Představení technologie JKali Framework

autor: Jiří Kalina
web: https://jkali.cz

Obsah

Úvod do projektu

JKali Framework (JKF) je programátorský nástroj pro usnadnění a zefektivnění tvorby informačních systémů. Z praxe programátora vím, že v dnešní době existuje spousta nástrojů, které se zabývají řešením problematik v různých oblastech. Programátoři tyto nástroje používají na projektech dohromady. Nástroje mají ale různé tvůrce, kteří se při vývoji soustředili hlavně na to, jak řešit danou problematiku, ale už ne tak intenzivně na synergii s nástroji od ostatních tvůrců. Za léta ve vývoji informačních systémů v jazyce Java a JavaScript jsem v týmech zpozoroval určité problémy, na kterých firmy neefektivně tráví spousty času tím, že nástroje nejsou v synergii. Netýká se to jen samotného programování, ale hodně i testování jako takového a i nedostatku používání automatizace při testování. JKF je nástroj, který nesměřuje pouze k usnadnění práce programátorům, ale k propojování celého týmu včetně analytiků.

Architektura běžných projektů

Problém 1 nároky na hardware: Jeden z problémů, na který jsem v praxi narazil byl například, že server při každém dotazu uživatele generuje pomocí komponentových struktur znova a znova HTML stránky a pamatuje si pro každého uživatele zvlášť nastavení daných komponent. To vede k problémům, že takovéto technologie nemůžete používat pro více uživatelů a nebo pokud to chcete udělat, stojí to zbytečné náklady na provoz dalších serverů určených jen pro komunikaci s uživateli (GUI).

Problém 2 náročná orientace v kódu: Používají se i systémy, kde se s komponenty nepracuje a programátor si skládá podle svého stylu kódy, jak uzná za vhodné. To jsou případy, kdy se třeba firma fixuje na určité zaměstnance, protože jen oni se v dané oblasti vyznají. Je potom finančně náročné zasvěcovat do projektu nové lidi, protože se musí s projektem seznámit. Takový programátor je zahrnutý zbytečně kódy, které by měl mít na starosti nějaký framework, co je přehledně uspořádá na vhodná místa. Ideálně mít tak přehledný kód, že do něj bude umět v případě potřeby nahlédnout třeba i analytik, který se normálně orientuje spíše na UML.

Problém 3 dva jazyky pracující se stejnou oblastí: Čím dál více se používá JavaScript pro programování klientské části na prohlížeči. JavaScript bývá výhradně a straně prohlížeče a ne na serveru, kde se používají běžně jiné jazyky. Uživatelské rozhraní oblastně ale zasahuje jak na stranu prohlížeče, tak na server. Například taková validace vstupních dat. Můžete jí kontrolovat už na straně prohlížeče (nezatěžujete výkonnostně server), ale to nestačí a je potřeba kontrola i na straně serveru v případě, že si někdo bude chtít poslat třeba programově něco nevalidního. V takovém případě bývají uživatelé tlačeni do zbytečné práce na dvou kódech a to souvisí i s plýtváním času v rámci udržování takto duplicitních oblastí.

Problém 4 nevyřešená práce s prostředími: Ve své praxi jsem se hodně potýkal s nedostatečně efektivní práci s prostředími. Při vývoji používáte různá prostředí jako třeba vývojové, testovací, provozní a podobně. Nástroje na správu aplikací jako třeba konzole aplikačních serverů pro nasazování aplikací na serveru a nebo různé pomocné skriptovací jazyky ať už Windows/Linux dávkové soubory, nebo multiplatformní Ant skripty, či balíčkovací systémy jako Maven a podobně. Tyto nástroje mají rozsah působnosti jen pro správu své oblasti, ale neznám nastroj, který by definoval něco jako správu prostředí, kde bude celý tým jasně vidět na jednom místě strukturovaně nastavení prostředí a bude je kdokoliv z týmu snadno umět použít bez nutnosti obtížného předávání know-how.

Architektura projektů psaných v JKali Framework

JKali Framework zastřešuje vývoj kompletně od webové části (GUI) dále serverovou část včetně práce se SQL databází, datovým modelem až po testovací nástroj a systémové ovládání pomocí JKali nabídek.

Webová část je generovaný JavaScript klient, který sestavuje všechna HTML pomocí přehledných objektově orientovaných komponent přímo na straně prohlížeče. Server není výkonnostně těmito procesy zatěžován, tak že je to efektivnější pro hardware. Jediné, co si prohlížeč jednorázově načte je jedna HTML stránka obsahující balík JavaScriptu a CSS stylů. Potom už probíhá jen dotahování čistých dat v JSON formátu k propagaci klientem.

Co se týče společné GUI oblasti pro prohlížeč i server, tak JKali Framework pro tuto oblast používá i na serveru jazyk JavaScript. Funguje to tak, že webová aplikace psaná v Javě má uvnitř sebe knihovnu, která funguje něco jako zavaděč JavaScriptových modulů pro serverovou část. Pro představu na serveru běží nějaký JavaEE aplikační server a do něj mohou vývojáři vzdáleně nasazovat/aktualizovat Java aplikace. Obdobně je to i pro JavaScript moduly na jednotlivých Java aplikací. Aplikace má v sobě tuto zaváděcí knihovnu a ta se chová jako takový malý aplikační server, ale spravuje JavaScript moduly. Uživatel může za běhu aplikačního serveru aktualizovat JavaScript moduly na vybrané aplikaci bez toho aniž by musel být zastavený celý aplikační server. To může urychlit nasazování. Tím, že prohlížeč i server sdílí pro GUI oblast stejný jazyk, nevznikají tak chaotické duplicity například v rámci validací a podobně. Ten samý validátor se dá pustit jak na serveru tak i na prohlížeči. Prohlížeč validuje data pro odlehčení zátěže serveru a server pak ve finále validuje jen kvůli bezpečnosti tím stejným validátorem.

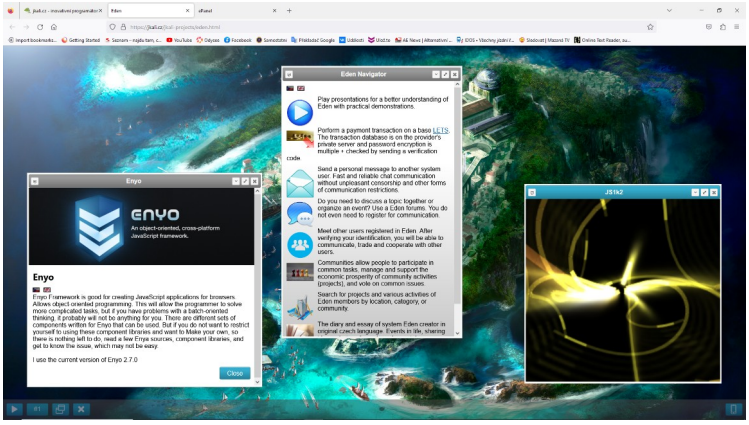
JKali Framework má vlastní systém ovládacích nabídek určených pro sestavování aplikace, nasazování modulů na server, spouštění testovacích scénářů a to vše s možností vybírat si pro jaké prostředí chcete akce provádět. Správa prostředí je jak jsem psal součástí frameworku.

Nový pohled na automatizované testování projektů. V praxi jsem se hodně potýkal s tím, že hodně času zabralo simulování různých problémů testery a vůbec opakované retesty po případných opravách brali celkem čas testerům. Tester si musel naklikat situaci v aplikaci či případně dělat ruční zásahy v databázi na testovacím prostředí a to stále spoustu času.

JKali Framework je stavěný na jiný přístup k testování. Více automatizovaný. Test jedné mé aplikace jsem prováděl tak, že jsem si v nabídce vybral testovací prostředí, která ale nemělo nikde žádnou existující testovací databázi. Ta databáze se vytvořila v okamžiku spuštění testu a to v RAM disku kvůli lepší rychlosti. Předgeneroval se počet uživatel a nějaká základní data a pak začalo provolávání GUI služeb předpřipravené testovací sekvenci. Provolávání služeb tak, jak by to dělali uživatelé. Tím se testují automatizovaně všechny možné situace. Čím je sekvence delší, tím je kvalitnější test. Poslání testera by nemělo být opakovaně manuálně něco naklikávat programátorům, ale napsat kvalitní testovací sekvenci a v případě problémů sdělit programátorovi v jakém místě sekvence nastává nějaký problémový případ. Programátor si test spustí u sebe se zarážkou v popisovaném místě a tam už má odsimulovanou problémovou situaci. Není potřeba žádné zbytečné opakované úsilí při testech.

Ukázka uživatelského rozhraní (GUI)

Mám na svém webu JKali dostupných pár ukázek aplikací v JKF. Server, kde mají běžet služby momentálně není zapnutý, ale to nijak nebrání uživatelům klienty používat, nastavovat si na nich vzhled prostředí a otevírat v nabídce okna, která fungují bez služeb jako například v sekci Others.

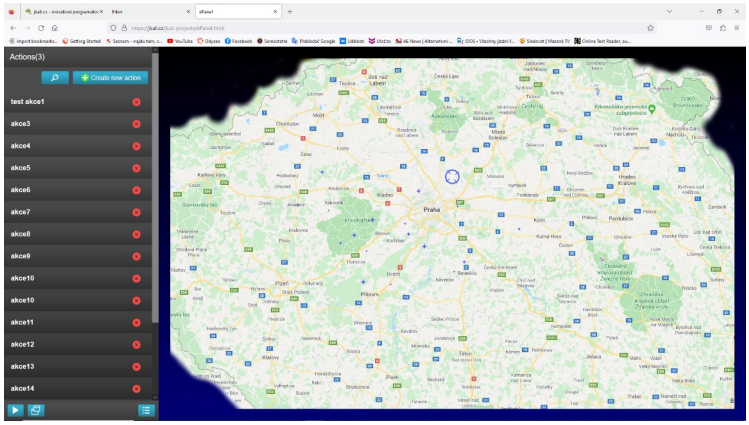


Ukázka LETS systému Eden je dostupná na tomto odkaze:

https://jkali.cz/jkali-projects/eden.html

V okně Eden Navigator je hned první tlačítko přechod na prezentace, kde jsou vidět ukázky používání systému.

Například zde je video, jak se rozhraní používá:
https://jkali.cz/data/video/eden/3_prezentace_prostredi.mp4



Na ukázce systému Eden je vidět rozvržení připomínající operační systém Windows, nebo Linux s možností přepnout se na mobilní zobrazení. Není nutné mít pouze takovéto rozvržení oken a programátor si může vytvořit vlastní třídu specifikující rozvržení. Další ukázka je s hlavním a postranním panelem. V hlavním panelu je vidět mapa a v postranním se přepínají okna. Ukázka přidružené pomocné aplikace ePanel k projektu Eden.

Ukázka je zde:

https://jkali.cz/jkali-projects/ePanel.html

V hlavní nabídce je vidět, že i v druhém rozvržení je možnost nastavení vzhledu. Přechody mezi okny jsou animované, což je jen takové zpestření.

Strukturování projektových souborů

Samotný JKF je převážně psaný v jazyce JavaScript a to samé platí i pro aplikace v něm vytvářené. JKF potřebuje na straně serveru mít jen zavaděč, který spustí JavaScript engine a umožní vývojáři nasadit vzdáleně JKR moduly (popsáno dále). To ale neznamená, že vývojář je vždy nucen psát vše jen v JavaScriptu. V mém případě mám zavaděč napsaný zatím jen pro Javu a Java je tedy na serveru hostitelský jazyk. Mohu si v Java aplikaci na serveru napsat libovolné funkce a ty potom volat z nasazených JKR modulů v JavaScriptu. Prvotně jsem uvažoval napsat zavaděč v jazyce PHP, protože má dostupnější hosting než Java a myslím, že by stálo za to to časem udělat, ale PHP byť údajně podporuje JS engine zvaný V8, tak se mi to tehdy v roce 2017 nepodařilo rozběhat. Jako Java programátor mám s tímto jazykem i více zkušeností a na něm se mi to rozběhat podařilo byť

server jsem si fyzicky musel zřídit vlastní.

