

```
out = "\n";
print = function(text) {
    out += text + "\n";
}

print("\n-----");
osoba = {
    jmeno: "Jiří",
    prijmeni: "Kalina",
    vek: 23
};

osoba.vek = 32;

print(osoba);
print(JSON.stringify(osoba));

print("\n-----");
osoby = [{
    jmeno: "Jiří",
    prijmeni: "Kalina",
    vek: 32
```

```

}, {
  jmeno: "Martin",
  prijmeni: "Novák",
  vek: 48
}, {
  jmeno: "Petr",
  prijmeni: "Bláha",
  vek: 25
}];

osoby.push({
  jmeno: "Bohouš",
  prijmeni: "Šmídů",
  vek: 41
});

osoby[1].vek = 35;

print(JSON.stringify(osoby));

print("\n-----");
for (i = 0; i < osoby.length; i += 1) {
  print(i + ": " + osoby[i].jmeno + " " + osoby[i].prijmeni + " (" + osoby[i].vek + ")");
}

print("\n-----");
for (i in osoby) {
  print(i + ": " + osoby[i].jmeno + " " + osoby[i].prijmeni + " (" + osoby[i].vek + ")");
}

out;

```

Výsledek:

```

"
-----
[object Object]
{"jmeno":"Jiří","prijmeni":"Kalina","vek":32}

-----
[{"jmeno":"Jiří","prijmeni":"Kalina","vek":32},
{"jmeno":"Martin","prijmeni":"Novák","vek":35},
{"jmeno":"Petr","prijmeni":"Bláha","vek":25},
{"jmeno":"Bohouš","prijmeni":"Šmídů","vek":41}]

-----
0: Jiří Kalina (32)
1: Martin Novák (35)
2: Petr Bláha (25)
3: Bohouš Šmídů (41)

-----
0: Jiří Kalina (32)
1: Martin Novák (35)
2: Petr Bláha (25)
3: Bohouš Šmídů (41)
"

```

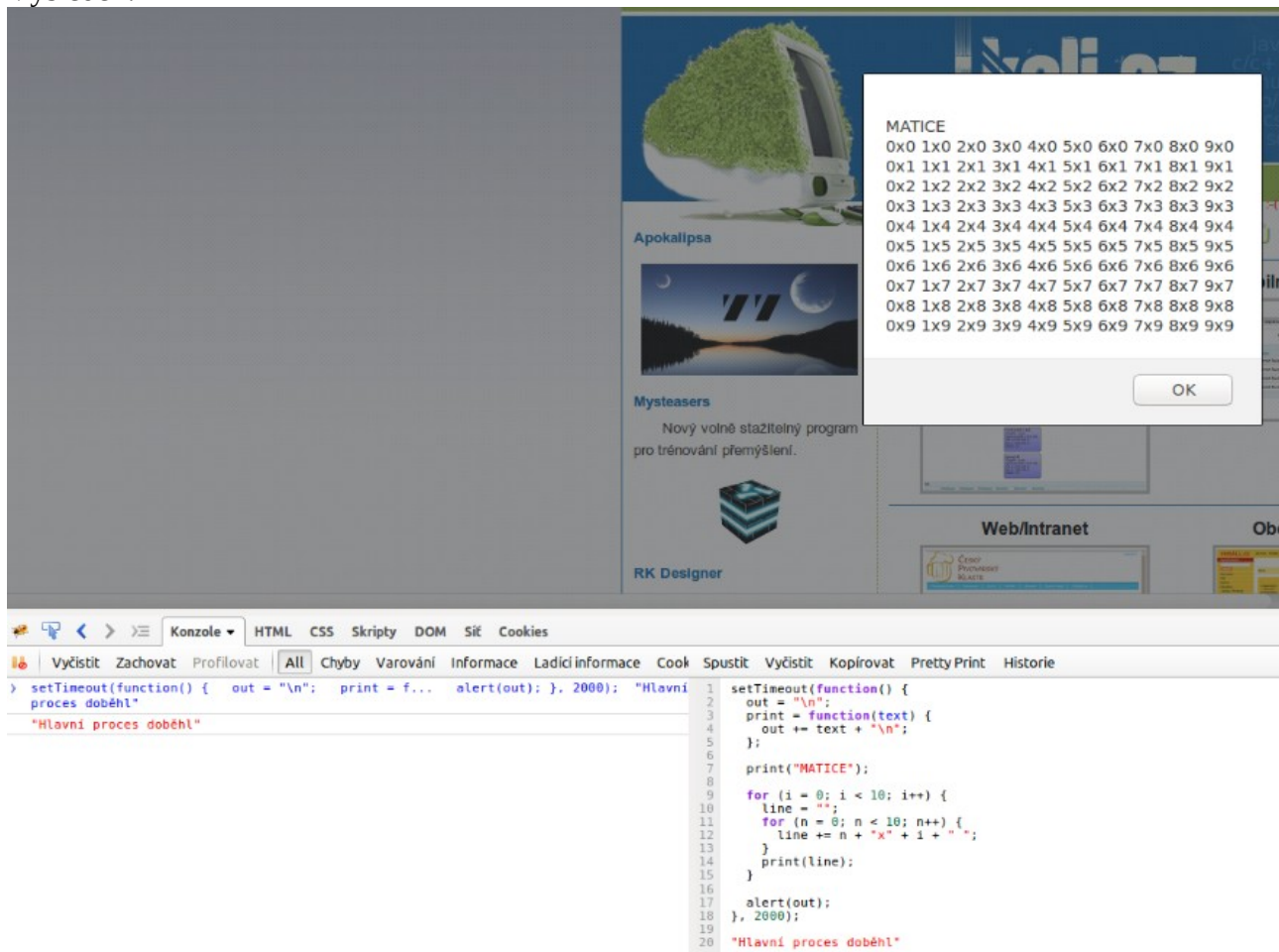
Asynchronní běh (procesy)

Další funkce není v podstatě celkem nic nového, protože v ní vypisují matici jako v kapitole o funkcích, ale vypisují jí v extra procesu. Ten vytvořím tak, že zavolám funkci **setTimeout**, kde první parametr je funkce, které se provede v novém procesu a druhý parametr je 2000, což je údaj pro procesor, který říká, zhruba kolik milisekund má procesor čekat, než se proces spustí, tak že dvě zhruba sekundy. Po zavolání **setTimeout** vypíšeme okamžitě řetězec „**Hlavní proces doběhl**“ a teprve po dvou sekundách se spustí funkce na vygenerování matice, která ale už nemá možnost vypsat řetězec do konzole, protože běží v jiném procesu (vlákně) a tak použijeme funkci prohlížeče **alert**, která zobrazí řetězec v informačním dialogu.

