VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYTVÁŘENÍ UMĚLÝCH DAT PRO TESTOVÁNÍ WEBOVÝCH APLIKACÍ

WEB APPLICATION TESTING WITH MOCKED DATA

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE — **Bc. TOMÁŠ BRUCKNER**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20194/2016/xbruck02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Bruckner Tomáš, Bc.**
Obor: Management a informační technologie
Téma: **Vytváření umělých dat pro testování webových aplikací**
Web Application Testing with Mocked Data
Kategorie: Počítačové sítě

Pokyny:

1. Analyzujte nástroje pro vytváření umělých dat pro testování aplikací se zaměřením na protokol OData.
2. Navrhněte knihovnu implementující CRUD operace nad umělými daty.
3. Implementujte navrženou knihovnu.
4. Otestujte správnost implementace a diskutujte výsledky.

Literatura:

- Lay, Steve. 2013. What is OData, and why is it important? | Getting Results -- The Questionmark Blog. *Questionmark blog*.
- Cupek, R. & Huczala, L., 2015. OData for service-oriented business applications: Comparative analysis of communication technologies for flexible Service-Oriented IT architectures. In *Industrial Technology (ICIT), 2015 IEEE International Conference on*. pp. 1538-1543.
- Handl, R., Pizzo, M. & Biamonte, M., 2014. OData JSON Format Version 4.0. *OASIS Standard*.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pluskal Jan, Ing.**, UIFS FIT VUT
Konzultant: Vopěnka Václav, Ing., UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá vytvářením umělých dat pro aplikace, které využívají REST rozhraní ke komunikaci mezi klientskou a serverovou částí. Z různých implementací REST rozhraní je práce zaměřena pouze na standard OData. Samotná práce je pod záštitou společnosti SAP, jejichž nástroje jsou použity i při vývoji výsledného řešení. Jedná se především o JavaScript framework SAPUI5. Přínosem této práce je vytvoření knihovny, která má za cíl usnadnit vývoj klientské části webové aplikace. Plně podporuje CRUD operace nad OData voláními. Oproti jiným knihovnám vytvářející umělá data nevrací vždy stejná statická data, nýbrž simuluje chování serverové části. Tedy při zavolání metody DELETE nad konkrétní entitou se daná entita opravdu smaže. Tato funkcionalita je umožněna tím, že se na klientské straně vytvoří databáze přímo v internetovém prohlížeči, která odpovídá databázi na straně serveru. Obdobná knihovna pro OData protokol zatím neexistuje, jedná se tedy o unikátní řešení. V rámci diplomové práce byla provedena validace knihovny na demonstrační aplikaci a výkonnostní analýza výsledného řešení.

Abstract

This work deals with creating and providing mocked data for applications that use REST interface to communicate between the client and server parts. From the various implementations of the REST interface, the work focuses only on OData standard. The project itself is mainly for SAP company. Naturally, even the libraries that are used in the final solution are from SAP. Primaly JavaScript framework SAPUI5 is used. The merit of this work is a library that facilitates the development of the client side of web applications. It fully supports CRUD operations over OData calls. Compared to other libraries creating mocked data that always return the same static data, this one simulates the behavior of the real server. So, when DELETE method is called for a specific entity, the given entity is deleted. This functionality is enabled by the client-side database created directly in the web browser, which corresponds to the database on the server side. A similar library for OData protocol does not exist, so it is a unique solution. The solution is verified using prepared web application.

Klíčová slova

REST, OData, umělá data, JavaScript, SAP, testování

Keywords

REST, OData, mock data, JavaScript, SAP, testing

Citace

BRUCKNER, Tomáš. *Vytváření umělých dat pro testování webových aplikací.* Brno, 2017. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Pluskal Jan.

Vytváření umělých dat pro testování webových aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Pluskala. Další informace mi poskytl Ing. Václav Vopěnka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Bruckner
22. května 2017

Poděkování

Chtěl bych poděkovat svým rodičům, kteří mi umožnili studovat na vysoké škole, a kteří mne podporovali psychicky i finančně. Dále bych chtěl poděkovat Ing. Janu Pluskalovi za vedení diplomové práce a Ing. Václavu Vopěnkovi za odbornou pomoc v průběhu vypracování celé diplomové práce.

Obsah

Kapitola 1

Úvod

V testování je síla. Vývoj informačních systémů zažil za posledních deset let obrovský pokrok. A právě z historie se dokážeme nejlépe poučit. Právě ona nám připomíná, jakým způsobem není vhodné vytvářet komplexnější řešení v IT odvětví. Krušná byla doba temna, kdy softwarový vývoj probíhal pomocí metodologie *Waterfall*. A právě tento způsob vývoje by splňoval pravidlo Occamovi břitvy. To nám přeneseně říká, že první řešení, které nás napadne, je často správné. Ve skutečném prostředí však tento model silně pokulhává, i když je tak přirozený.

Vytvořit požadavky, navrhnout řešení, implementovat dané řešení, otestovat a dát do produkce. Co by se mohlo nezdařit? Právě jasnost jednotlivých fází a nemožnost se mezi libovolně pohybovat byla záhuba pro mnohé IT projekty. Dnes už jsme chytřejší. Výhody agilního vývoje jsou již nezpochybnitelné a ověřené praxí. Mezi dnes nejpopulárnější metodologie patří Scrum, který jde ruku v ruce s TDD, nebo-li programování řízené testy[1].

Testování bude rovněž základem následující diplomové práce. Zaměříme se na informační systémy (IS), které jsou založeny na třívrstvé architektuře. Konkrétně na IS, kde klientská a serverová část spolu komunikují pomocí standardu OData[2]. Jedná se o způsob implementace REST aplikačního rozhraní, který je využíván moderními IS založenými na servisně orientované architektuře. Více v kapitole 3.

Cílem této práce bylo vytvořit knihovnu, která bude odstiňovat klientskou část od serverové. OData volání jsou knihovnou odchytána a vrací se data umělá. Rozdíl oproti jiným knihovnám a zároveň hlavní přínos práce je v tom, že nevrací pouze staticky definovaná data. Nýbrž se vytvoří vlastní klientská databáze, která plně podporuje všechny CRUD[3] operace viz kapitola 6. Nástroj by měl sloužit především pro vývojáře klientské části při vývoji, kdy serverová část ještě není implementována. Hlavní výhoda je tedy ve zvýšení nezávislosti na vývojářích serverové části. Pro oba týmy bude dostačující, když si definují rozhraní, kterým budou komunikovat.

Samotná práce je pak pod záštitou společnosti SAP. Samotný vývoj v této společnosti je rozdělen na spoustu menších aplikací, které dohromady fungují jako progresivní webová aplikace[4]. Aplikace jsou plně responzivní (adaptují vzhled na základě platformy, na které je aplikace puštěna). Vývoj v SAP probíhá v knihovně SAPUI5, která je také použita pro vývoj této diplomové práce. Více v podkapitole 3.4.

[1] V angličtině Test-driven development.
[2] Open Data Protocol.
[3] Z anglického Create, Read, Update a Delete.
[4] Více informací např. zde https://developers.google.com/web/progressive-web-apps.

Kapitola 2

Testování pomocí umělých dat

V této kapitole budou představeny nejpopulárnější nástroje pro vytváření umělých dat. Hlavní zaměření bude na nástroje využívané společností SAP. Kapitola by měla sloužit jako přehled již existujících řešení se zaměřením na jejich možnosti a limitace.

Samotný server pro vytváření umělých dat by měl splňovat tyto požadavky:

1. Automaticky vygenerovat umělá data bez nutnosti je definovat.
2. Automaticky zanalyzovat API bez nutnosti ho zvlášť definovat mimo soubor *metadata.xml* viz kapitola 3.
3. Na základě analýzy z bodu 2 odchytávat specifikované požadavky a vracet umělá data.

2.1 Nástroj Apiary

Mezi nejpopulárnější servery pro vytváření umělých odpovědí na HTTP žádosti patří Apiary. Celý projekt vznikl jako myšlenka na prvním hackathonu NodeJS[1]. Lidé projevili silnou pozitivní odezvu, projekt tedy pokračoval, až se z něj stal základ samostatné společnosti Apiary Inc. Ta byla 10. února 2017 koupena firmou Oracle za neznámou částku.

Z technického hlediska se jedná o jednoduchý server, který reaguje na HTTP a HTTPS volání. Vývojář definuje URL služby a HTTP metodu, na které má server naslouchá. Také je nutné definovat návratový kód a data.

Zdrojový kód 1 ukazuje příklad definice služby. Jednotlivé entity se definují ve formátu 1A[2], který nahradil původní implicitní formát. Na *řádku 3* je definice URI, na kterou bude server odpovídat. Na *řádku 5* je pak definovaná HTTP metoda, která bude povolena pro danou URL. Samotná data, která Apiary server vrátí, jsou popsána ve formátu JSON.

Apiary je možné provozovat ve třech režimech. Prvním z nich je využívat server umístěný na straně Apiary, který je poskytován i ve verzi zdarma. Druhým způsobem je využít placenou verzi, která umožňuje zakládat soukromé specifikace API, více uživatelů atd.[3] Poslední možností je provozovat si server na vlastním zařízení, což přináší ve většině případů rychlejší odezvy a stabilitu spojení.

[1] Dostupné z: https://www.webexpo.cz/praha2017/prednaska/from-zero-to-profit-apiary-s-start up-lessons-learned/.

[2] Dostupné z: https://github.com/apiaryio/api-blueprint/blob/master/APIBlueprintSpecificat ion.md.

[3] Dostupné z: https://apiary.io/plans.

Nástroj Apiary je mocný, nicméně má svoje omezení. Je u něj sice možné definovat odezvu na volání HTTP metody POST. Bohužel však nevytváří nová data do databáze, ale vždy vrací předem definovaná umělá data. Nelze tedy hovořit o plné podpoře CRUD operací[4]. Zároveň se jedná o obecný server. Lze ho tedy použít i pro webové aplikace využívající *OData* protokol. Není však navržen přímo pro *OData* protokol, takže si neumí vygenerovat sám. Nesplňuje tedy požadavky 2 a 3 definované výše v této kapitole.

```
FORMAT: 1A

## Users Collection [/users]

### List All Users [GET]

+ Response 200 (application/json)

{
    "users": [
        {
            "id": 1,
            "name": "Tom"
        }, {
            "id": 2,
            "name": "Vaclav"
        }
    ]
}
```

Zdrojový kód 1: Základní práce s lokálním úložištěm.

2.2 Knihovna OPA5

OPA5 (One Page Acceptance Tests) je součást SAPUI5 frameworku (více o SAPUI5 v podkapitole 3.4). Jedná se o knihovnu zaměřenou na integrační testování aplikací. Umožňuje testovat uživatelskou interakci, navigaci mezi stránkami či datovou vazbu[15]. Zdrojový kód 2 ukazuje základní strukturu OPA5 testu.

[4]Dostupné z: https://apiary.io/how-apiary-works.

```
jQuery.sap.require("sap.ui.test.Opa5");
jQuery.sap.require("sap.ui.test.opaQunit");

opaTest("Test for pressing the button", function (Given, When, Then) {
    // příprava
    Given.iStartMyApp();

    // akce
    When.iPressTheButton();

    // aserce
    Then.theButtonShouldChangeText();
});
```

Zdrojový kód 2: OPA5 test.

Životní cyklus OPA5 testu je rozdělen do tří fází:

- Příprava,
- akce,
- aserce.

Příprava je fáze, ve které je definováno prostředí testu. Je zde určen počáteční bod testu. Zdrojový kód 3 ukazuje vytvoření metody *iStartMyApp*. Ta vytvoří otevření prohlížeče s rámcem, ve kterém je spuštěna webová aplikace definovaná souborem *index.html*.

```
var arrangements = new sap.ui.test.Opa5({
    iStartMyApp: function () {
        return this.iStartMyAppInAFrame("index.html");
    }
});
```

Zdrojový kód 3: Přípravná fáze v OPA5.

Akce je druhá fáze v životním cyklu OPA5 testu. V této fázi je proveden úkon, který má být otestován integračním testem. Zdrojový kód 4 zobrazuje vytvoření metody *iPressTheButton*. Ta provede zmáčknutí tlačítka s identifikátorem *idButton*. Pokud není tlačítko na stránce nalezeno, tak je vypsán text v atributu *errorMessage*.

```
var actions = new sap.ui.test.Opa5({
    iPressTheButton: function () {
        return this.waitFor({
            viewName: "MainPage",
            id: "idButton",
            actions: new sap.ui.test.Press(),
            errorMessage: "Unable to find the button."
        });
    }
});
```

Zdrojový kód 4: Akce v OPA5.

Aserce je poslední fáze v životním cyklu OPA5 testu. V této fázi jsou ověřeny výsledky z akční fáze. Musí odpovídat předem definovaným hodnotám. Zdrojový kód 5 zobrazuje vytvoření metody *theButtonShouldChangeText*. Ta ověří, že tlačítko s identifikátorem *idButton* má definovanou textovou hodnotu *Pressed!*. V pozitivním případě je provedena metoda *success*. V opačném případě je vytisknuta zpráva definována atributem *errorMessage*.

```
var assertions = new sap.ui.test.Opa5({
    theButtonShouldChangeText: function () {
        return this.waitFor({
            viewName: "MainPage",
            id: "idButton",
            matchers: new sap.ui.test.matchers.PropertyStrictEquals({
                name: "text",
                value: "Pressed!"
            }),
            success: function (oButton) {
                Opa5.assert.ok(true, "New text:" + oButton.getText());
            },
            errorMessage: "The button's text did not change"
        });
    }
});
```

Zdrojový kód 5: Aserce v OPA5.

2.3 Knihovna Sinon.JS

Tato kapitola vychází z oficiální dokumentace[5]. Sinon.JS je pomocná JavaScript knihovna pro jednotkové testy v prostředí SAPUI5. Umožňuje nahrazovat metody objektů umělými. Také umožňuje několik způsobů odchycení *AJAX* požadavků na server.

Tyto způsoby jsou:

[5] Dostupné z: http://sinonjs.org/docs.

- Nahrazení *ajax* metody.
- Použití falešného požadavku.
- Použití falešného serveru.

Zdrojový kód 6 ukazuje příklad nahrazení *ajax* metody. Na *řádku 2* je metoda *getUser*. Ta provádí dotaz na server pomocí *ajax* metody. V případě úspěšného dotazu na server je provedena metoda *success*. Test začíná na *řádku 12*. Metoda *after* je provedena po uskutečnění testu a způsobí navrácení *ajax* metody do původního stavu. Na *řádku 16* začíná samotný test. Metoda *stub(object,"method")* nahradí metodu specifikovaného objektu uměle vytvořenou metodou. Ta následně udržuje informaci o tom, kolikrát byla volána či s jakými parametry. Na *řádku 18* je použita metoda *spy()*. Tato metoda vytvoří anonymní funkci, která funguje podobně jako metoda *stub()*. Na *řádku 20* metoda *calledWithMatch(object)* zkontroluje, zda byla *ajax* metoda provolána s očekávanými argumentem.

```
1  // funkce k otestování
2  function getUser(userID, handleData) {
3      jQuery.ajax({
4          url: "/user/" + userID,
5          success: function (data) {
6              handleData(data);
7          }
8      });
9  }
10
11 // test
12 after(function () {
13     jQuery.ajax.restore();
14 });
15
16 it("Get user 1337", function () {
17     sinon.stub(jQuery, "ajax");
18     getUser(1337, sinon.spy());
19
20     assert(jQuery.ajax.calledWithMatch({ url: "/user/1337" }));
21 });
```

Zdrojový kód 6: Náhrada ajax metody v Sinon.JS.

Zdrojový kód 7 ukazuje pokročilejší variantu odchytávání dotazů pomocí technologie *AJAX*. Knihovna *Sinon.JS* nahradí objekt *XMLHttpRequest* svým falešným objektem. *XMLHttpRequest* objekt slouží právě k asynchronní komunikaci se serverem ve stejné doméně. Je to základ technologie *AJAX*. Po vytvoření umělého objektu lze pak odchytávat volání metody *ajax*.

Na *řádku 4* je vytořen před začátkem testu falešný *XMLHttpRequest* objekt. Na *řádku 6* je zaregistrována metoda, která zpracuje serverové volání. V tomto příkladu je požadavek uložen do pole *requests*. Na *řádku 12* je obnoveno původní chování objektu *XMLHttpRequest* po ukončení testu.

Samotný test je na *řádku 15*. Na *řádku 18* je zkontrolován celkový počet požadavků na server. Na *řádku 19* je zkontrolována URL volání. Při prvním selhání aserce test končí a jsou vytisknuty podrobné informace o chybě. I při selhání je zavolána metoda *after*.

```
1  var xhr, requests;
2
3  before(function () {
4      xhr = sinon.useFakeXMLHttpRequest();
5      requests = [];
6      xhr.onCreate = function (req) {
7          requests.push(req);
8      };
9  });
10
11 after(function () {
12     xhr.restore();
13 });
14
15 it("Get user 1337", function () {
16     getUser(1337, sinon.spy());
17
18     assert.equals(requests.length, 1);
19     assert.match(requests[0].url, "/user/1337");
20 });
```

Zdrojový kód 7: Falešný požadavek v Sinon.JS.