JKF tedy nenutí uživatele používat pouze JavaScript, ale je to majoritní jazyk pro práci s projektovými moduly které bych přiblížil skoro k jakési budoucí UML odnoži, se kterou by se mohli naučit dělat celé týmy a lépe se tak orientovat v projektech.

Takto vypadá souborová struktura JKF projektů u mě. Není to nic pevně daného a každý si tu strukturu může udělat libovolně podle svého jak uzná za vhodné. V tomto JKF uživatele neomezuje. Modře jsou složky, zeleně soubory a popisky jsou šedivě:

projects
  eden – projekt Eden
    css – kaskádové styly
    database – obsahuje moduly formující datový model a databázové skripty
      tables – ve složce tables jsou moduly rozděleny do souborů podle názvů tabulek
        categories.js – soubor obsahuje modul table a localization. Moduly table slouží k vytvoření či případně modifikaci tabulek. Moduly localization jsou lokalizace jednotlivých sloupečků. Detailněji se tím budu zabývat dále. JKF takto umožní mít v tom samém souboru model tabulky a i zobrazitelné pojmenování sloupců. Hezky přehledně na jednom místě.
        …
      batchs.js – soubor obsahuje moduly batch. Ty popisují jednotlivé dávky databázových změn v závislosti na verzování aplikace.
    ePanel – přidružená pomocná aplikace ePanel k projektu Edenu. Klient s mapkou akcí.
      src – zdrojové kódy aplikace ePanel
        forms – obsahuje souboru s moduly form, tedy jednotlivá okna. Jsou jen 3 a zbytek ve shared.
          ActionForm.js – detail akce
          ActionsForm.js – seznam akcí
          MapForm.js – okno s mapou (je vidět napevno v pravém hlavím panelu)
        App.js – soubor obsahuje modul class. Hlavní třída eden.EPanelApplication klientské aplikace ePanel (tedy GUI). Týká se čistě strany prohlížeče.
    src - zdrojové kódy aplikace Eden
      components – obsahuje soubory s GUI komponenty použité na formulářích. Tyto komponenty jsou postavené v základě na EnyoJS frameworku a detailně se jimi zde nezabývám.
        …
      enums – obsahuje soubory s moduly enum, tedy výčty.
        QuestionEnum.js – obsahuje výčet doplňujících otázek při registraci uživatele + lokalizace.
        …
      forms - obsahuje soubory s moduly form, tedy jednotlivá okna projektu Eden + serverová část.
        category – okna týkající se kategorií + testy
          CategoriesForm.js – přehled kategorií
          CategoriesTest.js – tvorba testovací sekvence související s kategoriemi
          CategoryForm.js – detail kategorie
        user – okna týkající se uživatel + testy
          …
        voting – okna týkající se hlasování + testy
          …
        …
      services – obsahuje soubory s moduly class. Jedná se jen o statické funkce, které si sdílejí některé formuláře. Toto se týká jen strany serveru. Když píši formuláře, myslím tím formuláře i jejich s protějškem na straně serveru tedy tzv. rozhraní (popíši dále).

AuthService.js – obsahuje třídu s autorizačními funkcemi.
…
tasks – obsahuje soubory s moduly class a příznakem task. Tasky mají nastavenou časovou spoušť a provádějí automatizovaně po nastavené periodě nějaké úlohy. Něco jako je v PHP CRON a nebo v Javě knihovna Quartz a podobně.
VotingsTask.js – automatické zpracováni hlasování.
…
App.js – obdobně jako u ePanelu obsahuje hlavní zaváděcí třídu pro GUI projektu Eden + příslušné lokalizace. Lokalizace se u JKF projeků se dají přehledně psát v místech, kterých se týkají.
AppTest.js – obsahuje moduly test utvářející testovací sekvenci. Více dále.
.jkali – řídící soubor pro JKali Framework. Při skenování JKR modulů může určit například, jaké adresáře či JS soubory se mají v rámci projektu přeskočit. Složka ePanel se neskenuje při sestavení projektu Eden. Tento přidružený projekt má vlastní oddělené projektové spouštění.
ePanelProject.js – obsahuje modul project nesoucí informace o přidruženém projektu ePanel
project.js - obsahuje modul project nesoucí informace o projektu Eden
example1 – projekt s ukázkami 1
…
example2 – projekt s ukázkami 2
…
shared – zde jsou soubory s moduly sdílené mezi více projekty (například dokumenty a relax)
…
enviroments.js – soubor obsahující moduly connections (konektivita) a enviromants (prostředí)
menu.js – soubor obsahující moduly nabídek pro práci s projekty (sestavení, nasazování, testy, ...)

JKali Rockets (JKR) moduly jsou JSON struktury, které nesou různé typy zdrojových dat. Mohou to být třídy, tabulky, lokalizace, specifikace vstupů a jiné. Každý JKR modul má indikátory, kterými dáte najevo frameworku kam se má JKR modul nasadit. Zda je to pouze na klientskou část (prohlížeč), nebo serverovou, či případně pro JKali nabídky v příkazové řádce (nodejs). Mohou to být i všechny tři typy současně. V případě prohlížeče se moduly zabalují do výstupního HTML souboru s klientskou aplikací. V případě, že jsou určeny pro server, tak se posílají na server a zaváděč je ukládá do SQL databáze.

V dalších kapitolách popíši jednotlivé typy JKR modulů.

Soubor popisu projektu Eden (project.js)

JKR modul project nese informace o projektu jako je název, popis, verze, hlavní klientská třída, cesty pro skenování JKR modulů a jiná umístění. Soubor project.js je níže modře. Vidíme volání funkce jkr, která má v parametru JSON strukturu JKR modulu project.

```
jkr({project: {
        name: "eden",
        title: "Project Eden",
        version: "1.07.10",
        clientAppClass: "eden.Application",
        scaning: ["$project/../shared/forms",
                "$project/../shared/tables"],
        styles: ["$project/../shared/css",
                "$project/css"],
```

```
        assets: [{
                src: "$project/../shared/images",
                dest: "shared/images"
        }]
}});
```

Modul batch obsahující databázovou dávku (batchs.js)

V souboru batchs.js jsou moduly DB dávek. Ve struktuře souborů výše jsem poukázal na tabulku kategorií, tak ukážu dávku jen pro tuto tabulku. Níže je vidět, že dávka se vztahuje k verzi 1.07.07. Není o první DB dávka a ani poslední. Takovýchto dávek je v souboru batchs.js celá řada a každá z nich má sekvenci operací, které se mají danou dávkou vykonat. Jsou vidět dvě operace typu table. To znamená, že se vytváří a nebo modifikuje nějaká tabulka. Název tabulky je uveden níže a pod ním i její verze. Každá tabulka se v rámci JKF verzuje číslováním od 1. Vidíme, že se pracuje s tabulkami kategorií a projektů a obojí je verze 1, tedy teprve se tabulky zakládají.

Je i vidět, že před názvy tabulek je předpona „{eden}.”. Tato předpona dává JKF rozměr možností, se kterými jsem se v minulosti u nástrojů nesetkal. Tato předpona určuje název schématu a je proměnlivá podle toho, pro jaké prostředí projekt sestavujete. Můžete si nakonfigurovat prostředí takové, že třeba tablespaces budete mít jen v dočasném ram souborovém systému a použijete jednu databázi, kde budete mít ten samý projekt vícekrát ale izolovaně pro různá prostředí na různých schématech. Toto využívám při testování, že si vytvořím DB model na ram souborovém systému a tam provádím rychlé testy na čerstvě vygenerovaných datech. JKF při nastartování projektu projede všechny dávky a sestaví tak datový model odpovídající aktuální verzi projektu. Prostředími se ještě budu zabývat později.

…

```
jkr({batch:{
        version: "1.07.07",
        sequence: [{
                type: "table",
                name: "{eden}.categories",
                version: 1
        }, {
                type: "table",
                name: "{eden}.projects",
                version: 1
        }, {
                …
        }]
}});
```

…

Modul table a tabulka kategorií (categories.js)

V modulu table vidíme informace o tabulce jako název a verze. Je tam i prostředím definovaná proměnná obsahující název tablespace. Níže je pak struktura create, která nese informace o prvotním založení tabulky. Tedy jednotlivé sloupce. Pod sekcí create je sekce addFKs na vytvoření

cizích klíčů. Pokud by byla tabulka později modifikovaná, tak v tomto souboru bude více modulů table pro další úpravy sloupců. Modul table vytváří informace o datovém modelu, kterých se využívá i na jiném místě JKF a to ještě popíši. Strukturu datového modelu lze vygenerovat do JSON souboru pro přehled. JKF tyto moduly umí přeložit do SQL pro konkrétní dialekty. Já mám rozběhaný dialekt pro PostgreSQL, ale dialekty pro MySQL a Oracle nejsou dokončeny. Jejich dokončení, či případně tvorba dalších dialektů by závisela na tom, zda o to někdo bude mít zájem a zda by podpořil vývoj tohoto systému. Tyto table moduly se trochu myšlenkou podobají nástroji Liquibase, ale jsou součástí jednoho komplexnějšího frameworku.

Vespod vidíte i lokalizace jednotlivých sloupců pro češtinu a angličtinu. Je to vše pohromadě v jednom souboru. Tyto lokalizace mají návaznost na formující se datový model. To co je v lokalizacích v objektu labels pod objektem $, tak to je návaznost na datový model. Vývojář není nucen mít všechny lokalizace hromadně nepřehledně v jednom souboru, ale má je tam, co se píše daná tabulka. Pro případ, že by někdo chtěl lokalizace předat nějakému překladateli v jednom souboru jsem zvažoval nějaký export do jednoho celého JSON souboru, ale to udělané nemám. Pak by to museli od překladatele vývojáři převzít a nějak zpětně seskládat na jednotlivá místa. V praxi nevím, jak často se řeší něco jako hromadný překlad. Spíše si to vývojáři stejně překládají nějak samostatně a postupně.

Modul tabulek se týká jen serveru a tam se nijak indikátory neurčuje, kam má modul jít, ale u lokalizací je nodeSide a clientSide nastaveno. Tedy tyto názvy jsou k dispozici na klientu i v příkazové řádce. Lokalizace pro případ serveru se nastavuje v situacích, kdy třeba potřebujete odeslat ze serveru email s nějakou pevnou šablonou.

```
jkr({table:{
        name: "{eden}.categories",
        version: 1,
        tablespace: "{eden}",
        create: [{
                name: "id",
                type: "long",
                notNull: true,
                pk: true,
                autoIncrement: true
        }, {
                name: "name",
                type: "varchar",
                length: 40,
                notNull: true
        }, {
                name: "description",
                type: "text",
                length: 8000
        }, {
                name: "created",
                type: "timestamp"
        }, {
                name: "parent",
                type: "long",
                index: true
```

```
        }],
        addFKs: [{
                table: "{eden}.categories",
                columns: [{fk: "parent", pk: "id"}]
        }]
}});

jkr({localization: {
        name: "eden.CategoriesCZ",
        lang: "cz",

        clientSide: true,
        nodeSide: true,
        labels: {
                $: {
                        eden: {
                                categories: {
                                        id: "Id",
                                        name: "Název",
                                        description: "Popis",
                                        parent: "Kategorie"
                                }
                        }
                }
        }
}});

jkr({localization: {
        name: "eden.CategoriesEN",
        lang: "en",

        clientSide: true,
        nodeSide: true,
        labels: {
                $: {
                        eden: {
                                categories: {
                                        id: "Id",
                                        name: "Name",
                                        description: "Description",
                                        parent: "Category"
                                }
                        }
                }
        }
}});
```

Modul výčtu (QuestionEnum.js)

Zde je příklad použití funkce jk.enum pro vytváření výčtů. Funkce zavede zkráceným zápisem více modulů najednou a to jak výčet, tak českou a anglickou lokalizaci. Jedná se o výčet doplňujících otázek při registraci uživatele. V objektu values jsou jednotlivé položky, které je možno obohatit napojením na nějaká JSON data, ale to jsem zde nepoužil a jsou tam jen hodnoty null.

```
jk.enum({
	namespace: "eden",
	enum: "Question"
}, {statics:{

	values: {
		BOOK: null,
		FILM: null,
		ACTOR: null,
		PET: null,
		MOTHER: null
	}
}}, {

	cz: {
		BOOK: "Oblíbená kniha",
		FILM: "Oblíbený film",
		ACTOR: "Oblíbený herec",
		PET: "Jméno prvního mazlíčka",
		MOTHER: "Jméno matky za svobodna"
	},
	en: {
		BOOK: "Favorite book",
		FILM: "Favorite film",
		ACTOR: "Favorite actor",
		PET: "First pet's name",
		MOTHER: "Mother's name for free"
	}
});
```

Modul hlavní klientské třídy ePanelu (App.js)

V tomto modulu třídy popíši jen hlavičku modulu a sekci menu. Hlavní třída je jen jedna v celém klientovi a tak práce s ní není nic, co by se dalo brát jako rutinní činnost, co pomůže týmu jako celku chápat projekt. Práce s touto třídou bude hlavně pro programátora až na menu, které vnímám celkem přehledné, jednoduché a mohlo by zajímat i jiné lidi z týmu.