Skript:

```
setTimeout(function() {  
    out = "\\n";  
    print = function(text) {  
        out += text + "\\n";  
    };  
  
    print("MATICE");  
  
    for (i = 0; i < 10; i++) {  
        line = "";  
        for (n = 0; n < 10; n++) {  
            line += n + "x" + i + " ";  
        }  
        print(line);  
    }  
  
    alert(out);  
}, 2000);  
  
"Hlavní proces doběhl"
```

Výsledek:



Rozhraní prohlížeče (API)

Na závěr popíšu nějaké API (Application Programable Interface). Bez API by byl JavaScript jen tichý neviditelný nástroj, ve kterém se dá pracovat s pamětí, ale nikde není žádný výstup vidět. Například zmíněná funkce `alert` je součástí API internetového prohlížeče, která umožní zobrazit informační dialog uživateli, ale prohlížeč má mnohem víc funkcí. Když přijde HTML stránka do prohlížeče, tak prohlížeč vygeneruje strukturu DOM (Document Object Model), ze které se pak složí viditelný obsah stránky. Tento objektový model dokumentu můžeme upravovat pomocí JavaScript API internetového prohlížeče a tím tak měnit okamžitě obsah stránky. V poslední ukázce této kapitoly se získá z DOM objekt, které reprezentuje menu na mých stránkách. Menu je označené identifikátorem s hodnotou „menu“ a pomocí funkce **`getElementById`** v objektu **`document`** se dá tento objekt menu získat. Objekt menu má vlastnost **`innerHTML`**, což je vnitřní HTML kód, ze kterého je menu složeno. Jé této vlastnosti mohu přidělit hodnotu nového HTML, které obsahuje HTML značku **`spam`**, což je nespécifikovaný prvek, kterému nastavím styl (**`style`**) na červenou barvu (**`color: red`**) a velikost fontu 20 pixelů (**`font-size: 20px`**). Vnitřek HTML prvku **`spam`** bude text „Menu bylo nenávratně smazáno“. V dalším kroku se nadefinuje funkce **`animuj`**, která přidá do obsahu menu další text se zamračeným smajlíkem o velikosti fontu 12 pixelů a ten text **` `** (Non Breaking SPace) za smajlíkem oznamuje prohlížeči, že tam má být vidět mezera. Tato funkce se zavolá časovačem se sekundovým zpožděním a uvnitř funkce se znova zavolá přes časovač, tak že každou sekundu se přidá smajlík. Poslední věc je, že v objektu menu nastavíme funkci **`onclick`**, která se provede, když uživatel klikne na menu a vymaže celý vnitřní obsah.

Skript:

```
menu = document.getElementById("menu");  
menu.innerHTML = "<spam style='color: red; font-size: 20px>' +
```

```
"Menu bylo nenávratně smazáno</spam>";
```

```
animuj = function() {  
  menu.innerHTML += "<spam style='color: red; font-size: 12px'>:-(&nbsp;&nbsp;&nbsp;&nbsp;</spam>";  
  setTimeout(animuj, 1000);  
};
```

```
setTimeout(animuj, 1000);
```

```
menu.onclick = function() {  
  menu.innerHTML = "";  
};
```

Výsledek:



Instalace a něco málo o NodeJS

Máme tu ukázky programování v JS, ale není prostředí, kde by se to dalo lépe rozjet s bohatším API. Nejbohatší API má asi NodeJS, ve kterém se dá myslím udělat v JS plnohodnotná aplikace. NodeJS je zdarma a dá se stáhnout [zde](#). Po nainstalování bude potřeba otevřít příkazovou řádku ať už máte Linux, Windows či nějaký jiný operační systém. Do příkazové řádky napsat toto:
`node -e "console.log('Hello World.');"`

„**node**“ je spustitelný soubor aplikace NodeJS a příznakem **-e** dáme najevo, že chceme spustit skript uvedený v uvozovkách v parametru. To co je v uvozovkách je skript, který se vykoná. Je tam vidět volání funkce **log** s řetězcem „**Hello World**“ v objektu **console**. Console je objekt s API pro obsluhu konzole, což je příkazový řádek ve Windows a nebo terminál v Linuxu. Funkce log zobrazí v konzoli text na řádce, tak že je to obdoba funkce print, kterou jsem v ukázkách používal. Pokud příkaz node funguje a vidíme text „**Hello World**“, tak zkusíme teď v pracovním adresáři, kde je příkazový řádek, vytvořit soubor například s názvem test.js a do něj přepíšeme jednu z ukázek v předchozí kapitole. Například tuto s funkcemi:

```
out = "\n";  
print = function(text) {  
  out += text + "\n";  
}
```

```
print("FUNKCE -----");
```

```
for (i = 0; i < 10; i++) {  
  print("line " + i);  
}
```

```
print("\nMATICE -----");
```

```
for (i = 0; i < 10; i++) {
```

```

line = "";
for (n = 0; n < 10; n++) {
    line += n + "x" + i + " ";
}
print(line);
}

print("\nMOCNINA -----");

pow = function(x, m) {
    _x = x;
    for (i = 1; i < m; i++) {
        x *= _x;
    }
    return x;
}

print("6^2 = " + pow(6, 2));
print("3^3 = " + pow(3, 3));
print("2^4 = " + pow(2, 4));

```

out;

Schválně jsem označil červeně části kódu týkající se logování (tisk do konzole), které budou jinak, protože by se nám jinak nic nezobrazilo. Místo prvních čtyř červených řádků bude jen „**print = console.log;**“ a poslední červený řádek tam nebude. Takto upravený kód dát do souboru test.js a spustit v příkazovém řádku jen „node test.js“. Chování bude stejné. U ukázka s časovačem se přepíše kód takto:

```

setTimeout(function() {
    out = "\n";
    print = function(text) {
        out += text + "\n";
    };

    print("MATICE");

    for (i = 0; i < 10; i++) {
        line = "";
        for (n = 0; n < 10; n++) {
            line += n + "x" + i + " ";
        }
        print(line);
    }

    alert(out);
}, 2000);

```

"Hlavní proces doběhl"

Funkce alert je jen pro prohlížeč a je potřeba jí nahradit řádkem „**console.log(out);**“. Obdobně ten červený text nahradit příkazem „**console.log("Hlavní proces doběhl")**“.

Je vidět, že přenositelnost z jednoho API na druhé nemusí být až takový problém, ale také záleží v jaké je to situaci.

Následující kód udělá něco, co by si prohlížeč z bezpečnostních důvodů nemohl dovolit. Pomocí funkce `require("fs")` načteme objekt (knihovnu) `fs` (jako `FileSystem`), která má funkci `readdirSync`, co umožňuje vrátit pole názvů souborů a složek v uvedené složce uvnitř parametru. Složka se jménem `tečka` je tento pracovní adresář, ve kterém se nachází spuštěný skript a jiné věci. Pak už jenom stačí cyklem `for` vypsat pole do konzole. Kdyby prohlížeče umožňovali JavaScriptu takového API, tak skripty na různých stránkách mohou prohledávat souborový systém ve vašem počítači a odesílat informace přes internet na nějaký server. To je určitě nežádoucí.

```
var fs = require("fs");
```

```
var files = fs.readdirSync(".");
for (var i in files) {
  console.log(files[i]);
}
```

Závěr

Toto je v podstatě skoro vše, co člověk potřebuje znát k práci s JavaScriptem ať už je JS napojený na prohlížeč a nebo na jiné programy jako například grafické editory, generátory PDF, servery, telefony a nebo třeba v Raspberry počítači. Myslím si, že se JS celkem rozmáhá a je to podle mě budoucnost. Tento rychlokurz se podle mě neztratí.

JKali framework a NodeJS



Dnes je 1.7.2017. Pěkný datum. Do vývoje frameworku jsem se pustil 9.6.2017. To je nějakých 23 dní minus pár dní volno, ale nějaké promýšlení jsem zahájil pár dnů před tím. Je čas popsat, co už mám hotové. Co se dá udělat za necelý měsíc jednou osobou včetně návrhu, programování, testování a sepsání rychlokurzu na JS. Nebudu sám sebe hodnotit. To může každý sám.

Mluvil jsem tedy o jakémsi nástroji JKali-Rockets (JKR), který by měl urychlit nasazování úprav aplikací na server a celkově urychlit a zpřehlednit vývoj. Za tu dobu, co programuji, tak jsem nashromáždil různé zkušenosti a setkal jsem se s různými „nepohodlnostmi“ při vývoji aplikací, tak že toto je taková má „odpověď“.