Zdrojový kód 8 zobrazuje příklad vytvoření celého umělého serveru. Na *řádku 4* je vytvořen umělý server voláním *fakeServer.create(options)*. Při vytváření je použito nastavení *respondImmediately*, které automaticky a synchronně vrátí odpověď na uživatelský dotaz. Server funguje na principu definování žádostí a odpovědí. Je nutné mu definovat na jakou HTTP metodu a jakou URL má reagovat. Také je mu pro každé volání potřeba definovat požadovanou odpověď. Ta je složena z návratového kódu, hlavičky a těla odpovědi.

Umělému serveru lze definovat URL žádosti buď na základě řetězcové konstanty nebo regulárního výrazu. To umožňuje vysokou flexibilitu při definování žádostí a odpovědí. Při výsledném řešení diplomové práce bude použito odchycení serverových volání pomocí umělého serveru. Tomu bude potřeba nadefinovat všechny možné typy žádostí a patřičné odpovědi. Více v kapitole 3.

```
var server;

before(function () {
    server = sinon.fakeServer.create({respondImmediately: true});
});

after(function () {
    server.restore();
});

it("Get user 1337", function () {
    this.server.respondWith("GET", "/user/1337",
    [200, { "Content-Type": "application/json" },
    '{ id: 1337, name: "Tom" }']);

    var handleData = sinon.spy();
    getUser(1337, handleData);

    assert(handleData.calledOnce);
});
```

Zdrojový kód 8: Falešný server v Sinon.JS.

Kapitola 3

Protokol OData

Tato kapitola vychází z oficiální dokumentace[1]. Open Data Protocol (OData) je jedna z možných implementací REST rozhraní. Na tomto protokolu začala pracovat společnost Microsoft v roce 2007. Nyní již existuje čtvrtá verze, která byla standardizována společností OASIS.

Motivace k vytvoření tohoto protokolu s rostoucí popularitou REST rozhraní. Definovat pro každý projekt své vlastní REST rozhraní je zbytečné. Výhodnější je mít dobře zdokumentovanou a standardizovanou implementaci. Tím se zamezí předcházením chybám ze špatně navrženého REST rozhraní. Vývojářům to také zjednoduší práci, jelikož nebudou muset samotné REST rozhraní vymýšlet, ale již použijí ověřený standard. Na projektu se podílely i další společnosti, mimo jiné i SAP a IBM.

OData je právě ten protokol, který se snaží usnadnit a sjednotit komunikaci mezi aplikacemi, ať už běží u klienta, ve webovém prohlížeči, na cloudu či si ji uživatel pouští v telefonu. Základem protokolu jsou HTTP/HTTPS volání. Samotný obsah je pak předávám buď ve formátu JSON, XML nebo Atom. Méně známý je formát Atom[8], který vychází z formátu XML. Je kombinací dalších dvou standardů Atom Syndication Format (ASF) a Atom Publishing Protocol (APP). ASF definuje formát, v jakém jsou data službami poskytována[10]. APP definuje způsob, jakým se data můžou od poskytovatele číst a udržovat[5].

OData protokol podporuje tyto HTTP metody:

- GET - čtení entity či kolekce entit.
- POST - vytvoření nové entity.
- PUT - úprava existující entity přepsáním celé entity.
- PATCH - úprava existující entity přepsáním jen části entity.
- DELETE - smazání entity.

OData protokol je velmi obsáhlý, a proto zde bude uveden jenom výběr těch nejdůležitějších částí standardu. Ty budou sloužit jako základ pro implementaci viz kapitola 6.

3.1 OData parametry

OData protokol podporuje přidání volitelných parametrů nad dotazy. Tyto parametry upravují návratová data již na straně serveru. Respektive předdefinovaným způsobem modifikují

[1] Dostupné z: http://www.odata.org/documentation

formát odpovědi, která přijde na klientský dotaz. Většinou se jedná o operace nad kolekcí dat, nikoliv nad konkrétní entitou. Lze definovat i vlastní parametry.

Mezi operace patří například:

- $filter - filtruje hodnoty dle zadaného logického výrazu.
- $top - vrátí jen prvních N počet záznamů.
- $skip - vynechá prvních N počet záznamů.
- $select - vrátí jenom specifikované atributy entity.
- $orderby - seřadí hodnoty sestupně/vzestupně podle zadaného pole atributů.
- $format - nastaví typ formátu dat odpovědi (JSON, XML, Atom).
- $expand - nahradí odkaz zanořeného atributu skutečnou hodnotou.

3.2 Metadata dokument

Metadata je XML soubor popisující datový model pro OData. Jedná se o nejdůležitější stavební kámen celého systému a je naprosto esenciální pro fungování OData protokolu. Význam jednotlivých částí bude vysvětlen níže. Zdrojový kód 9 ukazuje příklad obsahu souboru metadat. Na *řádku 16* je definován *EntityContainer*, který by měl být v metadatech právě jeden. Zde jsou definovány všechny kolekce, funkce atp. Lze si to představit, že věci mimo *EntityContainer* pouze popisují struktury, ale obsah kontejneru jsou již reálná data, nad kterými budou vykonávány jednotlivá volání. Na *řádku 24* je entita *Annotations*, která většinou obohacuje již definované kolekce či entitní typy. V tomto případě je obohacen přímo celý kontejner o jednoduchý popis.

3.3 Strukturované typy

V této sekci se rozeberou jednotlivé struktury. Pro názornost jsou v této podkapitole uvedeny i ukázky jednotlivých struktur. Ty mají sloužit především k lepšímu pochopení filozofie a implementace REST rozhraní pomocí protokolu OData. Demonstrativní ukázka izolovaného použití často vede k jednoduššímu pochopení celého konceptu.

Datové entity jsou v OData definovány pomocí struktur *EntityType* a kolekcí *EntitySet*. Struktury *EntityType* jsou složeny z $0..N$ atributů. Jednotlivé atributy pak můžou být čtyř typů:

- Primitivní typ - jednoduché datové typy.
- Komplexní typ - struktura složená z primitivních i komplexních typů.
- Enumerační typ - číselník hodnot.
- Kolekce - kolekce primitivního, komplexního nebo enumeračního typu.

EntitySet kolekce má přiřazený jeden *EntityType*. Jedná se tedy o homogenní kolekci.

```xml
<edmx:Edmx xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
    Version="1.0">
    <edmx:DataServices
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    m:DataServiceVersion="4.0" m:MaxDataServiceVersion="4.0">
        <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"
        Namespace="ODataTest">
            <EntityType Name="Product">
                <Key>
                    <PropertyRef Name="ID"/>
                </Key>
                <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
                <Property Name="Name" Type="Edm.String"/>
                <Property Name="Description" Type="Edm.String"/>
            </EntityType>
            <EntityContainer Name="TestService">
                <EntitySet Name="Products" EntityType="ODataTest.Product"/>
            </EntityContainer>
            <Annotations Target="ODataTest.TestService">
                <ValueAnnotation Term="Org.OData.Display.V1.Description"
            String="Test Description"/>
            </Annotations>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>
```

Zdrojový kód 9: Metadata v OData.

Primitivní typ

Základním stavebním kamenem entit jsou primitivní typy. Ty jsou obdoba atomický hodnot v programovacích jazycích. Různé typy primitiv obsahují různé speciální atributy zvané *Facets*. Patří mezi ně maximální délka řetězce, přesnost u čísel v plovoucí čárce, zda je povolená *null* hodnota a další. Také lze definovat další vlastní atributy.

Demonstrativní výpis některých primitivních typů:

- Edm.Binary - binární data.
- Edm.Boolean - 1-bitová hodnota.
- Edm.Byte - 8-bitová hodnota neznaménkově.
- Edm.DateTime - datum bez offsetu časové zóny.
- Edm.DateTimeOffset - datum s offsetem časové zóny.
- Edm.Decimal - číslo v plovoucí řadové čárce s přesností na tři desetinná místa.
- Edm.Double - 64-bitová hodnota v plovoucí čárce.

- Edm.Guid - 128-bitová unikátní hodnota.
- Edm.Int16 - 16-bitová hodnota znaménkově.
- Edm.Int32 - 32-bitová hodnota znaménkově.
- Edm.Int64 - 64-bitová hodnota znaménkově.
- Edm.SByte - 8-bitová hodnota znaménkově.
- Edm.String - sekvence UTF-8 znaků.
- Edm.Time - hodinový čas od 00:00:00 do 23:59:59 ve formátu ISO 8601.

Komplexní typ

Komplexní typy umožňují definovat složitější struktury, na které se pak entity můžou přímo odkazovat. Usnadňují znovupoužitelnost a přehlednost v datovém modelu. Komplexní typ je identifikován pomocí atributu *Name*, který musí být unikátní v celém datovém modelu. Zdrojový kód 10 ukazuje příklad definice komplexního typu. Entita *Human* obsahuje jeden primitivní atribut *FullName* a komplexní atribut *HumanSizes* typu *Sizes*. Ten obsahuje dva primitivní atributy *Weight* a *Height* typu *Edm.Double*.

```
<ComplexType Name="Sizes">
    <Property Name="Weight" Type="Edm.Double" />
    <Property Name="Height" Type="Edm.Double" />
</ComplexType>

<EntityType Name="Human">
    <Property Name="FullName" Type="Edm.String" />
    <Property Name="HumanSizes" Type="Self.Sizes" />
</EntityType>
```

Zdrojový kód 10: Komplexní typ v OData.

Enumerační typ

Enumerační typ umožňuje definování číselníku hodnot ve tvaru klíč-hodnota. Samotná hodnota musí být nějakého primitivního typu. Jména musí být unikátní. Tento typ není společností SAP zatím podporován. Všechny číselníky jsou řešeny na úrovni databáze, nikoliv na úrovni samotného rozhraní. Na úrovni rozhraní je číselník typicky implementován jako příslušný *EntitySet* s odpovídajícím *EntityType* typem. Patří však ke klíčovým prvkům protokolu OData. Také je možnost, že do budoucna bude enumerační typ podporován. Z těchto důvodu je zde tento typ uveden.

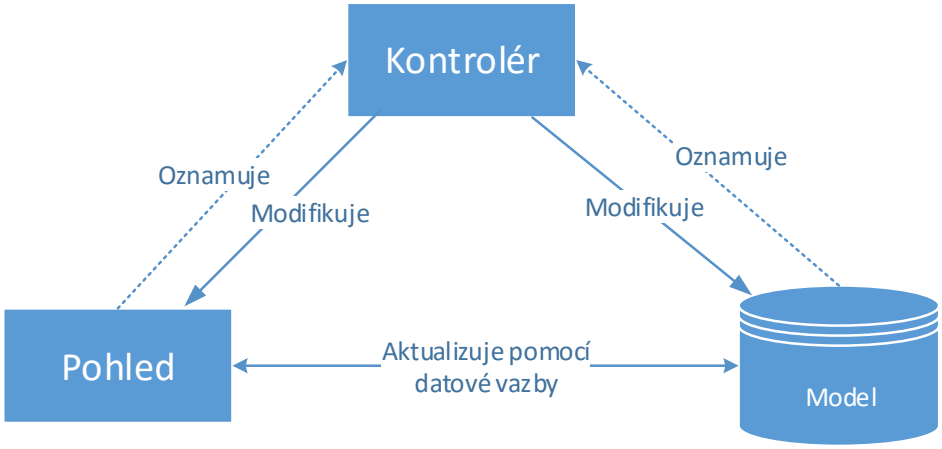
```
<EnumType Name="Colors" UnderlyingType="Edm.Int32">
    <Member Name="Blue"   Value="1" />
    <Member Name="Green"  Value="42" />
    <Member Name="Red" Value="1337" />
</EnumType>
```

Zdrojový kód 11: Enumerační typ v OData.

3.4 Knihovna SAPUI5

Tato podkapitola vychází z oficiální dokumentace[2]. OData protokol je implementován v různých knihovnách. V této práci bude představena JavaScript knihovna SAPUI5 od společnosti SAP[3]. V této knihovně bude implementována demonstrační aplikace pro verifikaci výsledného řešení.

SAPUI5 knihovna je založena na HTML5, jQuery a Model-Pohled-Kontrolér[4] návrhovém vzoru viz obrázek 3.1.



Obrázek 3.1: Model-Pohled-Kontrolér architektura.

Modelů zde může být několik. Existuje vždy jeden základní, který je reprezentací OData metadata souboru. Ten popisuje jednotlivé entity na straně serveru. Dále zde můžou být XML a JSON modely, které lze použít na datovou vazbu do pohledu. Lze tím ovlivňovat hodnoty elementů, jejich atributy (viditelnost, velikost, barvu atd.) či překlady. OData model reprezentuje data ze serveru. JSON a XML model reprezentuje data na klientovi. U modelů existují tři módy datové vazby:

[2] Dostupné z: https://sapui5.hana.ondemand.com.
[3] http://www.sap.com/index.html
[4] V angličtině Model-View-Controller.

- Oboustranná vazba[5] - změna v modelu je přepsána do pohledu a obráceně.
- Jednostranná vazba[6] - změna v modeluje přepsána do pohledu, nikoliv obráceně.
- Vazba jedenkrát[7] - hodnota v modelu je přepsána do pohledu pouze jednou, následná změna se neprojeví.

Pohled může být v SAPUI5 reprezentováno několika způsoby. Buď je možné definovat pohled přímo v JavaScriptu vytvořením instancí elementů. Ten je následně přidán do odpovídajícího místa ve stránce. Nebo lze použít tři způsoby definování šablony, která je použita jako předpis pro vzhled stránky. Šablona může být ve formátu XML, JSON nebo klasicky v HTML.

Kontrolér musí obsahovat každá stránka v SAPUI5. V něm se děje všechna logika na klientské straně aplikace. Kontrolér obsahuje čtyři základní metody, které reprezentují životní cyklus stránky:

- OnInit() - metoda, která je zavolána při vytvoření stránky.
- OnExit() - metoda, která je zavolána při zničení stránky.
- OnBeforeRendering() - metoda, která je zavolána před vykreslením stránky.
- OnAfterRendering() - metoda, která je zavolána při vykreslení stránky.

SAPUI5 přímo implementuje funkce pro zpracování OData volání, čímž značně usnadňuje práci programátorům. Ti si nemusí metody pro práci s OData protokolem implementovat sami. Další výhodou je, že všechny OData volání odchází na server z jednoho bodu. Je jím *XMLHttpRequest* objekt, který je základem technologie *AJAX*. Pro odchycení všech OData volání bude stačit nahradit daný *XMLHttpRequest* objekt uměle vytvořeným objektem. Ten nebude odesílat žádosti na server, ale pouze provede zachycené a náhradu uměle vytvořenou odpovědí. Princip je popsán v podkapitole 2.3.

[5] V angličtině two-way binding.
[6] V angličtině one-way binding.
[7] V angličtině one-time binding.

Kapitola 4

Typy webových databází

V této kapitole budou rozebrány možné způsoby uložení dat na straně uživatele přímo v internetovém prohlížeči. Konkrétně budou představeny technologie nativní. Charakter diplomové práce nedovoluje používat knihovny třetích stran. Především kvůli tomu, že výsledná implementace má být zaintegrována do technologií společnosti SAP. Ta má složitý proces schvalování knihoven třetích stran do svého produktu. Pravděpodobně by tedy nebylo možné integrovat výslednou knihovnu právě kvůli závislostem na knihovnách třetích stran. Proto je nutné použít čistý JavaScript.

4.1 Lokální a relační úložiště

Lokální úložiště[1] a relační úložiště[2] jsou nativně podporovány databáze typu klíč-hodnota. Rozdíl mezi lokálním a relačním úložištěm je, že relační se po skončení relace vymaže. Relace vydrží obnovení stránky. Pokud však stránku otevřeme v novém panelu či okně, pak se uživateli vytvoří nová relace. To je rozdílné oproti způsobu chování relace u cookies. Také je nutné brát v potaz, že pokud uživatel navštíví stránku pomocí protokolu HTTP, a následně ji navštíví přes zašifrovaný protokol HTTPS, tak se data v úložištích nesdílí. To platí pro lokální i relační úložiště. Oba protokoly mají svoje vlastní úložiště [16].

Zdrojový kód 12 ukazuje základní práci s lokálním úložištěm. Klíč a hodnota musí být oba řetězce, což silně limituje použití takového úložiště. Velikost lokálního i relačního úložiště je ve většině prohlížečů 5MB na každý zdroj[3] [14]. Internet Explorer má až 10MB [14]. Pro zjištění zbývajícího místa lze nad objektem úložiště zavolat metodu *reaminingSpace()*, což vrací počet UTF-16 znaků, které lze do úložiště ještě uložit. Pro odstranění všech položek slouží metoda *clear()*.

[1] V angličtině local storage.

