Univerzita Pardubice

Dopravní fakulta Jana Pernera

Framework pro tvorbu webových informačních systémů

Bc. Jan Jindra

Diplomová práce 2019

Univerzita Pardubice
Dopravní fakulta Jana Pernera
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Jindra**
Osobní číslo:	**D17443**
Studijní program:	**N3708 Dopravní inženýrství a spoje**
Studijní obor:	**Aplikovaná informatika v dopravě**
Název tématu:	**Framework pro tvorbu webových informačních systémů**
Zadávající katedra:	**Katedra informatiky v dopravě**

Zásady pro vypracování:

Cílem diplomové práce je navrhnout a implementovat prototyp frameworku pro usnadnění tvorby základního jádra a modulů webových informačních systémů. Výsledkem bude nástroj pro programátora, který na základě definovaných modulů pomůže předpřipravit základní rozvržení nového informačního systému.
Framework bude orientován především na administraci webových systémů, napříč kterými se často prolínají základní prvky (moduly). Výsledkem diplomové práce bude nástroj, který programátorovi nabídne seznam těchto modulů připravených dle požadavků navrhovaného informačního systému. Dále mu poskytne nástroje pro tvorbu unikátních částí systému, které se musí přizpůsobit požadavkům zadání.
Další požadavky na funkcionalitu frameworku:

- Možnost skládání jednotlivých modulů do finální podoby informačního systému.
- Vytváření libovolných kombinací modulů a jejich vzájemné propojení.
- Znuvupoužitelnost každého modulu a možnost si každý modul modifikovat pro dané použití.
- Snadné rozšiřování databáze existujících modulů.

Rozsah grafických prací:

Rozsah pracovní zprávy: **40 normostran**

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

Seznam odborné literatury:

WELLING, Luke a Laura THOMSON. Mistrovství PHP a MySQL. Přeložil Ondřej BAŠE. Brno: Computer Press, 2017. ISBN 978-80-251-4892-1.
ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.
ZAKAS, Nicholas C. JavaScript pro webové vývojáře. Brno: Computer Press, 2009. Programujeme profesionálně. ISBN 978-80-251-2509-0.

Vedoucí diplomové práce: **Ing. Stanislav Machalík, Ph.D.**
Katedra informatiky v dopravě

Datum zadání diplomové práce: **11. prosince 2018**

Termín odevzdání diplomové práce: **24. května 2019**

L.S.

doc. Ing. Libor Švadlenka, Ph.D.
děkan

doc. Ing. Karel Greiner, Ph.D.
vedoucí katedry

V Pardubicích dne 11. prosince 2018

Prohlašuji:

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

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

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012 Pravidla pro zveřejňování závěrečných prací a jejich základní jednotnou formální úpravu, ve znění pozdějších dodatků, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 5. 2019

Jan Jindra

Poděkování

Chtěl bych poděkovat panu Ing. Stanislavu Machalíkovi, Ph.D. za odborné vedení mé práce a za poskytnuté konzultace k danému tématu.

ANOTACE

Diplomová práce se zabývá tvorbou prototypu softwarového nástroje – frameworku, který usnadňuje a zefektivňuje vývoj informačních a administračních webových systémů. Výsledkem práce je nástroj pro programátora, který programátorovi dokáže ulehčit práci a přidat do ní více znovupoužitelnosti. V práci je uveden rozbor architektury, ze které je výsledný framework sestaven. Následně jsou zkoumány, rozebírány a modifikovány konkrétní nástroje, které napomáhají k fungování celého principu a pomáhají k tvorbě systémů, které mohou vznikat díky tomuto frameworku. Mezi tyto nástroje lze zařadit generátory, které dokáží velice jednoduše vytvářet formuláře či tabulky včetně veškeré jejich obsluhy. Ke konci práce je rozbor a popis následné modularizace a možnosti rozdělení systému do modulů, které lze navzájem kombinovat a využívat tak, aby došlo k zefektivnění práce programátora.

KLÍČOVÁ SLOVA

Tvorba systému, informační systémy, generování kódu, návrh architektury systému.

TITLE

Universal framework for the development of web information systems

ANNOTATION

I deal with development of prototype of software tool in my thesis. Specifically it is framework that makes developing the information and administration web systems easier and more effective. Result of my thesis is tool for programmer that can add his job easier and this tool can adds more reusability. There is architecture analysis in the thesis which the framework is composed. Next steps are specifically tools that are examined, dismantled and modified. These tools help with working principles and developing new systems that are based on this framework. Example of these tools is generator. Generator can construct forms and grids by simply way. Finally there is analysis and description of modularization with possibilities of separation to the modules. These modules can be combined and used for more effective work of programmer.

KEYWORDS

Software development, information systems, code generation, system architecture design

Obsah

Seznam ilustrací a tabulek

Terminologie

Agilní způsob programování – metodika vývoje softwaru, která popisuje spolupráci vývojového týmu a způsob, jak přistupovat k vývoji, a doporučuje základní principy, kterých se je třeba držet při tvorbě softwaru.

API - Application Programming Interface je rozhraní programu, které nám bez nutnosti znalosti vnitřní struktury umožňuje pracovat s programem.

B-Crypt – jedná se o kvalitní hashovací funkci. Často využívanou pro tvorbu hash otisků hesel.

Boolean – datový typ reprezentující stav dvou hodnot: pravdu „true“ (odpovídá 1), nepravdu „false“ (odpovídá 0).

Cache – mezipaměť pro ukládání dat, která je složité či časově náročné získat. Ukládají se ve zpracované podobě a mohou zrychlit přístup k těmto datům.

Callback – zpětné volání funkce bez toho, aby hlavní proces na výsledek funkce vyčkával.

Cookies – informace či data, která si vyměňují skrze protokol HTTP prohlížeč a server. Mohou uchovávat základní informace o instanci spojení a zároveň slouží jako malé persistentní úložiště, které lze využít pro uživatelská data.

CSS – Cascading Style Sheets nebo kaskádové styly, které lze chápat jako popisný programovací jazyk, který slouží k stylování a popisu grafického zobrazení HTML kódu.

DI - Dependency injection je programovací technika, která se dá volně přeložit jako vkládání závislostí, aniž bychom na tyto závislosti měli v místě inicializace reference.

Doktrína – v práci myšleno jako PHP knihovna pro práci s databází. Dostupná je na https://www.doctrine-project.org.

Entita – objekt, který uchovává data. Je to zobrazení reálného objektu v systému. V práci obvykle spojován jedna ku jedné s databázovou tabulkou.

Escapování – metoda pro odebrání speciálních znaků z řetězce, které jsou klíčové pro skládání programovacího kódu – nejčastěji SQL kódu.

Exception – výjimka či chyba programu, která může nastat při běhu. Výjimku lze zachytávat, aby nedošlo k pádu programu.

GitHub – webová služba pro verzování a webhosting open sorce projektů.

Hash, hashovací funkce – matematická funkce, které zašifruje vstupní data do otisku - hashe.

HTML5 – Hypertext Markup Language 5. Je specifikace značkovacího jazyka HTML, který se používá k popisu webových stránek. Verze 5 byla vydána 28. října 2014.

HTTP – Hypertext Transfer Protocol – protokol pro komunikaci mezi webovým klientem/prohlížečem a serverem. Slouží například pro přenos HTML dokumentů.

JS – JavaScript. Objektově orientovaný skriptovací jazyk vycházející z jazyka C/Java. Používá se pro programování na straně webového klienta/internetového prohlížeče.

JSON – JavaScript Object Notation je způsob zápisu objektu JavaScriptu do datové podoby, která se používá pro přenos dat.

Minifikace – způsob komprimace CSS nebo JS souborů do kompaktnější podoby. Zpravidla probíhá odebíráním nepoužitého kódu či zbytečných netisknutelných znaků.

MVC – model-view-controler – softwarová architektura, která rozděluje aplikaci na 3 vrstvy. První je datový model aplikace, druhý řídící logika a třetí uživatelské rozhraní.

MVP – model-view-presenter **–** upravená architektura MVC, kde řídící a kontrolní logika je nahrazena presenterem.

MySQL – databázový software.

Open source – často je myšlena licence, pod kterou je poskytován výsledný program či kód, kde základem je to, že je zdrojový kód veřejný a volně k užití.

PHP – Hypertext Preprocessor. Objektově orientovaný programovací jazyk pro tvorbu webových aplikací.

PHP objekt DateTime – objekt, který v jazyce PHP uchovává datum a čas.

React – JavaScriptový framework.

Refaktorovat kód – přepis zdrojového kódu pro vylepšení jeho vnitřní struktury bez vnějšího vlivu na chování aplikace.

Request – požadavek. Často myšleno jako HTTP Request, který je požadavkem od webového klienta (prohlížeče) na server pro provedení dané akce.

Response – odpověď na request – odpověď jako http Response na daný požadavek od serveru pro klienta (prohlížeč).

Session – relace dané webové stránky jako permanentní úložiště. Často se používá pro udržení přihlášeného uživatele do webových stránek.

Textbox input – HTML formulářové políčko pro zadávání textu.

TypeScript – programovací jazyk, který je nadstavbou JavaScriptu. Výsledný kód se kompiluje zpět do JS.

URL – Uniform Resource Locator. Doménová adresa, která udává specifikaci umístění zdroje v rámci protokolu HTTP.

XML – Extensible markup language – značkovací jazyk používaný pro serializaci dat.

Úvod

Cílem mé práce je navrhnout, zrealizovat a zhodnotit nástroj pro usnadnění tvorby webových systémů. K tomuto tématu jsem se dostal při vykonávání své profese programátora webových aplikací, kterou již vykonávám přes více než 5 let. Primární oblastí, kterou se ve své profesi zabývám, je tvorba webových aplikací pro malé a střední firmy. Zde z mé práce vyvstala idea stvořit nástroj či systém, který by mi usnadnil a pomohl s tvorbou dalších systémů.

Základním požadavkem většiny zákazníků je vyřešení nějakého problému či požadavku v jejich podnikání za pomoci sytému tzv. „na míru“. Často si nevybrali z nabídky již existujících systémů či programů řešících danou problematiku, anebo se jednalo o problematiku, která je unikátní či něčím speciální a systém pro ni ještě neexistuje. Proto přiblížit k tomu nějaký existující systém by bylo značně složité. V tomto případě pak přichází na řadu nechat si vytvořit systém úplně nový, který bude řešit jejich specifické požadavky a nebude mít extra omezení pro případnou dynamiku rozšiřování funkcionalit. Zároveň mají menší firmy, pro které je výsledné řešení určeno, často mnoho požadavků, protože v jejich podnikání je denním chlebem business, který se přizpůsobuje a proměňuje v čase na základě přání jejich zákazníků.

Nicméně vyvíjet takovéto systémy je celkem náročné, protože každý je upraven přesně podle požadavků daného zákazníka. Dle mých dosavadních zkušeností není zvykem, že by se dané řešení dalo použít pro více zákazníků bez složitějších úprav. Každý z těchto systémů se zabývá řešením něčeho úplně individuálního, co požadoval zákazník, a není úplně jednoduché nalézt pro tato řešení nějaký již hotový systém, který by si zákazník jednoduše zakoupil a případně si jej nechal lehce modifikovat. Pokud se do problematiky ponoříme trošku hlouběji, zjistíme, že existují určité sekce a principy, které se v některých řešeních opakují pořád dokola. V rámci jednotlivých systémů dochází u těchto sekcí a principů k modernizacím a úpravám, ale jejich základ zůstává vždy hodně podobný.

A z tohoto všeho vyplývá úkol mé práce. Cílem je tedy připravit a následně zhodnotit nástroj pro programátora (framework), který mu pomůže s usnadněním tvorby základního jádra a tvorby základních modulů daného webového systému, které se určitým způsobem mohou opakovat v různých modifikacích v jiných systémech, a tímto mu ulehčit práci při tvorbě webového systému „na míru“ pro zákazníka.

Pro to, abych dosáhl svého cíle, chci využít stávajících technologií a postupů práce, které již existují. A zároveň částečně modifikovat tyto existující technologie či vytvořit úplně nové, které by zefektivnily práci systému, a to od úrovně jádra systému až po konkrétní nástroje pro vývoj dalších nových systémů.

1 Softwarové technologie pro vývoj frameworku

Pro realizaci takovéhoto typu nástroje jsou stanoveny obecné požadavky, které buď vyvstaly přímo od zákazníků, anebo jsou dány například infrastrukturou, na které musí výsledné systémy běžet. Po zvážení všech těchto požadavků byl proto zvolen pro vývoj systému jeden z nejrozšířenějších postupů, a to kombinace jazyků PHP, JavaScriptu, MySQL, HTML a CSS. Avšak není třeba systém stavět úplně kompletně od začátku, lze využít už určité nástroje a dané postupy či frameworky postavené nad těmito technologiemi a z každého si vypíchnout a vzít to, co se reálně využije v novém nástroji. Cokoliv, co je použito, se snažím nejprve řádně zanalyzovat a prozkoumat, zda to vyhovuje mým nárokům a požadavkům na kvalitu a bezpečnost. Pokud z analýzy vyplyne, že daný nástroj pro mě není vhodný, či není tak efektivní, jak bych potřeboval, navrhnu a naimplementuji si nástroj nový.

K celému novému systému bych chtěl přistupovat aktuálně moderním přístupem a využít princip kombinace REST API na serveru a na ni napojit modul postavený pomocí JavaScriptového frameworku. Na API jsem se rozhodl využít český Nette framework, který u nás má velkou oblibu a je k němu mnoho materiálů a podkladů a celkově je považován za jeden z nejstabilnějších a neoblíbenějších PHP frameworků. Na JavaScriptového klienta jsem se rozhodl použít knihovnu či framework React od tvůrců Facebooku, a to hlavně díky jeho velké rozšířenosti, kompletní dokumentaci a stabilitě.