Moduly

Při vývoji aplikací je často člověk nucen používat určité druhy souborů. Něco z toho jsou zdrojové kódy programů, ale mohou to být i různé soubory obsahující různá nastavení, lokalizace, mohou to být SQL kódy, které čekají v dávkách pro aktualizaci databáze a podobně. Vývojář je pak nucen sepisovat určité druhy dat do stejných typů souborů, i když by bylo někdy přehlednější mít věci, které spolu souvisí třeba jen v jednom souboru. To nemluvím o tom, že by se leckdy i hodilo sdílet tytéž informace na více platformách. Například prohlížeč, server i nástroj v příkazové řádce.

Tímto problémem se právě zabývá nástroj JKali-Rockets, který je psaný v JS pro NodeJS ale částečně je i na serveru. Princip je takový, že řeknete nástroji umístění souboru s JavaScriptem a nebo celé složky obsahující JS, které má nástroj skenovat. JKR potom nad těmito soubory dovede provádět určité operace podle potřeb uživatele. V případě, že uživatel chce ve složce uvést konkrétní jména složek a nebo JS souborů, které má JKR ignorovat, tak stačí ve složce vytvořit

soubor se jménem „**jkr**“ a obsah souboru bude například řádka „**ignore=script1.js;script2.js;slozka1;slozka2**“. Tím oznámíme nástroji, že chceme ignorovat skenování dvou skriptů a dvou složek včetně jejich podsložek. JKR tedy prohledává JS soubory a hledá v nich právě zmíněné moduly. Je úplně jedno, jak si uživatel moduly rozmístí, zda chce mít obecné lokalizace pohromadě, zda chce mít specifické lokalizace pohromadě s moduly, kde se používají, zda chce mít databázové struktury u služeb a nebo samostatně, či zda chce mít serverovou část pohromadě s klientem a nebo odděleně.

Ted' příklad, jak vypadá obecný zápis modulu.

```
jkr({<typ modulu>:{
  name: „app1.modul1“,
  version: „1.00“

  nodeSide: true,
  serverSide: true,
  clientSide: true,

  <specifické vlastnosti modulu>: ....
});
```

Modul se zavádí funkcí **jkr**, které se v parametru předá objekt obsahující typ modulu s obecnými i specifickými vlastnostmi modulu. V názvu (**name**) ideálně používat nějaké označení aplikace, ke které se modul váže a pak za tečkou název modulu. Verze (**version**) je dobrá pro orientaci jak vývojářům, tak pro JKR, který se rozhodne, zda má modul poslat na server v aktualizaci a nebo ne. Příznaky **nodeSide**, **serverSide** a **clientSide** určují, zda modul má být dostupný pro nodejs, klienta a nebo server. Pokud se u příznaku uvede hodnota false a nebo se příznak neuvede, tak se modul na specifickou platformu nedostane.

Moduly tříd (OOP – Objektově Orientované Programování)

První druh modulů, které popíši, tak jsou určené pro programování, tak že uchovávají funkce, ale i proměnné. Je to jediný druh modulů určený pro programování. Ty ostatní moduly slouží k definici struktur, nastavení a k lokalizaci.

Ukázky programů, které jsem popisoval v rychlokurzu, tak jsou takzvaně jen problémově orientované. Používají se v nich funkce, které řeší určité věci (problémy), ale nejsou části kódu nijak seskupeny do svazků (tříd), které spolu nějakým způsobem souvisí. Toto seskupování kódů do tříd je tzv. objektově orientované programování, které mě osobně připadá přehlednější a bezpečnější, protože se nepletou mezi sebou proměnné a funkce řešící určité věci.

Přes OOP se dobře dají dělat grafické prvky. Vlastně tvůrce OOP Alan Kay je zároveň i vynálezce grafického rozhraní, které od firmy Xerox odkoupila firma Apple a celkem na tom zbohatla. Od Apple se to dostalo k firmě Microsoft a vznikl operační systém Windows.

Ted' ke konkrétní ukázce modulu jednoduché třídy:

```
jkr({class:{
  name: "app1.Test1",
  version: "1.00",

  nodeSide: true
}}, {

  main: function(argv) {
    log("Hello " + argv[2]);
  }
});
```

Třídy mají svůj obsah v samostatném druhém parametru funkce jkr. Nebudou se takto plést funkční věci se specifikací modulu, ale mé to i jiný technický důvod, který zde nebudu popisovat. Tato třída má funkci **main** s parametrem **argv**. Toto je klasická spustitelná metoda u programovacích jazyků jako C/C++, Java a asi i jiné. Tento modul mohu spustit z příkazové řádky následovně. Jen připomínám, že samotný NodeJS bez frameworku JKali, který jsem zatím nevydal, tak to umět nebude.

Příkaz:

`node <cesta k frameworku jkali.js> run <cesta ke skriptu nebo složce se skripty> app1.Test1 Jirka`

Výsledek:

Hello Jirka

Příkaz spustil NodeJS s prvním parametrem určující cestu k frameworku JKali. Další parametr je příkaz **run**, který určí co má framework udělat a má tedy spustit spustitelný modul, u kterého se očekává, že to bude třída obsahující instanční (ne statickou jako u Javy) metodu **main**. Název **modulu** **app1.Test1** je uveden v dalším parametru. Framework tedy prohledá skript nebo složku se skripty a spustí odpovídající modul. Takto snadno řekneme co chceme spustit a to ostatní se neřeší. Poslední parametr není povinný ale dal jsem ho tam jako informaci pro spustitelný modul. Tento parametr je k dispozici společně s ostatními ve funkci **main** v poli **argv** na třetím místě (index 2). První místo je cesta ke skriptu či složce a druhé název modulu, což se může hodit někdy pro použití. Framework pro logování do konzole používá jednoduše funkci **log**.

Ted' ukážu ukázkou na použití OOP v praxi, kde už budou dvě další třídy. Jedna bude **Osoba**, která obsahuje informace jméno, příjmení, ulice, město a psč. Umí naformátovaně vytisknout tyto údaje funkcí **vypis**. Druhá funkce u této třídy **Osoba** je **create**, které se říká konstruktor, protože při vytváření konkrétní instance (objektu) této třídy naplní z parametrů funkce jednotlivé vlastnosti instance. Pak vytvoříme třídu **OsobaRozsirena**, která bude tzv. potomkem třídy **Osoba** (to zajistí vlastnost **extends** ve specifikaci modulu). To znamená, že může využívat/modifikovat vlastnosti svého předka třídy **Osoba**. Objekt třídy **OsobaRozsirena** bude mít navíc vlastnost zaměstnání a bude jinak tisknout formátování funkcí **vypis**. Poslední třída **Test1** je spustitelná a udělá to, že pomocí klíčového slova **new** se v ní vytvoří dvě instance třídy **Osoba** a jedna instance třídy **OsobaRozsirena**. Tyto instance se pak vypíší.