[2] V angličtině session storage. V češtině neplést s matematickou relací. Jedná se zde o relaci ve smyslu relační vrstvy referenčního modelu ISO/OSI viz https://cs.wikipedia.org/wiki/Relační_vrstva.

[3] V angličtině origin.

```
var dbh = window.localStorage;
dbh.setItem("key", "value");

var x = dbh.key;  // x = "value"

dbh.removeItem("key");
```

Zdrojový kód 12: Základní práce s lokálním úložištěm.

Pro práci s objekty nejsou lokální a relační úložiště nativně připraveny. Tento nedostatek lze však kompenzovat serializací a deserializací. Zdrojový kód 13 ukazuje příklad serializace jednoduchého objektu pomocí volání statické metody *stringify(object)* nad objektem JSON. Následně pro deserializaci lze použít metodu *parse(string)*.

```
var obj = { foo: 42, bar: 1337 };
var dbh = window.localStorage;

dbh.setItem("key", JSON.stringify(obj));

var x = JSON.parse(dbh.getItem("key")); // x = { foo: 42, bar: 1337 }
```

Zdrojový kód 13: Pokročilá práce s lokálním úložištěm.

Celkově je technologie nejstarší ze zde uvedených. Podporují ji všechny moderní prohlížeče. Je podporována již od osmé verze Internet Exploreru[4]. Největší nevýhoda pro komplexnější použití je pouze naivní implementace úložiště klíč-hodnota.

4.2 Web SQL databáze

Mezi nativní databáze, které jsou podporovány přímo webovými prohlížeči, patří Web SQL databáze. Definici standardu lze nalézt na stránkách organizace World Wide Web Consortium[5]. Web SQL databáze je založena na tradičně relační SQL databázi. Jako svoji specifikaci si tvůrci Web SQL vybrali technologii SQLite[6].

K vytvoření databáze je provedeno volání *openDatabase(name, version [, description, size, callback])*, které vrací vytvořenou databázi. Volání je provedeno nad globálním objektem *window*. Zdrojový kód 14 zobrazuje příklad vytvoření databáze. První dva parametry jsou povinné a fungují jako identifikátor databáze. V případě, že bychom opět zavolali *openDatabase*, tak se již nevytvoří nová databáze, nýbrž je vrácena již v minulosti vytvořená databáze. Pokud bychom chtěli otevřít databázi se stejným jménem ale jinou verzí, došlo by k vytvoření nové databáze. Implicitní velikost databáze je 5MB. Prohlížeče upozorní uživatele, zda povoluje vytvoření větší databáze [13].

[4]Dostupné z: http://caniuse.com/#feat=namevalue-storage.
[5]Dostupné z: https://www.w3.org/TR/webdatabase.
[6]Dostupné z: https://www.sqlite.org/index.html.

```
1 const SIZE = 5 * 1024 * 1024;
2 var db = openDatabase("testdb", "1.0", "description", SIZE, function(db) {
3     alert("DB has been created");
4 });
```

Zdrojový kód 14: Otevření databáze ve WebSQL.

Pro modifikaci Web SQL databáze je použito volání nad objektem databáze *transaction (callback)*. Daný callback pak musí obsahovat jeden parametr, což je objekt, který umožňuje volání metody *executeSql(sqlStatement, placeholders, successHandler, errorHandler)*. Zdrojový kód 15 zobrazuje základní práci s naší testovací databází, která byla vytvořena výše. Na *řádku 4* si vytvoříme tabulku *users* s primárním klíčem *id* a sloupcem *name*. Můžeme si povšimnout, že lze není definice datového typu sloupců, jak je tomu v klasickém SQL prostředí. To je jedna z výhod SQLite. Ta si odvozuje datový typ sama, pokud není explicitně specifikován. Z technického hlediska nemá SQLite datové typy, ale používá třídy skladování[7] [9]. Patří zde *NULL*, *INTEGER*, *REAL*, *TEXT* a *BLOB*.

Na *řádku 5* je ukázka vložení jednoho řádku s pevně definovanými hodnotami. Pro použití parametrizované query se metodě *executeSql* předá jako druhý parametr pole hodnot, které je nahrazeno za otazníky v řetězci SQL příkazu. Ukázka použití parametrizované query je na *řádku 9*.

```
1 var db = window.openDatabase("testdb", "1.0");
2
3 db.transaction(function(tx) {
4   tx.executeSql("CREATE TABLE IF NOT EXISTS users (id PRIMARY KEY, name)");
5   tx.executeSql("INSERT INTO users (id, name) VALUES (0, 'Tom')");
6
7   const id = 1;
8   const name = "Kymbat";
9   tx.executeSql("INSERT INTO users (id, name) VALUES (?, ?)", [id, name]);
10 });
```

Zdrojový kód 15: Transakce ve WebSQL.

Pro čtení z databáze lze použít volání *transaction(callback)*, ale Web SQL nabízí i alternativu. Pokud transakce nebude modifikovat databázi, pak lze zavolat nad objektem databáze metodu *readTransaction(callback)*. Syntaxe volání je stejná jako pro *transaction (callback)*, nicméně je výrazně rychlejší, jelikož je optimalizována pro čtení. Zdrojový kód 16 zobrazuje příklad přečtení všech uživatelů z databáze. Následně se v *successHandler* funkci vytiskne počet nalezených uživatelů. Na *řádku 9* jsou pak vytisknuta jména všech uživatelů. Za zmínku stojí přístup k jednotlivým uživatelům. *Rows* není pole, ale *array-like* objekt. Má tedy atribut *length*, nicméně se k položkám nedá přistupovat jako v klasickém poli, ale musí být použito volání metody *item*.

Velkou nevýhodou této technologie je, že relační SQL přístup k databázi není příliš vhodný pro prostředí JavaScriptu. Nelze použít přímé mapování na objekty, jak je tomu

[7] Anglicky *storage classes*.

například v různých NoSQL databázích. Další nevýhoda je, že použití je velmi nepohodlné a nepřirozené v jazyce JavaScript. Formulace dlouhých řetězců definující jednotlivé select příkazy jsou v dnešní době již minulostí. Zvláště, pokud datový model není vyloženě vhodný pro relační databázi. Je to také důvod, proč roste popularita různých NoSQL databází. Pro menší databáze technologie MongoDB nebo pro větší úložiště technologie Hadoop.

```
var db = window.openDatabase("testdb", "1.0");

db.readTransaction(function(tx) {
    tx.executeSql('SELECT * FROM users', [], function (tx, results) {
        var len = results.rows.length;
        alert (`Found rows ${len}`);

        for (let i = 0; i < len; i++){
            alert(results.rows.item(i).name);
        }
    });
});
```

Zdrojový kód 16: Transakce čtení ve WebSQL.

Tato technologie se však stala velmi kontroverzní, zvláště kvůli odmítnutí tohoto standardu společností Mozilla [1], která zastupuje jeden z nejpopulárnějších webových prohlížečů Firefox[8]. Mozilla se následně dohodla se společností Microsoft, že nebudou danou technologii podporovat [1]. Z tohoto důvodu není Web SQL databáze podporována žádnou verzí webových prohlížečů Firefox, Firefox Android, Internet Explorer, Internet Explorer Mobile a Edge[9].

4.3 Databáze IndexedDB

IndexedDB je nejpokročilejší nativní technologie z pohledu flexibility a typu dat, které lze v této databázi ukládat. Jedná se o NoSQL databázi, která je plně asynchronní. To sebou nese výhody především v tom, že uživatel může dále pokračovat v práci, než jsou mu načtena data z databáze. Nevýhodou může být komplikovanější návrh a implementace práce s databází. Synchronní práce s IndexedDB by však dávala smysl, pokud by databáze byla puštěna v samostatném *WebWorker* procesu. IndexedDB používá na nižší vrstvě různé typy databáze dle internetového prohlížeče. Internet Explorer používá *Extensible Storage Engine*[10], Firefox používá *SQLite*[11] a Chrome používá *LevelDB*[12] [7].

Zdrojový kód 17 ukazuje vytvoření databáze IndexedDB. K vytvoření objektu reprezentující databázi je použito volání *open(name, version)* nad objektem *indexedDB* patřící do globálního prostoru. Nad databází lze následně zaregistrovat tři typy handlerů událostí:

- Onsuccess - provede se, pokud databáze byla vytvořena.

[8] Dostupné z: https://www.mozilla.org/en-US.
[9] Dostupné z: http://caniuse.com/#feat=sql-storage.
[10] Dostupné z: https://msdn.microsoft.com/en-us/library/gg269259(v=exchg.10).aspx.
[11] Dostupné z: http://www.sqlite.org.
[12] Dostupné z: http://leveldb.org.

- Onerror - provede se, pokud se nepodařilo databázi vytvořit.
- Onupgradeneeded - provede se, pokud je při volání *open(name, version)* nastavena vyšší verze databáze, než jaká je poslední verze již vytvořené databáze se stejným jménem.

Při prvním volání je tedy proveden callback na *onupgradeneeded* vždy, jelikož databáze zatím neexistuje. Jedná se zároveň o místo, kde jsou specifikovány změny ve schématu databáze. Zdrojový kód 17 zobrazuje triviální příklad vytvoření databáze s jednou tabulkou. Jelikož se však nejedná o relační databázi, nýbrž o NoSQL databázi, žádné tabulky zde neexistují. Jednotlivé entity jsou zde agregovány do tzv. skladišť[13]. Na *řádku 4* je ukázka zaregistrování handleru pro danou událost. V něm je následně na *řádku 6* zavolána metoda *createObjectStore(name [, options])*. První parametr reprezentuje název skladu, druhý parametr je volitelný objekt s nastavením. Atribut *keyPath* určuje klíčový atribut, který bude použit jako identifikátor při získávání dat z daného skladu. Atribut *autoIncrement* automaticky generuje tento klíčový atribut. Pokud je při vytváření skladu definován v nastavení atribut *keyPath* nebo *autoIncrement*, pak už do tohoto skladu nelze ukládat atomické hodnoty, ale vždy už jen a pouze objekty. Ty navíc musí obsahovat daný atribut specifikovaný nastavením atributu *keyPath*.

Na *řádku 7* je ukázka vytvoření indexu. Volání *createIndex(name, keyPath [, options])* vytvoří nad skladem index. To umožňuje ve skladu vyhledávat i podle jiných hodnot než podle *keyPath*. V příkladu je ukázáno vytvoření indexu na atribut *name*. V nastavení indexu je atribut *unique* nastaven na *false*. Tím zaručíme, že hodnoty vložených dat do skladu *users* nemusí mít unikátní hodnotu atributu *name*.

```
1  var dbh = indexedDB.open("testdb", 1);
2
3  // vytvoření databázového schéma
4  dbh.onupgradeneeded = function(event) {
5      var db = event.target.result;
6
7      var options = { keyPath: "id", autoIncrement: true };
8      var store = db.createObjectStore("users", options);
9
10     var index = store.createIndex("nameIndex", "name", { unique: false });
11 };
12
13 dbh.onerror = function(event) {
14     // kód při chybě
15 };
16
17 dbh.onsuccess = function(event) {
18     // kód při úspěchu
19 };
```

Zdrojový kód 17: Vyvoření schéma v IndexedDB.

[13] V angličtině stores.

Zdrojový kód 18 ukazuje základní způsoby vkládání a získávání dat z databáze. Vše se děje v transakcích voláním *transaction(stores [, mode])*. *Stores* je buď název řetězec skladu nebo pole řetězců názvů skladů, se kterými budeme v transakci pracovat.

Existují tři typy módů transakcí:

- Readonly - umožňuje čtení dat (implicitní hodnota).
- Readwrite - umožňuje čtení, modifikaci a úpravu dat.
- Versionchange - umožňuje modifikaci schématu databáze.

Na *řádku 8 a 9* jsou vytvořeny dva záznamy do skladu. Následně jsou ukázány dva způsoby, jak lze získat jeden konkrétní záznam ze skladu. První je na *řádku 11* pomocí skladu zadáním odpovídající hodnoty *keyPath*. Kvůli asynchronicitě jsou data získána až pomocí handleru události *success*. Obdobně je tomu za použití indexu. Na *řádku 15* je následně ukázka využití indexu k získání záznamu ze skladu na základě hodnoty nějakého jeho atributu. V případě, že by bylo potřeba získat všechny záznamy, tak lze použít metodu *getAll()*. Následně je na *řádku 20* zaregistrována handler funkce, která uzavře konektor do databáze. To je provedeno při ukončení námi definované transakce.

Z pokročilejších vlastností IndexedDB disponuje i možností použít databázové kurzory. Umožňuje to snížit zátěž na databázi a zrychlit odezvu stránky, jelikož není nutné získat všechny položky z databáze najednou, nýbrž postupně.

```
1  dbh.onsuccess = function(event) {
2      var db = event.target.result;
3
4      var tx = db.transaction("users", "readwrite");
5      var store = tx.objectStore("users");
6      var index = store.index("nameIndex");
7
8      store.add({ id: 1, name: "Tom" });
9      store.add({ id: 2, name: "Kymbat" });
10
11     store.get(1).onsuccess = function(event) {
12         alert(event.target.result.name); // zobrazí "Tom"
13     }
14
15     index.get("Kymbat").onsuccess = function(event) {
16         alert(event.target.result.name); // zobrazí "Kymbat"
17     }
18
19     // zavření databáze po skončení transakce
20     tx.oncomplete = function() {
21         db.close();
22     };
23 };
```

Zdrojový kód 18: Vyvoření schéma v IndexedDB.

IndexedDB má výrazně vyšší datové limity, než předchozí dva způsoby uložení dat. Dle specifikace má každá doména až 20% z volného místa na uživatelském pevném disku [17]. Celkově na všech doménách však nelze uložit více, než je polovina celkového volného místa na disku. Informaci o množství využité paměti zajišťuje *Quota Manager* proces prohlížeče.

Pro implementaci použiji tuto technologii k ukládání umělých dat, jelikož umožňuje nejkomplexnější řešení a flexibilitu. Velká výhoda je její nativní podpora v prohlížečích.

Kapitola 5

Dědičnost v knihovně SAPUI5

JavaScript je objektový jazyk, ale místo klasické třídní dědičnosti využívá dědičnosti prototypové. Nicméně knihovna *SAPUI5* používá názvosloví třídních objektově orientovaných jazyků. Z toho důvodu se i v tomto textu budu držet této praktiky. Pokud se tedy bude mluvit o třídě X, bude se v podstatě mluvit o prototypu. V obou případech se však jedná o jistý předpis, dle kterého jsou vytvářeny instance. Konkrétně u prototypové dědičnosti vznikají instance klonováním.

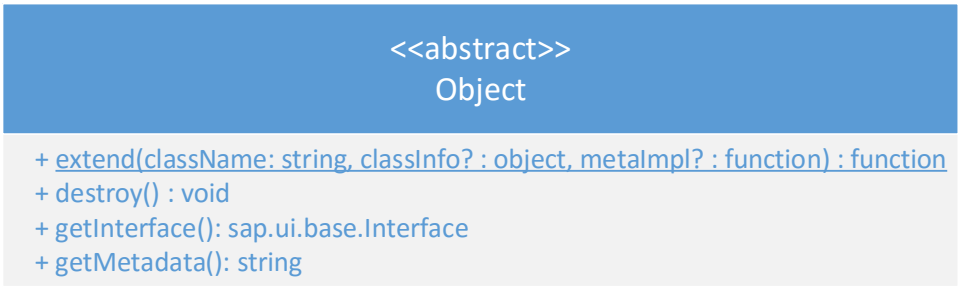
V této kapitole budou představeny existující třídy, které byly použity při implementaci komponenty na vytváření umělých dat. Všechny patří do knihovny *SAPUI5* až na objekt *null*, který je součástí nativního JavaScriptu. Jedná se zde o popis nejdůležitějších částí základních tříd *SAPUI5*, který sumarizuje informace z několika zdrojů včetně procházení zdrojového kódu. Tato kapitola slouží především k pochopení konceptu *SAPUI5* a bázových tříd, ze kterých následně dědí velká část dalších komponent včetně výsledné knihovny na vytváření umělých dat. Níže představené třídy jsou tedy použity ve výsledné implementaci knihovny viz kapitola 6.

5.1 Objekt Object

Vrchol prototypové dědičnosti tvoří objekt *Object*. Zde by mohlo dojít k chybné úvaze, že se jedná o objekt *Object*, který je nativní v jazyce JavaScript. JavaScript *Object* slouží jako základní objekt, ze kterého prototypově dědí všechny další objekty v JavaScriptu. Zde se však jedná o objekt *Object*, který patří do *SAPUI5*. Totožnost názvu s nativním objektem vznikla pravděpodobně z toho důvodu, že *SAPUI5 Object* rovněž slouží jako základní objekt, ze kterého dědí všechny další objekty *SAPUI5*.