V hlavičce máme jen název třídy, dále že je potomkem třídy jkali.Shell. Předpona @@ je direktiva pro JKF a znamená, že ve skenovaných cestách má najít modul jkali.Shell. Bylo by hezčí to tam nemít, ale to se mi technicky nepovedlo. Vlastnost requires je pole s moduly, které je potřeba k tomuto modulu připojit. Zde není vyžadována direktiva @@. Připojují se 3 třídy a 4 lokalizace. Vlastnost clientSide byla už popsána a určuje, že třída jde na klienta. Sestavování klientské aplikace do HTML souboru funguje tak, že se JKF podívá na tuto hlavní třídu a podle direktiv @@ připojuje

další klientské moduly obsahující také tyto direktivy. Je to rekurze, která se týká ale jen sestavování klienta a ne serveru. Do klienta jde tedy jen to, co je použit v hlavní třídě.

V sekci menu jsou specifikované jednotlivé položky hlavní nabídky klienta. Vidíme vlastnost title, kde pomocí speciálního výrazu $l{...} můžeme vložit klíč lokalizace. Vlastnost action je funkce, která se vykoná při stisknutí. Otevírají se jednoduše okna. Níže je vidět i snadné vytvoření podmenu vlastností submenu. Dále podmínění viditelnosti tlačítek vlastností visibility, kde se opět vyhodnocuje výraz napojený na datové struktury. V poslední položce je vidět volání rozhraní na straně serveru funkcí callInterface, kde se volá služba logout pro rozhraní okna eden.LoginForm. Rozhraním se v JKF rozumí protějšek na straně serveru. Něco jako webová služba, co přijímá a odesílá JSON data klientovi. Mám ve zvyku pro okno (formulář) dělat stejně pojmenované rozhraní na serveru, ale není to nutnost. Je to pro přehlednost.

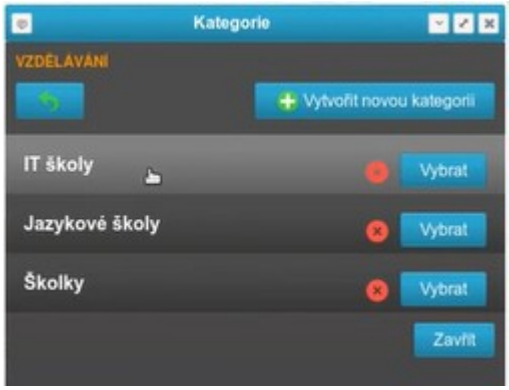
```
jkr({class:{
	name: "eden.EPanelApplication",
	extends: "@@jkali.Shell",
	requires: ["jkali.JKaliWeb", "jkali.Utils", "jkali.Localization", "jkali.CZ", "jkali.EN",
		"eden.AppCZ", "eden.AppEN"],

	clientSide: true
}}, {

	menu: [{
		title: "$l{eden.ActionsForm.title}",
		action: function() {
			this.app.openForm("@@eden.ActionsForm");
		}
	}, {
		title: "$l{eden.App.communication}",
		submenu: [{
			title: "$l{eden.App.themes}",
			action: function() {
				this.app.openForm("@@eden.ThemesForm");
			}
		}]
	}, {
		title: "Relax",
		submenu: [{
			title: "Ellipse",
			action: function() {
				this.app.openForm("@@jkali.EllipseForm");
			}
		}, {
			title: "Texture3D",
			action: function() {
				this.app.openForm("@@jkali.Texture3DForm");
			}
		}, {
			title: "JS1k",
			action: function() {
```

```
                        this.app.openForm("@@jkali.JS1kForm");
                    }
                }, {
                    title: "JS1k2",
                    action: function() {
                        this.app.openForm("@@jkali.JS1k2Form");
                    }
                }]
            }, {
                title: "$l{eden.App.login}",
                visibility: "${!jk.user}",
                classes: "green-item",
                action: function() {
                    this.app.openForm("@@eden.LoginForm", {
                        forcedPrompt: true
                    });
                }
            }, {
                title: "${jk.user.firstname} ${jk.user.surname}",
                visibility: "${jk.user}",
                classes: "green-item",
                action: function() {
                    this.app.openForm("@@eden.UserForm", {
                        data: {id: jk.user.userId}
                    });
                }
            }, {
                title: "$l{eden.App.logout}",
                visibility: "${jk.user}",
                classes: "red-item",
                action: function() {
                    var _this = this;
                    this.app.callInterface("eden.LoginForm", {
                        action: "logout",
                        data: {},
                        project: this.app.config.project,
                        version: this.app.config.version,
                        mode: this.app.mode
                    }, function() {
                        jk.user = null;
                        var a = _this.app; // po obnoce menu se ztrati reference na app z _this,
tak proto toto
                        a.refreshMenu();
                        a.refreshWindowsContents();
                    }, null, true);
                }
            }],
```

(dále už jen různé funkce a nastavování) …

```
});
```

Komplexní formulář přehledu kategorii (CategoriesForm.js) – moduly form, inputs a interface



Níže je ukázka kompletního souboru s tzv. komplexním formulářem, který uživateli zobrazuje stromový výběr kategorií. Uživatel může kategorii vybrat tlačítkem Vybrat, či případně přejít na přehled podkategorií kliknutím na její název. Formulář pak obnoví seznam a zobrazuje podkategorie. Tlačítkem se zelenou šipkou se vracíme zase o úroveň výše.
To co by v jiných nástrojích bylo nepřehledně rozstrkané v několika souborech, tak v JKF máme v jednom souboru. Serverovou i klientskou část.

První sekce komplexního formuláře je samotný formulář, co definuje GUI. Druhá sekce je struktura vstupních parametrů pro aktivity rozhraní určené pro formulář detailu kategorie. Formulář detailu kategorie je v jiném souboru, co neobsahuje už rozhraní ani definici vstupů. Nemusel bych to takto mít, ale rozjel jsem tento styl, že komplexní formuláře přehledu obsahuje věci týkající se detailu.

Třetí sekce je modul rozhraní, který vytváří služby na straně serveru pro tento formulář a celkově vlastně služby pro kategorie. Poslední sekce je související lokalizace.