Skript:

```
jkr({class:{
  name: "app1.Osoba",
  version: "1.00",

  nodeSide: true
}}, {

  jmeno: null,
  prijmeni: null,
  mesto: null,
  ulice: null,
  psc: null,

  zahlavi: "----- OSOBA -----",
  odsazeni: "-- ",

  create: function(jmeno, prijmeni, mesto, ulice, psc) {
    this.jmeno = jmeno;
```

```

        this.prijmeni = prijeni;
        this.mesto = mesto;
        this.ulice = ulice;
        this.psc = psc;
    },

    vypis: function() {
        log("");
        log(this.zahlavi);
        log(this.odsazeni + this.jmeno + " " + this.prijmeni);
        log(this.odsazeni + this.ulice);
        log(this.odsazeni + this.psc + " " + this.mesto);
    }
});

jkr({class:{
    name: "app1.OsobaRozsirena",
    extends: "app1.Osoba",
    version: "1.00",

    nodeSide: true
}}, {

    create: function(jmeno, prijeni, mesto, ulice, psc, zamestnani) {
        this.inherited(arguments);

        this.zamestnani = zamestnani;

        this.zahlavi = "***** OSOBA ROZŠ. *****";
        this.odsazeni = "";
    },

    vypis: function() {
        this.inherited();

        log(this.odsazeni + "Zaměstnání: " + this.zamestnani);
    }
});

jkr({class:{
    name: "app1.Test2",
    version: "1.00",

    nodeSide: true
}}, {

    main: function() {
        var osoba1 = new app1.Osoba("Martin", "Novák",
            "České Budějovice", "Uhlářova 21", 37005);
        var osoba2 = new app1.Osoba("Josef", "Prokop",
            "České Budějovice", "U mostu 14", 37027);
        var osoba3 = new app1.OsobaRozsirena("Jan",

```

"Hrdina", "České Budějovice", "U kostela 20", 37011, "prodavač");

```
        osoba1.vypis();
        osoba2.vypis();
        osoba3.vypis();
    }
});
```

Výsledek:

```
----- OSOBA -----
-- Martin Novák
-- Uhlířova 21
-- 37005 České Budějovice
```

```
----- OSOBA -----
-- Josef Prokop
-- U mostu 14
-- 37027 České Budějovice
```

***** OSOBA ROZŠ. *****

Jan Hrdina
U kostela 20
37011 České Budějovice
Zaměstnání: prodavač

Ve třídách se používá klíčové slovo **this**, které označuje, že chceme používat vlastnost právě této instance třídy. Další funkce je **this.inherited(arguments)**, která zavolá stejnojmennou funkci předka. Tím nám odpadne starost s opakovaným psaním stejného kódu pro každou třídu zvlášť. Tomuto kombinování tříd se říká polymorfismus. Mohu mít třeba jednu třídu obsahující hodně vlastností a složité funkce, do kterých nechci zasahovat psaním. Vytvořím si potomka této třídy, ve kterém udělám drobné úpravy a to usnadní spousty práce.

Další funkcionalita jsou statické vlastnosti tříd. V předchozí ukázce se používali jen instanční vlastnosti a to znamená, že se k nim dá přistupovat jen skrze konkrétní instanci (objekt), který je vytvořený konstruktorem. Někdy se ale hodí mít po ruce nějaké vlastnosti v třídách, které mohu používat bez toho, aniž bych vytvářel instanci objektu. Tyto statické vlastnosti jsou ve třídě dostupné jenom na jednom místě na rozdíl od objektů, kterých si mohu udělat kolik potřebuji. Statické vlastnosti se v obsahu třídy uvádí do objektu **statics**.

Skript:

```
jkr({class:{
    name: "app1.Test3",
    version: "1.00",

    nodeSide: true
}}, {

    statics: {
        vlastnost1: null,
        vlastnost2: null,

        funkce1: function() {
```

```

    },

    funkce2: function() {
    }

});

```

Použití statických vlastností by pak vypadalo takto:

```

app1.Test3.vlastnost1 = 12;
app1.Test3.vlastnost2 = „ab“;
app1.Test3.funkce1();
app1.Test3.funkce2();

```

JKali + Enyo



Enyo (EnyoJS) je stejně jako JKali framework, který je napsaný v JavaScriptu. Enyo se ale soustředí uje čistě jen na platformu webového prohlížeče, kdežto JKali už teď během necelého měsíce funguje na platformě NodeJS a také na vestavěném JS engine v Javě 8 (tam se používá myslím Nashorn engine od Mozilly). JKali bude zastřešovat i prohlížeč a to tak, že skrze Enyo. Například v Enyu, které také podporuje OOP, tak se nová třída vytváří funkcí „`enyo.kind({ <obsah třídy> });`“. Tato funkce kind na rozdíl od funkce jkr nepodporuje žádné specifikace verzí a ani informace o tom, jakým platformám je třída určena. Obsah třídy JKali je ale kompatibilní s Enyem. Vlastnosti **name** a **extends** jsou v případě Enya přímo v obsahu třídy stejně tak jako dostupnost sekce **statics**, konstruktor **create**, a i použití funkce **this.inherited(arguments)**. Já sice na platformě NodeJS a pro Javu nepoužívám Enyo pro zavádění tříd, ale snažil jsem se napodobit způsob zavádění tříd Enya. Všechny tučné slova v tomto odstavci jsou vlastně původem z Enya a mě se podařilo naprogramovat ekvivalentní chování. Možná si někdo řekne, proč jsem nepoužil Enyo. Enyo je stavěné hlavně pro prohlížeče. Je sice pro NodeJS extra skript, který dovede zavést používání Enya, ale toto podle mě není moc čisté řešení. Nepřipadá mi moc dobré natahovat spousta kódů určených pro prohlížeč na stranu serveru. Další problém se kterým bych se musel potýkat je rozběhání Enya na Javě, kde není ani ten zaváděcí skript dostupný. Teď ale jak to bude s JKali na straně prohlížeče? Mé řešení je následující. Funkce jkr zjistí, že se nachází na platformě prohlížeče a namísto toho, aby vytvořila třídu frameworkem JKali, tak použije funkci Enya enyo.kind. To znamená, že pouze vlastnost name a extends přeneše z objektu v prvním argumentu do objektu druhého argumentu a nad tím zavolá enyo.kind. Problém je vyřešen. Můžeme tedy na straně prohlížeče využívat všechny vlastnosti tříd Enya a přitom se k tomu dodají možnosti určit verzi třídy a i platformu, pro kterou je určená.

Jak to bude se skenováním? Mluvil jsem o tom, že JKali dovede skenovat soubory skriptů a celé složky, když hledá moduly. Enyo je ale v prohlížeči omezené a JavaScriptu není umožněno prohledávat seznamy souborů a složek. V Enyu se prohledávání řeší pomocí funkce `depend(<cesta k souboru či složce>, <cesta k souboru či složce>, ...)`. V každé složce se skripty v Enyu bývá zpravidla soubor package.js, ve kterém není nic jiného, než volání této funkce, ve které jsou uvedené názvy souborů či složek, které se mají načíst. Pokud tam název není, tak se nenačte. To je trochu nepohodlnost, protože zpravidla potřebujeme pokaždé načíst vše až na nějakou výjimku občas. JKali naopak čte automaticky vše a pokud výjimečně potřebujeme udělit výjimku, tak se uvede do souboru .jkali ve vlastnosti ignore. Na straně prohlížeče ale ani JKali nebude umět skenovat, tak že extra pro prohlížeče se bude používat funkce Enya depend v souborech package.js. Tato funkce depend bude ale v případě skenování na jiných platformách ignorována.

Jinak Enyo vyžaduje při definici nové třídy, aby případný předek byl už založen. JKali toto nevyžaduje a umí si s problémem poradit. Posloupnost načítání skriptů je u Enya stanovená tím, jak

jsou po sobě zadané názvy ve funkci `depend`. U JKali je posloupnost prohledávání skriptů ve složkách naprogramovaná tak, že prvně se ve složce prvně načtou všechny skripty a pak se teprve přechází na další pod složku, ve které se také prvně začíná načtením skriptů a pak teprve složek. U tříd je to frameworku JKali jedno, protože si umí poradit, ale například při načítání lokalizací už pořadí může hrát roli v případě, že načteme modlu s lokalizacemi, které mají totožné některé klíče a tím vlastně přepíší jejich texty. Lokalizace popíši v dalším článku.

Moduly popsané v následujících kapitolách ale už jsou jen čistě můj výtvor a s Enyem nemají nic společného, protože se z většiny týkají prostředí serveru.