Object je definován jako abstraktní třída, což v normální prototypové dědičnosti nedává smysl. Nic jako abstraktní třída obecně v jazyce JavaScript neexistuje. V *SAPUI5* se však toto označení používá jako informační označení objektů, které by se neměly instanciovat, i když prakticky to možné je. Jedná se tedy jen pro upozornění vývojáře, aby nedělal instanci tohoto objektu. *Object* navíc přidává metody:

- Metodu *destroy*, která je zavolána při zrušení objektu.
- Metodu *getInterface*, která vrátí všechny veřejné metody daného objektu.
- Metodu *getMetadata*, která vrací objekt popisující objekt, ze kterého byla vytvořena daná instance.



Obrázek 5.1: UML diagram SAPUI5 objektu Object.

- Statickou metodu *extend*, kterou si podrobně popíšeme níže.

U metody *getInterface* je nutné poznamenat, že nic jako veřejné a privátní metody v jazyku JavaScript neexistují. Jedná se o pouhý konsenzus, že pouze tyto metody by měli vývojáři aplikací volat. Pokud se rozhodnout volat privátní metody, pak musí nést následky. Například vývojáři komponent můžou kdykoliv změnit API privátních metod. Pro veřejné metody to již neplatí, ty již nelze měnit v okamžiku zveřejnění komponenty.

Nejdůležitější metodou objektu *Object* je statická metoda *extend*. Ta zajišťuje v *SAPUI5* dědičnost. Mějme libovolný objekt A, který dědí z objektu *Object*. Nad objektem A zavoláme metodu *extend*. Tím získáváme konstruktor nového objektu B, který dědí z objektu A. Parametrem metody *extend* je pak název, který identifikuje daný objekt, ze kterého je následně možné vytvářet další instance.

Druhým parametrem, který je volitelný, je objekt popisující nově vytvořený objekt. Obsahuje informaci, zda se jedná o abstraktní objekt, zda lze z tohoto objektu dědit, konstruktor metodu. Pokud není dodána konstruktor metoda, pak se vygeneruje obecný konstruktor pro daný objekt. Jediné co udělá je, že provolá konstruktor nadřazené třídy.

Poslední parametr je volitelný a definuje konstruktor funkci objektu metadat, který bude vracen při volání metody *getMetadata*. Samotná třída je reprezentována na obrázku 5.1 pomocí UML diagramu. Konkrétně byl použit diagram tříd.

5.2 Objekt EventProvider

Další třída v prototypovém řetězci je *EventProvider*. Opět se jedná o abstraktní třídu, ze které by se neměla dělat instance, i když samotný JavaScript toto omezení nemá. *EventProvider* objekt přidává tyto metody:

- Metodu *attachEvent*, která umožňuje zaregistrovat callback funkci pro názvem specifikovanou událost.
- Metodu *attachEventOnce*, která funguje obdobně jako *attachEvent*, ale daný callback se provede pouze při prvním vyvolání specifikované události.
- Metodu *detachEvent*, která odstraní danou callback funkci, čímž se zamezí jejímu provolání v případě vyvolání události, pro kterou byla zaregistrována. Parametrem je název události a reference na callback funkci, která má být odstraněna.

- Metodu *fireEvent*, která vyvolá specifikovanou událost. Lze definovat parametry, které se přidají do objektu události. Také lze určit, zda má event probublávat mezi otcem či otci.
- Metodu *getEventingParent*, která vrací otce, kterému daný objekt přeposílá své události. Ten je určen ve stromové struktuře. V případě že objekt neposlouchá na žádnou událost, pak tato metoda vrací *null*.
- Metodu *hasListeners* vrácí boolovskou hodnotu, která vrací pravdivou hodnotu v případě, že instance objektu má zaregistrovanou aspoň jednu callback funkci pro danou událost.
- Metodu *toString*, která vrací řetězec reprezentující daný objekt.

Struktura třídy *EventProvider* je znázorněna na obrázku 5.2 pomocí UML diagramu tříd. Je zde důležité upozornit na metodu *toString*, která by mohla některé čtenáře překvapit. Tato metoda je normálně dostupná jako metoda JavaScript objektu *Object*. Třída *EventProvider* ji obsahuje z toho důvodu, že třídy v *SAPUI5* nedědí z JavaScript objektu *Object*, ale ze *SAPUI5* objektu *Object*. Ten pak už dědí z JavaScript objektu *null*, který neobsahuje žádné metody.

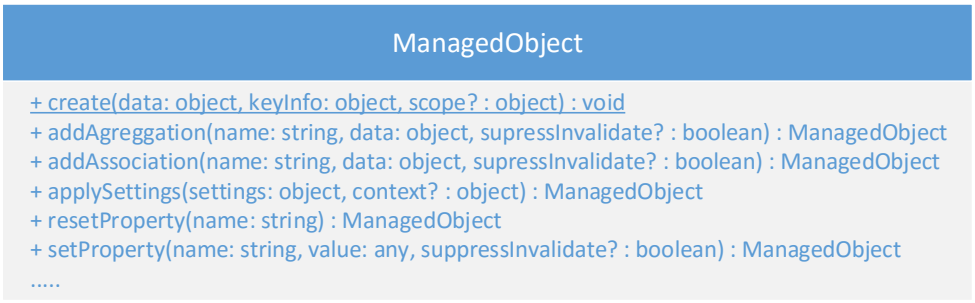
Obrázek 5.2: UML diagram objektu EventProvider.

Z tohoto důvodu bylo nutné dodat tuto užitečnou metodu přímo některé z hlavních tříd *SAPUI5*. Architekti této knihovny se rozhodli, že ji přidají právě do třídy *EventProvider*. Jedná se o jednu z nejzákladnějších tříd, ze které dědí téměř všechny další třídy.

Při registraci callback funkce pro danou událost použijeme funkci *attachEvent* nebo *attachEventOnce*. Parametry těchto funkcí je název události, pro kterou chceme zaregistrovat callback funkci. Dále samotná callback funkce. Také lze definovat data navíc, které se přidají k objektu události. Posledním parametrem je kontext, který se má použít při zavolání callback funkce. Tím je myšleno, na co bude odkazovat proměnná *this* uvnitř callback funkce. To je jedna z dalších unikátností jazyka JavaScript, který umožňuje libovolnou funkci zavolat v kontextu libovolného objektu. Pro čtenáře zvyklé pouze na třídní objektově orientované paradigma to může být zmatené. Lze si to představit tak, že JavaScript umožní dynamicky libovolně přiřadit libovolnou funkci a následně ji nad tímto objektem i zavolat. Po skončení provádění této funkce se pak zase následně dynamicky z tohoto objektu odebere. Technicky to takto nefunguje, ale pro přiblížení funkčnosti by to mohlo být dostačující.

5.3 Objekt ManagedObject

První neabstraktní třída v prototypovém řetězci. Jedná se o nejvíce komplexní třídu v celém *SAPUI5*. Komplexita vzniká hlavně díky tomu, že *ManagedObject* dynamicky generuje různé metody dle zadaných parametrů. To bude popsáno níže. *ManagedObject* je znázorněn na UML diagramu tříd na obrázku 5.3. Zde je uveden jenom demonstrativní výčet těch nejdůležitějších metod.



Obrázek 5.3: UML diagram objektu ManagedObject.

ManagedObject předefinovává statickou metodu *extend*. Rozšiřuje možnosti popisu objektu o další parametry. K již existujícím atributům objektu popisující nově vytvářenou třídu (definovaných v třídě *Object* v podkapitole 5.1) přidává navíc tyto atributy:

- *Aggregations* reprezentující dostupné agregace pro daný objekt.
- *Associations* reprezentující dostupné asociace pro daný objekt.
- *Events* reprezentující události, které bude daný objekt vyvolávat navíc oproti svým nadtřídám.
- *Library* reprezentující název knihovny, do které má být nově vytvořená třída zařazena.
- *Properties* reprezentující atributy nové třídy.

Properties definují aktuální stav daného objektu. Tyto atributy by měly být jednoduchého datového typu, tzv. řetězec, číslo, booleovská hodnota atd. Pro komplexní typy je vhodné použít asociace či agregace viz dále. Počet atributů není nijak omezen. Pro vygenerování se v objektu *Properties* definuje atribut, jehož název bude následně použit pro Při definici nového atributu lze zadat parametry:

- *Bindable*, který ovlivňuje vygenerování metod *bind{Name}* a *unbind{Name}*. {Name} zde nahrazeno názvem konkrétního atributu.
- *DefaultValue*, který reprezentuje výchozí hodnot v případě, že v konstruktoru objektu nebyla hodnota specifikována.
- *Group*, který reprezentuje sémantické rozdělení do skupin pro automaticky generované renderování.

- *Type*, který reprezentuje datový typ atributu. Jedná se o dynamický číselník s hodnotami:
  - *Any*, který reprezentuje libovolný datový typ.
  - *Boolean*, který reprezentuje booleovskou hodnotu.
  - *Float*, který reprezentuje číslo s plovoucí řádovou čárkou.
  - *Int*, reprezentující celé číslo.
  - *Object*, reprezentující libovolný objekt.
  - *String*, reprezentující řetězec.
  - Libovolný zaregistrovaný typ pomocí statické metody *DataType.createType*.
  - Pole libovolného výše zmíněného typu. Pro definici pole je nutné zadat [ ] za daný datový typ.

Ukázka definice atributů je na zdrojovém kódu 19. Zde byly pro novou třídu *Example* definovány tři atributy. První *myProperty* je definován zjednodušeným způsobem, kdy se jako hodnota pro klíč *myProperty* použije pouze datový typ atribut. Druhý atribut je definován plným způsobem, kdy je hodnota klíče objekt obsahující typ a výchozí hodnotu atributu. Třetím atributem je pole celých čísel s výchozí hodnotou prázdného pole.

ManagedObject pro takto definované atributy dynamicky vytváří přístupové a nastavovací metody. Pro každý atribut je vygenerována metoda *get{Name}()* a *set{Name}(hodnota)*, kde {Name} reprezentuje název atributu. Pro atribut *myProperty* byly automaticky vygenerovány metody *getMyProperty()* a *setMyProperty(hodnota)*. Tyto metody čistě získávají či nastavují hodnotu atributu ve vnitřním modelu. Pokud by vývojář chtěl nějakou složitější logiku, může si definovat vlastní *getMyProperty()* nebo *setMyProperty(hodnota)* metody, které přepíšou tyto dynamicky vygenerované metody.

Tyto dynamicky vygenerované metody nejsou nic jiného, než zavolání metod *getProperty("myProperty")* a *setProperty("myProperty", hodnota)* nad model třídy. Jedná se však o zjednodušení volání a hlavně tento přístup dynamického generování metod umožňuje definovat si vlastní komplexní logiku získávání a nastavování atributů, které drží konzistentní aplikační rozhraní. Vždy se volá stejná metoda *get{Name}()* nezávisle na tom, jestli má vlastní komplexní logiku.

Dobře specifikovat datový typ atributu je důležité proto, že *ManagedObject* automaticky validuje hodnoty, které jsou tomuto atributu přiřazeny. Pokud by si někdo chtěl zaregistrovat vlastní datový typ, pak musí definovat i validátor. Ten při každém nastavení hodnoty ověří, zda odpovídá nastavenému datovému typu.

```
1 ManagedObject.extend("Example", {
2     metadata: {
3         properties: {
4             myProperty: "string",
5             myIntProperty: { type: "int", defaultValue: 35 },
6             myArrayProperty: { type: "int[]", defaultValue: [] }
7         }
8     }
9 });
```

Zdrojový kód 19: Základní ukázka definice atributů v třídě *ManagedObject*.

Agreggations definují spravované objekty patřící danému objektu. Agregace může být vztahu 0 až 1 nebo 0 až N dle nastavené hodnoty *multiple*. Pokud je objekt A součástí agregace objektu B, tak již nemůže být součástí agregace jiného objektu C. Objekt B je tedy otec objektu A. Pokud je libovolný objekt zničen, tak jsou spolu s ním zničeny i všechny objekty jeho agregací. Pokud je zničen objekt, který je součástí něčí agregace, pak dochází k odstranění agregačního objektu z agregace svého otce. Pokud bychom měli objekt A, jehož otec je objekt B, a objekt A je zničen, tak destruktor automaticky odstraní referenci na objekt A z otcovského objektu B.

```
ManagedObject.extend("Example", {
    metadata: {
        aggregations: {
            singleAggregationExample: {
                type : "singleAggregationExample",
                multiple: false,
                visibility: true
            },
            multipleAggregationExamples: {
                type: "PropertiesExample2",
                singularName: "multipleAggregationExample",
                multiple: true,
                visibility: true
            }
        }
    }
});
```

Zdrojový kód 20: Základní ukázka definice atributů v *ManagedObject*.

Na ukázce zdrojového kódu 20 je vidět definice dvou jednoduchých agregací. Při definici nové agregace lze u ní nastavit tyto parametry:

- *Type*, reprezentuje datový typ objektů agregace.
- *Multiple*, který určuje, zda agregace bude několikanásobná či jednotná.
- *SingularName* pro několikanásobné agregace je vhodné definovat tvar jednotného čísla. *ManagedObject* dokáže název odvodit pro běžné názvy sám.
- *Visibility*, který definuje, zda agregace bude veřejná nebo privátní. Pro privátní se dynamicky negenerují přístupové metody viz níže.
- *Bindable*, který funguje obdobně jako v *Properties* viz výše.

Pokud je v agregaci nastaven parametr *multiple* na nepravdivou hodnotu a zároveň je viditelnost nastavena na pravdivou hodnotu, pak dochází k vygenerování několika veřejných metod. Pro agregaci *singleExample* ve zdrojovém kódu 20 se vygenerují metody:

- *getSingleExample()*, která vrací objekt agregace.

- *setSingleExample(objekt)*, která nastaví objekt agregace.
- *destroyItem(objekt)*, která zavolá destruktor nad objektem agregace.

Pokud je v agregaci nastaven parametr *multiple* na pravdivou hodnotu a zároveň je viditelnost nastavena na pravdivou hodnotu, pak se vygenerují další metody. Pro agregaci *multipleExamples* ve zdrojovém kódu 20 se navíc oproti agregaci *singleExample* vygenerují metody:

- *getMultipleExamples()*, která vrací všechny objekty agregace.
- *addMultipleExample(objekt)*, která vloží na konec nový objekt agregace.
- *insertMultipleExample(objekt, index)*, která vloží na začátek nový objekt agregace.
- *removeMultipleExample(objekt)*, která odstraní specifikovaný objekt z agregace.
- *removeMultipleExamples()*, která odstraní všechny objekty agregace.
- *destroyMultipleExamples()*, která zavolá konstruktor nad všemi objekty agregace.

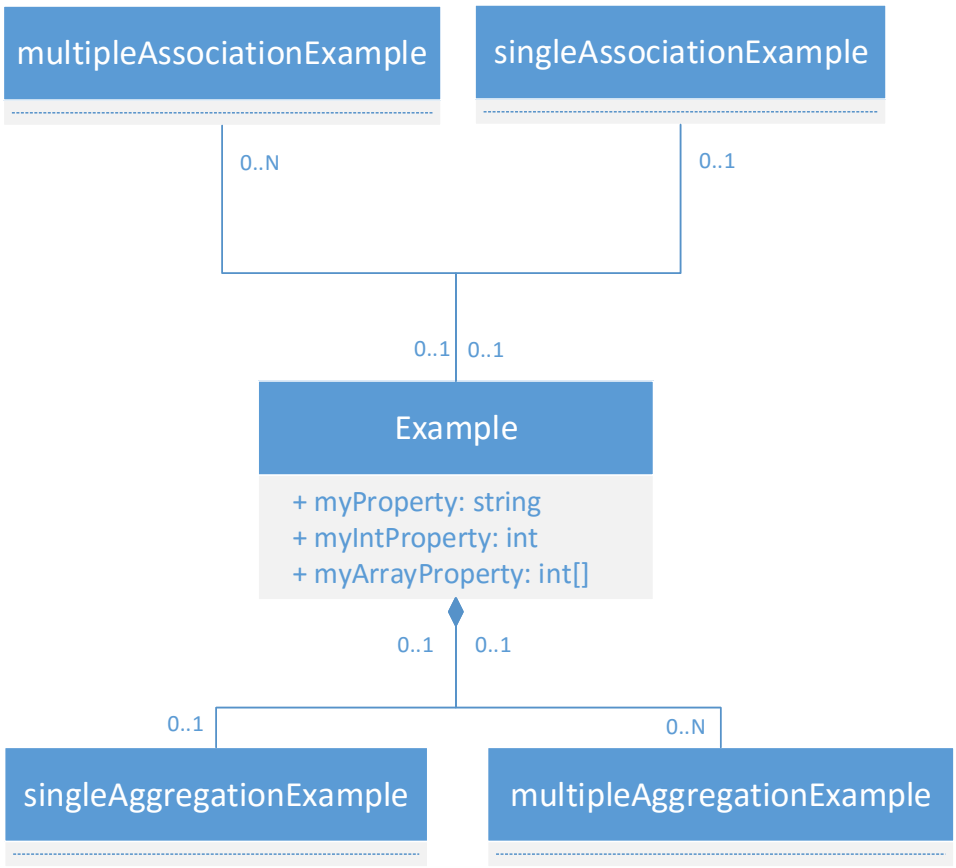
```
ManagedObject.extend("Example", {
    metadata: {
        associations: {
            singleAssociationExample: {
                type : "singleAssociationExample",
                multiple: false
            },
            multipleAssociationExamples: {
                type: "multipleAssociationExample",
                singularName: "multipleExample",
                multiple: true
            }
        }
    }
});
```

Zdrojový kód 21: Základní ukázka definice atributů v *ManagedObject*.