1.1 REST API

REST neboli Representational State Transfer je v podstatě princip či architektura, která nám popisuje, jak pomocí klasických HTTP requestů můžeme vytvořit, číst, editovat anebo smazat daná data na serveru. (Massé, 2012, s. 5-6)

API neboli Application Programming Interfaces je rozhraní, které programátor může využívat, aby mohl obsluhovat danou knihovnu, program či systém. Dle toho, jak je API pro daný systém/program definované, můžeme skrze něj jakkoliv využívat vnitřní funkcionalitu daného sytému/programu. API je v podstatě vnější rozhraní pro manipulaci s daným předmětem, které API poskytuje. (Massé, 2012, s. 5-6)

Kombinací API a REST získáváme architektonický styl používaný pro komunikaci mezi klientem a serverem. Server poskytuje API, na které se pomocí REST principu připojuje

klient a pomocí HTTP requestů s ním komunikuje – posílá mu požadavky. Jedná se v základu o jednostrannou komunikaci – klient se ptá a server odpovídá. (Massé, 2012, s. 5-6)

V dnešní době se nejčastěji používá právě tento princip, kde si server s klientem předává komunikaci v rámci volání HTTP requestů a data si předávají v JSON souborech – bez nutnosti ze serveru přenášet komplet celý html kód se posílají jenom data ve strukturovaném JSON soboru a klient na to reaguje například překreslením jenom dílčí části stránky (nepoužívá se refresh).

1.2 Composer, správce PHP knihoven

Composer je nástroj na správu knihoven a jejich závislostí v PHP. Dokáže sestavit strom závislostí z požadovaných knihoven PHP pro daný projekt a tyto knihovny dokáže aktualizovat tak, abychom pracovali s aktuálními a zároveň navzájem kompatibilními verzemi. Tento nástroj se instaluje samostatně do systému a nebývá součástí projektu. (getcomposer.org, Introdution)

Obsluhuje se přímo přes příkazovou řádku systému, kde se zadávají příkazy. Mezi ty hlavní a nejdůležitější patří příkaz na stažení PHP knihovny například z GitHubu, odkud danou knihovnu stáhne a nastaví pro využití v našem projektu. Knihovnu stáhne i včetně jejích závislostí a tyto závislosti správně nastaví, aby se programátor nemusel dále o nic starat a mohl čistě knihovnu využít. Knihovny stahuje na disk do adresáře vendor, který si vytvoří v kořenové struktuře projektu, a definice knihoven, které má nainstalované, tak ukládá do souboru composer.json. (getcomposer.org, Basic usage)

Z vlastní zkušenosti vím, že pomocí Composeru lze stahovat i celý PHP framework a pak do něj přidávat různé další knihovny, které potřebujeme využít. Anebo naopak odebírat knihovny, které v daném frameworku nevyužijeme. Composer by nám měl do určité míry zajistit, že všechny aktuální nainstalované knihovny budou funkční, správně nastavené a připojené k projektu.

1.3 PHP framework Nette

PHP framework Nette je soubor samostatně použitelných komponent pro PHP. Dohromady tvoří framework, který vyvíjejí už přes 10 let a dá se považovat za stabilní a široce používaný. Díky tomu, že je složen ze samostatně použitelných komponent/knihoven, jej lze využívat v kombinaci s jinými frameworky a nebo je využívat samostatně. Poskytován je pod Open

source licencí, a tudíž je zdarma dostupný i pro komerční využití. Zároveň nám pomáhá eliminovat bezpečnostní díry v programu. Celý framework je založen na objektovém principu MVP (model-view-presenter), který je odvozen od klasického MVC (model-view-controler). (nette.org, 2019)

1.3.1 Využité komponenty z Nette

Níže jsou zmíněné vybrané komponenty/knihovny z balíčků Nette frameworku. O ostatních knihovnách se můžete dočíst na online oficiální dokumentaci na adrese: https://doc.nette.org/.

„nette/di“ neboli knihovna pro řešení Dependency Injection (DI):

„Podstatou Dependency Injection (DI) je odebrat třídám zodpovědnost za získávání objektů, které potřebují ke své činnosti (tzv. služeb), a místo toho jim služby předávat při vytváření.“ (nette.org, Dependency Injection)

DI nám dovolí vyžádat si kdekoliv v programu danou PHP třídu bez toho, abychom museli znát vše, co je potřeba pro její inicializaci. Například pro inicializaci třídy, která nám řeší připojení k databázi, tu vždy musíme předat konfiguraci pro připojení k dané databázi (minimálně typ databáze, přihlašovací údaje a název dané databáze). Pokud ale využíváme DI, nemusíme. V kódu si vyžádáme instanci třídy pro práci s databází a všechno potřebné nastavení za nás už DI vyřídí a my dostaneme již vše připravené. (nette.org, Dependency Injection)

„nette/caching“ neboli knihovna řešící cache:

Cache slouží k uložení dat, která jsou náročná na získání či výpočet. Obecně se cache používá ke zrychlení systému uložením nějakých stavů či dat. Probíhá vždy jako ukládání klíč – hodnota a funguje tak, že například místo volání funkce se nejprve zkontroluje, zda výsledek funkce nemáme již v cache uložišti. Pokud ano, vrátí nám tato data a už funkci znovu nevykonává. Pokud tam tato data nemáme, funkce se vykoná a před tím, než data vrátí, uloží je do cache. Nette má pro práci s cache tuto knihovnu a díky ní zajišťuje plnou atomicitu operací. Navíc můžeme nativně skrze tuto knihovnu provádět různé typy cachování. Podporuje například i MemcachedStorage, který ukládá data přímo do RAM paměti. Cache lze díky této knihovně jednoduše expirovat a invalidovat. Expirace nejčastěji bývá časová – tzn. můžeme si definovat, že určitý typ cache má danou životnost, která se nejčastěji uvádí v minutách. V praxi to vypadá tak, že se do cache uloží daný záznam a po uplynulých

minutách ztratí svou platnost a je potřeba vytvořit cache novou a tou nahradit stávající záznam. Invalidace (řízené mazání) probíhá nezávisle na čase a můžeme jej zavolat kdekoliv z kódu a řídí se zadanými parametry. To se nejčastěji používá tak, že si dáme příznaky (tagy) k dané cache a pokud nám ji někdo změní (upraví třeba záznam v databázi), zavoláme na ni v metodě ukládání dané entity invalidaci. (nette.org, Cache)

„tracy/tracy“ ladící nástroj Tracy:

Používá se k ladění PHP kódu a vizualizaci při vývoji a logování chyb v produkčním běhu systému. Dále slouží ke sledování a měření času běhu skriptu, sledování provedených SQL dotazů, načtených tříd pomocí DI a mimo jiné i pro „dumpování“ proměnných (obdoba PHP příkazu var_dump). (nette.org, Tracy)

„neon/neon“ neboli knihovna pro konfigurační soubory:

Používá se pro práci se soubory s příponou .neon, které se v Nette používají pro zápis konfigurací. Obsahuje funkce pro práci s tímto typem souboru včetně serializace a validace dat v těchto konfiguračních souborech. (nette.org, Neon)

„dibi/dibi“ knihovna pro práci s databází Dibi:

Dibi slouží jako abstraktní databázová vrstva pro práci a přístup k databázi. Zjednodušuje zápis SQL příkazů a zefektivňuje práci programátora. Dibi není ani knihovna pod Nette, ale stojí spíše samostatně, nicméně je od stejných autorů, jako je Nette, lze ji dobře používat a má přímou integraci do knihoven Nette. (Grudl, dibiphp.com, 2008-2019)

1.4 Správce JS balíčků NPM

NPM slouží pro správu balíčků JavaScriptu. Funguje v podstatě podobně jako Composer, akorát na JavaScript. Pod NPM je veřejný registr Open source balíčků, které si můžeme do našeho projektu natáhnout. Po stažení balíčku se nám stáhnou i všechny jeho závislosti a balíček se připraví k využití. (docs.npmjs.com, About npm)

1.5 jQuery a AJAX

jQuery je jedna z nejrozšířenějších knihoven JavaScriptu, která zjednodušuje práci se základním JavaScriptem. Dokážeme skrze ni přistupovat přímo k elementům dokumentu bez nutnosti psát spoustu řádků navíc. Využívá selektory pro přístup k modelu DOM

(Document Object Model) podobným selektorům v CSS. Dále zjednodušuje práci pro obsluhu reakcí od uživatele. Kde nabízí oblíbený způsob jak zachytávat a obsluhovat dané události (např. „onClick“ – při kliknutí na daný element, „onChange“ – při změně daného textového pole apod.). Zjednodušeně řečeno, knihovna jQuery dává programátorovi velice mocný nástroj na práci s JavaScriptem. (Chaffer, Swedberg, 2013, s. 23-25)

AJAX neboli asynchronní JavaScript a XML (Asynchronous JavaScript and XML) slouží k načtení dat ze serveru či odeslání dat na server bez nutnosti obnovit stránku. AJAX je základní součástí JavaScriptu a v současné době představuje základní princip jak komunikovat mezi klientem a serverem z JavaScriptových aplikací, které běží čistě v prohlížeči uživatele. jQuery nabízí plnohodnotnou podporu AJAXu a nabízí k němu velice robustní metody jak jej používat a velice zjednodušuje práci s ním. (Chaffer, Swedberg, 2013, s. 23-25)

1.6 Webpack a Gulp

Webpack je knihovna pro kompilaci JavaScriptu. JS frameworky anebo obecně i JavaScript se nemusí zapisovat pouze do souborů s příponou .js, ale existují i další standarty a metody jak psát JS. Například TypeScript, který je třeba kompilovat do JS, aby bylo možné jej spustit v prohlížeči. Při kompilaci může zároveň probíhat i minifikace výsledného JS souboru. Webpack tedy funguje tak, že na vstupu vezme náš projekt napsaný například v TypeScriptu, který je rozdělen do desítek či stovek souborů, a tyto soubory zkompiluje (převede vše do zápisu v čistém JS) a vytvoří nám jeden výsledný minifikovaný JS soubor, ve kterém je celá naše JS aplikace. (webpack.js.org)

Knihovna Gulp funguje hodně obdobně jako knihovna Webpack. Obecně je knihovna Gulp nástroj pro automatizaci úkolů a slouží pro kompilaci souborů z preprocesorů do jednoho CSS souboru a pro kompilaci JS souborů. Gulp lze nastavit tak, že je v režimu watch – což znamená, že dané soubory sleduje a jakmile dojde do některého z nich k zápisu a uložení, provede automatickou rekompilaci (pozn.: i knihovna Webpack má režim watch). (Linha, blog.wedream.cz, 2016)

Já osobně knihovnu Webpack používám ke kompilaci projektů v JS frameworcích a knihovnu Gulp ke kompilaci CSS souborů a externích JS souborů do jednoho CSS a jednoho JS minifikovaného souboru.

1.7 Framework React

React je v současné době velice oblíbená JavaScriptová knihovna, která slouží pro vytváření komponent. Oproti jiným robustním JS frameworkům se React spíše řadí mezi knihovny než úplně frameworky. Slouží pro reprezentaci prezentační vrstvy. Největší výhoda je ta, že za nás React kompletně řeší komunikaci s DOM JavaScriptu. Protože největší úskalí JS DOMu je to, že je pomalý. React si vytváří svůj virtuální DOM a řeší nad ním aktualizaci a překreslování svých komponent. (Facebook Inc., reactjs.org, 2019)

1.8 JS knihovna Axios

Pro komunikaci se serverovým API nepožívám v Reactu klasický jQuery AJAX, ale malou knihovnu Axios, která umožňuje výměnu informací mezi serverem a klientem v podobě Reactové aplikace. Axios je jednoduchá alternativa k realizaci této komunikace. Výhodou využití Axios oproti jQuery je velikost. jQuery je komplexní knihovna pro práci s JavaScriptem, což nám v JS aplikaci zajišťuje nadstavba Reactu. Nicméně sám React není nijak závislý na jQuery, takže proto tuto knihovnu jej nemusíme využívat a na komunikaci mezi server – klient můžeme využít například zmíněný Axios. (www.npmjs.com, axios, 2018)

1.9 HTML, CSS a JS knihovna Bootstrap

Bootstrap je Open source nástroj pro tvorbu a vývoj s pomocí HTML, CSS, JS. Pomáhá nám ve stavbě uživatelského prostředí za pomoci připravených nástrojů. Automaticky již řeší responzivní design pro všechny možné velikosti displejů a je založený na předpřipravených CSS souborech a jQuery knihovnách. (getbootstrap.com)

Tento nástroj má v základu předdefinované základní CSS styly a dokáže programátorovi pomoci s tvorbou základního GUI. V Bootstrapu lze napsat jakoukoliv šablonu webu. Ať už se jedná o designový web anebo základní administraci. Pokud si nechceme psát všechny CSS a JS soubory od začátku, Bootstrap není špatnou volbou.

1.10 JSON Schema

JSON Schema jsou v podstatě pravidla pro to, jakým způsobem pracovat a validovat JSON soubory. Není to ještě ověřený standard, ale obecně se dá říci, že je uznávaný. JSON Schema nám udává, jak mají vypadat data zapsaná v JSON souboru. (Droettboom, json-schema.org, 2019)

2 Bezpečnostní rizika webových aplikací

V každé nově vznikající internetové aplikaci či systému by se každý programátor měl zamyslet nad tím, jak řešit a eliminovat bezpečnostní rizika a díry ve svém programu. Moderním trendem je odpovědnost za bezpečnost, která se často přenáší na nějaký určitý framework či knihovnu. Tento přístup je pohodlný, nicméně každý programátor si musí uvědomovat alespoň základní bezpečnostní rizika, která existují, a měl by vědět, jak se jich vyvarovat a zároveň jak psát aplikaci, šifrovat hesla a ukládat data. V dnešní době je běžné, že i největší a nejdokonalejší systémy či frameworky v sobě mohou mít chyby a bezpečnostní nedokonalosti.

V práci jsou uvedeny základní a veřejně známé chyby a rizika z hlediska webové aplikace. Neuvádím problémy, které se týkají serverové infrastruktury či zabezpečení jednotlivých uživatelských rolí a přístupů v rámci systému.