Modul lokalizace a výjimky

Programátor se někdy setkává se situací, kdy potřebuje, aby program generoval zprávy uživateli v jazyce, který si nastaví. K tomu se používají lokalizace. Ve skriptu programátor namísto zpráv používá jen unikátní identifikaci (klíče) zpráv, podle kterých může systém dohledat překlad do konkrétního jazyka. Jednotlivé překlady zpráv se píší do modulů `localization`, kterých pro jeden jazyk může být více a nebo jen jeden podle potřeby programátora. Ve výsledku nástroj JKR bude umět spojit všechny moduly lokalizace stejného jazyka do jednoho. Toto spojování zatím nemám stejně tak propojení frameworku s Enyem, protože toto se týká hlavně strany prohlížeče, na kterou jsem se zatím nedostal. Samotné lokalizace mi už fungují.

Následující ukázka obsahuje modul s lokalizací pro anglický jazyk (vlastnost **lang** = „en“). Dále obsahuje vlastnost **labels**, která je specifická pro modul lokalizací. Je to objekt obsahující hierarchickou strukturu klíčů a jejich zpráv pro konkrétní jazyk. První sekce by měla být název aplikace (**app1**), ve které mohou být přímo uvedeny všechny zprávy, ale mohou si udělat i podsekce, jako například v ukázce je objekt **exceptions**, který obsahuje zprávy problémů. Ve zprávách se občas vyskytuje symbol `%s`, který slouží jako označení místa, kam se vloží nějaká specifická hodnota. V samotné spustitelné třídě **Test1** se používá funkce **getLabel**, která nám vrátí výslednou zprávu aktuálně vybraného jazyka pro klíč uvedený v prvním parametru a se specifickými hodnotami v poli uvnitř druhého parametru, které se vkládají tak, jak jdou po sobě. Při spuštění aplikace je ve výchozím stavu nastaven anglický jazyk a podle toho se prohledávají lokalizace jen v angličtině. V ukázce je dále vidět, jak se dá poslat výjimka určená pro lokalizaci její informace. Pošle se funkcí **jk.error**, kde první parametr je klíč lokalizace a druhý případné pole specifických hodnot. Při odchycení mohou přeložit výjimku pomocí funkce **ex2s** (jako **ExceptionTwoString**). Unikátní identifikace výjimky je právě název její lokalizace a později ukážu, jak podle ní identifikovat problém. Další možnost odeslání výjimky je bez lokalizace a to tak, že se přímo v ní uvede text, který se má zobrazit bez jakéhokoliv překladu. Tato výjimka se dá poslat metodou **jk.errorMessage**. V druhé ukázce odchycení výjimky je použita funkce **ex2s** pro převod výjimky na text s druhým parametrem na `true`. Toto znamená, že chceme vypsát i takzvaný **stacktrace** výjimky. Programátor potom může dohledat v jaké části kódu výjimka vznikla a jak postupně putovala a ukončovala funkci po funkci až do místa jejího odchycení. Je to šikovná věc při hledání chyb. V druhé fázi programu přepneme jazyk na češtinu funkcí **setLanguage(„cs“)** a provedeme v podstatě to samé ještě jednou, ale výstup bude česky, jak je vidět ve výsledku. Lokalizaci pro češtinu jsem napsal schválně pod modul s třídou **Test4**, aby bylo vidět, že je jedno, kam lokalizace umístím. Mohou opravdu být v jednom souboru společně s čímkoliv jiným.

Skript:

```
jkr({localization: {  
    name: "app1.loc",  
    version: "1.00",  
    lang: "en",  
  
    nodeSide: true,  
    labels: {
```

```

    app1: {
        msg1: "Message1 attribute1 = '%s', attribute2 = '%s'",
        msg2: "Message2 attribute1 = '%s', attribute2 = '%s'",

        exceptions: {
            ex1: "Problem1 attribute1 = '%s'"
        }
    }
});

jkr({class:{
    name: "app1.Test4",
    version: "1.00",

    nodeSide: true
}}, {

    main: function() {
        log("-- EN --");
        log(getLabel("app1.msg1", ["value1", "value2"]));
        log(getLabel("app1.msg2", ["value1", "value2"]));
        try {
            throw jk.error("app1.exceptions.ex1", ["value1"]);
        } catch (e) {
            log(ex2s(e));
        }
        try {
            throw jk.errorMessage("Problem2 ...");
        } catch (e) {
            log(ex2s(e, true));
        }

        setLanguage("cs");
        log("-- CS --");
        log(getLabel("app1.msg1", ["hodnota1", "hodnota2"]));
        log(getLabel("app1.msg2", ["hodnota1", "hodnota2"]));
        try {
            throw jk.error("app1.exceptions.ex1", ["hodnota1"]);
        } catch (e) {
            log(ex2s(e));
        }
        try {
            throw jk.errorMessage("Problém2 ...");
        } catch (e) {
            log(ex2s(e, true));
        }
    }
});

jkr({localization: {
    name: "app1.loc",

```

```

    version: "1.00",
    lang: "cs",

    nodeSide: true,
    labels: {
        app1: {
            msg1: "Zpráva1 vlastnost1 = '%s', vlastnost2 = '%s'",
            msg2: "Zpráva2 vlastnost1 = '%s', vlastnost2 = '%s'",

            exceptions: {
                ex1: "Problém1 vlastnost1 = '%s'",
                ex2: "Problém2 vlastnost1 = '%s'"
            }
        }
    }
});

```

Výsledek:

```

-- EN --
Message1 attribute1 = 'value1', attribute2 = 'value2'
Message2 attribute1 = 'value1', attribute2 = 'value2'
Problem1 attribute1 = 'value1'

```

Problem2 ...

Error

```

    at error (/home/kali/JSoft/JavaScript/jkali/jkali.js:1031:70)
    at errorMessage (/home/kali/JSoft/JavaScript/jkali/jkali.js:1035:15)
    at main (/home/kali/JSoft/JavaScript/examples/example2/test.js:34:13)
    at runJKali (/home/kali/JSoft/JavaScript/jkali/jkali.js:1120:25)
    at Object.<anonymous> (/home/kali/JSoft/JavaScript/jkali/jkali.js:1140:2)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)

```

-- CS --

```

Zpráva1 vlastnost1 = 'hodnota1', vlastnost2 = 'hodnota2'
Zpráva2 vlastnost1 = 'hodnota1', vlastnost2 = 'hodnota2'
Problém1 vlastnost1 = 'hodnota1'

```

Problém2 ...

Error

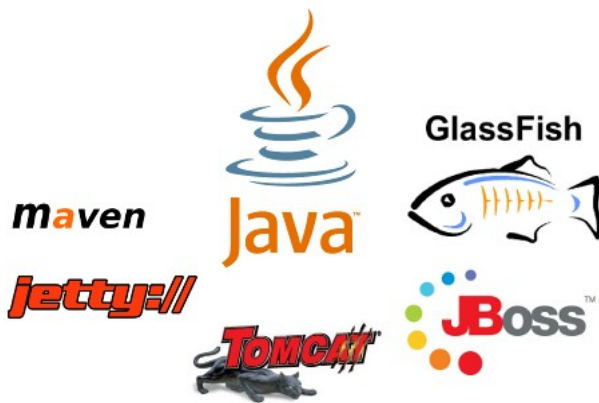
```

    at error (/home/kali/JSoft/JavaScript/jkali/jkali.js:1031:70)
    at errorMessage (/home/kali/JSoft/JavaScript/jkali/jkali.js:1035:15)
    at main (/home/kali/JSoft/JavaScript/examples/example2/test.js:49:13)
    at runJKali (/home/kali/JSoft/JavaScript/jkali/jkali.js:1120:25)
    at Object.<anonymous> (/home/kali/JSoft/JavaScript/jkali/jkali.js:1140:2)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)

```

at Function.Module._load (module.js:438:3)

JKali framework a Java webový server



Konečně se dostávám k zajímavější části frameworku a tou je komunikace se serverem a práce s databázemi. Dnes je pondělí 3.7.2017 a tato kapitola může zároveň být takový status. Od včerejška je u nás na návštěvě bratranec. Bude tu pár dní, tak si dám trochu volno před tím, než se pustím do podkapitol.