Associations je další důležitá součást *ManagedObject* třídy. Funkcí je asociace velmi podobná agregaci, ale jsou zde menší rozdíly. Pokud je objekt A asociací objektu B a objekt B je zničen, tak objekt A není zničen. V agregaci by došlo ke zničení objektu A. Také není parametry definice nejsou tak obsáhlé jako u agregace. Obsahují tyto parametry:

- *Type*, reprezentuje datový typ objektů agregace.
- *Multiple*, který určuje, zda asociace bude několikanásobná či jednotná.
- *SingularName* pro několikanásobné asociaci je vhodné definovat tvar jednotného čísla. *ManagedObject* dokáže název odvodit pro běžné názvy sám.

Již tedy nelze definovat viditelnost. Z toho plyne, že všechny asociace jsou veřejné. Zároveň zmizel parametr *bindable*, který umožňoval navázat na vnitřní model objektu. Pro asociace platí, že *ManagedObject* může držet pouze jejich reference, ale nemůže je mapovat na vlastní vnitřní model. Ukázka definice asociací je na zdrojovém kódu 21.



Obrázek 5.4: UML diagram tříd pro *ManagedObject*.

U klonování dochází ke dvou situacím. Je klonován objekt A, který má asociaci na objekt B. V takovém případě je naklonován pouze objekt A a dostane referenci na již existující objekt B. V případě, že je klonován objekt A, který má agregaci na objekt B a ten má jako asociaci objekt C, pak se klonují všechny tři objekty.

Vztah mezi asociacemi, agregaci a atributy je znázorněn na obrázku 5.4. Na obrázku je ukázka třídy definované ve zdrojových kódech 19, 21 a 20.

Events jsou další parametry, které lze specifikovat při vytváření nové třídy, která dědí z *ManagedObject*. Události nám umožňují upozornit všechny posluchače, že nastala daná událost. Při definici události lze definovat parametry:

- *AllowPreventDefault*, který definuje, zda má být zabráněno výchozímu chování události, pokud nějakou má.
- *Parameters*, který definuje parametry, které se přidají spolu do objektu události.

```
ManagedObject.extend("Example", {
    metadata: {
        events: {
            beforeRequest: {
                allowPreventDefault : false,
                parameters: {}
            },
            click: {
                click : true,
                parameters: {}
            }
        }
    }
});
```

Zdrojový kód 22: Základní ukázka definice událostí v *ManagedObject*.

Definice dvou událostí je ukázaná ve zdrojovém kódu 22. Zde je definována událost *beforeRequest* a událost *click*, u které je navíc zabráněno výchozí chování. Pokud by tedy uživatel kliknul na HTML reprezentaci objektu *Example*, tak by se prohlížeč nezachoval dle výchozího nastavení. To může být například přesměrování na jinou URL, pokud by uživatel kliknul na odkaz. Pro každou zaregistrovanou událost *click* se vytvoří tyto metody:

- *AttachClick(funkce, objekt)*, která umožňuje zaregistrovat callback funkci, která se má provolat při vzniku události. Druhý parametr specifikuje kontext callback funkce.
- *DetachClick(funkce, objekt)*, který odregistruje specifikovanou callback funkci.
- *FireClick()* provolá všechny callback funkce zaregistrované pro danou událost.

Samotný *ManagedObject* definuje několik svých vlastních událostí. Mezi ně patří:

- *FormatError*, která nastane, když selže validace některého atributu při získávání hodnoty z modelu.
- *ModelContextChange*, která nastane, když se změní vnitřní model objektu.
- *ParseError*, která nastane, když selže parsování některého atributu při ukládání do modelu.
- *ValidationError*, která nastane, když selže validace některého atributu při ukládání do modelu.
- *ValidationSuccess*, která nastane, když se nová hodnota úspěšně uloží do modelu.

Kapitola 6

Implementace knihovny

V následujících kapitolách bude popsáno samotné řešení vytvořené knihovny pro generování umělých dat. Hlavní a jediná třída implementace je třída *MockServer*, která dědí z třídy *ManagedObject*. Tato třída využívá z tohoto objektu hlavně možnost dynamického generování pro nastavování a získávání hodnot atributů. Samotná třídní hierarchie je zobrazena na obrázku 6.2. Tento obrázek poskytuje pohled nejvyšší úrovně na vytvořenou knihovnu. Při implementaci byl také použít návrhový vzor Jedináček viz podkapitola 6.1.

6.1 Odchytávání dotazů

Pro odchytávání dotazů na server byla použita knihovna *Sinon.JS* viz podkapitola 2.3. Ta má však své nedostatky. Například nelze použít několik *FakeServer* instancí patřící do *Sinon.JS*. Tento bug je nahlášený přímo na Github repozitáři projektu a prozatím se neplánuje ho řešit[1]. Z toho důvodu bylo nutné nějak vyřešit problém v případě více instanci *MockServeru*, který vnitřně používá právě *FakeServer* z knihovny *Sinon.JS*.

Nakonec se k tomu použil návrhový vzor Jedináček. V třídě *MockServer* je pomocí uzávěru[2] definována proměnná pro *FakeServer*. Nově vytvořená instance *MockServer* třídy zavolá statickou metodu *_getInstance()*. Ta zkontroluje, zda je v uzávěru uzavřené proměnné již objekt *FakeServer* třídy. Pokud ne, tak ho vytvoří, v opačném případě vrátí již existující instanci. Ukázka vytvoření dvou instancí *MockServer* třídy je zobrazena na obrázku 6.1.

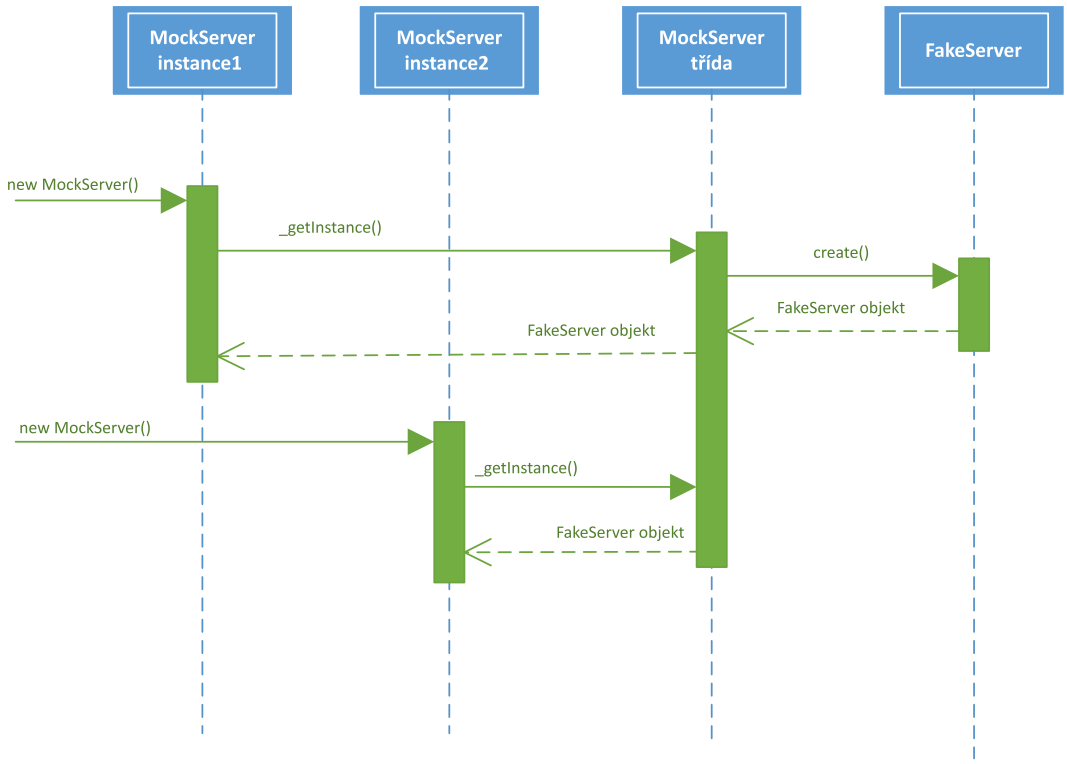
Samotný uzávěr je technika v některých programovacích jazycích včetně JavaScriptu, která umožňuje udržovat referenci na proměnnou, jejíž standardní životní cyklus skončil. Respektive funkce, pro jíž kontext byla proměnná definována, již skončil. Ukázku lze vidět ve zdrojovém kódu 23. Zde je využito techniky uzávěru k vytvoření iterátoru. Funkce, která je vrácena funkcí *closure* má odkaz na proměnnou *array*. Při zavolání funkce *closure* se do proměnné *iterator* uloží nová funkce. Ta při svém zavolání odebírá prvky z pole *array*.

Výhoda této techniky je, že se již nedá žádným způsobem dostat k proměnné *array*. Nehrozí tedy, že by ji kdokoliv modifikoval nebo si ji četl. V jazyce JavaScript se pomocí této techniky dosahuje privátních funkcí a proměnných, ke kterým se již za běhu programu nedá dostat.

Zároveň je udržována statická proměnná reprezentující všechny instance *MockServer* třídy. Toto pole je tam z toho důvodu, že *MockServer* má statické metody na zapnutí,

[1] Viz https://github.com/sinonjs/sinon/issues/211.

[2] Anglicky closure viz https://developer.mozilla.org/cs/docs/Web/JavaScript/Closures.



Obrázek 6.1: UML diagram návrhového vzoru Jedináček.

vypnutí či zničení všech *MockServer* instancí. Dále si staticky udržuje pole filtrů pro *FakeServer*. Pomocí nich lze specifikovat dotazy na server, které *FakeServer* neodchytí, ale pošle je dál na skutečný server[3].

Kvůli nemožnosti mít více instancí *FakeServer* třídy tedy nelze mít ani separátní filtry pro jednotlivé instance *MockServer* třídy. Z tohoto důvodu vzniká omezení, že všechny filtry jsou společné pro všechny instance *MockServer* třídy.

[3]Dostupné z: http://sinonjs.org/releases/v2.1.0/fake-xhr-and-server/#filtered-requests.

```
function closure() {
    var array = [ "Tom", "Kymbat" ];
    return function() {
        return array.pop();
    };
}

var iterator = closure();

iterator.pop();  // "Kymbat"
iterator.pop();  // "Tom"
iterator.pop();  // undefined
```

Zdrojový kód 23: Ukázka uzávěrů.

6.2 Databáze

V této části bude vysvětlena implementaci databáze. Data jsou uloženy i vypnutí a zapnutí aplikace. Původně se k tomuto měla použít databáze *IndexedDB* viz podkapitola 4.3. Postupem však nastalo několik problémů. Zvláště kvůli asynchronní podstatě všech operací v *IndexedDB* a nemožnosti vynutit synchronní chování. Pokud se v *IndexedDb* zavolá metoda *get* na nějakou entitu, tak i ta se provede asynchronně. Došlo tedy ke komplikacím.

Při zavolání vzdálené služby v aplikaci dojde při odchycení volání k zavolání nadefinované funkce *respond*, která zašle požadavek do databáze. Ten však byl opět asynchronní. To znamenalo, že se klientovi vrátili v odpovědi na dotaz prázdná data.

Dalším problémem s *IndexedDb* byl ten, že pokud nastala situace, že uživatel zavolal smazání entity, tak se zavolala *delete* metoda nad databází. Pokud však během toho uživatel zavřel internetový prohlížeč, tak se transakce už nevykonala. Tento problém měla vyřešit globální událost JavaScriptu *onbeforeunload*. Ta zabraňuje uživateli zavření internetového prohlížeče, než se operace dokončí. Často se využívá v případě, že uživatel upravuje nějaký formulář, ale zapomněl ho uložit.

Tato událost však nevedla k řešení. Již několik let je na to nahlášený bug přímo na stránkách *Chromium*[4]. Byla možnost však využít globální JavaScript událost *onunload*, která by měla sloužit na podobné události. Později se však ukázalo, že zde záleží na velikosti ukládaných dat.

Nelze predikovat, jestli se poslední transakce uloží nebo ne. Jedná se o naprosto nekonzistentní chování, které je závislé od aktuálního vytížení procesoru a internetového prohlížeče[5]. Proto bylo nakonec v práci použito místo databáze *IndexedDb* lokální úložiště viz podkapitola 4.1.

Není však použito dle původního návrhu při každém dotazu na server. Místo toho je celá databáze uložena v paměti počítače uvnitř JavaScript proměnné. Až při vypnutí aplikace je obsah dané proměnné serializován a uložen do lokálního úložiště. Při opětovném zapnutí

[4]Dostupné z: https://bugs.chromium.org/p/chromium/issues/detail?id=144862

[5]Dostupné z: http://vaughnroyko.com/offline-storage-indexeddb-and-the-onbeforeunloadunload-problem/.

aplikace se tyto data opět deserializují a uloží do JavaScript proměnné. Díky synchronní podstatě lokálního úložiště zde nedochází k problémům jako u *IndexedDb*.

Je nutné mít na paměti, že lokální úložiště má omezení velikosti 5MB na jednu doménu. Tato velikost by však měla být plně dostačující. Zvláště pokud vezmeme v úvahu využití této knihovny. Neočekává se produkční nasazení u klienta, nýbrž bude sloužit jen pro vývojáře k testování aplikace a při jejím vývoji. Vyvíjené aplikace jsou z větší části menšího rázu s řádově jednotkami *EntitySet* množin. Aby se naplnila kapacita lokálního úložiště, muselo by se jednat o aplikaci s řádově desítkami tisíc záznamů a větší množstvím entit. Pro účely této knihovny bude tedy lokální úložiště plně dostačující.

6.3 Generování dat

MockServer automaticky generuje umělá data pro všechny *EntitySet* množiny. V některých případech však není vhodné generovat vždy data automaticky, ale vývojář by si rád specifikoval svá vlastní data. Pro tento případ je zde vytvořena metoda *simulate*. Ta bere dva parametry. První z nich je URL, na kterém se nachází soubor *metadata.xml*. Druhým z nich je řetězec, který reprezentuje bázová URL, na které se budou nacházet uživatelem specifikovaná data.

MockServer knihovna následně načte *metadata.xml* soubor, který pomocí volání metody *parseXML(řetězec)* deserializace do objektu reprezentující dané XML. Tento objekt se projde a najdou se všechny *EntityType* typy a *EntitySet* množiny. Pokud byla definována bázová URL, na které se nachází uživatelsky definovaná data, pak se pro každou *EntitySet* množinu zavolá synchronní požadavek na danou URL.