Tato bezpečnostní rizika či opatření, která jsou zde uvedena, jsou ve výsledném systému řešena a eliminována. V systému je několik míst náchylných například k odposlouchávání komunikace, takže na výsledných systémech musí být vždy nasazen SSL certifikát pro HTTPS protokol, který nám zajistí šifrování této komunikace. Dále je dosti důkladně řešen SQL Injection, a to hlavně v místech, kde se v systému otevírá přímý přístup k databázi tak, aby případný útočník této rozšiřující funkcionality nemohl zneužít ve svůj prospěch a ze systému získat data či jej poškodit.

2.1 HTTPS

První a základní věcí, jak můžeme zabezpečit svůj webový projekt, je nasadit si šifrovaný protokol HTTPS. Tento protokol není nic jiného než klasický HTTP, ale poslaný skrze šifrovaný kanál za pomoci protokolu SSL (Secure Socket Layer). Toto šifrování chrání pouze komunikaci mezi klientem a serverem. Neřeší vůbec to, že by potencionální útočník napadl přímo klienta anebo server. Prakticky to znamená, že pokud se bude někdo snažit odposlouchávat vaši komunikaci, dostane se pouze k šifrovaným datům. Pokud používáte pouze klasický HTTP protokol, při odposlouchávání dostává kompletní data včetně dat odeslaných ve formuláři – což u přihlašovacího formuláře do webového systému znamená, že se zadarmo dostal k heslu uživatele. (Huseby, 2006, s. 26-27)

Nasazení HTTPS na web je v dnešní době jenom otázkou konfigurace, protože existují i certifikáty, které jsou vydávané bezúplatně. Během své praxe jsem se nesetkal se zneužitím webové stránky skrze odposlouchávání nezabezpečeného protokolu HTTP. Je to však způsobeno tím, že na všech projektech, na kterých jsem dosud pracoval, byl použit zabezpečený protokol HTTPS. Avšak ve svém okolí nacházím i nové projekty, které vývojáři spouštějí pod nezabezpečenou formou protokolu. Myslím si, že je to v dnešní době již bezpečnostní prohřešek.

2.2 SQL Injection

SQL Injection je problém, který vzniká na úrovni serveru, ovlivňuje databázi a patří mezi závažné bezpečnostní problémy. Jedná se o to, že útočník nám dokáže ovlivňovat SQL příkazy, které jdou do databáze prostřednictvím základních funkcí aplikace. Nejčastěji zneužívá skládání řetězců do SQL příkazu a vnoří si do nich svůj kód navíc. Problém vzniká ve chvíli, kdy na úrovni aplikace nejsou ošetřeny speciální znaky, které se používají pro tvorbu SQL příkazů. (Huseby, 2006, s. 32)

Pokud útočník v daném systému nalezne neošetřený vstup, který má možnost ovlivňovat databázi, díky SQL Injection dokáže získat kompletní přístup ke všem datům, která se v dané databázi nachází. Nebo lze pomocí něj daná data modifikovat, zašifrovat či smazat.

Prevence SQL Injection se řeší především tak, že se buď „escapují“ speciální znaky, či se využije knihovna pro práci s databází, která za nás bude tento problém řešit. Nicméně je třeba u každé takovéto knihovny řádně studovat dokumentaci, aby programátor věděl, jak korektně ji používat a jak tomuto problému zabránit.

2.3 Cross-site Scripting

Cross-site Scripting u webových aplikací spočítá v tom, že nám útočník do HTML kódu dostane svůj vlastní škodlivý kód v JavaScriptu. Nejčastěji tento typ útoku vzniká od neošetřeného vstupu. Takže útočník nám podstrčí svůj skript, který pak webový server odesílá uživatelům v rámci HTML kódu. Účelem takovéhoto skriptu je nějak ovlivnit či zneužít klientskou část aplikace. Tento typ útoku tedy spíše cílí na uživatele než na webovou aplikaci. Útočník může v tomto případě zneužít celou řadu věcí. Například přihlašovací formuláře, cookies, díky kterým mu může ukrást jeho identitu či relaci (session), může pozměnit obsah stránky či provést přesměrování. JavaScript je plnohodnotný programovací

jazyk a díky Cross-site Scriptingu útočník získává kompletní nástroj k manipulaci s infikovaným webem. (Huseby, 2006, s. 99-100)

Tomuto typu útoku se dá poměrně snadno zabránit a dnešní frameworky tento problém dobře znají a mají proti němu dobrou prevenci. Nicméně ve své praxi se setkávám s případy, kdy programátor neúmyslně obchází zabezpečení frameworku a sám otevírá díru, kterou lze využít pro Cross-site Scripting, i když využívá framework, který garantuje bezpečnost proti tomuto typu útoku.

2.4 Hashování hesel

Hashovací funkce je funkce, která má pouze jednocestný způsob výpočtu hashe. Tzn. na vstupu přijímá text a na výstupu vrací hash o většinou pevné délce, ze kterého nelze spočítat původní vstup. (Huseby, 2006, s. 134-135)

Hashování ve webové aplikaci využíváme nejčastěji pro zašifrování/zahashování hesel tak, aby nebyla v čitelné podobě uložené v databázi. Nette Framework nám nabízí základní hashovací funkci B-Crypt, která je dostatečně bezpečná. Hesla hashujeme už jenom z principu, že únik osobních dat uživatelů je velmi citlivý problém. (doc.nette.org, Hashování hesel)

Ještě mnohem horší než únik hesel v hashované podobě je únik hesel uživatelů v čitelné podobě, protože uživatelé často používají stejná hesla pro různé systémy a pokud útočník získá heslo z jednoho sytému, často se dostane i do dalších systémů, kde je uživatel zaregistrován. Dnes se již naštěstí nesetkávám s případy, kdy dané webové programy či systémy používají nebezpečný způsob ukládání hesel v čitelné podobě. Poznat takový systém však není úplně snadné, pokud nemáme přístup ke zdrojovým kódům a jsme omezeni pouze z pohledu uživatele.

Indikátorem špatně naimplementovaného ukládání hesel může být i funkcionalita, která je zprvu myšlena jako vylepšení systému. Konkrétně mohu uvést příklad, se kterým jsem se již setkal. Obnova hesla funguje tím způsobem, že se na uživatelův email při žádosti o obnovu hesla zašle jeho aktuální heslo. Toto je jednoznačný identifikátor toho, že je ukládání a hashovaní hesel v systému špatně vyřešeno, protože systém by obecně neměl ukládat podobu hesla v čitelné podobě. Pokud heslo dokáže takto jakýmkoliv způsobem poskytnout, může zde

být zásadní problém v bezpečnosti a hesla mohou být uložena přímo v databázi v čitelné podobě, a tudíž bez zabezpečení proti zneužití.

3 Základní rozvržení a návrh systému

Základní filozofie většiny webových systémů, které tvořím, je v tom, že každý systém má většinou 2 základní části. První část je ta, která je dostupná pro zákazníka, a druhá část je ta, kam má přístup obsluha. Nejlépe si to lze asi představit na jednoduchém příkladu e-shopu, kde část, kam má přístup zákazník, je samotný e-shop s nabídkou produktů. Část, kam má přístup obsluha, je administrace, kde obsluha vyřizuje objednávky, spravuje produkty atd. Dalším příkladem jsou systémy založené čistě na administraci – lze si je představit jako systémy bez veřejné prezentace pro koncové zákazníky – například různé informační systémy a systémy řídící výrobu.

Jedním z mých úkolů bude zamýšlet se nad prvky, které mohu v rámci systémů znovu používat. Větší část těchto prvků je přirozeně v administraci. Už jenom z principu, že zákazníkovi je většinou jedno, jakým způsobem bude fungovat administrační rozhraní systému. Pro moji cílovou skupinu zákazníků je nejdůležitější právě ta část programu, kde je například uvedena nabídka produktů, a ne tolik jaká je obsluha v administraci. Na tu mají většinou požadavky pouze toho rázu, aby byla přehledná a snadno obsluhovatelná. Co se týká typů systémů založených na bázi informační či řídící (bez veřejné prezentace), tam jsou podmínky na administraci mnohem specifičtější. Avšak i zde nalezneme uplatnění pro znovupoužitelnost daných prvků.

Díky tomuto se v diplomové práci zaměřuji právě na tvorbu administračních prostředí, pro která má být i tento framework primárně určen, a snažím se na nich vše demonstrovat. Nicméně to však neznamená, že se určité části nedají využít i pro tvorbu prezentačního modelu daných systémů.

Konkrétně si to lze představit na webových řešeních typu: e-shop pro prodej izolačních bloků dle parametrů kupujícího, systém pro evidenci dotačních programů, internetová seznamka či informační systém pro plánování zemních vrtů. Z prvního pohledu se jedná o absolutně diferenciované systémy, které mají společného jen málo. Protože každý z nich se zabývá řešením něčeho úplně individuálního, co požadoval zákazník, a není úplně jednoduché nalézt pro tato řešení nějaký již hotový systém, který by si zákazník jednoduše zakoupil a případně si jej nechal lehce modifikovat.

Po hlubší analýze takovýchto typů systémů ale nakonec dojdeme k několika sekcím či principům, které budou stejné či podobné. První věc, která bude určitě v každém z uvedených systémů, je přihlašování uživatelů a nějaká jejich evidence a správa. To je sekce, kterou budou mít v různých modulacích všechny tyto jmenované systémy. Konkrétně u e-shopu bude nějaká administrace, kde se spravují produkty a vyřizují objednávky. Do této administrace je třeba se nějak přihlašovat a ve větších e-shopech tam má mít přístup více lidí. Obdoba administrace jako na e-shopu bude určitě i na internetové seznamce. To samé bude mít i informační systém pro plánování vrtů či systém pro evidenci dotačních programů, kde správa uživatelů nemusí sloužit jenom pro přihlašování čistě nějakých správců, ale pro přihlašování všech uživatelů, kteří s daným systémem budou pracovat.

3.1 Návrh systému

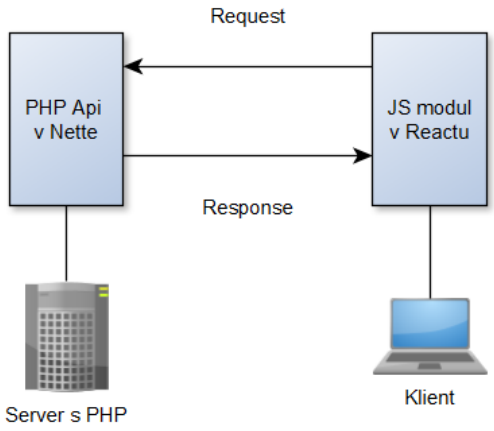
Životní cyklus projektu dle metodiky Unified Process (UP) je rozdělen na 4 fáze: zahájení (období plánování), rozpracování (období architektury), konstrukce (počátky provozuschopnosti) a zavedení (nasazení produktu). Všechny tyto fáze by měly končit nějakými milníky a měly by být indikátory postupu v projektu. Dále se napříč těmito fázemi průběžně, v různé intenzitě, věnujeme jednotlivým aktivitám, a to: sbírání požadavků, analýze, návrhu, implementaci a testování. (Arlow, Neustadt, 2007, s. 61-62)

Při návrhu takto specifického systému se pokouším přidržovat postupů či návodů dle metodiky UP. Z praxe však vím, že je třeba ji brát vždy jenom jako dobré doporučení a ne jako striktní dogma. Zkouším si postupně projít požadavky a budu se je snažit analyzovat a vymýšlet řešení a případně si budu dané části již i implementovat a testovat a následně budu tyto dílčí části skládat do finálního programu. Pokud narazím při implementaci či při skládání daných dílčích částí programu na problém či nový požadavek, nebudu se bát kompletně refaktorovat svůj kód.

3.1.1 Základní koncept systému

V návrhu zohledňuji vybrané technologie a principy, ve nichž budu vyvíjet systém. Díky předdefinovaným požadavkům a požadavkům zákazníků vyvíjím v jazyce PHP. Neprogramuji úplně v čistém jazyce, ale sáhnu po nějakém základu, který mi pomůže ošetřit základní problémy a ušetří práci. Zvolil jsem si Nette framework. Vybral jsem si z něj nějaké knihovny a odstranil věci, které nebudu potřebovat. Úplně celé view a šablony, jež nahradím

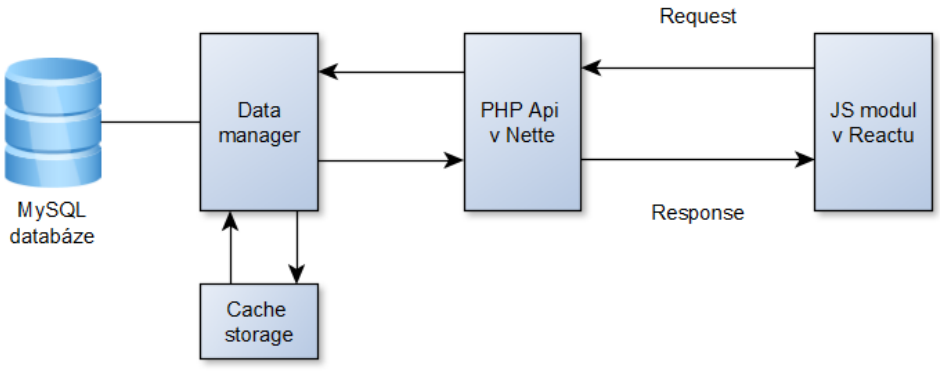
JavaScriptovým frameworkem React, který mi zajistí částečnou logiku a view na frontu. Znázornění základních modulů systémů jsem zobrazil na obrázku č. 1.



Obrázek 1: Znázornění základních modulů systému

Můj další požadavek je na rychlost systému. V konceptu se soustředím především na rychlost a náročnost pro získání dat z databáze, které bývají největším problémem. Z vlastní zkušenosti vím, že pokud systém nějakou dobu běží, používá se a přibývají do něj data, bez indexace databáze a bez zavedení cache se může za čas zpomalit. Předpokládám nyní modelový případ, že máme metodu na Api, která se ptá na data a její kompletní vyřízení trvá cca 2 vteřiny času. Dvě vteřiny nejsou dlouhá doba, ale pokud je má uživatel čekat při každém kliknutí myši, bude to pro něj nekomfortní.