Komentáře v kódu jsou šedivě.

```
jk.complexForm({
	namespace: "eden",
	form: "Categories",
	requires: ["eden.CategoriesCZ", "eden.CategoriesEN"] // Lokalizace k db tabulce kategorií.
}, {

	statics: { // Statické třídní vlastnosti formuláře s rozměry a vzhledem.
		width: 450,
		height: 400,
		theme: "dark"
	},

	menu: [{ // Nabídka okna s tlačítkem pro založení kategorie. Funkce createCategory je níže.
		title: "$l{eden.CategoriesForm.create}",
		action: function() {
			this.owner.form.createCategory();
		}
	}],

	// Přehled specifických vlastností formuláře. Není nutné pro funkci, aby to tu bylo.
	selector: null, // GUI komponenta výběru kategorie, která formulář otevřela.
	selectCallback: null, , // Callback o výběru kategorie pro otevírací komponentu.
	filter: null, // Objekt filtru seznamu, který se přenáší na databázi.
```

```
view: { // Tato struktura definuje uspořádání GUI komponent formuláře.
	components: [{
		// První komponenta nemá uvedený kind, jde tedy jen o prostý HTML DIV.
		classes: "jkali-content-panel", // CSS třída jkali-content-panel.
		components: [{
			classes: "jkali-section-title", // CSS třída jkali-section-title.
			content: "$s{this.owner.data.category.name}" // Cesta k textu.
		}]
	}, {
		// Druhá komponenta má uveden kind a je to komponenta s tlačítky.
		kind: "@@jkali.ButtonPanel",
		leftComponents: [{ // Definice tlačítek na levé straně panelu.
			ontap: "back", // Funkce back se zavolá při stisku tlačítka.
			visibility: "$s{this.owner.data.category}", // Tlačítko je viditelné jen pokud byla dohledána rodičovská kategorie.
			components: [{
				tag: "img", // Komponenta je prostý HTML img.
				style: "margin-right: 2px;", // CSS styl.
				attributes: { // HTML vlastnosti.
					src: "shared/images/back.png",
					width: "20px"
				}
			}]
		}],
		components: [{
			ontap: "createCategory", // Stisknutelný HTML DIV.
			components: [{
				tag: "img",
				style: "margin-right: 2px;",
				attributes: {
					src: "shared/images/add.png",
					width: "20px"
				}
			}, {
				// Text je z lokalizace tohoto formuláře.
				content: "${getForm(this).getLabel('create')}"
			}]
		}]
	}, {
		kind: "@@jkali.Loop", // Komponenta jkali.Loop generuje seznam z dat.
		data: "$s{this.owner.data.categories}", // Cesta k datům.
		classes: "selectable-list",
		components: [{ // Toto je specifikace komponent jedné položky seznamu.
			classes: "selectable-item jkali-gradient",
			ontap: "select", // Stiskem HTML DIVu voláme funkci select.
			components: [{
				classes: "item-buttons",
				components: [{
					tag: "img",
					style: "margin-right: 10px; vertical-align: middle;",
```

```
                                visibility: "${{hasRight({type: 'edit', table: '{eden}.categories'})}}", // Viditelnost tlačítka pro editaci jen pokud máme právo upravovat tabulku.
                                ontap: "edit",
                                attributes: {
                                    src: "shared/images/edit.png",
                                    title: "$l{jkali.common.edit}",
                                    width: "24px"
                                }
                            }, {
                                // Ikonka na mazání je vidět vždy.
                                tag: "img",
                                style: "margin-right: 10px; vertical-align: middle;",
                                ontap: "remove",
                                attributes: {
                                    src: "shared/images/close.png",
                                    title: "$l{jkali.common.remove}",
                                    width: "24px"
                                }
                            }, {
                                kind: "@@jkali.Button", // Komponenta tlačítka.
                                content: "$l{jkali.common.open}",
                                ontap: "open"
                            }]
                        }, {
                            // Název položky.
                            classes: "item-title",
                            content: "${this.owner.item.name}"
                        }, {
                            // Popis položky.
                            classes: "item-details", //
                            // Ve výrazech můžeme volat libovolné funkce.
                            content: "${jkali.Utils.htmlify(this.owner.item.description)}",
                            allowHtml: true // Povolíme vykreslení HTML prvků pro DIV.
                        }]
                    }]
                }, {
                    // Teď nevím, proč není tato komponenta zachycená na fotce výše, ale jedná se o univerzální stránkovací posuvník viditelný ve videoukázkách Edenu na přehledech.
                    kind: "@@jkali.ListControls",
                    value: "${this.owner.filter}",
                    classes: "jkali-content-panel",
                    onChange: "load" // Funkce volaná při stránkování.
                }, {
                    kind: "@@jkali.ButtonPanel",
                    components: [{
                        content: "$l{jkali.common.close}",
                        ontap: "close"
                    }]
                }]
```

```
	},

	// Teď následují jednotlivé GUI funkce formuláře.

	// Funkce load načte/obnoví ze serveru seznam kategorií a detail rodičovské kategorie
	load: function(form, p) {
		if (this.isInitLoad(p)) { // Jde o prvotní načtení formuláře?
			this.filter = initFilter(); // Připravíme strukturu obecného filtru
			var category = null;
			if (this.selector) { // Zkusíme z výběrové komponenty získat data kategorie.
				category = this.selector.getValue();
			}
			// Do filter.categoryId zkusíme nastavit id rodičovské kategorie.
			this.filter.categoryId = category ? category.parent : null;
		}

		var _this = this;
		// Teď zavoláme akci load z příslušného rozhraní na serveru. Název rozhraní je stejný jako jméno komponenty formuláře (this.kindName).
		this.callAction(this.kindName, "load", {
			filter: prepareFilter(this.filter) // Korekce a nastavení filtru do requestu
		}, function(res) { // Zpětné volání s odezvou ze serveru
			_this.data = res; // Celý response objekt v res dáme do proměnné data tohoto formuláře.
			_this.filter.total = res.total; // Nastavíme celkový počet záznamů pro stránkování filtrem.
			_this.refresh(); // Obnova zobrazení formuláře.
			_this.proceed(p); // Spuštění dalšího async. postupu JKF. Detaily zde nebudu popisovat. Je to rutinní volání na funkci load.
		}, null, form);
	},

	// Funkce edit jen otevře formulář detailu pro úpravu.
	edit: function(sender, event) {
		this.openForm("@@eden.CategoryForm", {
			data: {id: event.originator.owner.item.id}
		});

		return true; // Tato návratová hodnota způsobí, že kliknutí na ikonu editace s nebude volat i kliknutí rodičovské komponenty pro přechod na nižší úroveň kategorií.
	},

	// Obdobné volání akce jako u load, ale tentokrát mažeme kategorii. Akce při mazání vrací rovnou aktuální seznam kategorií. Není tedy nutno po mazání znova volat server pro obnovu. Dvě db operace jednou akcí je urychlení.
	remove: function(sender, event) {
		var _this = this;
		this.callAction(this.kindName, "remove", {
			filter: prepareFilter(this.filter),
			id: event.originator.owner.item.id
```

```
		}, function(res) {
			_this.data = res;
			_this.filter.total = res.total;
			_this.refresh();
		});

		return true;
	},

	// Přechod na podkategorii obnovením přehledu.
	open: function(sender, event) {
		this.filter.categoryId = event.originator.owner.item.id;
		this.load(this);

		return true;
	},

	// Výber kategorie pro otevírající komponentu. Předáme data vybrané kategorie otevírateli, či zavoláme nastavený selectCallback a zavřeme okno.
	select: function(sender, event) {
		if (this.selector) {
			// FIXME pri prechodu z mobilniho rezimu na desktop se dela vormular znova a ma nova data (InputsGrid), tak ze toto uz ukazuje jinam
			this.selector.setValue(event.originator.owner.item);
		} else {
			if (this.selectCallback) {
				this.selectCallback(event.originator.owner.item, this);
			}
		}
		this.closeForm();
	},

	// Vrátíme se o úroveň výše a obnovíme přehled.
	back: function() {
		this.filter.categoryId = this.data.category.parent;
		this.load(this);
	},

	// Otevřeme detail kategorie bez předaného id. Tím formulář funguje jako zakládající a ne jako upravující kategorii.
	createCategory: function() {
		this.openForm("@@eden.CategoryForm", {
			data: {parent: this.data.category}
		});
	}
}, {

	// Toto je sekce modulu inputs. Definice vstupů je k dispozici jak na serveru, tak na GUI. Na serveru i na GUI slouží jako validátor vstupů. Na GUI se používá i v komponentě pro automatické generování vstupních polí formulářů.
```

```
	tables: { // Mapa zkratek názvů db tabulek. Zkratka l bude reprezentovat název tabulky kategorií. To bude copy/paste chyba z lokalit. Tady mělo být c a ne l.
		l: "{eden}.categories"
	},

	inputs: [{
		// První vstup je filtrační položka id rodičovské kategorie.
		name: "categoryId", // Název vstupu.
		value: "${this.filter.categoryId}", // Cesta v request datech.
		table: "l", // Vlastnosti tohoto vstupu natáhneme automaticky z databázového modelu tabulky kategorií. Tím nemusíme už duplicitně zde uvádět o jaký typ jde. Tato validační informace je k dispozici už v modelu. Toto se týká i informací o lokalizaci. Tato položka table je nepovinná a pokud jí neuvedeme, můžeme si typ zde specifikovat detailně, ale tento případ jsem zde na těchto vstupech neuvedl.
		column: "parent", // Název sloupce v tabulce se ale liší od názvu vstupu categoryId a tak jej zde musíme upřesnit, jak se jmenuje v db tabulce. tedy parent.
		profile: "filter" // Každý vstup se může značkovat jedním a nebo více profily. Díky tomuto označování mohu pak názvem profilu vybrat odpovídající vstupy pro danou akci. Toto je profil filtru. Vstup se týká tedy jen filtrace dat. Názvy profilů vstupů si můžu vymyslet jak mě napadne. To není nic předem stanoveného.
	}, {
		name: "id", // ID záznamu kategorie.
		value: "${this.id}", // Umístění už není v objektu filtru, ale venku.
		table: "l", // Opět dědíme vlastnosti z datového modelu kategorie.
		required: false, // V data modelu je id sice povinné, ale my tuto povinnost v tomto vstupu chceme potlačit kvůli formuláři zakládající novou kategorii. Ten id kategorie neodesílá.
		// Profily jsou zde 3 a to validace při načtení detailu kategorie a pak zápis a mazání.
		profiles: ["loadDetail", "save", "remove"]
	}, {
		name: "name",
		value: "${this.name}",
		table: "l",
		// Name má profil detail pro automatické vykreslení na formuláři a pak pro uložení.
		profiles: ["detail", "save"]
	}, {
		name: "description",
		value: "${this.description}",
		table: "l",
		fieldType: "textarea", // Jako typ automaticky generovaného vstupního pole zvolíme textareu, tedy víceřádkové textové pole.
		// Profily obdobně jako u name.
		profiles: ["detail", "save"]
	}, {
		name: "parent",
		value: "${this.parent}",
		table: "l",
		// Profily obdobně jako u name.
		profiles: ["detail", "save"]
	}],
```

```
	// Rutinní manipulace s filtrem. Detaily zde nebudu popisovat.
	init: function() {
		addFilterInputs(this.inputs);

		this.inherited(arguments);
	}
}, {

	// Zde už jsem v sekci rozhraní. Tedy výhradně serverová část.  Vlastnost actions je pole názvů akcí. Tím se myslí názvy funkcí tohoto objektu, které jsou volatelné jako služby přes internet. Kdyby někdo chtěl volat akci s jiným názvem, server to odmítne.
	actions: ["load", "loadDetail", "save", "remove"],

	// Jsme v rozhraní pro kategorie, tak si sem dám obecnou proměnnou tableName s názvem tabulky. To není žádná funkcionalita JKF a je to jen pro to mít název tabulky pro akce na jednom místě.
	tableName: "{eden}.categories",

	// Akce load načte seznam a celkové množství pro stránkování. Načte i informace o rodičovské kategorii.
	load: function(data, ctx) {
		// Validační mechanizmus na serveru je úplně jednoduchý. Validujeme request v proměnné data a to vstupy profilu filter. O nic víc se uživatel nestará. V případě nevalidních dat se odešle chyba na klienta a JKF tuto chybu umí zpropagovat uživateli včetně lokalizovaných názvů vstupů. Těch názvů, co jsou uvedeny u datového modelu.
		getInputs(this.inputsName).check(data, ["filter"]);

		// Příprava klauzule where. V tomto případě se jedná jen o obyčejný text, ale JKF umí zpracovat i složitější JSON struktury. To vysvětlím dále.
		var where = q.eq("parent", data.filter.categoryId);

		// Volání SQL selektu pro celkové množství položek kvůli stránkování. Proměnná q (query) na straně serveru nese databázové funkce.
		var total = q.selectCount({
			from: this.tableName,
			where: where
		}, ctx, true);

		// Volání dalších SQL dotazů už provedeme v konstrukci návratové JSON struktury.
		return {
			// Funkce selectQuerySingle vrátí null, pokud nenajde záznam.
			category: data.filter.categoryId ? q.selectQuerySingle({
				columns: "*",
				from: this.tableName,
				where: q.eq("id", data.filter.categoryId)
			}, ctx) : null,
			// Výběr kategorii je obohacen voláním  applyFilter na JSON struktuře selektu. Mohl bych používat skládané SQL pomocí textů, ale JSON struktury jsou lepší. Dají se snadno programovatelně modifikovat a dá se jimi zapojit i používání datového modelu. Konkrétně práce s relacemi. Relace konkrétně v této ukázce použiti nejsou.
```

```
                categories: q.selectQuery(applyFilter(data.filter, total, {
                        columns: "*",
                        from: this.tableName,
                        where: where,
                        sort: [{
                                column: "name"
                        }]
                }), ctx),
                total: total
        };
},

        // Zde máme už volání SQL, kde používáme usnadnění datového modelu. Dotažení detailu kategorie pro formulář s detaily.
        loadDetail: function(data, ctx) {
                // Validujeme vstupní request tentokrát profilem loadDetail
                getInputs(this.inputsName).check(data, ["loadDetail"]);

                return q.selectQuerySingle({
                        columns: ["l.*", { // Vracíme všechny sloupce kategorie, ale navíc vrátíme i sloupce id a name u aliasu p. To je rodičovská kategorie.
                                as: "p", columns: ["id", "name"]
                        }],
                        from: [{
                                // Zde je klauzule from složitější než jen název tabulky. Název tabulky je ve JSF vlastnosti base, ale ještě žádáme JSF o připojení tabulky s aliasem p navázané na cizí klíč parent. Všimněte si, že já nemusím nikde uvádět, jak se ta cizí tabulka jmenuje a ani připojovací podmínku. O to se automaticky postará JKF díky tomu, že zná datový model. Já tedy v JKF mám svobodu používat JSON SQL struktury obohacené o pomoc JKF s modelem a k tomu navíc si mohu místy do těchto struktur vkládat SQL jaké mě bez omezení napadne.
                                base: this.tableName + " l", joins: [{
                                        leftFK: "l.parent", as: "p"
                                }]
                        }],
                        where: q.eq("l.id", data.id)
                }, ctx, false, true); // Poslední true atribut způsobí, že pokud selectQuerySingle nenalezne hledaný záznam, nevrátí null, ale vyhodí uživateli chybu.
        },

        // Ukládání nové a nebo úprava stávající kategorie.
        save: function(data, ctx) {
                var inputs = getInputs(this.inputsName);
                inputs.check(data, ["save"]); // Validace vstupů profilu save

                // Tato operace vyžaduje už přihlášení. Touto metodou zkontrolujeme, zda je uživatel v systému přihlášený. Proměnná ctx obsahuje i session informace.
                eden.AuthService.loadRights(ctx);

                if (data.id) { // Pokud přišlo id kategorie pro úpravu, musíme zkontrolovat i oprávnění měnit tuto kategorii.
```

```
                hasRight({
                        type: "edit",
                        table: this.tableName,
                        context: ctx,
                        throwExcepion: true
                });
        } else {
                var count = q.selectCount({
                        from: this.tableName
                }, ctx);
                // Pokud zakládáme novou kategorii, zkontrolujeme konfigurační proměnnou prostředí, zda je počet kategorií v limitu. Pokud ne, vyhodíme výjimku eden.CategoriesForm.maxCategories. Tato výjimka je jednoduše jen klíč lokalizace s parametrem. Klíč lokalizace je současně i jasně čitelný chybový kód, co se pošle na klienta a tam ho JKF zpropaguje uživateli.
                if (count >= jk.getEnviroment(ctx).config.maxCategories) {
                        throw jk.error("eden.CategoriesForm.maxCategories", [jk.getEnviroment(ctx).config.maxCategories]);
                }
        }

        // Teď už jen si připravíme objekt row s daty pro uložení. Přenesení dat zajistí funkce inputs.toDB s profilem vstupů save.
        var row = {};
        inputs.toDB(data, row, ["save"]);
        if (data.id) {
                // JKF za pomocí znalostí datového modelu umí aktualizovat záznam v tabulce.
                q.executeUpdate([row], this.tableName, ctx);
        } else {
                row.created = formatDate(new Date(), "json");
                // JKF za pomocí znalostí datového modelu umí uložit nový záznam v tabulce.
                q.executeInsert([row], this.tableName, ctx);
        }

        // Akce je volaná z formuláře přehledu, tak na závěr rovnou aktualizujeme a vrátíme přehled.
        return this.load(data, ctx);
    },

    // Jednoduché mazání záznamu kategorií
    remove: function(data, ctx) {
        getInputs(this.inputsName).check(data, ["remove"]);

        q.executeRemove([data], this.tableName, ctx);

        return this.load(data, ctx);
    }
}, {
```

```
	// Poslední sekcí jsou lokalizace.
	cz: {
		title: "Kategorie",
		back: "Zpět",
		create: "Vytvořit novou kategorii",
		maxCategories: "Nejvyšší počet kategorií je momentálně %s"
	},
	en: {
		title: "Categories",
		back: "Back",
		create: "Create new category",
		maxCategories: "Maximum count of categories is now %s"
	}
});
```