Úvod do problematiky JavaEE (Java Enterprise Edition) a hledání řešení pro JavaScript

Musel jsem se na začátku tedy zabývat otázkou, na jaké platformě budou běžet JavaScripty na straně serveru. Jedna možnost by byla samotné NodeJS, ve kterém jsem už něco ukázel. Java (konkrétně edice EE (Enterprise Edition)) má oproti NodeJS celkem dobře našlápnuto co se týče různých serverových knihoven. Ano v NodeJS si mohu opravdu naprogramovat HTTP server a v něm skripty spouštět, ale jsou pro NodeJS dostupné léty odzkoušené a vyvíjené konektory pro různé databáze? Je k dispozici prověřená knihovna pro práci s Connection Pools, které umí udržovat aktivní spojení s databází a poskytují je uživatelské vrstvě podle potřeby, čímž se nemusí stále navazovat nové a nové spojení? Jsou pro NodeJS dostupné nějaké funkcionality pro práci s http relací či cookies a co třeba taková problematika nahrávání souborů přes http protokol, na kterou je potřeba extra knihovna a nebo vyšší verze implementace J2EE. Já mám obavy, že dělat server čistě na NodeJS by obnášelo zbytečnou práci s řešením všelijakých problémů, které už jsou prošlápnuté na vhodnější platformě pro tuto oblast. NodeJS na serveru používat samostatně tedy raději nebudu.

Nabízí se další řešení, které mě napadlo asi jako první. Databáze PostgreSQL sama o sobě umožňuje jazyk PLV8, což je ovládání databáze v JavaScriptu. Máme tu zase podobný problém s všelijakými udělatky, které bych v PLV8 asi špatně dělal a nebo špatně napojoval na Javu. No a co kdyby za mnou přišla nějaká firma a řekla mi, že se jim líbí můj framework, ale jsou věrní zákazníci firmy Oracle, kdežto framework funguje jen na PostgreSQL. Tím pádem ztrácím potenciální zakázku já a nebo obecně ten, kdo framework používá. Kdyby ostatní databáze podporovali obdobou PLV8, tak to budu extra programovat rozhraní pro každou databázi zvlášť, aby se framework dal jednotně používat? To také není dobré řešení.

Co když mě přestane například vyhovovat databáze PostgreSQL a budu chtít přejít na Oracle a nebo MySQL či něco jiného? To budu muset přepisovat všechny SQL dávky do jiné databáze? Kde pak. Jenom specifikuji jiný typ databáze a všechny dávky se vygenerují pro ní.

Jaké mám tedy udělané stávající řešení strany serveru, které už mi funguje?

Mám napsanou JavaEE aplikaci, která využívá správce balíčků **Maven**. Maven usnadní vývojáři připojování Java knihoven k projektu. Vývojář nemusí nikde nic stahovat ale pouze v projektu nspecifikuje jakou knihovnu potřebuje s jakou verzí a při překladu aplikace do spustitelného stavu si Maven sám sežene potřebné knihovny pro projekt. Pomocí nástroje Maven sestavím tedy spustitelnou webovou aplikaci, kterou mohu nahrát na libovolný aplikační server pro platformu JavaEE. Mohu použít servery **Jetty**, **Tomcat**, **JBoss**, **GlassFish** a jiné edice. Server Jetty nemusím dokonce ani instalovat a ani ho nijak extra spouštět. Jetty je přídatný modul pro Maven a tak mohu

webovou aplikaci spustit přímo pomocí Mavenu na serveru Jetty, ale toto je vhodné jen pro mě jakožto vývojáře webové aplikace pro JavaEE. Běžný uživatel frameworku JKali nebude nijak moci zasahovat do zdrojových kódů Java části frameworku.

Jak to tedy vlastně funguje?

Popíši na příkladu. Vývojář si chce udělat webovou aplikaci v JKali. Nainstaluje si tedy Javu a nějaký Java aplikační server. Vezme webovou aplikaci napsanou v Javě, která je součástí frameworku a tu pouze nainstaluje na aplikační server pod názvem kontextu, který mu vyhovuje. Pokud jí tam dá například pod kontextem „**Obchod1**“, tak bude aplikace dostupná na adrese „[http://<host serveru>\[:<port web vrstvy>\]/Obchod1](http://<host serveru>[:<port web vrstvy>]/Obchod1)“. Takovýchto aplikací si může nahrát vývojář na server kolik chce s různými názvy kontextu. **Kontext** je tedy jakási vyhraněná oblast paměti na serveru, kde budeme mít nahrané jednotlivé JavaScriptové aplikace podle potřeby a by se v rámci jednoho kontextu budou moci navzájem využívat ale do jiného kontextu už nebudou moci zasahovat.

Jak se dostane aplikace napsaná v JS na server do konkrétního kontextu? K tomuto účelu slouží takzvaná **prostředí**. Vývojář si tedy už připravil kontext, ale zatím v něm nejsou nahrané žádné aplikace v JS. Vývojář bude mít někde na disku složku obsahující všelijaké projekty napsané v JS. V této složce bude, mimo jednotlivých složek s projekty, jeden důležitý JS soubor obsahující specifikaci prostředí. Mohu mít například jednu aplikaci napsanou v JS, ale rozběhám si jí na několika prostředích, které budou třeba i databázově oddělená. To se využívá, když potřebujete například jedno prostředí pro vývoj, kde se mohou vyskytovat častěji chyby, druhé prostředí pro testování, kde testeři testují a chtějí, aby se tam nedostávali čerstvé chyby způsobené při vývoji. Další prostředí může být třeba provozní, kde je také ta samá aplikace, ale už opravdu musí projít testem, než se do tohoto prostředí aplikace dostane a je tak dostupná běžným uživatelům. V specifikaci prostředí si tedy vývojář nadefinuje jednotlivá prostředí, kde určí, zda chce, aby se aplikace nahrávala na server vzdáleně (například nahrání na testovací prostředí) a nebo aby si server rovnou prohledával místní soubory projektu (což je pohodlnější pro vývoj). Dále se v prostředí specifikuje mimo projektu, o který se jedná, tak také kontext do kterého se aplikace nahraje, ale je tam i uvedený jednak databázový uživatel který přistupuje k datům a dokonce i takzvaná mapa schémat, ale ke schématům se dostanu až budu databází.

Vývojář jediné co tedy udělá, tak je to, že spustí v příkazové řádce framework přes NodeJS, kde uvede, že chce provést operaci „**update**“, čímž se aktualizuje vybrané prostředí. Uvede pouze cestu k souboru obsahující specifikaci prostředí a pak už jen uvede název prostředí, které chce aktualizovat. To je celé a vše ostatní je už automatizované. **JKali-Rockets** se připojí na server s tím, že chce aktualizovat konkrétní prostředí a dotáže se na stavy. Webová aplikace odpoví, že není nainstalovaný systém JKali a tak nemůže nic udělat a to ani vrátit stavy o prostředí, protože skripty generující stavy nejsou nahrané. Po příchodu této odpovědi ze serveru systém pošle sám sebe na server v dalším kroku a na serveru se jednak spustí, ale také připraví svůj vlastní import do databáze, aby se podruhé už nemusel znova odesílat. Po spuštění systému už vrátí stavy prostředí, na které nástroj na klientské straně odpoví tím, že připraví import do databáze změn samotného prostředí, které se odešlou na server, nahrají do databáze a JavaScript engine a serveru se restartne, aby vše načetl.