Hodí se určitá volání Api cachovat, aby se zvýšila jejich rychlost odezvy. Cachovat na Api lze pouze ta volání, která nám vrací některá data. Výsledek, kterého je třeba dosáhnout, je na obrázku č. 2 (zanedbám tu již infrastrukturu pod aplikací – server a klient, ale dále si je tam lze představit).



Obrázek 2: Základní model s Data managerem

Celý model, jenž znázorňuje obrázek č. 2, jsem navrhl tak, že JS modul vyšle request (požadavek) na data. Api modul se na tato data zeptá Data manageru. Ten si spočítá unikátní klíč, dle kterého se dotáže s konstantní náročností do cache, a v případě, že se mu z cache vrátí data, pošle je skrze Api až JS modulu. Pokud by nenašel data v cache, vytvoří SQL dotaz, ten pošle na databázi a data získá. Nicméně než data odešle zpět až ke klientovi, uloží si je pro příští využití do cache pod předem spočítaným klíčem složeným z hashe (otisku) dotazu. Pro Cache storage systém využívá „nette/caching“ knihovnu včetně funkcionality pro expiraci a invalidaci dat.

3.1.2 Rozšíření základního konceptu o Application storage

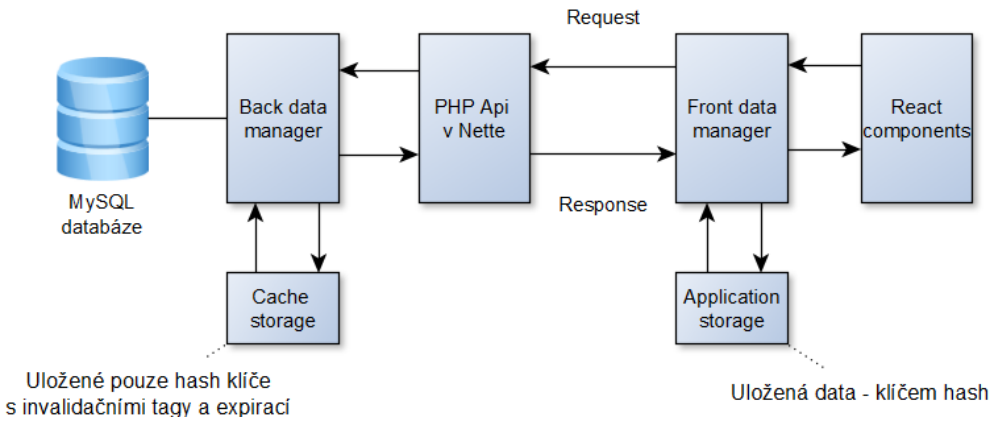
Ze základního modelu s Data managerem pro zavedení cachování nad Api vznikl již dobrý model, jak by výsledný systém mohl fungovat. Nadále tu zůstává problém s Api, která nám vrátí data za 2 vteřiny, což se stává, když nejsou v cache uložená či jsou již expirovaná/invalidovaná. Pokud bude systém vícekrát zrcadlit data z databáze do cache a pokud bude databáze větších rozměrů, naroste i velikost cache. To nemusí být problém, když bude využit modul cache, který ukládá data přímo na disk. Pro pružnou aplikaci je dobré si cache držet přímo v RAM paměti, například skrze MemcachedStorage, který poskytuje Nette cache.

V HTML5 existuje objekt localStorage, který umožňuje ukládat persistentní data na straně klienta. Funguje tak, že za danou doménu si program může do localStorage uložit data a přistupovat k nim principem klíč – hodnota. Data v objektu localStorage jsou uchovávána do té doby, dokud nejsou kódem Javascriptu odstraněna či je uživatel sám nevymaže z mezipaměti prohlížeče. V novodobých prohlížečích lze do localStorage uložit data až o velikosti 10 MB. U prohlížečů pro mobilní zařízení či starších prohlížečů může být tato velikost menší. (Zakas, 2009, s. 679-681)

Po důkladné analýze základního modelu s Data managerem jsem tento základní koncept rozšířil a vylepšil. Princip vylepšení je v tom, že se povinnost cachování některých dat přenese na klienta, a tudíž na JS modul v Reactu. Ten si je uloží do localStorage. Nad localStore jsem vytvořil nový modul, a to Application storage, který pracuje hodně obdobně jako Cache storage na úrovni serveru. To znamená, že může serializovat daná data a uložit je do localStorage pod klíčem, zároveň si o nich uchovává nějakou časovou značku pro expiraci a tagy pro invalidaci. Dále řeší ošetření ukládání dat na velikost localStorage a zachytává se exception při přetečení localStorage. Při přetečení se podívá na uložené značky

expirace a dle nejstaršího data uvolní z localStorage. Nad Application storage jsem postavil také vlastní Data manager, který řeší získávání dat pro JS modul. Tento koncept je vidět na obrázku č. 3.

Můj koncept přidání Application storage má tu výhodu, že ušetří přenos dat mezi serverem a klientem. Zároveň ušetří místo na serveru a data rozloží mezi klienty, což v původním modelu nešlo. Nicméně tu vzniká problém s invalidací a aktuálností dat na straně klienta.



Obrázek 3: Rozšířený model o Application storage

Aby se tento problém vyřešil, musel jsem upravit Cache storage tak, že se v něm neukládají data, ale jenom hash klíče k datům. Hash klíče se generují už na úrovni Front data manageru, který se nejprve zeptá do Application storage, zda daná data má. Pokud ano, vrátí callback s daty do React komponenty, která si je vyžádala. Pokud data ne, nevrací nic a zavolá Api s požadavkem o data, ke kterému přidá hash klíč. Back data manager data připraví a vrátí skrze Api zpět. Nicméně než data vrátí, uloží si daný hash klíč do RAM cache včetně expiračních a invalidačních tagů pro daná data.

Nyní má tedy JS modul daná data v Application storage, nemusí se na něj již ptát serveru a vrací je klientovi v podstatě v reálném čase bez jakéhokoliv čekání na server. Což je to, čeho jsem v tomto modelu chtěl dosáhnout. Funguje to tak, že si React komponenta vyžádá data na Front data manageru a ten je nyní najde v Application storage. Callbackem je vrátí do komponenty a asynchronním voláním (bez toho, aby klient čekal – na pozadí) zavolá danou Api na získání dat znovu, ale s příznakem, ať mu ověří, že jsou v Cache storage dle daného hashe ještě platná. Back data manager se podívá do RAM paměti a v případě, že jsou data platná, ihned mu pošle zpět response ve smyslu, že ano. Pokud se data změní, Back data manager automaticky získá nová data, ta pošle zpět Front data managerovi a ten zavolá znovu callback, aby React komponenta data uživateli překreslila.

Konkrétně to znamená, že pokud má JS modul stará data v Application store a přijde na ně požadavek od komponenty na vykreslení pro uživatele, Front data manager je prostě vrátí a komponenta je vykreslí. Nicméně na pozadí běží požadavek na server, který může poslat data nová a okamžitě jakmile přijdou, uživateli se pod rukama aktualizují. Tzn. uživatel si zobrazí například přehled objednávek. Začne se rozkoukávat, během pár milisekund mu přiskočí do tabulky nová objednávka a data má v tom okamžiku aktuální. Výsledek je tedy ten, že uživatel vůbec nečeká na nějaké načtení, ale dostane poslední stav, který se ověří a v případě potřeby lehce aktualizuje. Což považuji za stěžejní výhodu pro uživatele systému.

Posledním problémem je úvodní načtení. Tzn. první vyžádání dat, které může trvat například 2 vteřiny. Elegantní řešení spočívá v rozšíření Front data manageru o predikci toho, co bude uživatel po spuštění potřebovat. Systém si to zkusí přednačíst dopředu v době, kdy uživatel čeká na spuštění aplikace. Obecně bývá u JavaScriptových aplikací úvodní načítání do prohlížeče, kde si již v rámci tohoto načítání můžeme natáhnout základní balík dat, který si uložíme do Application storage. Tím se vyřešil problém s prvotním načítáním dat, která je obtížné získat.

V praxi to bude vypadat tak, že se uživatel například přihlásí do administrace e-shopu. Přihlašování bývá většinou ve front-end části a aplikace se bude načítat až po přihlášení JS aplikace pro administraci. Při tomto načítání budeme mít statickou predikci toho, že obecně víme, že nejčastější věc, kterou uživatel po přihlášení dělá, je, že zpracovává nové objednávky. Takže si můžeme všechny nové objednávky načíst předem do Application storage, uživatel bude mít po načtení a naběhnutí aplikace vše dostupné a bude si moci pohodlně procházet a zpracovávat objednávky bez toho, aby musel na systém jakkoliv čekat.

Díky přidání Application storage a predikci přednačítání dat vznikl celkem robustní nástroj k zajištění běhu aplikace pro uživatele v podstatě v realtime, aniž by musel čekat na data. Po zhodnocení výsledku mohu konstatovat, že se tento můj koncept architektury dobře osvědčil a aplikace působí díky němu velice pružně a má rychlé odezvy.

3.1.3 Možnost zdokonalení rozšířeného konceptu

Základní koncept, který je hotový, by ještě šlo v budoucnu rozšířit a zdokonalit. Tyto návrhy vzešly z mého testování a analýzy výsledku. Nejvyšší přidanou hodnotu by mohly mít následující funkcionality.

První věcí je manipulace s daty. To je operace, která při určitých okolnostech může být také dosti náročná a uživatel na ni může čekat. Jako manipulaci s daty uvažujme souhrnně: přidání nového záznamu, úpravu stávajícího a smazání záznamu z databáze. Aby na tyto operace uživatel nemusel čekat, šel by na straně Back data manageru realizovat jakýsi zásobník operací, které se mají posloupně vykonat nad databází. Zásobník by byl asynchronní a tím pádem by klient nemusel čekat, až jeho akce dosáhne fyzického uložení v databázi. Ve větších systémech by zde bylo třeba se ještě zamyslet, jak řešit případné kolize mezi jednotlivými operacemi od více uživatelů.

Dále by bylo možné zefektivnit invalidaci a expiraci dat na straně Application storage. Problematické je zde hlavně to, že můžeme uživateli po krátkou chvilku poskytnout neplatná data, než se stihnou na serveru ověřit a u uživatele překreslit aktuálními. Aktuálnost dat si totiž nyní zjišťujeme, až když si uživatel data vyžádá, a zjišťujeme ji dotazem od klienta na server, což je klasický model HTTP komunikace – klient se dotazuje a server odpovídá.

V HTML 5 je definován objekt WebSocket. Ten umožňuje obousměrnou komunikaci mezi prohlížečem a serverem na základě speciálního protokolu. V podstatě mezi sebou uzavřou pomyslný tunel, který je neustále otevřený a skrze něj probíhá oboustranná a velice rychlá komunikace. Často se používá v systémech, kde je potřeba okamžité výměny zpráv či u realtime webových her. (Zakas, 2009, s. 748-749)

Díky WebSocketu bychom mohli zajistit, aby se kontrolovala invalidace dat. Klient by se serverem otevřel spojení a server má uložené hashe dat včetně tagů pro invalidaci. Pokud by se tato data změnila a on zneplatnil nějaký hash, mohl by sám poslat zprávu klientovi, že používá stará data, a ten by si je mohl okamžitě překreslit.

V praxi by toto vypadalo tak, že si uživatel otevře tabulku s objednávkami a aniž by něco dělal, budou mu v ní samy přibývat či ubývat nové objednávky dle toho, jak je například jeho kolega vyřizuje. Prostě by před sebou viděl živý systém a nebyl nucen do něj zasahovat, aby získal nová data.

4 Nástroje pro jednodušší tvorbu systému

Po důkladné analýze systémů, které jsem kdy realizoval, jsem došel k závěru, že je třeba zautomatizovat či připravit funkcionality pro usnadnění tvorby problematik. Níže uvádím čtyři oblasti, pro které jsem vytvořil vlastní nástroje pro jejich řešení.

- **EntityManager** = nástroj pro obecnou práci s databází – získávání a ukládání dat, mapování na PHP objekty a práce s nimi. Správa a manipulace s databází. Operace a skládání dotazů nad databází.
- **EntityAccessPoint (EAP)** = nástroj pro řešení přímé komunikace databáze skrze API a snadná obsluha pro manipulaci s daty.
- **Form generator** = nástroj pro to, jak data upravovat a přidávat – tzn. řešení pro formuláře.
- **Grid generator =** nástroj, jak zobrazovat kolekce dat a jak jimi procházet – filtrovat, stránkovat.

4.1 Řešení databázových entit a EntityManager

Po prozkoumání a analýze knihoven a řešení pro práci s databází jsem vytvořil vlastní způsob pro přístup k databázi. Základ je postaven na knihovně Dibi. Je možné využívat všechny základní principy definované v této knihovně, plus je tu rozšíření pro snadnější manipulaci s daty. Toto rozšíření spočívá v konfiguracích databázových entit a ve třídě EntityManager.

Základní myšlenka celého konceptu je ve snaze minimalizovat počet dat, která je třeba přenášet z databáze a zpět. Klasické PHP knihovny pro práci s databází (doktríny) a mapování na entity používají způsob toho, že entita v databázi se rovná PHP třídě. Sloupce v databázi odpovídají vlastnostem dané třídy a pokaždé, když chceme nějak manipulovat s daty, je třeba přenášet v podstatě celý řádek z databáze. Když je třeba vypsat hodnotu z jednoho sloupce a databázová tabulka jich má 50, pro výpis jedné hodnoty se z databáze natáhne celý řádek, který se serializuje do PHP třídy a my dostaneme její instanci.

Prakticky to lze ukázat na příkladu, kde vypisujeme jméno a příjmení uživatele na stránce. Známe jednoznačný identifikátor (primární klíč) uživatele (nejčastěji sloupec Id). Dle tohoto identifikátoru se dotážeme databáze a ona vrátí celý sloupec, který obsahuje 50 polí a serializuje se do PHP třídy. Z této třídy si následně vezmeme pouze jméno a příjmení, to vypíšeme a zbytek alokovaných dat v podstatě zahodíme a nijak dále nevyužijeme. Místo

jednoho uživatele vypíšeme tímto způsobem všechny, které máme v databázi, a je jich například jeden tisíc. Konkrétně: pro každého uživatele natahujeme 50 polí, abychom využili dvě z nich (respektive tři – jednoznačný identifikátor potřebujeme většinou pokaždé). Znamená to, že jsme si pro výpis těchto uživatelů vyalokovali v operační paměti 50 tisíc proměnných, což je o 47 tisíc více, než reálně potřebujeme, i když by stačilo alokovat jenom 3 tisíce.