Komplexní formulář detailu kategorie (CategoryForm.js)

Tento komplexní formulář nemá vyplněnou sekci se vstupy a ani s rozhraním. Má ale nadefinované vlastní lokalizace.

```
jk.complexForm({
	namespace: "eden",
	form: "Category"
}, {

	statics: {
		width: 450,
		height: 300,
		theme: "dark"
	},

	view: {
		name: "content",
		components: [{
			classes: "jkali-content-panel",
			components: [{
				// Tato komponenta je zásadní pro formuláře s detaily. Generuje automaticky vstupní políčka podle inputs modulů a uvedených profilů. Vstupní políčka si mohu zadat napevno pomocí komponent, ale toto je hezčí cesta.
				kind: "@@jkali.InputsGrid",
				localization: "${getForm(this).kindName}", // Umístění lokalizací vstupních políček. Konkrétně zde se nevyužívá, protože lokalizace jdou z datového modelu.
				class: "${getForm(this).parent.inputsName}", // Specifikace vstupů u rodičovského formuláře (přehled kategorií).
				inputs: "${this.owner.data}", // Cesta k napojeným datům
				labelsWidth: "150px", // Šířka titulků
				profiles: ["detail"] // Vybrané profily pro generování. Vstupy se generují v takovém pořadí, v jakém jsou uvedeny v modulu inputs.
			}]
```

```
		}, {
			kind: "@@jkali.ButtonPanel",
			components: [{
				content: "$l{jkali.common.ok}",
				ontap: "ok"
			}, {
				content: "$l{jkali.common.close}",
				ontap: "close"
			}]
		}]
	},

	onShow: function() {
		this.app.adaptHeight(this, this.$.content);
	},

	load: function(form, p) {
		if (!this.data.id) { // Pokud není uvedeno id kategorie, rovnou pokračujeme na otevření formuláře bez volání akce na serveru.
			this.proceed(p);
			return;
		}

		var _this = this;
		// Voláme akci loadDetail rodičovského komplexního formuláře.
		this.callAction(this.parent.kindName, "loadDetail", this.data, function(res) {
			// Rutinní nastavení odezvy do vstupů
			if (_this.$.inputsGrid) {
				_this.$.inputsGrid.setInputs(res);
			} else {
				_this.data = res;
			}
			_this.refresh();
			_this.proceed(p);
		}, null, form);
	},

	ok: function() {
		// Provedeme obecný mechanizmus validace všech GUI komponent vstupů formuláře. Profil detail. Tato kontrola se provede v prohlížeči ještě před voláním serveru.
		if (!this.check(["detail"])) {
			return;
		}

		// Rutinní optimalizace vstupních dat pro request a následně volání akce save.
		var data = getInputs(this.parent.inputsName).optimalize(this.data, ["save"]);
		data.filter = prepareFilter(this.parent.filter);
		if (!data.id && !data.parent) {
			data.parent = this.parent.filter.categoryId;
		}
```

```
                var _this = this;
                this.callAction(this.parent.kindName, "save", data, function(res) {
                        _this.parent.data = res;
                        _this.parent.filter.total = res.total;
                        _this.parent.refresh();

                        _this.closeForm();
                });
        }
}, null, null, {

        cz: {
                title: "Vytvořit novou oblast"
        },
        en: {
                title: "Create new category"
        }
});
```

Služba s autorizačními funkcemi (AuthService.js)

Jen malá ukázka, jak na serveru vytvářet třídy, které nesou obecně používané funkce. Třeba služba eden.AuthService má mimo jiné funkce login a logout. Funkce neprovádějí kontrolu hesla a podobně, ale jen nastaví session a vrátí požadovaná data. Nejedná se o rozhraní, které by bylo napřímo volané odněkud z webu.

```
jkr({class:{
        name: "eden.AuthService",
        version: "1.00",

        serverSide: true
}}, {statics:{

        login: function(userId, ctx) {
                // Nastavíme userId do objektu session.
                ctx.session.userId = userId;

                // Aktualizujeme čas přihlášení u uživatele.
                q.executeUpdate([{
                        id: userId,
                        last_success_login: formatDate(new Date(), "json")
                }], "{eden}.users", ctx);

                // Načteme informace o uživateli.
                var user = q.selectQuerySingle({
                        columns: ["nick", "firstname", "surname", "verified_by"],
                        from: "{eden}.users",
                        where: q.eq("id", userId)
                }, ctx);
```

```
                // Načteme informace o ověřovateli uživatele.
                var verification = q.selectQuerySingle({
                        columns: "verifier",
                        from: "{eden}.verifications",
                        where: [
                                q.eq("applicant", userId)
                        ]
                }, ctx);

                // Sestavíme výsledný objekt pro návratovou hodnotu.
                var result = {
                        userId: userId,
                        firstname: user.firstname,
                        surname: user.surname,
                        nick: user.nick,
                        verifiedBy: user.verified_by,
                        rights: this.loadRights(ctx),
                        verification: verification,
                        unreaded: this.unreadedCount(ctx)
                };

                result.wcrCount =
q.selectCount(eden.WaitingCommunityRequestsForm.getCRFromWhere(ctx), ctx);

                // Toto volání funkce release se na serveru vyskytuje. Používám ho pro rozložení
datových struktur, aby se ulehčilo garbage collectoru pro JavaScript.
                release(user);

                return result;
        },

        logout: function(ctx) {
                ctx.session.userId = null;
                ctx.session.extraAuth = null;
        },

        …

});
```

Pravidelné spouštění úloh (VotingsTask .js)

Modul třídy s příznakem task = true. Je to automaticý časovač, který na serveru po stanovených intervalech pouští funkci processTask. Vnitřek funkce zde není ale k dispozici jsou podobné prostředky jako na rozhraních a to práce s databází a podobně.

```
jkr({class:{
        name: "eden.VotingsTask",

        task: true,
```

```
        seconds: [0, 15, 30, 45] // Spouštěč se volá každou čtvrt minutu.
}}, {

        processTask: function(ctx) {

                …

        }
});
```

Správa prostředí (enviroments.js)

Pro spravování prostředí se používají JKR moduly variables, connections a enviroments. Modul variables jsou proměnné s obecnými hodnotami, které se na jednotlivých prostředí nemění. Třeba cesty, kam si chci nechat sestavovat projekty. Modul connections jsou informace o připojení na jednotlivé zavaděče v jednotlivých serverech. Jak jsem psal na začátku server má nějaké aplikace napsané v hostitelském jazyce konkrétně u mě zatím jen Java. Každá tato aplikace, která má připojený JKF zavaděč, tak je možné do ní vzdáleně instalovat JKF projekty. Jeden zavaděč na jedné hostitelské aplikaci může mít nainstalováno klidně více JKF projektů najednou. Modul enviroments je něco jako matice, která definuje konfigurace jednotlivých projektů pro různé zavaděče.

```
jkr({
        variables: {
                tmpDirPath: "/home/kali/DL/ram", // Cesta k pomocnému RAM adresáři ještě co jsem měl Linux. Adresář sloužil při sestavování projektů.
                jkaliFrameworkExamplesDevDir: "jkali-projects-dev", // Podadresář pro sestavení vývojového prostředí
                jkaliFrameworkExamplesTestDir: "jkali-projects-test",  // Podadresář pro sestavení testovacího prostředí
                jkaliFrameworkExamplesProductionDir: "jkali-projects",  // Podadresář pro sestavení provozního prostředí
                …
        },

        connections: [{
                name: "jkali-projects", // Název konektivity na zavaděč u mě na vývoji.
                host: "localhost", // Host zavaděče.
                port: 8090, // Port pro zavaděč.
                context: "jkali-projects", // Web kontext konkrétní Java aplikace se zavaděčem.
                password: "...", // Heslo k přístupu na zavaděč.
                SQLGenerator: "pgsql" // Typ databáze.
        }, {
                name: "jkali-projects-test", // Zavaděč pro testovací prostředí
                host: "localhost",
                port: 8091,
                context: "jkali-projects-test",
                password: "...",
                SQLGenerator: "pgsql"
        }, {
```

```
            name: "jkali-projects-production", // Zavaděč pro vývojové prostředí
            host: "192.168.0.100",
            port: 8080,
            context: "jkali-projects",
            password: "...",
            SQLGenerator: "pgsql"
        }, {
            …
        }],

        enviroments: [{
            name: "example1", // Prostředí projektu examples1 pro vývoj
            projectDir: "$enviroments/example1", // Cesta ke složce projektu relativně od umístění souboru s prostředími.
            buildDir: "${this.tmpDirPath}/ex1-build", // Sestavovací adresář
            // server
            connection: "jkali-projects", // Název konektivity na zavaděč
            useFilesystem: true, // Má zavaděč načítat projekty ze souborového systém? V opačném případě by je načítal z databáze. U vývojového prostředí, kde programátor na projektech pracuje je potřeba načítání ze souborů.
            datasource: "testdb", // Název zdroje dat pro zavaděč. Tyto zdroje dat nejsou v souboru s prostředími, ale přímo v domovském adresáři uživatele ve složce .jkali. Zde jsou properties soubory k jednotlivým kontextům hostitelských aplikací a navíc každý properties soubor může mít definováno více datasource pro různé databázové uživatele. Je to obdoba jako datasource na Java aplikačních serverech. Konektivity do místní uživatelovo databáze mohou být u každého uživatele jiné a proto jsou odděleně v home adresářích. Názvy datasourců by ale měli být u všech uživatel stejné. To dělá ze souboru enviroments.js přenositelný celek. Přijde například nový člověk do projektu a vy mu můžete poslat projekt jako celek včetně nastavení pro jednotlivá prostředí. To je výhoda. Je to takto více přehledné a snadnější pro ovládání.
            schematics: { // Mapa schémat pro prostředí. Schémata při tvorbě datového modelu mohou využívat proměnné a tyto proměnné se nastavují zde.
                this: "testdb"
            }
        }, {
            name: "example1-production", // Ukázky pro provozní prostředí
            projectDir: "$enviroments/example1",
            buildDir: "${this.tmpDirPath}/ex1-production-build",
            // server
            connection: "jkali-projects-production",
            composing: true, // Zde se nepoužívá načítání ze souborového systému, ale z databáze, kam vývojář nahraje vzdáleně aplikaci.
            allInOne: true, // Všechny moduly v jednom DB záznamu.
            datasource: "testdb",
            schematics: {
                this: "testdb"
            }
        }, {
            name: "eden", // Projekt Eden pro vývoj.
            projectDir: "$enviroments/eden",
            project: "${this.ePanel ? 'ePanel' : 'eden'}", // Eden má podprojekt ePanel a ten se dá
```

```
vybrat nastavením příznaku ePanel v parametru volání sestavení projektu.
                buildDir: "${this.tmpDirPath}/${this.ePanel ? 'ePanel' : 'eden'}-build",
                // client
                clientTitle: "${this.ePanel ? 'ePanel' : 'Project Eden'} [dev]", // Nastavíme titulek
HTML stránky. Přípona [dev] ukazuje, že jde o vývoj.
                outDir: "${this.tmpDirPath}/${this.jkaliFrameworkExamplesDevDir}",
                outFile: "${this.ePanel ? 'ePanel' : 'eden'}-dev.html",
                production: "${this.production == 'true'}", // Předáme vstupní parametr. Pokud je
production nastaveno, dělá se jedno HTML obsahující vše. Toto sestavení klienta provádí EnyoJS.
                config: { // Zde si pro prostředí můžeme konfigurovat přímo aplikaci. Tato
konfigurace je dostupná jak na klientu, tak na serveru.
                        serviceURL: "http://localhost/jkali-projects/service.php", // Specifikace
servisní URL. PHP skript přeposílá data Java serveru. Měl jsem tam technický problém, který řešil
tento skript.
                        multipartServiceURL:
"http://localhost:8080/jkali-projects-redirect/Service", // URL pro multipart data.
                        enviroment: "eden", // Název prostředí.
                        enabledTasks: "*", // Pro prostředí je povoleno pouštění tasků.

                        maxCategories: 100000, // Již zmíněná konfigurace maximálního počtu
kategorií pro dané prostředí.

                        …
                },
                // server
                connection: "jkali-projects",
                useFilesystem: true,
                //showSQL: true, // Při nastavení showSQL server do konsole tiskne volaná SQL.
                datasource: "shared",
                schematics: { // Tato proměnná už byla zmíněna v tabulce kategorií.
                        eden: "shared"
                },
                tablespaces: { // Mapa tablespaců. Obdobné chování jako u schémat.
                        eden: "${this.mysql == 'true' ? '' : 'jkali_projects'}"
                }
        }, {

                …

        }]
});
```