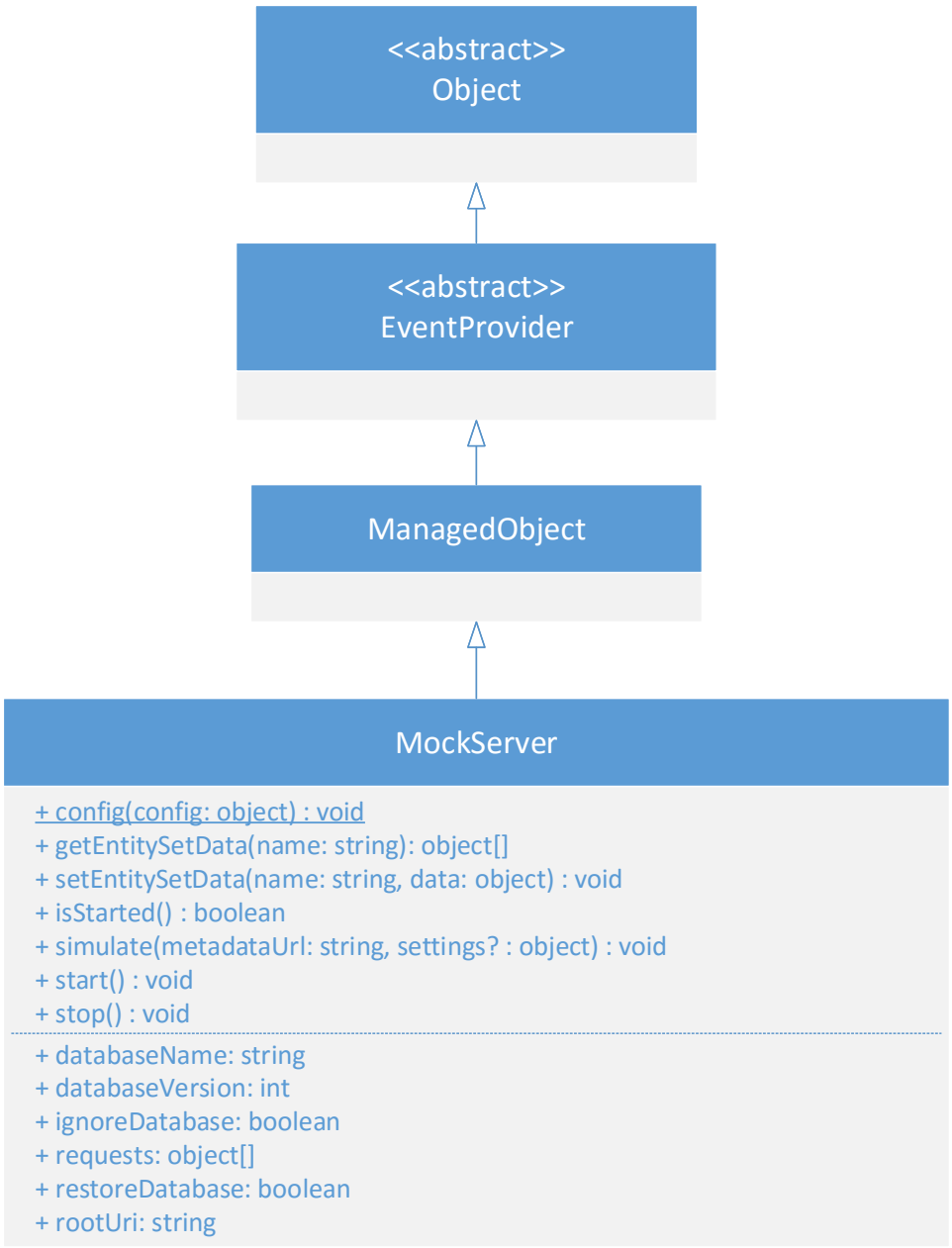
Data jsou očekáváná ve formátu JSON. To je i koncovka, která se očekává na konci názvu souboru. Ten musí být opět pojmenován stejně jako název dané *EntitySet* množiny. Například předpokládejme, že existuje *EntitySet* množina s názvem *UsersSet*, která je definovaná v *metadata.xml* souboru. Uživatel vývojář zavolá metodu *simulate* nad *MockServer* instancí a jako parametr pro bázovou URL předá řetězec *"/test/data/"*. Pak *MockServer* bude hledat uživatelsky definovaná data pro *EntitySet UsersSet* na adrese *"/test/data/UsersSet.json"*.

V případě, že je najde pro tento daný *EntitySet*, pak už je automaticky negeneruje. Pokud by na této adrese žádná data nenašel, pak předpokládá, že uživatel vlastní data pro daný *EntitySet* nedefinoval a vygeneruje je sám.

Další možností je při zavolání metody *simulate* nedat bázovou URL, ale místo toho předat parametrem adresu konkrétního JSON souboru. Zde se pak očekává, že bude obsahovat data pro všechny *EntitySet* množiny. V takovém případě je to jediný dotaz na server, který se vykoná. Například by tato adresa mohla být *"/test/data.json"*. Z tohoto souboru se následně získají data pro všechny *EntitySet* množiny. V případě, že v tomto souboru nebyly definovány data pro některý *EntitySet*, pak jsou opět dogenerovány automaticky *MockServer* objektem.

Pro generování komplexních typů (viz výše) se využije toho, že každý komplexní typ se skládá z několika jednodušších typů. Vygenerují se tedy hodnoty jednoduchých typů jako je číslo, řetězec nebo booleovský hodnota. Z těchto hodnot se následně složí hodnota komplexního typu. *MockServer* knihovna neposkytuje možnost zanoření komplexních typů. Tedy atribut komplexního typu již nemůže být komplexní typ, což se v praxi ukázalo, že není omezující. V SAP se komplexní typy téměř nepoužívají, natož zanořené.

Ze samotného generování dat stojí za zmínku generování datového typu *GUID*. Formát je specifikován jako *xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxx* [11]. Kde *x* reprezentuje hexade-



Obrázek 6.2: UML diagram tříd dědičnosti.

cimální jednociferné číslo. Tedy číslo z rozsahu 0 až F. Znak y reprezentuje hexadecimální číslo z rozsahu 8 až B. Ke generování byl použit již známý algoritmus[6].

[6] Viz http://stackoverflow.com/questions/105034/create-guid-uuid-in-javascript.

6.4 Registrace požadavků

U *FakeServer* třídy knihovny *Sinon.JS* je nutné zaregistrovat všechny požadavky, které mají být odchyceny. Mezi tyto požadavky patří především dotaz na *metadata.xml* soubor, požadavek na autorizační žeton[7] viz dále. Dále je nutné zaregistrovat všechny operace nad *EntitySet* množinami. Mezi ně patří získání entity pomocí HTTP metody GET, přidání entity pomocí HTTP metody POST, upravení entity pomocí metody PUT nebo MERGE, a smazání entity pomocí HTTP metody DELETE.

U získání entity je nutné registrovat i dotaz na entitu, na kterou má daná entita navázaný cizí klíč. Mějme *EntitySet* množinu Auto a *EntitySet* množinu Barva. Existuje auto s primárním klíčem 3, které má referenci na element množiny Barva s primárním klíčem Modrá. Pak se lze dotázat na entitu Barva dotazem */Auto('3')/Barva*. Všechny cizí klíče pro danou *EntitySet* množinu jsou definovány v *metadata.xml* souboru jako *NavigationProperty*. Pro každý navigační atribut musí existovat asociační množina, která obsahuje tento sdílený primární klíč. Jedná se o typické řešení M ku N vztahů v databázi.

U HTTP POST požadavku je možné definovat, že skutečná HTTP metoda je jiná. Slouží k tomu v hlavičce atribut *x-http-method*, který definuje skutečnou metodou. Možnost definovat jinou metodu vznikla z toho důvodu, že bezpečnostní opatření některých sítí nedovolují posílat jiné požadavky než GET a POST[8].

MERGE a PATCH HTTP metody fungují dle definice stejně. Z toho důvody jsou i implementace totožné. PATCH metoda byla přidána až v *OData* verzi 3[9]. SAP stále používá metodu MERGE, ale pro budoucí použití se implementovala i metoda PATCH.

S tím také souvisí, že na jeden *batch* požadavek odejde ze serveru jeden požadavek. Ta musí opět obsahovat odpovědi na všechny požadavky, které obsahoval předchozí *batch* požadavek. Oddělovač požadavků je specifikován jako regulární výraz:

$$--batch\_[a-z0-9-]*$$

U odpovědi se následně definuje oddělovač mezi jednotlivými požadavky. Typ dat je vždy v hlavičce definován jako *multipart/mixed*.

Součástí odchytávání požadavků je i kontrola, zda nepřišel dotaz na nastavení *X-CSRF* autorizačního žetonu. Ten umožňuje ochranu proti *Cross-Site Request Foregery* útoku [3]. V našem případě však neexistuje server, který by autorizační žeton generoval nebo ověřoval. Z toho důvodu není důležitá hodnota žetonu, proto se může použít libovolná náhodná hodnota. Konkrétně byla použita hodnota 42, protože to je odpověď na základní otázku života, vesmíru a vůbec.

Výsledná odpověď má následující strukturu. Každá entita obsahuje hodnoty svých atributů. Dále obsahuje atribut *__metadata*, jehož hodnota je objekt obsahující URI na samotnou entitu a její *EntityType*. Pro *NavigationProperty* atributy je uložena URI, na které lze získat referencovanou entitu přes cizí klíč.

Při získávání hodnot z databáze je nutné vyhodnocovat výrazy. Například pokud bychom měly *EntitySet* Faktury, tak lze v požadavku definovat, že chceme všechny faktury, které jsou vystaveny na cenu více jako 100 euro a zároveň menší hodnotu než 500 euro. Mezi podporované operace patří všechny klasické porovnávací operace, hledání prefixu, sufixu a podřetězce. Všechny operace jsou implementovány pomocí jednoduchého parsování řetězce a hledání klíčových slov.

[7] V angličtině token.
[8] Viz https://msdn.microsoft.com/en-us/library/dd541471.aspx.
[9] Dostupné z: https://msdn.microsoft.com/en-us/library/dd541276.aspx.

Kapitola 7

Implementace aplikace

Pro potřeby validace implementace knihovny byla vytvořena aplikace v *SAPUI5*. Tato knihovna umožňuje jednoduchým způsobem nadefinovat progresivní webovou aplikaci [2]. Progresivní aplikace je jedna z implementací Jednostránkových aplikací[1]. Jednostránkové aplikace jsou webové aplikace, které jsou podobné desktopovým aplikacím [6]. Uživateli se při vstupu na hlavní stránku načtou všechny potřebné knihovny a obsah hlavní stránky.

V okamžiku, kdy uživatel přejde na některou další stránku patřící do stejné aplikace, tak se již znova nenačítají potřebné knihovny. Pouze se stáhne obsah stránky, na kterou uživatel přešel. Tímto způsobem je možné zachovat obsah a stav i z předcházejících stránek. Pokud by se tedy uživatel vrátil zpátky na hlavní stránku, tak se již nenačítá nic. Také si aplikace pamatuje nastavení z původních akcí na hlavní stránce. Mezi to může patřit například zakliknutí některých položek v tabulce, stav rozkliknutí ve stromové struktuře vyhledávání a další.

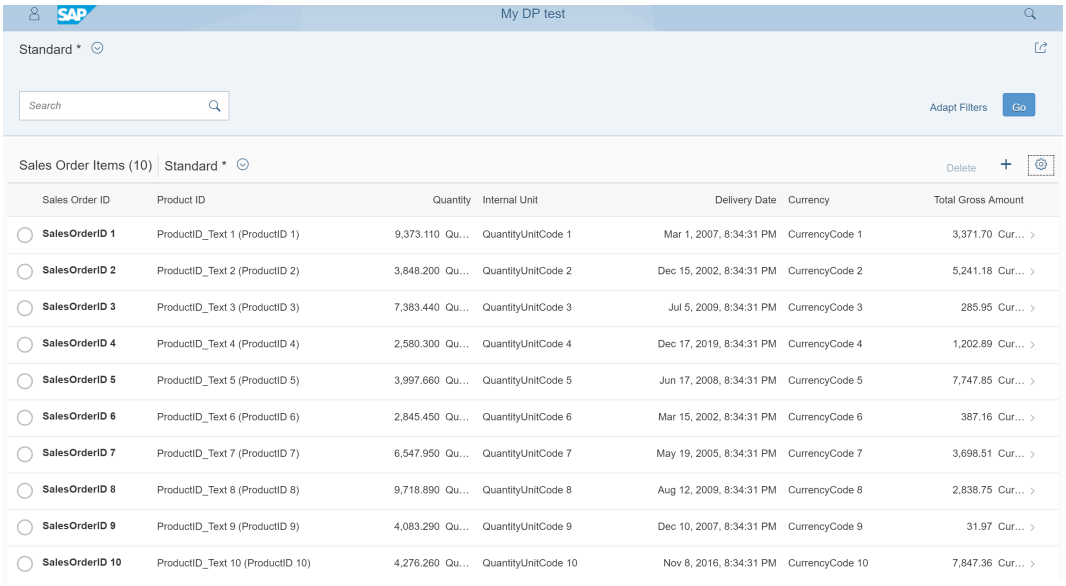
Jednostránkové aplikace pro komunikaci se serverovou částí silně využívají technologii *AJAX*. Jedna z hlavních výhod je velmi rychlá odezva při delší práci na stránce, jelikož se vše načte pouze jednou. Hlavní nevýhoda je především pomalejší první načtení stránky. Je to způsobeno nejčastěji vyšší velikostí JavaScript knihoven, které je nutné poslat uživateli. Tyto knihovny jsou nutné pro vykonávání složité logiky na klientské části aplikace.

Progresivní webové aplikace jsou jedna z implementací jednostránkových aplikací. Většinou se jedná o soubor několika aplikací, které jsou součástí jedné větší aplikace. Existuje zde nějaká obálka, která řídí celé zobrazování a nahrávání. Do této obálky se následně dynamicky stáhne vždy potřebná aplikace [6]. Lze si to představit jako plochu na počítači s několika zástupci na různé aplikace. V okamžiku, kdy uživatel klikne na jednu z těchto aplikací, tak se mu stáhne její obsah a zobrazí se. Pokud by ji vypnul a znova zapnul, tak se již opětovně nestahuje většina aplikace. Stáhnou se pouze uživatelská data, nikoliv už potřebné JavaScript knihovny, CSS styly, HTML obsah, SVG ikony a další. Pokud bychom měli například aplikaci pro zobrazení faktur, tak se stáhnou pouze informace o jednotlivých fakturách. Obecně však není nutné stahovat ani tyto data, pokud má uživatel možnost si vynutit znovunačtení těchto dat ze serveru.

[1] V angličtině Single Page Application (SPA).

7.1 Implementace v knihovně Fiori Elements

Součástí knihovny *SAPUI5* je i knihovna *Fiori Elements*[2]. Ta umožňuje deklarativním způsobem vygenerovat progresivní aplikaci, která vychází z designu *Fiori*. *Fiori* je designový koncept, který vyhrál prestižní cenu *Red Dot*[3]. Zaměřuje se hlavně na uživatelský prožitek[4]. Tento design je plně responzivní, takzvaně se snaží poskytnout ideální použití aplikace na počítači, tabletu i telefonu. To je důležité především se vzrůstajícím počtem uživatelů na mobilních zařízeních. Design těchto aplikací se tedy často zaměřuje nejdříve na mobilní zařízení a až v druhé řadě na stolní počítače[5].



Obrázek 7.1: Hlavní stránka aplikace za použití *List Report* stránky.

Samotná knihovna*Fiori Elements* umožňuje použít tři předdefinované aplikace:

- *List Report* aplikace, jejíž základem je tabulka[6].
- *Object Page* aplikace, jejíž základem je detail jedné entity[7].
- *Overview Page* aplikace, jejíž základem je rychlý přehled o sledovaných aktivitách[8].

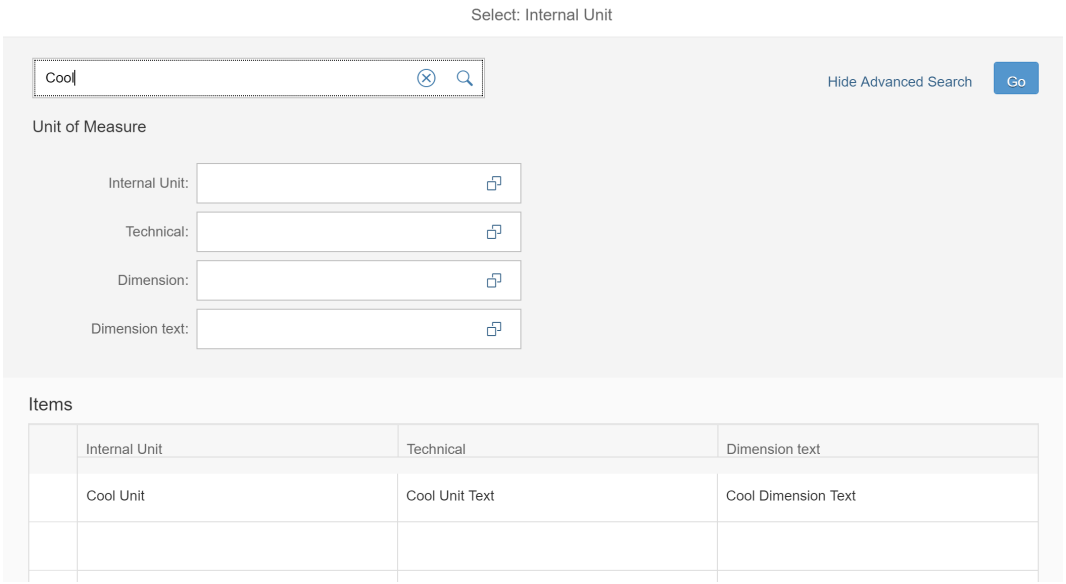
Tyto aplikace se již z větší části nemusí programovat. Místo toho jsou deklarativním způsobem nakonfigurované, jak mají vypadat. Tento způsob vývoje aplikací agreguje logiku spousty aplikací na jedno místo, čímž se omezuje jejich chybovost. Pokud by se v některé aplikaci našla chyba, tak ji lze centrálně opravit pro všechny ostatní aplikace. Tento přístup

[2]Dostupné z: https://experience.sap.com/fiori-design-web/smart-templates.
[3]Dostupné z: http://news.sap.com/sap-wins-prestigious-red-dot-award-design-concept.
[4]V angličtině User Experience (UX).
[5]V angličtině design Mobile First. Dostupné z: https://experience.sap.com/fiori-design-web/mobile-first.
[6]Dostupné z: https://experience.sap.com/fiori-design-web/smart-template-list-report.
[7]Dostupné z: https://experience.sap.com/fiori-design-web/object-page.
[8]Dostupné z: https://experience.sap.com/fiori-design-web/overview-page.

se hodí pro případy, kdy je nutné vytvořit velkou spoustu podobných aplikací, které mají pouze pracovat nad jinýma datama.