4.1.1 Databázové entity jako konfigurační soubor

Daný problém s mapováním a serializací dat do PHP tříd při práci s databází je v práci vyřešen tak, že neexistují PHP třídy odpovídající entitám, ale pouze konfigurační soubory. Ty stanovují, jaké sloupce jsou v které databázové entitě a jsou zde rozšířeny o další dodatečné informace, které se využívají napříč systémem. Ukázka takovéhoto konfiguračního souboru je na obrázku č. 4.

User.neon

```
table: User
primaryKey: id
parameters:
    id:
        type: int
        title: 'Id'
    name:
        type: string
        title: 'Jméno'

    surname:
        type: string
        title: 'Příjmení'

    email:
        type: string
        title: 'E-mail'

    password:
        type: string
        title: 'Heslo'
        hash: true
        disableSelect: true

    created_date:
        type: date
        title: 'Datum založení'
        format: 'j. n. Y H:i'

    update_date:
        type: date
        format: 'j. n. Y H:i'
        title: 'Datum upravení'
```

Obrázek 4: Ukázka konfiguračního souboru entity

Konfigurační soubory entit využívají Neon soubory z Nette, které má nad nimi dobře zpracované operace pro kontrolu syntaxe a rychlé zpracování. Jak je z obrázku č. 4 vidět, konfigurační soubory definují strukturu databázových sloupců dané entity, plus je doplňují

o další podpůrné informace, které nám ušetří následnou práci. Mohou obsahovat další konfigurace pro práci s danou entitou – například způsob, jak s daným polem/sloupcem z databáze pracovat, jak jej formátovat či jakým způsobem se k němu chovat při ukládání a komu dovolit, aby s ním mohl manipulovat.

Můj navržený koncept pro práci s entitami založenými a definovanými v konfiguračních souborech funguje tak, že se daná entita neserializuje do PHP třídy, ale zůstává v datovém typu asociativního PHP pole, kde je uložena jako klíč – hodnota. Klíč je název sloupce z konfiguračního souboru (koresponduje s názvem sloupce v databázi) a hodnota je konkrétní záznam z entity. Problém se serializací celé entity je vyřešen tak, že se do asociativního pole entity natahují pouze ty sloupce, které jsou vyžádány. Tzn. pole nemá pevnou strukturu, jako je PHP třída. Proto sloupce, které nejsou vyžádány, v poli nejsou, a tudíž se zbytečně nepřenáší.

S tímto faktem je třeba při práci s poli entit počítat, protože nemusí mít kompletní sloupce. Pokud vyvstane potřeba si nějaký sloupec z databáze dotáhnout, musí se přidat do žádosti o data.

4.1.2 EntityManager – manažer pro komunikaci s databází

Entity jsou samy o sobě pouze konfigurační soubory, které mapují či opisují strukturu databázových tabulek. Jeden z nástrojů, který s nimi dokáže pracovat, je EntityManager.

Implementuje metody pro získání jedné entity či kolekce entit, zjištění počtu záznamů v dané tabulce, vrácení dat asociativně (jako jiný explicitní klíč – hodnota). Také implementuje metodu pro získání a manipulaci s datovým zdrojem pro generované zobrazovací tabulky systému.

Obsahuje metody pro manipulaci s daty, a to obecnou metodu pro uložení dat a pro smazání. Obecná ukládací metoda funguje na principu, že EntityManager dostane informaci o tom, jaká entita se má uložit, a zároveň dostane associativní pole s danými hodnotami. Toto pole zkontroluje dle anotací z konfiguračního souboru. Případně přetypuje některé položky a ověří, že daný uživatel má právo s nimi takto manipulovat, a následně, pokud je uveden jednoznačný identifikátor, provede úpravu daného záznamu v databázi. Pokud jednoznačný identifikátor není uveden, tato data vloží do databáze jako nový záznam.

Všechny tyto metody předpokládají manipulaci na základě definovaných entit, které tyto metody svazují a doplňují o potřebné informace, jak s nimi pracovat. Například datumové položky je možné díky těmto definicím do databáze ukládat buď jako klasický PHP objekt DateTime, nebo jako řetězec podle uvedeného formátu v konfiguračním entitním souboru.

Hlavní účel EntityManageru je ten, že nám dokáže vyřešit valnou většinu komunikace s databází a je navržen pouze pro základní operace, které se nejčastěji nad databází provádí. Jeho účelem není nahradit základní adaptér pro manipulaci s SQL, ale jen ho doplnit o elegantní způsob, jak pracovat s entitami. Je to takto koncipováno schválně, abychom se jej nesnažili využívat na pokročilejší SQL dotazy, které například nějak spojují více databázových tabulek, či dotazy s nějakou složitější agregací dat. Takovéto dotazy je třeba psát v čistém SQL, například přes Dibi, kde už si můžeme pohlídat, co přesně chceme nad databází provést, či jaká data chce získat.

4.2 EntityAccessPoint (EAP) – univerzální přístupový bod skrze API

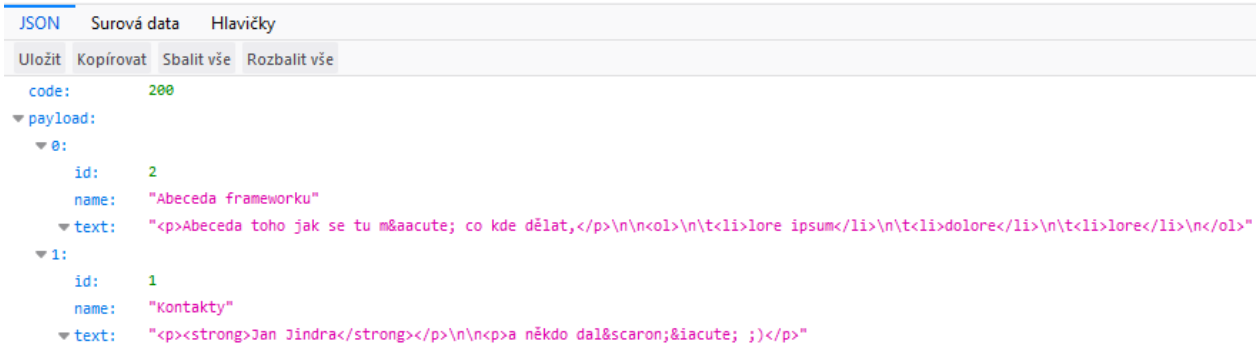
Systém jako celek funguje na principu REST API, kde je na serveru vystavěna PHP aplikace, která poskytuje API, a na frontu běží v JavaScriptová aplikace, která daná data prezentuje a pracuje s nimi. Přišel jsem s inovací jak zjednodušit to, abychom nemuseli vytvářet API pro každou entitu, se kterou budeme potřebovat na straně JS aplikace pracovat. U administračních prostředí, na která se systém výhradně zaměřuje, se vše točí okolo práce s daty. Potřebuje často data získávat, a to jak v kolekcích dat, tak i samotných entit. Dále s daty často manipulovat – přidávat nové záznamy do databáze, odebírat či modifikovat stávající. Obecně platí, že pokud máme aplikaci založenou na REST API, píšeme si k jednotlivým entitám vždy vlastní API obsluhu, kterou modifikujeme dané entitě přesně „na míru“. Důsledek je však takový, že píšeme z valné většiny času pořád to samé dokola. Většina těchto API tříd je obsahově hodně podobná a rozdíl bývá často pouze v tom, že manipulují každá s jinou entitou.

Systém zachovává základní principy výše zmíněné architektury. Ale rozšířil jsem ho o princip univerzálního přístupového bodu, který jsem nazval EntityAccessPoint (EAP). Tento EAP v podstatě opisuje schopnosti EntityManageru, zpřístupňuje jej skrze veřejnou API ven a doplňuje o další funkcionalitu. Tento princip považuji za jeden ze základních kamenů systému, který jsem vytvořil.

Velkou sílu EAP vyobrazuje následující příklad. Pokud na EAP pošlu klasický GET požadavek s nastavením zobrazeným na obrázku č. 5, API vrátí JSON odpověď, která je zobrazená na obrázku č. 6.

```
1
2   ## Příklad parametrů pro volání GET api
3
4   Url:    /api/uni/get    # url v rámci systému, kde je EAP
5   Metoda: GET             # HTTP metoda volání
6
7   Parametry:
8   type:       find                # získá celou kolekci dat
9   entity:     web_page            # hledá v dané entitě
10  fields:     id|name|text        # které parametry chceme vrátit
11  condition:  {"active":1}        # předpokládá JSON pole s podmínkama výběru
12  orderby:    {"priority":"ASC"}  # JSON pole určující řazení záznamů
13  limit:      10                  # maximální počet záznamů, které vrátí
14  offset:     0                   # offset posunutí kolekce
15
16
```

Obrázek 5: Ukázka nastavení volání na EAP



Obrázek 6: Výsledek volání základního dotazu na EAP

V požadavku je řečeno, že chceme získat kolekci dat z entity „web_page“. Konkrétně sloupce id, name a text. Parametr „condition“ nám říká, že se mají vybrat pouze ty záznamy, které mají atribut „active“ rovný jedničce. Výsledek se má seřadit dle atributu „priority“ sestupně a vrátit maximálně 10 nalezených záznamů jako JSON odpověď.

Hlavním přínosem je to, že EAP je univerzální nástroj k manipulaci s daty skrze API a nenutnosti vytvářet dodatečnou funkcionalitu pro každou entitu zvlášť. Často programátor potřebuje na straně JavaScriptové aplikace získat nějakou entitu či kolekci entit. Díky EAP si o ně prostě požádá a dostane je, aniž by musel ještě něco programovat na straně PHP. Stejně jako pro získávání dat je EAP dobrý i pro jejich ukládání či mazání, které funguje na analogickém principu.

4.2.1 Zabezpečení EAP

U EAP je důležitou a základní věcí oprávnění a zabezpečení. EAP je v podstatě veřejné Api, takže jej může volat úplně kdokoliv bez nutnosti být v systému autorizován či ověřen. Nemusí jej volat pouze JavaScriptová aplikace, ale slouží i jako základní Api rozhraní celého systému. Takže je třeba důsledně dbát na to, aby se u každé entity definovalo, jaké oprávnění má jaký typ uživatele z hlediska pohledu systému. V systému v základní konfiguraci si lze nadefinovat základní typy přístupových rolí uživatelů systému. Tyto role si může každý přizpůsobit tak, jak potřebuje. Existují zde 2 základní role, a to superadministrator a host neboli „guest“. Pokud má uživatel roli superadministrator, je mu povoleno úplně vše bez vlivu autorizace. Pro všechny ostatní role, které si nadefinujeme, to platí přesně naopak – vše je zakázáno a povoleno je jenom to, co se nastaví v autorizaci entit. Role hosta (guest) je výchozí pro všechny nepřihlášené uživatele. Tzn. je to každý, kdo přijde k systému a není autorizován – používá se často, pokud má daný systém nějakou veřejnou prezentaci (např. e-shop).

Příklad nastavení autorizací entity je vidět na obrázku č. 7. Důležitá je položka „authorization“, která nám říká, že uživatelé se základní rolí „guest“ anebo „administrator“ mají nějaká práva na to k entitě přistupovat a pracovat s ní. U role „administrator“ je zřejmé, že je povolena kompletní manipulace s danou entitou. Tzn. má právo „write“ i „read“ zapnuté a může si s danou entitou skrze EAP dělat vše, co uzná za vhodné. U role „guest“ je z uvedené konfigurace zřejmé, že uživatel s touto rolí nemůže nic měnit či zasahovat do dat. Nicméně může získat přístup pro čtení pro určité parametry (jedná se tu o entitu webové stránky, která slouží pro uchovávání textových článků stránky, které jsou veřejně dostupné).

web_page.neon

```
table: web_page
primaryKey: id
parameters:
    id:
        type: int
        title: 'Id'

    name:
        type: string
        title: 'Název'

    text:
        type: string
        title: 'Text'

    menu:
        type: int
        title: 'V menu'
        data: [[id = '1', mask = 'Horní menu'], [id = '2', mask = 'Dolní menu']]

    active:
        type: bool
        title: 'Publikováno?'

    created_date:
        type: date
        title: 'Datum vytvoření'
        format: 'j. n. Y H:i'

    update_date:
        type: date
        title: 'Datum upravení'
        format: 'j. n. Y H:i'

authorization:
    guest:
        read: [id, name, text] # omezení dostupných polí
        write: false

    administrator:
        read: true
        write: true
```

Obrázek 7: Ukázka konfiguračního souboru entity s nastaveným oprávněním

Dále je možné toto základní zabezpečení rozšířit o AuthMiddleware, který lze přesně specifikovat. Zde jsem ho navrhnul a implementoval tak, že na základě definic v entitě můžeme zavolat funkci přímo z konfigurace entity. To znamená, že kterékoliv entitě mohu zkontrolovat či omezit přístup k operacím EAP pro určité role či konkrétní uživatele.

```
authorization:
    customer:
        form: [id, name, surname, email, password, street, city, zip, country]
        read: onlyOwn(id,id)
        write: onlyOwn(id,id)
```

Obrázek 8: Ukázka konfiguračního souboru entity s pokročilým zabezpečením

Na obrázku č. 8 je zobrazeno nastavení pro čtení a zápis funkcí „onlyOwn“ pro roli „customer“. Funkci „onlyOwn“ jsem na autorizačním middlewaru definoval jako omezení,

které kontroluje, že se skrze EAP kterýkoliv uživatel s rolí „customer“ dostane pouze k jedné jediné entitě, a to je ta s daty o něm. Příklad je vyjmut z entity uživatele a toto nastavení přímo říká, že uživateli povolíme přístup pro zápis a čtení pouze jeho vlastního profilu skrze EAP. Dále tu přibyl ještě atribut „form“, který říká, které položky má mít uživatel s rolí „customer“ dostupné ve formuláři. Tzn. je to omezení polí, které může editovat v dané entitě.