Formulářový testovací modul (CategoriesTest.js)

Zmiňoval jsem se, že JKF zahrnuje i tvorbu testování. Testování se dělá pomocí sekvence volání akcí. Testování tedy zahrnuje jen chování serveru, ale to je při testech zásadní. Klient je se dá brát jako rutinní používání komponent, které si jednou napíšete, odladíte a pak už nenastávají moc často problémy a tím spíše ne, pokud pro automatické generování vstupů používáte datový model. Na serveru ale měníte v průběhu času datový model a to může nečekaně ovlivňovat různé situace při přístupu k databázi. JKR modul test obsahuje tedy sekvenci operací, které volají akce serveru a i

takové, které nevolají akce serveru. Zapisuji moduly tak, že vedle hlavního komplexního formuláře pro přehled položek tabulky vytvářím jeden soubor s koncovkou Test.js. Tento soubor obsahuje jeden a i více testovacích modulů. Záleží na tom, zda testovací sekvence má být spojitá a nebo je narušená sekvencemi z jiných testovacích souborů. Může přeci nastat situace, kdy se mi třeba zaregistruje jeden uživatel, pak provádí testování za jeho profil nějaké aktivity na jiných formulářích a následně se registruje další uživatel, což je opět test v souboru pro uživatele. Každý testovací modul má tedy pořadí provádění, jak uvidíte níže. To, že seskupuji volání testovacích akcí do souborů podle tabulek a příslušných formulářů je z toho důvodu, že když se mění něco na volací akci, tak je mám všechny pohromadě a snadno bez složitého hledání upravím na všech místech. Přijde mi to i přehlednější až na to, že nevidíte na první pohled průběh celé testovací sekvence. Proto by měl podle mě správně tester tu sekvenci dokumentovat bokem, aby byla jasně přehledná pro ostatní z týmu. Zde je ukázka testování kategorií.

```
jkr({test:{
        name: "CategoriesTest",
        order: 10, // Toto je pořadí volání této testovací sekvence.
        // prev: votings 9, next: projects 11 // Vidíte, že v kódu mám komentář s popisem, která oblast sekvencí je předchozí a která následující, abych i bez dokumentace mohl dohledat celkovou sekvenci modulů. Číslo sekvence order může mít i desetinou hodnotu. Pokud budu chtít přidat zde sekvenci, která má být mezi pořadím například 27 a 28 a tyto dvě sekvence nejsou zde v tomto souboru, tak není třeba řadu posouvat a jen udělám novou sekvenci s průměrem obou pořadí, tedy 27.5.
        sequence: [{
                run: function(ctx) {
                        ctx.session = ctx.opSession; // První krok této testovací sekvence je obyčejné provedení skriptu. Nastavíme simulovanou testovací session na tu, kterou jsem si připravil v předchozích krocích a to na uživatele Ondřeje Pokorného. Něco takového například známý testovací nástroj SoapUI nedovede, že nasimuluje session data. Nebo asi hodně těžko. Toto je možné snadno provádět díky tomu, že testovací systém je součástí JKF a je tam synergie. Testujete tak jednou sekvencí přístup více uživatelů. To je nebývalé.
                }
        }, {
                category: "save-data", // Další krok je už volání akce serveru. Akce save pro ukládání nové kategorie na rozhraní s názvem eden.CategoriesForm. Vidíte, že já si mohu testovací kroky členit i do kategorií. To se hodí, když chci test spustit jen pro sekvenci s vybranými kategoriemi. Toto je například kategorie save-data, která vypovídá, že krok má zásah do změn dat. Jsou i takové kroky, které nemění data a jen testují případné chyby. Pokud chci test provádět rychleji a zajímá mě jen tvorba dat, mohu ty ostatní kroky přeskočit díky tomuto kategorizování.
                callAction: {
                        interface: "eden.CategoriesForm",
                        action: "save",
                        data: function(ctx) { return {
                                // 1 // Jen si poznamenávám pro přehled automaticky generované id
                                name: "Výroba",

                                filter: ctx.filter // V kontextu testování mám předpřipravený obecný stránkovací filtr, který snadno jen předám do requestu akce.
                        }; },
                        response: function(data, ctx, proceed) {
                                proceed(); // Test proběhl a bez kontrol odezvy v proměnné data
```

```
přejdeme na další krok funkcí proceed.
                        }
                    }
            }, {
                    category: "save-data", // Obdobně voláme založení další kategorie a tak dále...
                    callAction: {
                            interface: "eden.CategoriesForm",
                            action: "save",
                            data: function(ctx) { return {
                                    // 2
                                    name: "Služby",

                                    filter: ctx.filter
                            }; },
                            response: function(data, ctx, proceed) {
                                    proceed();
                            }
                    }

            }, {
                    …
            }]
}});
```

Hlavní aplikační testovací modul (AppTest.js)

Ještě bych rád ukázal rozměrnější pohled na testovaní a to jak vlastně testování začíná. Jakými kroky a co je možné nastavit. Níže jsou vidět první kroky celkové testovací sekvence. Úplně první krok je, že si nastavím nějaké kontextové proměnné.

```
jkr({test:{
        name: "AppTest",
        order: 0,
        // next: locations 1
        sequence: [{
                run: function(ctx) {
                        ctx.usersCreateLoops = 5; // Kolik chceme předgenerovat uživatel? Kolik generačních cyklů? Nejedná se o registraci uživatele voláním akce, ale přímo voláním SQL.
                        ctx.usersCreateLoopSize = 200; // Kolik uživatel vygeneruje jeden cyklus?

                        if (ctx.profile == "save-data") {
                                ctx.categories = ["save-data"]; // Pokud je vstupní parametr testování profile na hodnotě save-data, tak testovací kategorie budou jen save-data.
                        } else if (ctx.profile == "experimental") {
                                ctx.categories = ["experimental"]; // Pokud je profil experimental, testuje se jen experimentální test, který má jen jeden krok viz. níže.
                                ctx.noRunNonCategory = true; // Zároveň při experimentálním testování nebudeme provádět kroky s neuvedenou kategorií jako třeba nastavování session.
                        }
```

```
                        // Nastavíme si do kontextu čas konce hlasování a sbírek o 14 dní do budoucna. Jen ukázka, kam až při generování dat jdu.
                        var nowplus14 = new Date();
                        nowplus14.setDate(nowplus14.getDate() + 14);
                        var nowplus14Str = jkali.Utils.formatDate(nowplus14, "json");
                        ctx.votingTime1 = nowplus14Str;
                        ctx.collectionTime1 = nowplus14Str;

                        // Dáme si do kontextu výčty. Při volání akcí se vstupními daty výčtů je používám. Kdybych pak nějaký výčet změnil, nahlásí mi to při testu chybu, než když bych vkládal přímo textovou hodnotu.
                        ctx.projectTypeEnum = getEnum(eden.ProjectTypeEnum);
                        ctx.budgetTypeEnum = getEnum(eden.BudgetTypeEnum);
                        ctx.payerTypeEnum = getEnum(eden.PayerTypeEnum);
                        ctx.recipientTypeEnum = getEnum(eden.RecipientTypeEnum);
                        ctx.payTypeEnum = getEnum(eden.PayTypeEnum);

                        ctx.filter = { // Již zmíněný objekt filtru.
                                first: 0, // pozice stránky
                                count: 20 // velikost stránky
                        };
                }
        }, {
                category: "experimental", // Hned druhý krok je volání akce experimental na rozhraní eden.AppTest, které je součástí tohoto souboru a je popsané níže. Tento krok se volá vždy a v testovacím profilu experimental jen a pouze tento krok. Když chci rychle zkusit zavolat nějaký experimentální kód na serveru a nechci se proklikávat k nějakému testovacímu formuláři, jednoduše si ho otestuji v této akci.
                callAction: {
                        interface: "eden.AppTest",
                        action: "experimental",
                        response: function(data, ctx, proceed) {
                                proceed();
                        }
                }
        }, {
                category: "save-data", // Volání akce clearTables maže obsah všech tabulek datového modelu před zahájením generování dat.
                callAction: {
                        interface: "eden.AppTest",
                        action: "clearTables",
                        response: function(data, ctx, proceed) {
                                proceed();
                        }
                }
        }]
});

jkr({test:{
        name: "AppTest",
```