Importy samotného systému a i změn prostředí se generují v SQL pro konkrétní vybranou databázi (PostgreSQL, MySQL nebo Oracle). Vše mi funguje. V rámci importu se nenahrávají jenom skripty, ale už funguje i vytváření či modifikace SQL tabulek a import jejich dat.

Takovýto je momentálně stav.

Mám programátorsky teď volno, ale napsal jsem nějaké zamyšlení do dokumentu na projekt Eden <http://jkali.cz/data/eden.pdf>.

Status 10.7.2017

Dnes je 10.7.2017. Minulý týden jsem si dal přestávku od programování až na včerejšek, kdy jsem si rozběhal a seznámil jsem se s aktuální verzí Enya 2.7.0. Začal jsem se tedy zabývat částí programování prohlížeče, což je poslední část frameworku, aniž bych dokončil podrobnější popis toho, co mi už funguje na straně serveru. Přehodnotil jsem priority a myslím si, že marketingový génius Bill Gates by mě asi nepochválil za můj dosavadní postup, kdy pracně vymyšlený návrh produktu prezentuji už v době jeho vzniku. Sice neotevírám zdrojové kódy, ale prezentuji koncept používání, kterého se může někdo chytit a já budu sice pracovat, ale nic nevydělám. Naopak to chce vydělat co nejvíce peněz, které budou použity na podporu projektu Eden. Je potřeba hlavně se zaměřit na to, co vidí koncoví uživatelé a to budou především běžní lidé využívající projekt Eden, skrze který bude framework vlastně prezentovaný. Pak by teoreticky mohl vzrůst zájem o weby nové generace, které bude možné programovat v JKali frameworku a podobných, pokud vzniknou. Až teprve potom budou vývojářské společnosti poptávat dodavatele těchto frameworků, jako jsem například já a jejich technologové se budou rozhodovat, jakému dodavateli dát přednost na základě „vnitřností“, které jsem chtěl prezentovat dále, ale prozatím nebudu.

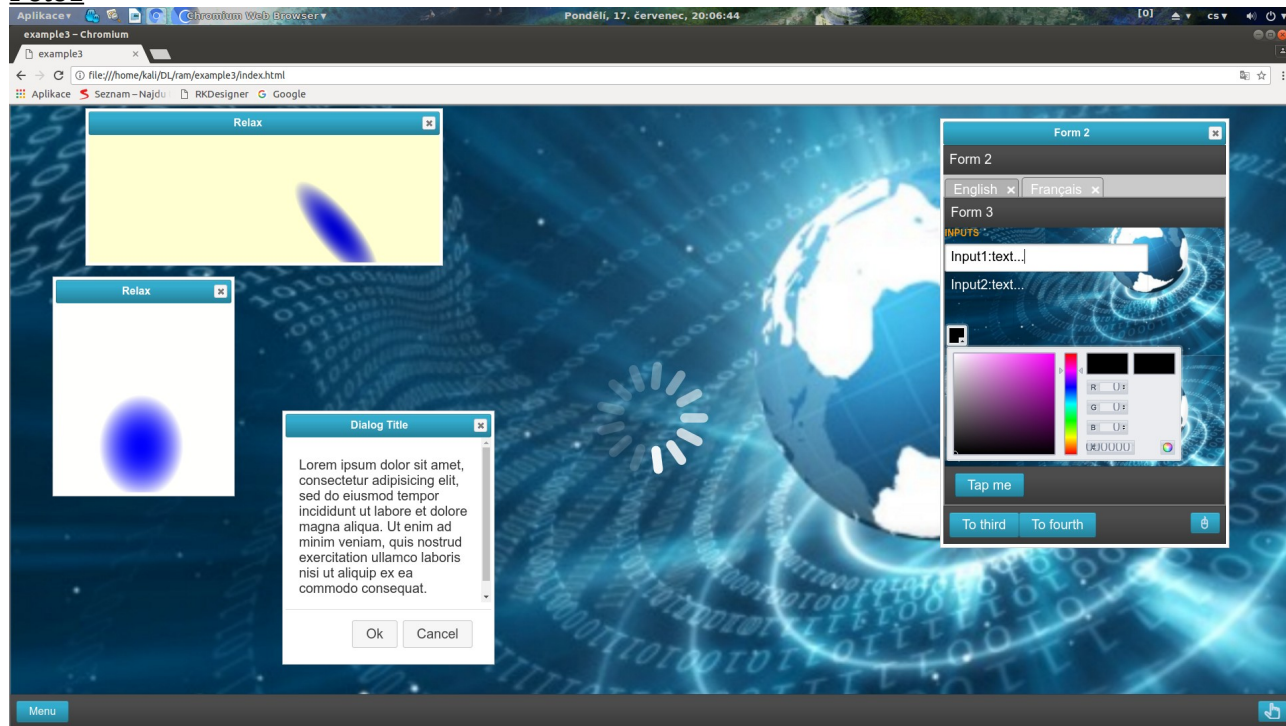
Začal jsem se tedy zabývat uživatelským rozhraním, což je část, která by neměla být podceňována v případě, že člověk chce dosáhnout nějakého úspěchu. Lidi mnohdy nezajímá to, co se děje uvnitř, ale jak věci vypadají. Tak jsme sám zvědavý, co budu mít hotové za týden.

Status 17.7.2017

Je přesně 17.7.2017 20:00 a já se pouštím do psaní. Pěkný datum. Od minulého týdne jsem udělal docela velký pokrok na straně vývoje uživatelského rozhraní. Myslel jsem, že už dnes budu moci poslat odkaz na něco funkčního k prohlédnutí, ale zatím to k tomu není. Není to ve stavu, kdy by se to mohlo už zkoušet uživateli. Každopádně během pár dní snad něco konkrétního k ozkoušení už snad bude.

Přikládám alespoň nějaké fotky z testování na prohlížeči chrome.