4.3 Form generator – řešení, obsluha a generování formulářů

Jedním ze základních stavebních kamenů každého informačního systému či rozhraní pro správu systému je formulář. Formulář je tu myšlen jako nástroj pro editace stávajících dat či přidávání nových dat. Primárně se tu snaží systém pomoci s tvorbou a obsluhou formulářů korespondujících s entitami. Tzn. máme pro danou entitu nějaký formulář, který nám řeší přidávání nových záznamů či editaci existujících. Tyto dva formuláře se většinou od sebe extra neliší. Rozdíl v nich bývá jenom na pozadí, editační formulář je po inicializaci vyplněn daty z databáze včetně skrytého políčka s jednoznačným identifikátorem, dle kterého se rozhoduje, který záznam v databázi bude modifikován. Formulář pro přidávání nových záznamů bývá prázdný a políčko s jednoznačným identifikátorem bývá nevyplněné – tak systém rozezná, co má s daty z formuláře udělat – jestli upravit stávající anebo vložit nový záznam.

Pro tento typ formulářů jsem zde navrhnul a implementoval nástroj, který jsem nazval generátor formulářů, neboli Form generator. Tento generátor formulářů má 2 základní části. První z nich je definice formuláře na entitě, kterou skrze EAP (EntityAccessPoint) API poskytuje pro JavaScriptovou aplikaci. Dále je v první části i zpětná obsluha formuláře. Tzn. EAP jsem rozšířil o metodu ukládání skrze definici formuláře, kde řeší validaci dat a oprávnění k dané operaci. Druhá část je samotný generátor na straně JavaScriptové aplikace, který se stará o vykreslení formuláře dle definic, jeho validaci, předvyplnění při editaci a obsluhu ukládání.

Konfigurace na entitě probíhá zápisem anotací do konfiguračního souboru entity. Příklad těchto rozšířených anotací můžeme vidět na obrázku č. 9. Pro každý atribut v entitě si přidáme nastavení „form“, kde si definujeme základní požadavky na to, jak se má zobrazit a jestli má vůbec dané pole figurovat ve formuláři.

Mezi základní „form“ parametry, které můžeme nastavovat, jsem v systému připravil například:

- *„type – int | float | string | text | bool | date | hidden | select ...“*; Tento atribut nám určuje typ políčka ve formuláři. Například při uvedení „string“ bude políčko klasický textbox input. Při uvedení „bool“ to bude zaškrtávací políčko a při uvedení „date“ dostaneme políčko s kalendářem pro výběr data.
- *„required: 'Pole název je povinné!' “*; Atribut udává, zda je políčko v daném formuláři povinné k vyplnění či nikoliv. A pokud jej uživatel nevyplní a pokusí se formulář odeslat, zobrazí se mu zpráva uvedená v tomto atributu.
- *„placeholder: 'Cena v Kč' “*; Atribut vyplňuje placeholder (místodržitele/zástupný text) u daného formulářového políčka.
- *„rule“*; Tento atribut očekává další nastavení, které je uvedeno jako pole. A jedná se o rozšířenou validaci. Zde můžeme například uvádět, aby nám validoval, že v políčku má celé číslo či zadaný e-mail a nebo že políčko má mít minimální počet znaků.

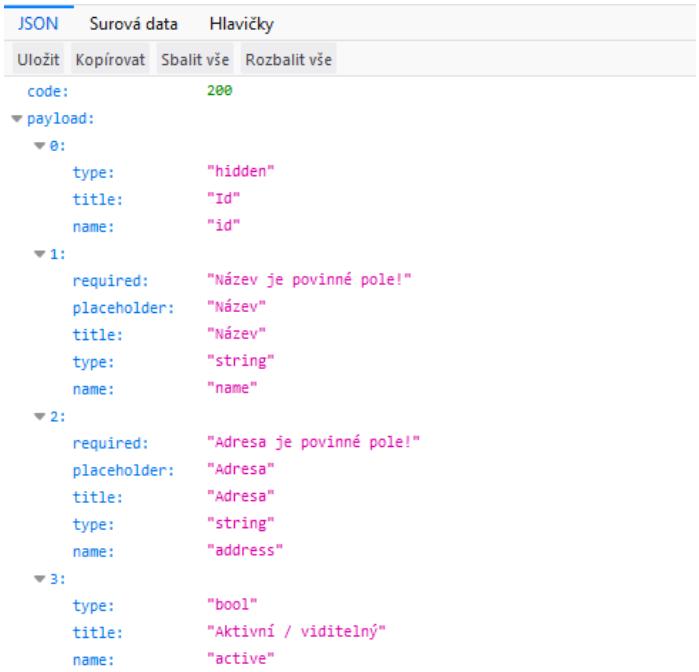
Těchto anotací má formulář generátor ještě celou řadu. Na obrázku č. 9 je vidět základní příklad nastavení těchto anotací. Konkrétně zobrazuje, že formulář bude mít 4 políčka a z toho jedno skryté pro uchování jednoznačného identifikátoru. Dvě další políčka budou klasická textová a jedno bude zaškrtávací (konkrétně v tomto případě bude zaškrtávací políčko nahrazeno za přepínač on/off).

center.neon

```
primaryKey: id

parameters:
    id:
        type: int
        title: 'Id'
        form:
            type: hidden

    name:
        type: string
        title: 'Název'
        form:
            required: 'Název je povinné pole!'
            placeholder: 'Název'

    address:
        type: string
        title: 'Adresa'
        form:
            required: 'Adresa je povinné pole!'
            placeholder: 'Adresa'

    active:
        type: bool
        title: 'Aktivní / viditelný'
        form:
            type: 'bool'
```

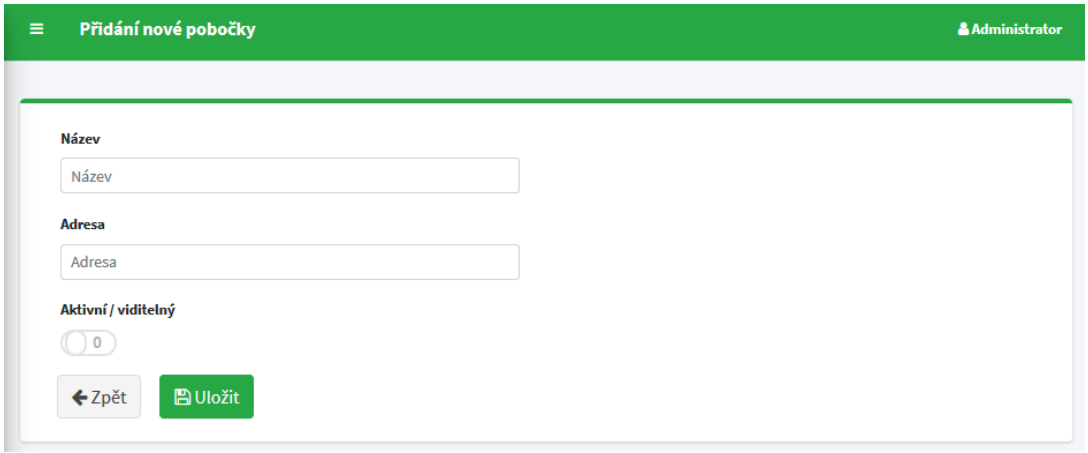
Obrázek 9: Nastavení anotací k formuláři na entitě

Tyto anotace jsou jednoduchou metodou převedeny na upravené JSON schéma, které poskytuje EAP pod GET metodou „/api/uni/get-form-schema“. Metodě je třeba předat název entity do parametru „entity“. Předpokládá se entita, která má definované formulářové anotace. Výsledek tohoto volání nám vrátí JSON v podobě vyobrazené na obrázku č. 10. Na něm je vidět, že se z konfiguračního souboru vzaly čistě anotace pro formulář a daly se k dispozici pro JavaScriptový generátor, který skrze ně vystaví formulář.



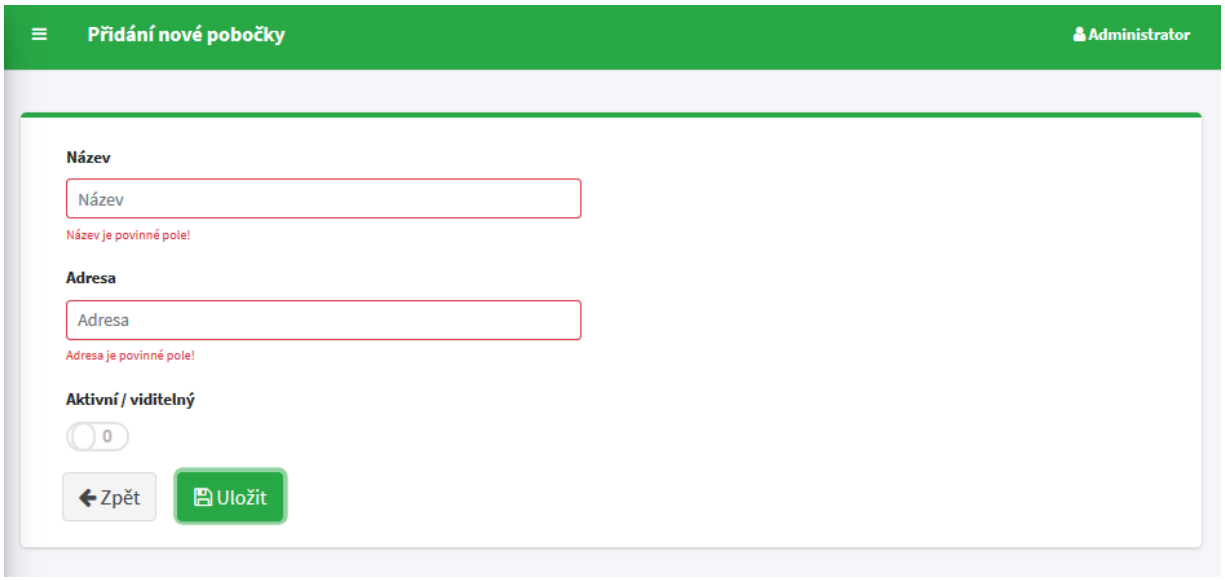
Obrázek 10: Nastavení anotací formulářů jako JSON schéma

Dále proces funguje tak, že si generátor v JS aplikaci vezme toto schéma (buď z volání z API anebo z cache, kde ho má také uložené) a na základě něj vystaví kompletní formulář. Výsledek tohoto vidíme na obrázku č. 11., kde je již vidět formulář, který je vytvořený skrze anotace z konfigurace entity.



Obrázek 11: Formulář vytvořený pomocí anotací

Formulář je vygenerován včetně validačních pravidel na vyplněnost, která jsou zadaná v anotacích. To je vidět na obrázku č. 12, kde se uživatel pokusil odeslat formulář prázdný.



Obrázek 12: Formulář vytvořený pomocí anotací – validace

Zápis kódu v JS React aplikaci výše zmíněného formuláře je zobrazen na obrázku č. 13, kde je útržek kódu uvedeného formuláře. Vše jsem udělal tak, aby byl zápis v JavaScriptu přehledný a jednoduchý. Na tomto obrázku jsou dva html „div“ tagy, které jsou definované z knihovny Bootstrap a vytváří blok či prostor, kde vykreslíme formulář. A tag „SchemaForm“ je již třídou generátoru formuláře naprogramovaného v JS. Tato třída u takto základního formuláře předpokládá předání definice typu entity, což je v tomto případě „center“. Dále je tam atribut id, který nám z proměnné z Reactu dodává jednoznačný identifikátor (v případě obrázku č. 12 nebyl vyplněn, protože se do formuláře nedoplnila data existujícího záznamu). Posledním uvedeným atributem je „linkBack“, který nám udává URL, na které má po uložení (či kliknutí na tlačítko „Zpět“) směřovat uživatele.

```
<div className="row">
    <div className="col-md-6">
        <SchemaForm
            entity="center"
            id={this.state.id}
            linkBack="/centra"
        />
    </div>
</div>
```

Obrázek 13: Zápis formuláře vytvořeného anotacemi v Reactu

V praxi jsem si ověřil, že generátor formulářů je dobrým výsledkem mé práce, protože takto vygenerované formuláře jsou velkým přínosem a hlavně velkým ulehčením práce programátora. Největší úspora je v množství kódu a počtu operací, které by normálně musel naprogramovat, než by takovýto formulář vytvořil.

V podstatě mu stačí, když si sepíše na entitě konfiguraci na formulář a v JS si jej jenom umístí pomocí jednoho tagu, a to „SchemaForm“. Díky tomuto má zajištěné, že se formulář vykreslí včetně všech validací (serverová na API při ukládání a zároveň i validace na straně JS). Dále k tomuto formuláři nemusí již psát ukládací metodu, protože on sám už podle konfigurace ví, jak se má v základu ukládat. Stejně jako u EAP či EntityManageru je generátor formulářů jenom nástrojem k ulehčení určité dílčí operace, kterou tu jsou základní formuláře. Není určen k tomu, aby vyřešil všechny formuláře v daném systému, ale vypomohl u těch, kterých je nejvíce. Pokud bude v systému například specifický formulář pro vytvoření objednávky, je třeba jej udělat pomocí klasických přístupů. Napsat mu jeho vlastní ukládací metodu, vlastní validace atp.

4.4 Grid generator – řešení, obsluha a generování tabulek

Dalším důležitým stavebním kamenem každého informačního systému jsou tabulky či gridy pro zobrazení dat. Mám zkušenosti s tím, že většina informačních systémů či systému sloužícího jako administrace pro správu je složena téměř pouze z tabulek a formulářů. Pokud je návrh takovéhoto systému dobrý, můžeme zde aplikovat základní princip toho, že většina přehledových tabulek bude zobrazovat právě data z jedné entity. Přičemž se samozřejmě nevylučuje základní spojení na další entity doplňující zobrazení této hlavní. Například na e-shopu v tabulce přijaté objednávky máme sloupec, kde je uveden zákazník, který je z jiné entity než objednávka. Spojení 1:N či 1:1 jsou akceptována a je na ně pamatováno.