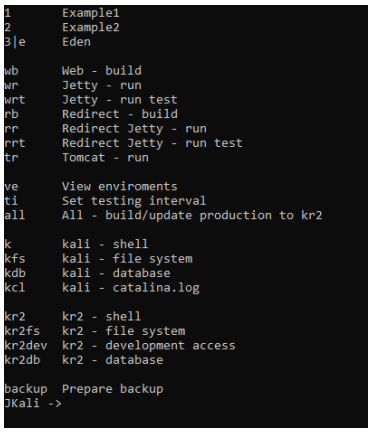
```
	order: 2,
	// prev: locations 1, next: users 3
	sequence: [{
		category: "save-data", // V předchozím kroku, který není zde vidět se vygenerovali lokality. Teď přejdeme k předgenerování uživatel do těchto lokalit akcí createUsers.
		callAction: {
			interface: "eden.AppTest",
			action: "createUsers",
			data: function(ctx) { return { // Předáme data z kontextu do requestu.
				usersCreateLoops: ctx.usersCreateLoops,
				usersCreateLoopSize: ctx.usersCreateLoopSize,
				cblloc: ctx.cblloc,
				cbrloc: ctx.cbrloc
			}; },
			response: function(data, ctx, proceed) {
				proceed();
			}
		}
	}]
}});

… // Další testovací kroky v tomto souboru přeskočím a jdu k samotnému rozhraní eden.AppTest.

jkr({class:{
	name: "eden.AppTest",
	extends: "jkali.ActionInterface", // Rozhraní dědí z třídy jkali.ActionInterface, která se používá na komplexních formulářích pro vyvolávání akcí.
	version: "1.00",
	interface: true // Indikátor, že se jedná o rozhraní.
}}, {

	// Seznam akcí na rozhraní, ale všechny popisovat nebudu.
	actions: ["experimental", "clearTables", "createUsers", "createRights", "createConfirmRight", "createUserCommunities"],

	// Funkci receive, kterou JSF volá na rozhraní pří volání požadavku z internetu přetížíme a to tak, že pokud se nejedná o testovací prostředí, tak se nepokračuje a žádná z akcí se neprovede. Volání rozhraní pak skončí nelokalizovanou chybou. To, že je prostředí testovací se nastavuje v prostředích v sekci config vlastností testing = true. To nebylo na ukázkách prostředí výše vidět.
	receive: function(request, ctx) {
		if (!jk.isTesting(ctx)) {
			throw jk.errorMessage("No testing enviroment");
		}

		return this.inherited(arguments);
	},

	experimental: function(data, ctx) {
		//log("Experimental testing..."); // Zde máme prostor psát a snadno testovat experimentální kód.
```

```
	},

	clearTables: function(data, ctx) {
		q.clearTables(ctx); // Mažeme obsahy tabulek v datovém modelu. Mazání dělá funkce JKF, kterou mám otestovanou jen pro PgSQL.
	},

	createUsers: function(data, ctx) {
		// Pole jmen pro generování.
		var fnames = ["Jiří", "Ondřej", "Milan", "Petr", "Michal", "Jaroslav", "Miroslav", "Čestmín", "Tomáš", "Otakar", "Pavel", "Luboš", "Václav", "Jonáš",
			"Matěj", "Svatoslav", "Martin", "Marcel", "Libor", "Karel", "Adam", "Adrian", "Arnošt", "Cyril", "Svatopluk", "Bedřich", "Dalibor", "Jan", "Jaromír",
			"Matouš", "Lukáš", "Josef", "Jeremiáš", "Igor", "Ivan", "Marek", "Ladislav", "Vladislav", "Gabriel", "Bohdan", "Jindřich"];
		// Pole příjmení pro generování.
		var snames = ["Adamec", "Silný", "Bořivoj", "Krátký", "Vávra", "Tušný", "Rychlí", "Perný", "Hojný", "Medvěd", "Alcior", "Bohatý", "Beran", "Cwerdyk", "Čipera", "Elevar",
			"Farný", "Funštát", "Falský", "Gruber", "Gayzar", "Arský", "Giridza", "Horník", "Hašler", "Hakyn", "Hořký", "Irisa", "Irven", "Udatný", "Sládek", "Studený",
			"Šilhavý", "Bohdal", "Boháč", "Veselý", "Podlaha", "Lorenc", "Gondík", "Železný", "Vondráček", "Tercl", "Šedivý", "Hubený", "Chvátal", "Pelikán", "Kropáček",
			"Podhorský", "Podlešák", "Novák", "Nevosád", "Urban", "Milotivý", "Macura", "Marný", "Ungnad", "Novotný", "Světlý", "Horní"];

		// Vygenerujeme podle konfigurace uživatele a uložíme je do databáze. Počáteční id začíná od 100000000000000. Eden umožňuje v rozsahu do 100000000000000 zadávat volitelná id při registraci.
		var i = 0;
		for (var j = 0; j < data.usersCreateLoops; j++) {
			var rows = [];
			for (var n = 0; n < data.usersCreateLoopSize; n++) {
				rows.push({
					id: i + 100000000000000,
					firstname: fnames[i % fnames.length],
					surname: snames[Math.floor(i / fnames.length) % snames.length],
					password: "1234",
					location: i % 2 == 0 ? data.cbrloc : data.cblloc,
					sentinel: i % 4,
					medic: (Math.floor(i / 4)) % 4,
					question: 0,
					answer: "...",
					credit: 0,
					cover: 0,
					diligence: 70,
					fail_logins: 0,
					fail_logins_wait: 5,
					created: formatDate(new Date(), "json"),
					active: true,
```

```
                        deactive_time: 0,
                        deactivation_requested: false,
                        change_notify: false
                    });
                    i++;
                }
                q.executeInsert(rows, "{eden}.users", ctx);
                release(rows);
            }

            // Teď ještě v případě, že se jedná o PgSQL, tak změníme sekvenci tabulek na konec vygenerovaných záznamů. Zde je vidět, že můžeme na JKF rozhraních volat SQL libovolně sestaveným textem.
            if (serverInfos.SQLGenerator == "pgsql") {
                var p = jk.replaceSchema("{eden}.users", ctx.enviroment).split(".");
                var seq = format("%s." + q.sequencePrefix + "%s" + q.sequencePostfix, p[0], p[1]);
                var exec = q.createExecutions();
                q.query("ALTER SEQUENCE " + seq + " RESTART WITH " + (i + 100000000000000), exec);
                q.runExecutions(exec);
            } else {
                throw jk.errorMessage("Database is not implements ALTER SEQUENCE");
            }
        },

        …
});
```

JKF umí uživateli nabídnout i volání testovací sekvence v pořadí omezeném intervalem. Například si spustíte test od pořadí 0 do 10. Potom si něco budete klikat v aplikaci ručně a na závěr si necháte pustit test od kroku 11 do konce. Neřešíte situace, kdy potřebujete zbytečně čas testera, aby vám manuálně nasimuloval nějaký problém. Ten tester by opravdu měl mít na starosti jen tvorbu testovací sekvence a její dokumentování. Pokud tester narazí na problém, tak jediné co udělá je, že informuje o problému. O tom co přesně udělal a v jakém to bylo testovacím kroku. Případně po opravě rozšíří testovací sekvenci, tak že testy budou kvalitnější.

Modul nabídek (menu.js)



Teď už se dostáváme k poslední otázce. Jak se to celé vlastně ovládá? JKF nabízí vlastní poměrně přehledný způsob ovládání a opět postavený na JavaScriptu. K ovládání se používá textové rozhraní Linux terminálu a nebo Windows příkazové řádky. Obojí se pouští v JavaScript engine NodeJS. JKR moduly se jmenují menu. Nalevo vidíte hlavní nabídku mých JKali projektů, jak se zobrazuje v textovém rozhraní příkazové řádky na Windows. Platforma NodeJS je multiplatformní, tak že to samé jde spustit i v Linuxu a i jinde. V nabídce vidíme nějaké položky. Na levé straně je zkratka, kterou volíme položku a napravo je název položky. Teď ukážu modul hlavní nabídky. Těch nabídek je v souboru menu.js více a dají se i

parametrizovat. O tom více níže.

```
jkr({menu: {
	name: "main",
	title: "JKali", // Název nabídky vidíme úplně dole vedle výzvy k zadání povelu.
	include: ["$menus/enviroments.js"], // Připojujeme modul s prostředími.
	items: [{
		title: "Example1", // První položka je přechod na podmenu s projektem Example1.
		shortcuts: "1", // Zkratka pro povel je 1.
		menu: "enviroment", // Jedná se o přechod na menu s názvem enviroment.
		args: {
			envName: "example1" // Parametr podmenu envName je example1.
		}
	}, {
		title: "Example2", // Další položka je obdoba té výše popsané.
		shortcuts: "2",
		menu: "enviroment",
		args: {
			envName: "example2"
		}
	}, {
		title: "Eden", // Přechod na projekt Eden, který má specifické menu eden.
		shortcuts: ["3", "e"], // Dvě možné klávesové zkratky.
		displayShortcuts: "3|e", // Jak vidět v nabídce zkratku?
		menu: "eden"
	}, {
		spacer: true // Toto je jen prázdná řádka.
	}, {
		title: "Web – build", // Tato položka není přechod na podmenu, ale už volání funkce pro sestavení Java aplikace se zavaděčem.
		shortcuts: "wb",
		execute: function(ctx, callback) { // I menu má svůj kontext, kam si dávám pomocné proměnné a funkce, co používám na akcích menu položek.
			ctx.jkaliWebBuild(ctx.jkaliWeb, "jkali-web", "jkali-projects", callback);
			return true;
		}
	}, {
		title: "Jetty – run", // Zde pouštíme Jetty server na specifických portech. Vidíte, že i pouštění serverů je záležitostí JKali nabídek. Já pomocí JKali nabídek pouštím třeba prohlížeč s odkazem na jízdní řády a nebo nějakou hru. Zástupci na ploše jsou pro mě už minulostí.
		shortcuts: "wr",
		execute: function(ctx, callback) {
			ctx.jkaliRunJetty(ctx.jkaliWeb, "jkali-projects-target", "jkali-projects", 8090, 10000, callback);
			return true;
		}
	}, {
		…
	}],
```

```
	// Zde si nastavíme kontextové proměnné a funkce této nabídky.
	setup: function(ctx) {
		ctx.jkaliNode = "..."; // Nastavení všelijakých cest pro servery, Javu, NodeJS a jiné.
		ctx.jkaliHome = "...";
		ctx.jkaliJS = ctx.jkaliHome + "...";
		ctx.jkaliTemp = "...";
		ctx.jkaliTomcatParent = "...";
		ctx.jkaliTomcat8 = "...";
		ctx.jkaliWeb = "...";
		ctx.jkaliJDK8 = "...";
		ctx.jkaliPsql = "...";
		ctx.maven = "...";

		ctx.localDBAdmin = "...";
		ctx.localDBName = "...";
		ctx.localDBPort = "...";

		// Funkce specifická pro Linux volá systémový příkaz spuštěný v gnome-terminálu.
		ctx.terminalExec = function(command) {
			fs.writeFileSync(this.jkaliTemp + "/command", command);
			cli.exec("gnome-terminal -e \"" + this.jkaliHome + "/accessories/command.sh
" + this.jkaliTemp + "/command\"");
		};

		…

		// Vrací systémový příkaz. V Linuxu si interaktivním editorem sed upravím
buildovací soubor pro Maven, aby cílový adresář šel jinam.
		ctx.correctPom = function(targetName) {
			return "cp pom.xml _pom.xml;" +
				"sed -i 's/<build>/<build><directory>" + this.jkaliTemp.replace(/\//g,
"\\/") + "\\/" + targetName + "<\\/directory>/g' _pom.xml;";
		};

		// Vrací systémový příkaz na mazání buildovacího souboru _pom.xml.
		ctx.restorePom = function() {
			return "rm _pom.xml;";
		};

		// Funkce zavolá systémový příkaz Linuxu pro sestavení zavaděče. Sice utvářím
systémové příkazy, ale vidíte, že mohu pro jejich tvorbu alespoň používat JavaScript a ne zase
nějaký komplikovaný dávkovací jazyk. Je to usnadnění vedoucí k jednotě. Takovýchto funkcí v
nabídkách vytvořím jen pár a pak už je jen opakovaně volám JavaScriptem viz. akce menu výše.
		ctx.jkaliWebBuild = function(webPath, webName, targetNamePrefix, callback) {
			cli.exec("cd " + webPath + ";" +
				"export JAVA_HOME=" + this.jkaliJDK8 + ";" +
				this.correctPom(targetNamePrefix + "-target") +
				this.maven + " -f _pom.xml clean package;" +
				this.restorePom() +
				"cd " + this.jkaliTemp + "/" + targetNamePrefix + "-target;" +
```

```
                        "mv " + webName + "*.war " + targetNamePrefix + ".war;", true, true,
callback);
                };

                // Pouštím Jetty server
                ctx.jkaliRunJetty = function(webPath, targetName, webName, webPort, debugPort,
callback) {
                        cli.exec("cd " + webPath + ";" +
                                "export JAVA_HOME=" + this.jkaliJDK8 + ";" +
                                "export MAVEN_OPTS=\"-Xmx512m -XX:MaxPermSize=128m -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=" + debugPort + "\";" +
                                this.correctPom(targetName) +
                                "sed -i
's/<contextPath>\\/to-rename-by-jkali<\\/contextPath>/<contextPath>\\/" + webName +
"<\\/contextPath>/g' _pom.xml;" +
                                this.maven + " -f _pom.xml jetty:run -Djetty.port=" + webPort + ";",
true, true, function() {
                                        cli.exec("rm " + webPath + "/_pom.xml");
                                        callback();
                                });
                };
        }
});});
```

```
d          Dev enviroment
dp         Dev ePanel enviroment
rddb       Restore dev database
m          Dev Mysql enviroment

t          Test enviroment
tp         Test ePanel enviroment
(c|d)ts    Create/Drop testing tablespace
(d|r)db    Dump/Restore test database
rtd        Run test create data
rte[r]     Run test experimental [with reload]
rta[r]     Run test all [restore database before]

p          Production to kali enviroment
pp         Production ePanel to kali enviroment
pl         Production to kali (local) enviroment
dpd        Dump kali production database
pk         Production to kr2 enviroment
JKali Eden -> _
```