K samotné deklaraci se využívá soubor *metadata.xml*, který definuje základní datový model aplikace. Dále lze u jednotlivých *EntitySet* množin definovat definovat další atributy, podle kterých je následně webová aplikace vykreslena, a které ovlivňují samotné chování aplikace. Pro *EntitySet* množiny lze pomocí atributů s prefixem *sap:* definovat například tyto vlastnosti:

- *Sap:create* definuje, zda lze přidávat nové entity do dané množiny.
- *Sap:deletable* definuje, zda lze mazat existující entity v dané množině.
- *Sap:updatable* definuje, zda lze upravovat existující entity v dané množině.
- *Sap:searchable* definuje, zda lze nad entitami v dané množině vyhledávat.

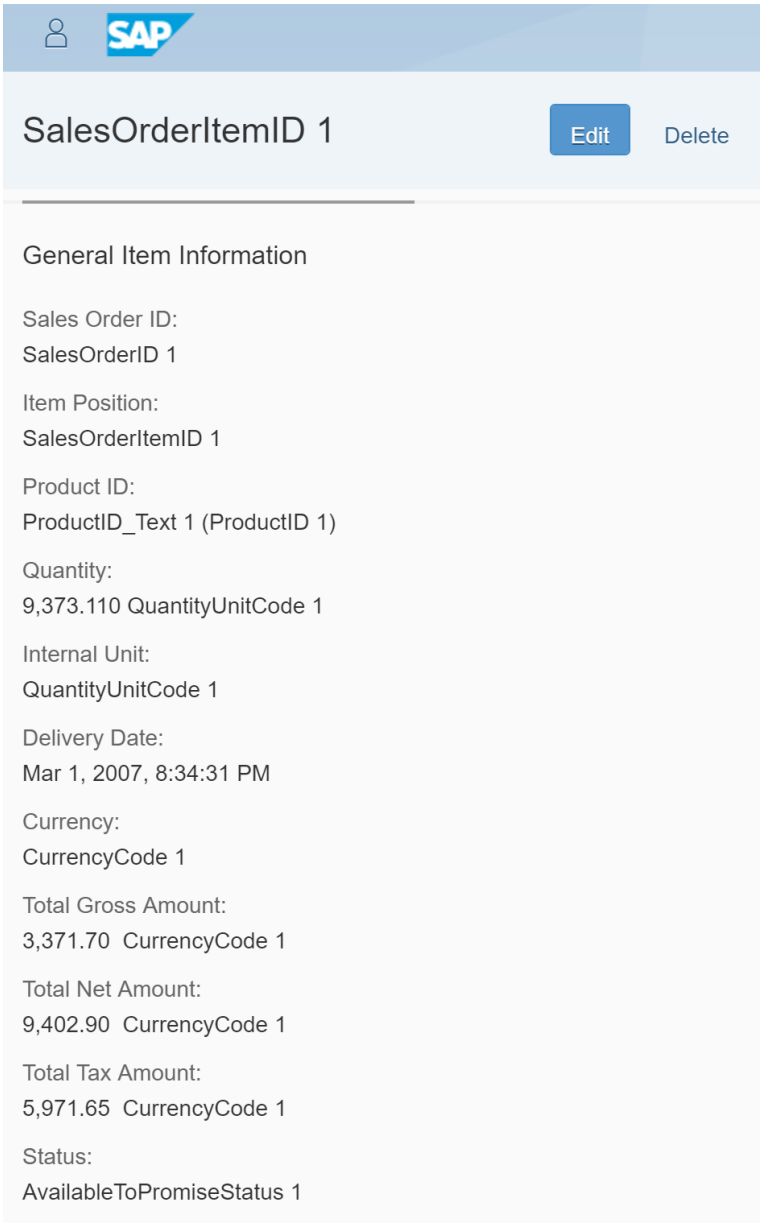


Obrázek 7.2: Automaticky vygenerované modální dialogové okno s vyhledáním hodnoty.

Pro *EntityType* atributy lze navíc definovat především tyto informace:

- *Sap:label* definuje textový popisek pro vykreslené vstupní políčko.
- *Sap:display-format* definuje, zda se má text vykreslit normálně či pouze velkými písmeny.
- *Sap:unit* definuje, zda se jedná o nějaký typ měrné jednotky či měnu.
- *Sap:quickinfo* definuje textový popisek pro nápovědu[9].

[9] V angličtině tooltip.



Obrázek 7.3: Stránka detailu objednávky za použití *Object Page* stránky.

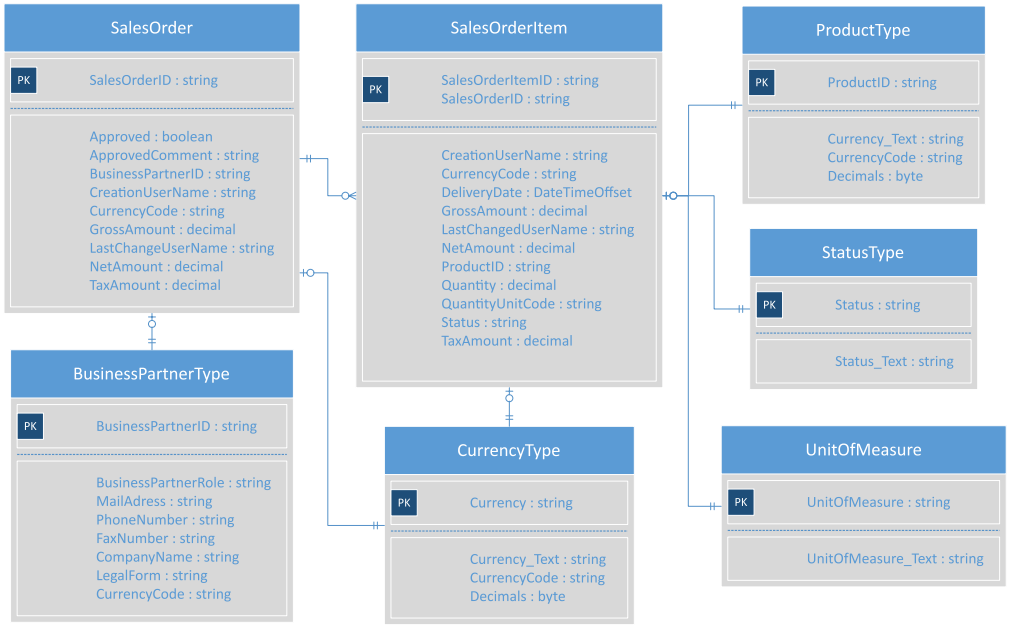
Dále lze v *metadata.xml* souboru definovat anotace, které dále obohacují sémantický význam. Příkladem může být definování některé atributy *EntityType* jako povinných. Dále lze pomocí anotací definovat *ValueHelp* dialog okno. Pomocí tohoto dialogového okna lze vyhledat v *EntitySet* množině konkrétní hodnotu s konkrétní hodnotou atributu. Například máme *EntityType* reprezentující uživatele, který má atributy jméno a příjmení. Nyní chceme do formuláře doplnit příjmení konkrétního uživatele, ale bohužel známe jen jméno.

U vstupního políčka bude ikona, která otevře dialogové okno s nápovědou *ValueHelp*. Zde můžeme zadat námi známé jméno a v tabulce v daném okně se nám zobrazí všechny příjmení, které patří uživatelům s daným jménem. Také lze vyhledávat, pokud známe aspoň část hodnoty požadovaného atributu. Ukázka automaticky vygenerovaného modálního di-

alogového okna je na obrázku 7.2. Lze zde vidět jednoduchý *EntityType Unit* a hledáme hodnotu atributu *Internal Unit*.

Lze použít obecné vyhledávání nad všemi políčky, které lze vidět v horní části dialogového okna. Momentálně se vyhledává nad hodnotou *Cool*. Dále lze zadat konkrétní hodnotu pro konkrétní atribut. V ukázce jsou použity atributy *Internal Unit, Technical, Dimension* a *Dimension Text*. V tabulce dole jsou pak zobrazeny všechny entity, které odpovídají hledaným parametrům. Na ukázce lze vidět případ, kdy chceme zobrazit méně sloupců, než nad kterými vyhledáváme. Zde je odstraněn ze sloupců tabulky atribut *Dimension*, i když se pomocí něho dá hledat.

Datový model vzorové aplikace je zobrazen na obrázku 7.4. Jedná se o jednoduchou aplikace, která zobrazuje jednotlivé položky faktury. Každá faktura má nějakého obchodního partnera, který ji k ní přiřazen. Každá faktura obsahuje nula až N položek. Tyto položky jsou zobrazeny na hlavní stránce, která využívá výše zmíněnou tabulkovou stránku *ListReport* viz obrázek 7.1.



Obrázek 7.4: Automaticky vygenerované modální dialogové okno s vyhledáním hodnoty.

Kapitola 8

Výkonnostní analýza implementace

Výkonnostní analýza je v počítačové vědě spuštění již vytvořeného programu, skupiny programů nebo jejich částí za účelem zhodnocení relativního výkonu modulu, objektu či funkce. K uskutečnění validní analýzy je nutné mít jednotný vzorek dat, vůči kterému se budou jednotlivé části aplikace zkoumat. Také je nutné vždy měřit výkon v co nejpodobnějších podmínkách. Ideálně na stejném počítači se stejným vytížením procesoru, grafické karty, disku a dalších částí.

Pro měření byl použit přenosný počítač Surface Book s následující konfigurací:

- Windows 10 64 bit.
- Intel Core i7 6600U CPU.
- 16 GB RAM.
- 512 SSD.
- nVidia GeForce 940M.

V následujících částech se objevují různé webové prohlížeče. Zaměření bylo pouze na nejnovější verze, jelikož starší verze již nejsou v *SAPUI5* podporovány. Pro měření byly použity tyto webové prohlížeče v následujících verzích:

- Chrome ve verzi 58.0.3029.96.
- Edge ve verzi 25.10586.672.0.
- Firefox ve verzi 53.0.2.
- Internet Explorer ve verzi 11.873.10586.0.
- Opera ve verzi 44.0.2510.1449.

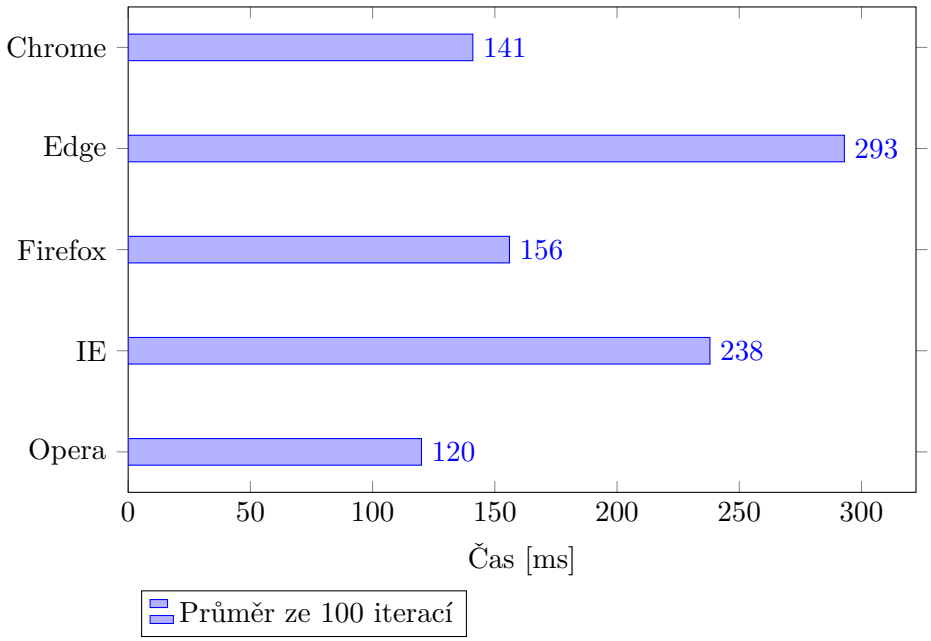
8.1 Odezva databáze

Cílem testování databáze bylo zjistit chování databáze dle různé velikosti ukládaných dat. Také mi přišlo zajímavé podívat se na rozdíly mezi jednotlivými webovými prohlížeči. Lze očekávat, že nejhorší výsledky bude mít prohlížeč Internet Explorer. Firma Microsoft tento

projekt už dále nevyvíjí a pouze opravuje kritické chyby. S příchodem operačního systému Windows 10 nahradil Internet Explorer nový prohlížeč Edge[1]. Ten přinesl především podporu standardu ECMAScript 6, který není podporován v prohlížeči Internet Explorer téměř vůbec. Projekt i testy jsou však psány ve standardu ECMAScript 5, jelikož knihovna *SAPUI5* zaručuje podporu pro Internet Explorer. To patří mezi velké omezení při vývoji knihovny a aplikací v *SAPUI5*.

První test porovnává výkon databáze v jednotlivých webových prohlížečích. Předem připravená data se zpracovávala následujícím způsobem:

1. Data jsou získána z lokálního úložiště.
2. Data jsou deserializována z řetězce.
3. Data jsou jako objekt uložena do proměnné.
4. Objekt je opět serializován do řetězce.
5. Serializovaný objekt je opět uložen do lokálního úložiště.



Obrázek 8.1: Výsledky měření načítání a ukládání dat do lokálního úložiště.

Připravená data obsahují vzorek dat ze vzorové aplikace viz kapitola 7. Výsledky měření jsou zobrazeny na obrázku 8.1. Časy jsou uvedeny pro 100 iterací výše zmíněného testu. Každý test byl navíc puštěn 100 krát a spočítal se jejich průměr. Časy jsou uvedeny v milisekundách. Pro měření byla použita nativní JavaScript metoda *performance.now()*, která měření čas s přesností na pět mikrosekund[2]. Z výsledků je jasně vidět, že nejpomalejší jsou webové prohlížeče firmy Microsoft. Jako výrazně nejpomalejší se ukázal webový prohlížeč Edge, i přesto, že je výrazně modernější a novější než jeho předchůdce Internet Explorer. Jako nejrychlejší se ukázal webový prohlížeč *Opera*.

[1]Dostupné z: https://www.microsoft.com/en-us/windows/microsoft-edge.
[2]Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

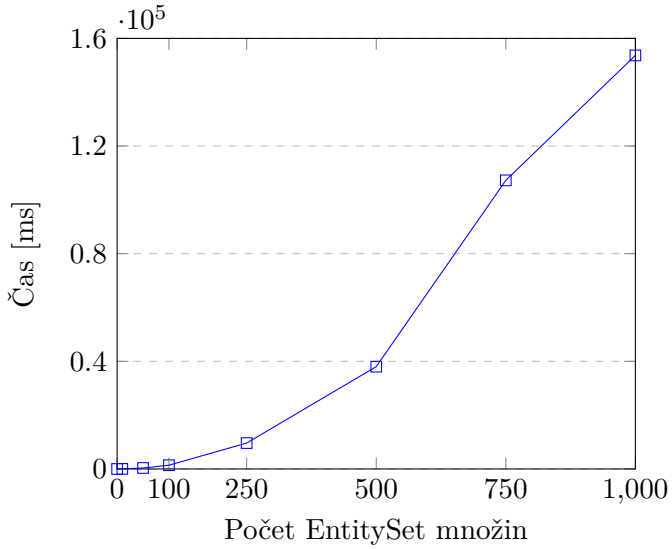
Při testování jsem narazil na jednu chybu, která se objevila v prohlížečích Internet Explorer a Edge. Pokud jsem HTML stránku s testovacím skriptem pustil v těchto prohlížečích jako lokální soubor, tak vyskočila chyba. Prohlížeče neznaly lokální úložiště a hlásili, že neexistuje. Po delším zkoumání se ukázalo, že se jedná o dlouho nahlášenou chybu, která existuje již v prohlížeči Internet Explorer od verze 9[3]. Z toho důvodu bylo tedy nutné pustit aplikační server a nechat si poslat testovací stránku z localhostu. Tím se problém vyřešil.

Druhý test na databázi byl zaměřen na složitost ukládání a načítání dat do databáze. Test má podobnou strukturu jako předcházející test. Rozdíl je v tom, že jsou pro jednotlivé testy dynamicky měněny počty *EntitySet* množin. První test obsahuje jednu množinu a poslední tisíc množin. Výsledky měření jsou zobrazeny v Tabulce 8.1.

Počet množin	Test 1 [ms]	Test 2 [ms]	Test 3 [ms]	Test 4 [ms]	Test 5 [ms]	Průměr [ms]
1	0.38	0.23	0.19	0.19	0.19	0.24
10	25.8	23.1	23.4	14.6	17.5	20.9
50	347.4	414.2	365.2	352.0	336.0	363.0
100	1443.0	1418.6	1390.8	1366.0	1323.3	1388.4
250	9954.5	9492.5	9797.0	9444.1	9429.0	9623.4
500	38174.8	38115.8	38065.9	37890.4	37855.2	38020.4
750	106163.2	108732.4	106858.3	105152.9	109315.3	107244.4
1000	153207.3	154663.8	153659.3	152845.3	153988.2	153672.8

Tabulka 8.1: Výsledky měření výkonnostní analýzy databáze.