Základní zobrazovací tabulka (grid) potřebuje umět zobrazit určená data z dané entity. Doplnit je o sloupce z připojených entit. Přidat stránkování při větším rozsahu dat. Mít možnost definovat filtry a řazení nad jednotlivými sloupci. Dále stránkování, filtrování a řazení je třeba řešit na úrovni databáze. Není akceptovatelné, aby si celou kolekci dat systém přenesl ke klientovi a tyto operace probíhaly až v JavaScriptu. Toto je akceptovatelné u menšího rozsahu dat. Avšak rozsah dat v systémech, pro které je výsledný framework určen, bývá dost velký, způsoboval by problémy a enormně zpomalil celý systém.

Pro tento případ jsem vymyslel koncept generátoru gridů, který slouží obdobně jako generátor formulářů k elegantnímu vyřešení dané problematiky.

Stejně jako u formulářů se opřeme o základ, který máme pro tento systém připraven, a to o konfigurační soubory entit a o EAP.

Tabulky (gridy) využívají stejně jako formuláře určité definované anotace v entitách. Na obrázku č. 14 je vidět základní princip užití. Entitu „center“ jsem rozšířil kromě formulářových anotací ještě o anotace pro grid. Ty jsou o trochu jednodušší než anotace pro formulář. V základu definují typ filtru a také to, zda se daný sloupec bude či nebude řadit. Pokud není anotace pro grid uvedena, daný sloupec se v tabulce neobjeví.

center.neon

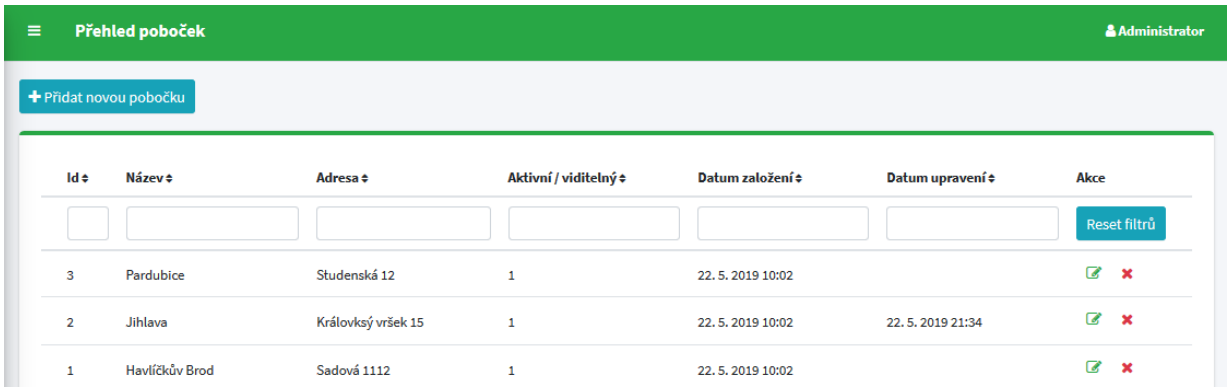
```
primaryKey: id

parameters:
    id:
        type: int
        title: 'Id'
        form:
            type: hidden
        grid:
            filter: text
            sort: true
            styleHeader:
                width: "70px"
    name:
        type: string
        title: 'Název'
        form:
            required: 'Název je povinné pole!'
            placeholder: 'Název'
        grid:
            filter: text
            sort: true

    address:
        type: string
        title: 'Adresa'
        form:
            required: 'Adresa je povinné pole!'
            placeholder: 'Adresa'
        grid:
            filter: text
            sort: true

    active:
        type: bool
        title: 'Aktivní / viditelný'
        form:
            type: 'bool'
        grid:
            filter: text
            sort: true

    created_date:
        type: date
        format: 'j. n. Y H:i'
        title: 'Datum založení'
```

Obrázek 14: Nastavení anotací ke gridu v konfiguračním souboru entity

Tyto anotace převádím v systému stejně jako u formulářů do principu upraveného JSON schématu, který poskytuje EAP pod GET metodou „/api/uni/get-grid-schema“. Je třeba předat parametr „entity“, který předpokládá název entity s anotacemi pro grid. Úryvek výsledku volání této EAP metody je vidět na obrázku č. 15.

Obrázek 15: JSON schéma pro grid

Toto schéma dostane z API (nebo z cache) JS generátor gridu v React aplikaci a na základě něj vystaví tabulku. Tabulku vystaví i včetně všech funkcionalit a callbacků na řazení, filtrování a stránkování (stránkování se zobrazí až od určitého počtu záznamů). Lze také doplnit odkaz na editační formulář, který již odkazuje na stránku s vygenerovaným formulářem z generátoru formulářů. Další akce, která je dostupná, je přidání tlačítka na mazání s callbakem na EAP, které zajistí smazání daného záznamu. Dále vygenerovanému gridu EAP dává možnost datového zdroje skrze API, který dodává data a má připravené metody pro funkce gridu, aby dle toho mohl posílat vyfiltrovaná, seřazená či offsetovaná data.



Obrázek 16: Náhled vygenerovaného gridu z anotací

Kód, který pak používá programátor v Reactu, vyobrazuje obrázek č. 17. Samotný grid je vygenerován pomocí třídy BaseGrid. Tento BaseGrid vyžaduje předat název entity, kterou chceme vykreslit. Dále kolik má být limit položek na jednu stránku. A závěrem jaká tlačítka se mají použít. Tlačítka je možné přidávat jakákoliv při dodržení zobrazeného vzoru.

```
30              <div className="card-body">
31                  <BaseGrid
32                      entity="center"
33                      limit="40"
34                      buttons={[
35                          {
36                              link: '/centrum/uprava/:id',
37                              text: '',
38                              icon: 'pencil-square-o',
39                              className: 'text-success'
40                          },
41                          'deleteModal'
42                      ]}
43                  />
44              </div>
```

Obrázek 17: Zápis gridu vytvořeného anotacemi v Reactu

Z popsaných principů vychází, že je nyní velice jednoduché vytvořit tabulku pro zobrazování dat z definic entit. Programátorovi stačí jen doplnit do konfiguračního souboru anotace pro grid a v React aplikaci si umístit třídu BaseGrid, která se mu už postará o zbytek.

4.5 Pokročilé nastavení grid a form anotací a generátorů

Na následujících případech se pokusím nastínit další rozšiřující funkcionality týkající se tabulek a formulářů a jejich generátorů.

4.5.1 Anotace pro cizí entity

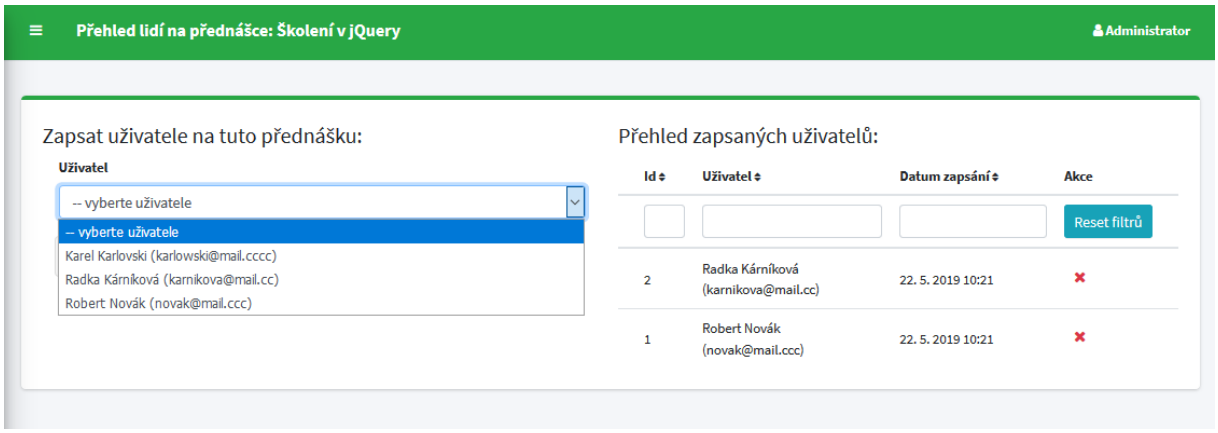
Na předchozích příkladech jsem ilustroval, jakým způsobem využívat form a grid generátor svázaný s anotacemi na entitě. Rozšiřující možností anotací je spojení na cizí entitu naší vypisované entity. Toto je řešené přímo skrze anotace, což uvedu na příkladu na obrázku č. 18, kde lze vidět, že vypisujeme z databáze celočíselnou hodnotu „author_id“, která je zároveň cizím klíčem k tabulce s administrátory/uživateli systému. Díky uvedení anotačního atributu „foreign“ nám EAP dokáže pro datové zdroje formuláře a gridu napojit na naši entitu cizí entitu. Na pozadí se udělá klasické spojení tabulek s tím, že se v selekci vyberou z cizí entity pouze sloupce uvedené v atributu „fields“. Anotace „mask“ nám udává, jakým způsobem se má entita vypsat. Konkrétně udává, v jakém tvaru se mají vytažené hodnoty za daného uživatele vypsat – proběhne tam nahrazení názvů sloupců oddělených znakem procenta za reálnou hodnotu (například výsledkem „mask“ může být: *„Jan Jindra (jindra-*

jan@email.cz)"). Funkce „mask" se aplikuje pro formulář i pro grid, kde by na ni mělo jít aplikovat filtrování a řazení.

```
24      user_id:
25          type: int
26          title: 'Uživatel'
27          foreign: user
28          fields: 'name|surname|email'
29          mask: '%name% %surname% (%email%)'
30          form:
31              type: select
32              prompt: '-- vyberte uživatele'
33              orderBy:
34                  surname: ASC
35              preload: true
36              required: 'Uživatel je povinný!'
37          grid:
38              type: text
39              filter: text
40              sort: true
```

Obrázek 18: Anotace pro spojení na cizí entitu

Výsledek formulářové i gridové anotace je vidět na obrázku č. 19, kde jsou již komponenty vykresleny. Vlevo máme formulářový prvek select, který obsahuje kolekci uživatelů, v systému doplněný skrze anotaci. Vpravo je vidět vygenerovaný grid, který obsahuje sloupec „Uživatel", který je z té samé entity a je vygenerován taktéž z anotace.



Obrázek 19: Náhled gridu a formuláře s napojením na cizí entitu

4.5.2 Vlastní schéma pro formulář, jiná ukládací metoda

Aby generátor formuláře nebyl úplně jednoúčelový, je připraven i na některé pokročilejší formuláře a lze do něj vložit i vlastní schéma nezávislé na schématu pocházejícího z entit. To se definuje jako JSON a nebo v JavaScriptu jako pole objektů a je uloženo přímo v třídě

SchemaForm, respektive BaseForm, která obsahuje základ generátoru formulářů. Pokud tyto třídy zdědíme, můžeme si vytvořit vlastní formulář nezávislý na schématu z entity.

```
19      /**
20       * Load schema for form
21       * @override
22       */
23      public loadSchema() {
24          // manual load schema for form
25          let schema = [
26              {
27                  name: 'login',
28                  required: 'Login je povinné pole!',
29                  title: 'Login',
30                  placeholder: 'Login',
31                  type: 'string'
32              },
33              {
34                  name: 'password',
35                  required: 'Heslo je povinné pole!',
36                  title: 'Heslo',
37                  placeholder: 'Heslo',
38                  type: 'password'
39              }
40          ];
41          let data = this.setDefaultsData(schema);
42          this.setState({schema: schema, data: data, loading: false});
43      }
```

Obrázek 20: Vlastní schéma pro formulář

Na obrázku č. 20 je vidět úryvek kódu, který je ve formulářové třídě, jež dědí od základního generátoru a přepisuje metodu na získávání schématu pro tvorbu formuláře. Takto vytvořený formulář již můžeme využít pro cokoliv – v tomto případě je zrovna využit jako přihlašovací formulář, který v podstatě nic neukládá do databáze, ale jenom autorizuje daného uživatele.

Pro tyto vlastní formuláře ale často nejde využít původní ukládací metoda na EAP, takže je potřeba jim napsat vlastní API pro jejich obsluhu a změnit jejich ukládací callback nastavením jiného – v tomto případě API, která ověří zadaného uživatele a v případě kladného vyřízení uživatele autorizuje a vpustí do systému.

4.5.3 Jiný způsob vykreslení formuláře generovaného ze schématu

V základním nastavení se formulář vykresluje postupně, tak jak jsou za sebou pole uvedená ve schématu, a dodržuje pořadí schématu. Aby bylo možné formulář vykreslit jiným způsobem, vymyslel jsem zde metodu, díky které lze každé formulářové políčko vykreslovat zvlášť. S tím, že si musíme vytvořit novou komponentu v Reactu, tu podědit ze základní třídy SchemaForm a přepsat si metodu pro vykreslování formuláře, ve kterém si nadefinujeme pomocí funkce „renderOneInput“, kde bude dané políčko vykresleno.

```
export class UserForm extends SchemaForm {
    constructor(props: any) {
        super(props);
    }

    /**
     * Render form
     * @override
     * @param state
     * @returns {any}
     */
    public renderForm(state) {
        return <div className="row">
            {this.renderOneInput('id')}
            <div className="row col-md-12 mb-4">
                <div className="col-md-6">
                    {this.renderOneInput('name')}
                    {this.renderOneInput('surname')}
                    {this.renderOneInput('email')}
                    {this.renderOneInput('password')}
                </div>
                <div className="col-md-6">
                    {this.renderOneInput('street')}
                    {this.renderOneInput('zip')}
                    {this.renderOneInput('city')}
                    {this.renderOneInput('country')}
                </div>
            </div>
```

Obrázek 21: Vlastní vykreslení formulářových prvků

5 Základní moduly

Po přípravě všech zmíněných principů a nástrojů k usnadnění tvorby sytému nám vznikají moduly, které byly jedním cílů práce. Aktuálně dokážeme velice rychle připravit základní kostru systému. Vytvořit si jakékoliv administrační číselníky, které se skládají z tabulek a formulářů a spravují danou agendu dat. Nicméně je třeba ještě jedna věc, a to pomoci programátorovi se stavbou konkrétních systémů založených na mém frameworku pro tvorbu informačních systémů. Vše je od začátku koncipované tak, že hlavní logika a hlavní rozhodovací části systému jsou postavené na API a JavaScriptová aplikace v Reactu sice přebírá část logiky, ale ta je většinou pouze doplňková.