Další nabídkou je podmenu projektu Eden. Vidíme dole u výzvy, že je to podmenu za hlavní nabídkou JKali.

```
jkr({menu: {
  name: "eden",
  title: "Eden",
  items: [{
                title: "Dev enviroment", // Podmenu pro
vývojové prostředí Edenu.
                shortcuts: "d",
                menu: "enviroment",
                args: {
                        envName: "eden"
                }
        }, {
                title: "Dev ePanel enviroment", // Podmenu pro vývojové prostředí ePanelu.
                shortcuts: "dp",
                menu: "enviroment",
                args: {
                        envName: "eden",
                        subtitle: "ePanel",
                        clientOnly: true, // Parametr nabídky prostředí, že projekt má jen klientskou
část.
                        systemArgs: "-DePanel=true"
                }
        }, {
```

```
            …
        }],

        // Opět pomocné proměnné pracující se systémem i databází.
        setup: function(ctx) {
            ctx.jkaliEdenPrepareSchema = function(name, user) {
                return "CREATE SCHEMA " + name + " AUTHORIZATION jkali;" +
                    "GRANT USAGE ON SCHEMA " + name + " TO " + user + ";" +
                    "GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
IN SCHEMA " + name + " TO " + user + ";" +
                    "GRANT SELECT, UPDATE ON ALL SEQUENCES IN SCHEMA "
+ name + " TO " + user + ";";
            };

            ctx.jkaliEdenDropSchema = function(name) {
                return "DROP SCHEMA " + name + " CASCADE;";
            };

            ctx.jkaliEdenRestoreTestDatabase = function() {
                cli.exec("psql -d jkali jkali -c \"" + this.jkaliEdenDropSchema("eden_test") +
this.jkaliEdenPrepareSchema("eden_test", "eden_test") + "\";" +
                    "cat " + ctx.jkaliTemp + "/eden.db | " +
                    "sed 's/_jkali_tablespace_replacement/jkali_projects_test/g' | " +
                    "sed 's/_jkali_schema_replacement/eden_test/g' | psql -d jkali jkali");
            };
        }
}});
```

```
u     Server update
un    Server undeploy and drop tables
dt    Drop tables

c     Build client
cp    Build client production pack

f     View in Firefox
ch    View in Chrome
re    Reload
a[f]  Availability from now [+ <some minutes>] on l
ocal
JKali Eden 'eden' enviroment -> _
```

Poslední nabídka je ta, která se dá parametrizovat a to nabídka pro prostředí.

```
jkr({menu: {
        name: "enviroment",
        title: "'${jk.menus.enviroment.env.name +
(jk.menus.enviroment.args['subtitle'] ? '-' +
jk.menus.enviroment.args['subtitle'] : '')}'
enviroment", // Titulek složíme zase opět s využitím
výrazů, které JKF používá i jinde.
        items: [{
            title: "Server update", // Tato akce aktualizuje vzdáleně na serveru serverové moduly.
            shortcuts: "u",
            disabled: "${jk.menus.enviroment.args.clientOnly}", // Položka není viditelná, pokud
má projekt pouze klienta.
            execute: function(ctx) {
                cli.exec(jk.menus.enviroment.updateSpec + " -Dupdate-only=server");
            }
        }, {
            title: "Server undeploy and drop tables", // Bez akce. Tuto položku jsem očividně
ještě nenaprogramoval.
            shortcuts: "un",
```

```
                disabled: "${jk.menus.enviroment.args.clientOnly}"
        }, {
                title: "Drop tables", // Také nenaprogramované.
                shortcuts: "dt",
                disabled: "${jk.menus.enviroment.args.clientOnly}"
        }, {
                spacer: true,
                disabled: "${jk.menus.enviroment.args.clientOnly}"
        }, {
                title: "Build client", // Sestavení klienta.
                shortcuts: "c",
                disabled: "${!jk.menus.enviroment.env.outDir}",
                execute: function(ctx) {
                        cli.exec(jk.menus.enviroment.updateSpec + " -Dupdate-only=client");
                }
        }, {
                title: "Build client production pack", // Sestavení balíčku do jednoho HTML souboru.
                shortcuts: "cp",
                disabled: "${!jk.menus.enviroment.env.outDir}",
                execute: function(ctx) {
                        cli.exec(jk.menus.enviroment.updateSpec + " -Dupdate-only=client -
Dproduction=true");
                },
                spacer: true
        }, {
                title: "View in Firefox", // Zobrazit klienta ve Firefoxu
                shortcuts: "f",
                disabled: "${!jk.menus.enviroment.env.outDir}",
                execute: function(ctx) {
                        cli.exec("setsid firefox file://" + jk.menus.enviroment.clientUrl + " &", true);
                }
        }, {
                title: "View in Chrome", // Zobrazit klienta v Chromu
                shortcuts: "ch",
                disabled: "${!jk.menus.enviroment.env.outDir}",
                execute: function(ctx) {
                        cli.exec("setsid chromium-browser file://" + jk.menus.enviroment.clientUrl +
" &", true);
                }
        }, {
                title: "Reload", // Přenačtení JavaScript enginu zavaděče.
                shortcuts: "re",
                disabled: "${jk.menus.enviroment.args.clientOnly}",
                execute: function(ctx) {
                        cli.exec(ctx.jkaliNode + " " + ctx.jkaliJS + " reload " + ctx.jkaliHome +
"/projects/enviroments.js " + jk.menus.enviroment.env.name);
                }
        }, {
                title: "Availability from now [+ <some minutes>] on local", // Nastavení informace o
nedostupnosti JavaScript enginu. Zavaděč funguje, ale po nějaký čas vrací informaci o
```

```
nedostupnosti serveru. Třeba pokud probíhají nějaké změny na databázi.
                shortcuts: "a",
                displayShortcuts: "a[f]",
                disabled: "${jk.menus.enviroment.args.clientOnly}",
                execute: function(ctx) {
                        cli.exec(ctx.jkaliNode + " " + ctx.jkaliJS + " availabilityFrom " +
ctx.jkaliHome + "/projects/enviroments.js " + jk.menus.enviroment.env.name);
                }
        }, {
                shortcuts: "af", // Tato položka funguje, ale nemá viditelný titulek. Je to rozšíření
akce výše jen o přičtený čas.
                disabled: "${jk.menus.enviroment.args.clientOnly}",
                execute: function(ctx) {
                        ctx.terminalExec("echo -n \"How minutes: \"; read outagemins;" +
                                ctx.jkaliNode + " " + ctx.jkaliJS + " availabilityFrom " +
ctx.jkaliHome + "/projects/enviroments.js " + jk.menus.enviroment.env.name + " \"`date -
d \"$outagemins minutes\" +'%Y-%m-%d %k:%M:00.000'`\";" +
                                "echo -n \"Press enter to close\"; read key");
                }
        }],

        // Zase nějaké nastavení kontextových proměnných tentokrát včetně kontroly, zda prostředí
vůbec existuje.
        setup: function(ctx) {
                var envName = this.args.envName;

                if (!jk.enviroments[envName]) {
                        throw jk.errorMessage(format("Enviroment '%s' not exists.", envName));
                }

                this.env = jk.enviroments[envName];
                this.updateSpec = ctx.jkaliNode + " " + ctx.jkaliJS + " update " + ctx.jkaliHome +
"/projects/enviroments.js " +
                        this.env.name + (this.args.systemArgs ? " " + this.args.systemArgs : "");
                this.clientUrl = getExpr(this.env.outDir, jk.variables) + "/" +
getExpr(this.env.outFile, jk.variables);
        }
});});
```

JKali nabídky se ovládají celkem příjemně a má to vysokou přehlednost a snadnou přenositelnost až na to používání systémových příkazů. Všechny tyto nabídky jsou v souboru menu.js. Akce nabídek je možné vyvolávat i systémovým příkazem, kde zadáte cestu k akci skrze jednotlivé podnabídky. Využití je i pro systémové programování. Hlavně máte většinu v JavaScriptu a přehledně. Nač používat nějaký další systémový skript? Proč to zbytečně komplikovat?

Závěr

Jak už jsem psal na začátku. JKF je něco jako nástroj, který by mohl nadefinovat standard blízký UML a přístupu k vývoji informačních systému budoucnosti. Řekl bych, že vyniká díky synergii a vysoké přehlednosti v několika směrech. I co se týče zátěže serveru, tak je na tom myslím velmi dobře a má moderní animovaný vzhled.

Je to JavaScript a ne kompilovaný jazyk. Nevýhoda je v tom, že vám editor neprozradí chyby ještě před kompilaci. To ne, ale JKF je velice stručný a to nedává příliš prostoru vývojáři vytvářet velké množství chyb. Navíc ten systém testování je také velké urychlení. Editor vám chybu možná neukáže hned, ale spustíte test a přijdete na to i tak celkem rychle.

Silnou stránkou je i jeho napojitelnost. Používáte to, co chcete používat. Pokud vás zajímá jen klient a to co s ním souvisí, uděláte si pomocí komplexních formulářů to, co se týká GUI a potom na straně serveru už nemusíte SQL z JKF volat. Budete používat vlastní libovolné knihovny v hostitelské aplikaci. Správu datového modelu v JKF odříznete. Testy také nemusíte používat. Pokud vás nezajímá klient, ale líbí se vám serverová část. Můžete používat jen serverovou část a klienta si napojíte nějakého vlastního který bude posílat JSON requesty na rozhraní zavaděče. Ta napojitelnost tu je myslím celkem vysoká.

Uvolnění zdrojových kódů pro veřejnost je podmíněné tím, že bych chtěl vybrat finance na pokrytí nákladů na vývoj projektu do stávající podoby. Tím myslím vývoj JKF včetně ukázek, ale ne projektu Eden, co je v něm napsaný. Cením si to na **800 000 Kč** po zdanění za uvolnění kódů, ale celkově pokud má být dotační částka minimálně 1 000 000 Kč, což převyšuje cenu v četně 15% zdanění, tak bych to přijal se závazkem, že dokončím na projektu ještě nějaké věci. S tím i počítám, že to budu dále rozvíjet. Mohla by to být i dobrá cesta, jak založit komunitní projekt, který si najde svojí komunitu, co ho bude vyvíjet dále.