Data z této tabulky jsou použit na vytvoření grafu na obrázku 8.2. Pro menší počet množin křivka připomíná kvadratickou funkci. Pro větší počet množin však začíná připomínat lineární funkci. Pro asymptotickou složitost je typické, že naměřená funkce má ze začátku výkyvy. Pro nekonečno však jasně vychází složitost lineární. Tedy patří do třídy složitosti $\mathcal{O}(n)$.



Obrázek 8.2: Složitost výsledku měření výkonnostní analýzy databáze.

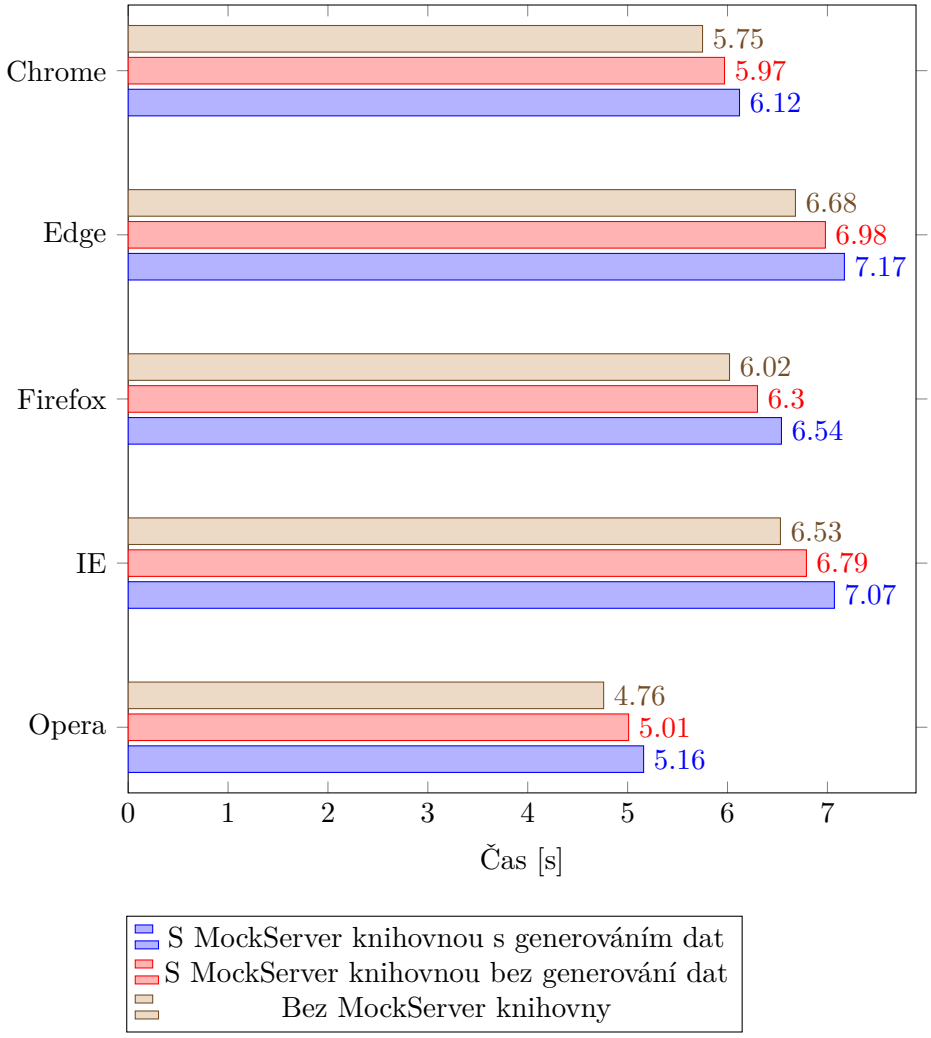
[3]Dostupné z: https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/650482/.

8.2 Načtení aplikace

Další test byl zaměřen na dobu načtení vzorové webové aplikace viz kapitola 7. V prvním případě se pustila aplikace bez knihovny *MockServer*, tedy se nenačetla žádná data. V druhé případě se použila knihovna *MockServer* a data se místo generování načetla z databáze. V posledním testu se použila knihovna *MockServer* a data se generovala dynamicky knihovnou.

K měření byly použity všechny webové prohlížeče definovány výše. Čas byl měřen až do úplného načtení stránky. Tento čas byl měřen pomocí vestavěným nástrojům k profilaci webové prohlížeče. Internet Explorer a Edge tuto informaci přímo nezobrazuje. Je potřeba si vyexportovat záznam z profilování prohlížeče do *.HAR* souboru. V tomto záznamu pak lze nalézt časová razítka pro první a poslední požadavky.

Z výsledků vychází, že nejpomalejší je verze s generováním umělých dat pomocí *MockServer* knihovny a nejrychlejší je verze bez *MockServer* knihovny. Rozdíly nejsou však příliš výrazné. Pro využití této knihovny jsou tyto časy naprosto dostačující. Vzhledem k tomu, že použití bude především pro vývoj a automatické testování.



Obrázek 8.3: Výsledky měření načítání vzorové aplikace bez a s *MockServer* knihovnou.

8.3 Požadavky na server

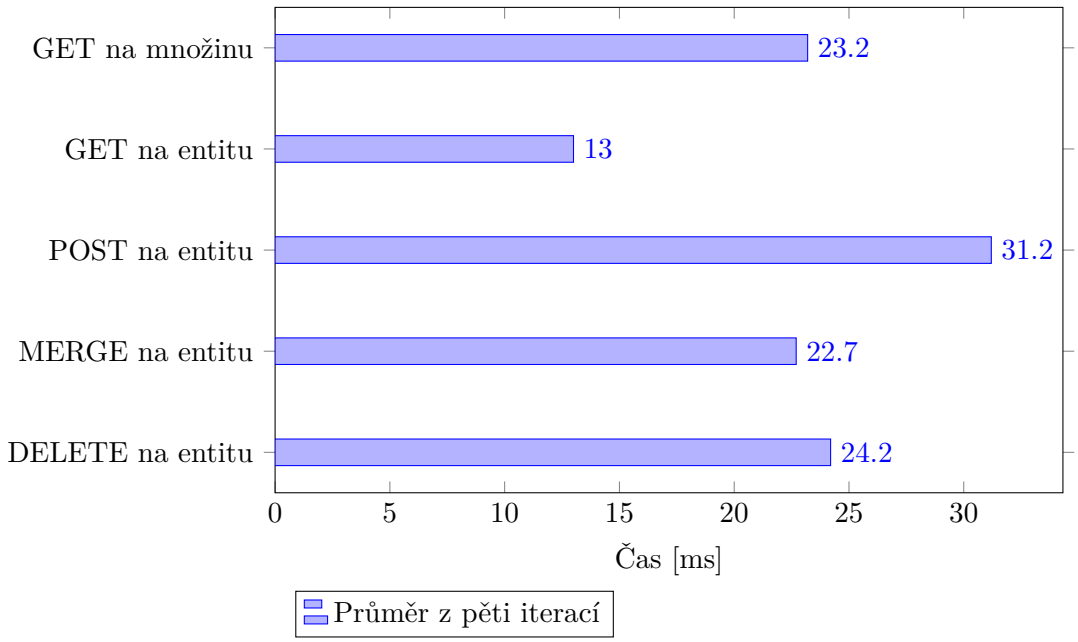
Poslední část byla zaměřena na testování délky zpracování jednotlivých požadavků na server. K měření byla použita vzorová aplikace viz kapitola 7. Měření probíhalo ve webovém prohlížeči Chrome ve verzi uvedené výše. Měřily se tyto požadavky:

- HTTP GET požadavek, který vrátí všechny entity z dané množiny.
- HTTP GET požadavek, který vrátí konkrétní existující entitu.
- HTTP POST požadavek, který vytvoří novou entitu.
- HTTP MERGE požadavek, který upraví existující entitu.
- HTTP DELETE požadavek, který smaže existující entitu.

Každý požadavek byl nejdříve zabalen do *batch* požadavku viz výše. Výsledky měření jsou zobrazeny v Tabulce 8.2. Tyto hodnoty byly zavedeny do grafu na obrázku 8.4.

Typ požadavku	Test 1 [ms]	Test 2 [ms]	Test 3 [ms]	Test 4 [ms]	Test 5 [ms]	Průměr [ms]
GET na množinu	22.9	21.2	22.1	24.1	25.9	23.2
GET na entitu	12.1	16.6	14.0	12.9	9.7	13.0
POST na entitu	32.8	28.0	29.4	33.7	32.5	31.2
MERGE na entitu	21.5	18.4	25.7	25.2	22.8	22.7
DELETE na entitu	23.5	24.3	21.2	26.0	26.3	24.2

Tabulka 8.2: Výsledky měření požadavků na server.



Obrázek 8.4: Výsledky měření načítání a ukládání dat do lokálního úložiště.

Z výsledků nejhůře vyšel požadavek na vytvoření nové entity. Nejrychleji je zpracován požadavek na konkrétní entitu. Všechny požadavky jsou zpracovány v dostatečně rychlém čase, aby byla knihovna použita při vývoji a testování. V mnoha případech se může ukázat, že lze knihovnu *MockServer* použít i v případě, že serverová část již existuje. Pokud má problém s vytížeností, tak lze použít knihovnu *MockServer*. Ta má konstantní čas odpovědi nezávisle na zatížení serverů.

Kapitola 9

Závěr

V této práci byly analyzovány nástroje pro vytváření umělých dat se zaměřením na protokol *OData*. V kapitole 2 byly prozkoumány dostupné nástrojů a došlo se k závěru, že zatím neexistuje řešení, které by plně implementovalo CRUD operace nad *OData*. Existují pouze možnosti, jak pro jednotlivá volání vracet staticky definovaná data, ale možnost skutečné umělé databáze, která bude správně reagovat na přidávání nových entit či mazání, neexistuje.

V kapitole 3 byl představen samotný protokol *OData* se zaměřením především na jeho klíčové aspekty. Vzhledem k obsáhlosti standardu by nebylo ani možné v této práci rozebrat všechny jeho elementy. Protokol *OData* je navíc velmi flexibilní a umožňuje si různé atributy a struktury dodatečně definovat dle potřeby.

Pro návrh bylo nutné vyřešit způsob uložení databáze na udržení umělých dat. Tato problematika byla rozebrána v kapitole 4, kde se jako nejpraktičtější jevila *NoSQL* databáze *IndexedDB*. Při následné implementaci knihovny se však ukázala tato databáze nevhodná a místo ní bylo použito lokální úložiště viz kapitola 6.

Další překážkou výsledného řešení byla chybějící logika ze serveru. Implementovaná knihovna umí validovat data na základě *metadata.xml* souboru. Bude ji však chybět serverová logika, která může obsahovat další validace. V ideálním případě by však data měla být validována samotným vývojářem klientské části aplikace. Příkladem může být číslo, které může obsahovat pouze sudé hodnoty. Takové omezení nelze v *metadata.xml* definovat, tzv. to nedokáže ověřit ani výsledná knihovna. Ta všechny omezení získává pouze z metadatového souboru.

Práce se také zaměřila na knihovnu *SAPUI5*, která byla použita při implementaci. Tato JavaScript knihovna společnosti SAP plně podporuje protokol OData. Použité třídy a dědičná hierarchie je popsána v kapitole 5. V této části je naznačen koncept celé knihovny *SAPUI5*, do které bylo výsledné řešení integrováno.

Samotná implementace je popsána v kapitole 6. Zde byl popsán základní tok knihovny. Od vytvoření instance konstruktorem přes registraci požadavků, které se mají odchytávat, až po samotné generování dat. Mezi ty zajímavější generování patří vytváření hodnot globálně unikátních identifikátorů dle RFC 4122. Dále je zde popsáno jaké požadavky je nutné odchytávat a jakým způsob samotné odchytávání probíhá.

Pro demonstraci výsledného řešení byla vytvořena ukázková webová aplikace viz kapitola 7. Pro validaci výsledné knihovny byly použity i další interní webové aplikace společnosti SAP.

Výsledné řešení bylo podrobeno výkonnostní analýze viz kapitola 8. Z výsledků bylo zjištěno, že odezva databáze je velmi rychlá. To je důležité zvláště pro jednotkové testy.

Jeden jednotkový test by neměl trvat déle než několik milisekund, jelikož je při vývoji dobrým zvykem psát stovky jednotkových testů. Ty se navíc spouštějí několikrát denně. Při vývoji klientské části webové aplikace je také nutné pouštět všechny jednotkové testy v pěti nejpoužívanějších webových prohlížečích. To několikanásobně prodlužuje čas vykonávání těchto jednotkových testů.

Dále se analyzovala délka načtení aplikace. Pokud vývojář bude používat pro vývoj klientské části místo reálného serveru vytvořenou knihovnu, tak nechce být omezován příliš dlouhým načítáním aplikace. Při samotném vývoji dochází často k opakovanému načítání webové aplikace po každé drobné změně v kódu. Pokud by byla výsledná knihovna špatně navržena a výrazně by zpomalovala načtení aplikace, tak by mohla vývojáře odradit od jejího používání. Z výsledků analýzy však plyne, že načtení webové aplikace je zpomaleno pod jednu vteřinu. To se dá považovat za téměř neznatelné a vývoj aplikace by neměl omezovat.

V rámci pokračování na práci by bylo možné rozšířit práci o *OData* verzi 4, na kterou se v budoucnu chystá SAP přejít. Dále by bylo možné rozšířit *MockServer* o vrácení hodnot ve formátu XML. SAP dnes používá pouze formát JSON, ale samotný *OData* formát podporuje i XML. Také by bylo možné vytvořit podporu pro typ *EnumType*, který rovněž není podporován společností SAP. Toto rozšíření by mohlo být užitečné pro aplikace, které používají *OData* protokol, ale nepoužívají SAP technologie.

Literatura

[1] Arocena, G. O.; Mendelzon, A.; Mihaila, G. A.: Applications of a Web query language. *Computer Networks and ISDN Systems*, ročník 29, 1997: str. 1305–1316.

[2] Ater, T.: *Building Progressive Web Apps: Bringing the Power of Native to the Browser*. O'Reilly Media, 2017, ISBN 1491961651.

[3] D.Kombade, R.; Meshram, B.: CSRF Vulnerabilities and Defensive Techniques. *IJCNIS*, ročník 4, č. 1, 2012: s. 31–37.

[4] Feathers, M. C.: *Working effectively with legacy code*. Prentice Hall PTR, 2005, ISBN 978-0131177055.

[5] Gregorio, J.; de hOra, B.: The Atom Publishing Protocol. RFC 5023, RFC Editor, October 2007.

[6] Jadhav, M. A.; Sawant, B. R.; Deshmukh, A.: Single Page Application using AngularJS. *International Journal of Computer Science and Information Technologies*, ročník 6, 2015: str. 2876–2879.

[7] Kimak, S.; Ellman, J.: The role of HTML5 IndexedDB, the past, present and future. *International Conference for Internet Technology and Secured Transactions*, ročník 10, 2015: str. 379–383.

[8] Kirchhoff, M.; GEIHS, K.: Semantic description of OData services. *Proceedings of the Fifth Workshop on Semantic Web Information Management*, 2013: s. 1–8.

[9] Kreibich, J.: *Using SQLite: Small. Fast. Reliable. Choose Any Three.* O'Reilly Media, 2010, ISBN 9780596521189.

[10] Nottingham, M.; Sayre, R.: The Atom Syndication Format. RFC 4287, RFC Editor, December 2005.

[11] Nottingham, M.; Sayre, R.: A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, RFC Editor, July 2005.

[12] Osherove, R.: *The art of unit testing: with examples in C#*. Manning, 2014, ISBN 9781617290893.

[13] Sharp, R.: Introducing Web SQL Databases. HTML5 Doctor, 2010, [Online; navštíveno 7.1.2017].
URL http://html5doctor.com/introducing-web-sql-databases/

[14] West, W.; Puliwood, S.: Analysis of Privacy and Security in Html5 Web Storage. *Journal of Computing Sciences in Colleges*, ročník 1, č. 8, 2011.

[15] Integration Testing with One Page Acceptance Tests (OPA5). SAP SE, 2016, [Online; navštíveno 30.12.2016].
URL https://sapui5.hana.ondemand.com/#docs/guide/2696ab50faad458f9b4027ec2f9b884d.html

[16] SessionStorage. Mozilla Developer Network, 2017, [Online; navštíveno 7.1.2017].
URL https://developer.mozilla.org/cs/docs/Web/API/Window/sessionStorage

[17] Browser storage limits and eviction criteria. Mozilla Developer Network, 2015, [Online; navštíveno 9.1.2017].
URL https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria

Přílohy

Příloha A

Obsah přiloženého paměťového média

Obsahem přiloženého paměťového média je:

- Text diplomové práce ve formátu PDF.
- Zdrojové soubory diplomové práce pro systém LaTeX.
- Zdrojové soubory aplikace.
- Zdrojové soubory knihovny.
- Zdrojové soubory testovací sady.
- Soubor *Readme.txt*.