Základní moduly fungují na principu osamostatnění vytvořených částí systému tak, aby se daly znovu používat v dalších systémech. A jde tu konkrétně o princip toho, že se na frameworku postaví systém a při stavbě se již myslí na to, že dané moduly potom rozšíří danou nabídku modulů frameworku, které budou mít možnost si vybrat programátoři a na jejich základě vystavět jiný systém.

Všechny moduly jsou koncipovány tak, že budou sloužit čistě jako podklad pro programátora při tvorbě nového systému, který bude tvořit zákazníkovi „na míru“. Tzn. každý modul musí být editovatelný a fungovat tak, že pouze doplní základní kostru systému a následně nechá programátorovi volné ruce si jej jakkoliv editovat či upravit. Na základě této editace modulu pak může programátor modul vzít a udělat z něj nový, který dá k dispozici ostatním pro stavbu dalšího systému.

5.1 Druhy modulů

5.1.1 Jednoduché číselníkové moduly

Nejjednodušší moduly systému jsou jednoduché číselníky. Pod tímto pojmem si můžeme představit jednoduchou správu emailů pro rozesílání newsletterů (zamýšlena je tu pouze evidence emailů a ne realizace rozesílání), ve které máme tabulku/grid, kde je seznam emailů s datem, kdy byly přidány. Dále tato sekce obsahuje formulář, který nám umožní daný newsletterský email opravit, či přidat nový. V podstatě si pod jednoduchým číselníkem můžeme představit jednu entitu, která neobsahuje žádné vazby na jiné entity (jiné entity na ni mohou mít vazbu, ale ona nemá na jiné). V tomto případě je separace modulu velmi jednoduchá, protože všechny tyto informace včetně toho, jak bude vypadat grid a formulář,

jsou obsažené v konfiguračním souboru entity a v React aplikaci je v podstatě jenom základní vykreslení těchto dvou komponent, které se ani nemusí separovat.

Tyto moduly se tedy skládají čistě jenom z jednoho souboru, a to konfiguračního souboru pro entitu (plus případně i importního SQL souboru pro tvorbu tabulky do databáze, protože entity fungují zatím pouze na bázi mapování a ne správy tabulek dle zadané konfigurace).

Jako připravené moduly typu jednoduché číselníkové máme v základním frameworku tři. První je popisovaná správa newsletterů. Další je správa poboček a třetí je jednoduchá správa webových stránek.

5.1.2 Složitější moduly se závislostmi

Tyto moduly již nelze stavět pouze na bázi separace jednoho konfiguračního souboru entity, jako tomu je v jednoduchých číselníkových modulech.

Jako příklad je zde uveden jeden modul tohoto typu. Jedná se o evidenci přednášek, pod ním si můžeme představit nějaké vypisované akce. Přednáška/akce má evidované různé parametry od kapacity, ceny za vstup či doby trvání atd. Dále má vazbu na entitu administrátorů a entity rezervací míst na přednášce a ta má vazbu na entitu uživatelů (administrátor = člověk přihlašující se do administrace – spravuje přednášky, uživatel = člověk z vně systému – přijde na přenášku).

Na tomto příkladu je vidět, že se jedná o mnohem komplexnější modul než jednoduchý číselník. Nicméně separace tu opět není složitá. Separují se dva entitní konfigurační soubory, a to entitní soubor samotných přenášek a entitní soubor rezervací míst na přednášce. Dále je nutné přidat závislosti daného modulu. A to závislost na modulu administrátorů a modulu uživatelů, které musí být v systému taktéž dostupné, či si je programátor po použití tohoto modulu musí upravit a závislosti odebrat.

V konečné fázi modul obsahuje 2 konfigurační sobory entity (plus případně SQL import tabulek), popis nutných závislostí na jiné konkrétní moduly a popis, jakým způsobem vykreslit v Reactu dané komponenty z modulu.

5.1.3 Speciální a komplexní moduly

Posledním typem z definované třídy jsou komplexní moduly, které jsou často hodně specifické pro daný systém. Do nich patří například pokročilé správy či procesy

pro zpracování či řízení nějakého konkrétního problému v systému. Tzn. jsou to moduly, které mají spoustu závislostí a naprogramované vlastní obsluhy pro formuláře/gridy, či mají naprogramované nějaké vlastní komponenty.

Jako příklad uveďme naprogramovanou správu uživatelů či správu administrátorů v systému. Správa administrátorů kromě evidence lidí, kteří mají přístup do administrace, obsahuje i autorizační modul pro přihlašování a správu skupin oprávnění pro dané administrátory.

Správa uživatelů tu nemá vlastní závislosti na jiné entity, ale zato obsahuje nahrávání a evidenci profilové fotografie, která je vytvořena jako vlastní komponenta v Reactu a má naprogramovanou i vlastní API obsluhu.

Obecně je u těchto typů modulů separace složitá a přenos mezi systémy taktéž složitější. Separovat se tu musí nejenom konfigurační soubory entit, ale i vlastní API soubory, modelové soubory na straně PHP. Na straně JS Reactové aplikace je pak třeba taktéž vyseparovat vlastní komponenty, které souvisejí s těmito moduly (např. nahrávání profilových fotografií či samotné přihlašování administrátorů).

5.2 Vyhodnocení modularizace

Modularizace nám přidává do frameworku možnost částečně si poskládat základní kostru systému z již dříve hotových systémů. Nicméně jsem ji navrhl a připravil tak, aby nebyla omezující a programátorovi pomohla. A zároveň aby nebyla nutnou podmínkou pro tvorbu nového systému – aby si programátor mohl sám vybrat, zda nějaký modul využije, anebo si napíše své vlastní bez toho, aby sáhl do základní databáze již hotových modulů.

Princip celé modularizace je v tom, že si programátor ze seznamu modulů vybere ty, které se mu hodí do jeho systému. Nahraje si je do svého projektu a následně si každý modifikuje a upraví pro dané použití. Tyto moduly se díky závislostem mohou kombinovat. Programátor může spojovat existující moduly, které předtím spojeny nebyly, a z nich vytvořit nové, o které by pak měl rozšířit databázi všech modulů.

Největší přidaná hodnota modularizace je vidět například na vzorovém příkladu, kdy budu potřebovat postavit novou administraci pro e-shop, který je tvořen zákazníkovi „na míru“. Tento systém začnu stavět tak, že si vezmu základní jádro vytvořeného frameworku. Pak si

otevřu a projdu základní databázi modulů a vyberu: správu uživatelů, správu administrátorů, správu newsletteru, správu poboček a správu webových stránek.

Správu uživatelů si modifikuji na správu zákazníků. Vyloučím z nich některá pole, která se mi do nového systému nehodí, a naopak přidám pole, která budou potřeba. Dále vezmu modul poboček a předělám ho na správu kamenných prodejen, kam přidám otevírací doby a možnost nahrát fotografii pobočky. Dále si vytvořím nový modul pro správu produktů a nový modul pro správu objednávek. Modul objednávek sváži s modulem produktů a modulem zákazníků atd. I přes nutné modifikace stávajících modulů došlo k výrazné úspoře času, než kdybych musel všechny vytvářet úplně znovu.

Nově vzniklé moduly by mohly v rámci tohoto vzorového příkladu rozšířit databázi modulů, aby byly pro další systémy znovu k dispozici. Jakmile získá tato databáze modulů na objemu, bude zveřejněna, aby si z ní mohl kterýkoliv programátor vybírat a používat již vytvořené moduly, které jsou co nejpodobnější jeho řešené problematice, a mohl je upravovat a modifikovat pro jeho řešený problém a tím zároveň vytvářet moduly nové a dále rozšiřovat databázi.

6 Závěr

Výsledný systém, který se mi podařilo vytvořit, hodnotím velmi kladně a již nyní je pro mě velkým přínosem. Myslím si, že jsem cíl práce splnil. Aktuálně na frameworku pro tvorbu webových systémů, který jsem ve své diplomové práci vytvořil, vzniká již první systém pro koncové využití. Pro mě samotného je velkým přínosem, a to hlavně z důvodu, že se mi podařilo vytvořit a zkonstruovat nástroj postavený na principech, které jsem v této práci navrhl a vymyslel.

V práci řeším návrh a implementaci systému, a to už od nástinu základní architektury, na které systém funguje. Základ jsem postavil na principech PHP Rest API s JavaScriptovou aplikací pro prezentaci dat. Pro konkrétní komunikaci obou částí jsem navrhl a zrealizoval pokročilý model pro výměnu dat založený na JSON schématu s využitím metody cachování dat s predikcí pro přednačítání a cachování dat už na úrovni JavaScriptu. Výhoda tohoto modelu je v tom, že nám umožňuje, aby nám webová aplikace běžela v realtimu, což klasické webové aplikace v základu nenabízejí. Výsledkem modelu je stav, kdy by uživatel neměl na systém čekat a měl by se mu výrazně zvýšit komfort práce s ním. Na principu tohoto modelu bych chtěl dále pracovat, ještě jej zdokonalit o přidání spojení skrze technologii HTML5 WebSoketu a jeho hlavní výhody ještě více zdokonalit.

Dále jsem v práci navrhl a implementoval nástroje, které pomohou a zefektivní práci programátora při tvorbě informačních a administračních systémů. Tyto nástroje pomáhají optimalizovat určité operace, se kterými se programátor setkává při tvorbě systémů. Mezi tyto nástroje patří například Form generator, u kterého vidím největší výhodu v tom, že programátorovi ušetří spoustu práce a poskytne mu nástroj pro efektivní a rychlou tvorbu formulářů v systému. Také jsem vyřešil problematiku univerzálního přístupového bodu k API, kterou jsem nazval EntityAccessPoint (EAP). Tento nástroj slouží pro práci s daty skrze veřejné API a jeho největší výhodou je, že programátorovi nabízí mocný a bezpečný nástroj pro práci s databází, který může využívat přímo z JavaScriptové aplikace.

Vytvořený framework pak směřuje k modularizaci částí systémů, které jsou díky jeho nástrojům, a dobrému návrhu architektury, tvořeny. Pomalu již vzniká databáze modulů, které je možné využít pro rapidní urychlení vývoje a sestavení základní kostry systému. Moduly lze kombinovat, doplňovat a jakkoliv modifikovat a tím pádem vytvářet i nové a těmi zpětně doplňovat databázi modulů k využívání.

Mým plánem do budoucna je vyzkoušet vytvořený framework pro tvorbu webových systémů řádně v praxi a postavit na něm několik systémů. Dodělat do něj vylepšení, která určitě vzejdou z praxe, a následně jej uvolnit a dát k dispozici i ostatním programátorům, aby jej mohli využívat a modifikovat.

Použité zdroje

ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

Bootstrap [online]. [cit. 2019-05-16].
Dostupné z: https://getbootstrap.com/

Composer - Dependency Manager for PHP: Introduction [online]. [cit. 2019-05-15].
Dostupné z: https://getcomposer.org/doc/00-intro.md

Composer - Dependency Manager for PHP: Basic usage [online]. [cit. 2019-05-15].
Dostupné z: https://getcomposer.org/doc/01-basic-usage.md

DROETTBOOM, Michael. *Understanding JSON Schema: What is a schema?* [online]. 5. 2. 2019 [cit. 2019-05-16].
Dostupné z: https://json-schema.org/understanding-json-schema/about.html

FACEBOOK INC. *React: Tutorial: Intro to React* [online]. 2019 [cit. 2019-05-16].
Dostupné z: https://reactjs.org/tutorial/tutorial.html

GRUDL, David. *Dibi* [online]. 2008-2019 [cit. 2019-05-15].
Dostupné z: https://dibiphp.com/cs/documentation

HUSEBY, Sverre H. *Zranitelný kód*. Brno: Computer Press, 2006. ISBN 80-251-1180-6.

CHAFFER, Jonathan a Karl SWEDBERG. *Mistrovství v jQuery: [kompletní průvodce vývojáře]*. Brno: Computer Press, 2013. Mistrovství. ISBN 978-80-251-4103-8.

LINHA, Pepa. *Blog.webdream: Gulp.js - nástroj pro automatizaci úkolů* [online]. 2016 [cit. 2019-05-16].
Dostupné z: http://blog.webdream.cz/ostatni/gulp-js-nastroj-pro-automatizaci-ukol

MASSÉ, Mark. *REST API Design Rulebook*. 1. O'Reilly Media, 2012. ISBN 978-1-449-31050-9.

Nette [online]. 2019 [cit. 2019-05-15].
Dostupné z: https://nette.org/

Nette: Cache [online]. [cit. 2019-05-15].
Dostupné z: https://doc.nette.org/cs/3.0/caching

Nette: Dependency Injection [online]. [cit. 2019-05-15].
Dostupné z: https://doc.nette.org/cs/3.0/dependency-injection

Nette: Neon [online]. [cit. 2019-05-15].
Dostupné z: https://doc.nette.org/cs/3.0/neon

Nette: Tracy [online]. [cit. 2019-05-15].
Dostupné z: https://tracy.nette.org/cs/guide

Nette: Hashování hesel [online]. [cit. 2019-05-16].
Dostupné z: https://doc.nette.org/cs/3.0/passwords

Npm: About npm [online]. [cit. 2019-05-16].
Dostupné z: https://docs.npmjs.com/about-npm/

Npm: axios [online]. 2018 [cit. 2019-05-16].
Dostupné z: https://www.npmjs.com/package/axios

Webpack [online]. [cit. 2019-05-16].
Dostupné z: https://webpack.js.org/concepts

ZAKAS, Nicholas C. *JavaScript pro webové vývojáře*. Brno: Computer Press, 2009. Programujeme profesionálně. ISBN 978-80-251-2509-0.

Přílohy

Příloha CD – Zdrojové kódy frameworku pro tvorbu webových systémů.