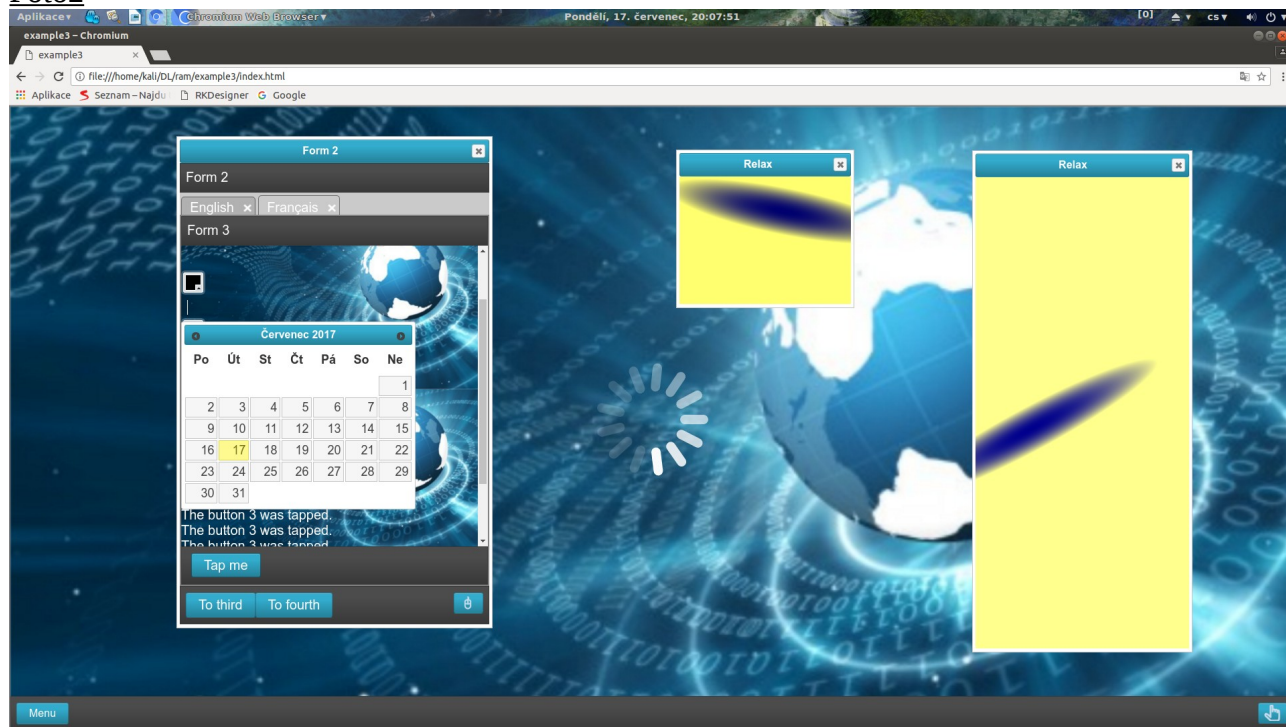
Foto1



Na první fotce je vidět systém při práci v režimu pro stolní počítače. Vypadá to trochu jako operační systém. V levém spodním panelu je nabídka, ve které se dají otevírat okna (formuláře). Jsou

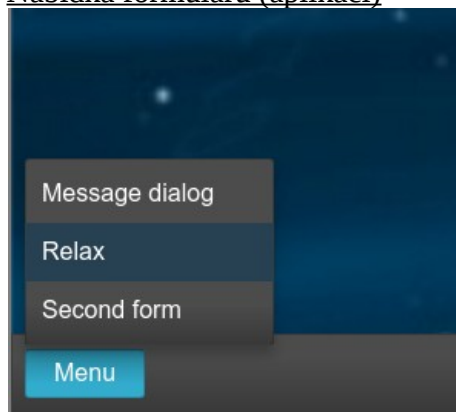
otevřené dvě okna s relaxační animací, pak je tam dole dialog s textem a vpravo je okno, ve kterém jsou vidět nějaké ovládací prvky. Je tam výběr barvy, textová políčka a zkušební tlačítka. Veprostřed na ploše se točí jen tak zkušebně spinner, který se bude používat normálně jako indikátor, že se něco děje. Vpravo dole pod plochou je modré tlačítko s rukou. Tím se celé zobrazení přepne do režimu vhodném pro dotyková zařízení, tak že tam nebude komfort oken, ale bude možné obsluhovat stejné aplikace, pouze přizpůsobené pro dotyková zařízení. To co se zobrazuje v okně vpravo je vlastně ten režim pro dotyková zařízení, ale zatím není hotový. Proto je v okně vpravo dole vidět modré tlačítko s myší pro přepnutí na stolní počítače, ale je samozřejmě nesmysl tento režim zobrazovat v okně. To jsou jen experimenty co vše je možné v JKali frameworku.

Foto2



Na další fotce je nový jen výběr data.

Nabídka formulářů (aplikací)



Takto vypadá nabídka.

Během pár dní snad bude k dispozici první zkušební verze.

.....

Je 21.7.2017 8:20. Stále pracuji na mechanismu přepínání oken jak v režimu pro počítače, tak pro telefony a ladím různé chyby. Čekal jsem, že dám něco k prezentaci dříve, ale věci se někdy zkomplikují. S tím už se nedá nic dělat. Každopádně na to, jak jsem si nasadil laťku vysoko, tak postupuji řekl bych i tak celkem rychle a navíc to funguje stejně na prohlížečích Chrome, FireFox, Internet Explorer, Opera, Safari a i na tom mém zastaralém telefonu SonyEricsson Xperia Pro, kde mám ještě Android 2.4 (tam zatím zlobí jen výběr barvy prstem). Někdy je asi lepší mít na testování nějaké starší zařízení, aby byla motivace optimalizovat výkon. Prezentace snad bude už při pondělním statusu, ale jisté to není. Raději věnovat více času k dosažení dokonalosti, než se spokojit s průměrem.

Status 24.7.2017

Dnes 24.7.2017 sem nebudu nic komentovat. Jen vkládám odkaz na první demoverzi aplikace ve frameworku Jkali, kde nějaké komentáře už jsou.

<http://jkali.cz/examples/1/>

Status 1.8.2017

Včerejší status (pondělí 31.7.2017) jsme vynechal. Výjimka z pravidla. Dnes je prvního srpna 2017 a detailnější informace jsou přímo v ukázce.

<http://jkali.cz/jkali-framework-examples/example2.html>

Status 9.8.2017

Opět mám trochu zpoždění a píši status po osmy dnech (o den navíc). Už ale mám první verzi projektu, která komunikuje se serverem a je v ní naprogramované fórum, které už neslouží jen jako ukázka, ale je řekl bych celkem dobře použitelné. Jenom to naplnit nějakými daty. Tomu se budu věnovat v následujících dnech.

Tato třetí funkční ukázka začíná být první verzí projektu Eden. Více informací v aplikaci.

Máme tu pěknou shodu číslic. Jedná se o třetí funkční ukázku a je den devátého. Já bydlím V. Volfa 39 v třetím patře v bytě číslo 9.

<http://jkali.cz/jkali-framework-examples/eden.html>

Status 14.8.2017

Mám další vydání aplikace Eden verze 1.01. Nový je pouze proklik na fórum z přehledu všech zpráv, překlad úvodní stránky do angličtiny a nějaké drobnosti. Odkaz je stále stejný. Neplánuji už zachovávat starší verze.

U verze 1.00 jsem si všiml zajímavé věci. Aplikace (soubor eden.html) měla 770 KB. Neudělal jsem to schválně, že bych tam přidal nějaké mezery nebo se snažil psát kód, aby to vyšlo (to by bylo dost obtížné). Ted' je 14.8.2017. Když k dnešnímu datu přičtu 7 dnů, tak co mi vyjde? 21.8.2017. Toto datum už jsem přeci zmiňoval v prvním díle příběhu na stránce 44. Rozebíral jsem tam jedno video, na kterém vypravěč říkal, že nastane „77dmá Saros série 145“ (nějaký hvězdný úkaz) a asi začne něco, co snad nemělo doposud obdoby. Takových předpovědí ale už asi bylo více. Zmiňoval jsem tam také, že jsem měl v porodnici náramek s číslem 77. Také zajímavá shoda náhod :).

Ted' si asi dám týden pauzu. Já vlastně už nemohu pokračovat až jak se uvidí, zda o to někdo stojí. Framework nemám v plánu uvolňovat. Považuji tak trochu za zázrak, že jsem to dokázal vymyslet, naprogramovat, otestovat a odladit sám za pouhé dva měsíce. Byla to tvrdá práce, ale bez nadpřirozených duchovních sil byl neměl asi šanci. Navíc ted' podle té velikosti první verze 770 KB

to vypadá, že to bylo už dávno vymyšlené a propočítané včetně velikosti okolních knihoven, které jsem použil. Mojžíš měl obyčejný klacek a dělal s ním divy jenom pro to, že mu kryl záda Bůh. Asi tu hůl také nedával jen tak z ruky.

<http://jkali.cz/jkali-framework-examples/eden.html>

Instalace a něco o webovém serveru Tomcat

<bude doplněno později>

Moduly připojení na server a specifikace prostředí

<bude doplněno později>

Modul serverového rozhraní

<bude doplněno později>

Příprava databází PostgreSQL, MySQL, Oracle a něco o nich

<bude doplněno později>

Modul databázové dávky

<bude doplněno později>

Modul databázové tabulky

<bude doplněno později>

Modul databázových operací

<bude doplněno později>

Testovací modul

<bude doplněno později>