|  |  |  |
| --- | --- | --- |
| **Univerzita Hradec Králové Fakulta informatiky a managementu Katedra informatiky a kvantitativních metod** | | |
| **Vývoj portálových aplikací**  Diplomová práce | | |
| Autor: Zdeněk Věcek Studijní obor: Informační management | | |
| Vedoucí práce: doc. Ing. Filip Malý, Ph.D. | | |
| Hradec Králové | duben 2015 | |
| Prohlášení:  Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury. | | | |
| V Hradci Králové dne 18.1.2015 | | Zdeněk Věcek | |
| Poděkování:  Chtěl bych upřímně poděkovat panu doc. Ing. Filipu Malému, Ph.D. za odborný dohled, cenné rady, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnoval. | | | |

Anotace

Diplomová práce se zabývá

Annotation

Title: Portal application development

The

Obsah

[1 Úvod 8](#_Toc409351665)

[2 Systémy pro správu obsahu – CMS 8](#_Toc409351666)

[3 Portál a portlety 12](#_Toc409351667)

[3.1 Portál 12](#_Toc409351668)

[3.2 Portlet kontejner 12](#_Toc409351669)

[3.3 Portlet 13](#_Toc409351670)

[3.3.1 Životní cyklus portletu 14](#_Toc409351671)

[3.4 Portletové specifikace 15](#_Toc409351672)

[3.4.1 JSR 168 15](#_Toc409351673)

[3.4.2 JSR 286 16](#_Toc409351674)

[4 Konkurence na trhu 17](#_Toc409351675)

[4.1 Backbase 18](#_Toc409351676)

[4.2 IBM WebSphere 18](#_Toc409351677)

[4.3 eXo platform 19](#_Toc409351678)

[5 Funkční popis aplikace 19](#_Toc409351679)

[6 Výběr JS frameworku 20](#_Toc409351680)

[6.1 ReactJS 21](#_Toc409351681)

[6.2 Flux 23](#_Toc409351682)

[7 Architektura aplikace Evrem 26](#_Toc409351683)

[7.1 Frontend 26](#_Toc409351684)

[7.2 Backend 30](#_Toc409351685)

[7.2.1 Apache Maven 34](#_Toc409351686)

[7.3 Administrátorské nastavení 37](#_Toc409351687)

[8 Ukázky aplikace 37](#_Toc409351688)

[9 Zdroje 37](#_Toc409351689)

[10 TODO 38](#_Toc409351690)

Seznam obrázků

Seznam tabulek

# Úvod

Portálové aplikace zastávají již delší dobu klíčovou roli ve sféře enterprise systémů. Rozsáhlé korporace obsahují stovky malých, středních, ale i velice komplexních webových aplikací, ke kterým musí zaměstnanec denně přistupovat. V takovém to prostředí je uživatel nucen u každé aplikace poskytnout své autentizační údaje na různých přístupových adresách či pracovat na různých platformách. Kooperace těchto aplikací je řešena pomocí rozmanitých řešení. Vývoj probíhá v různých jazycích, kde znalosti nebývají přenositelné. To mnohdy vede ke ztrátě cenného know how po odchodu vývojáře, jež se jen velmi těžko jeho nástupcům získává. Rozmach v oblasti podnikového softwaru musel být postupně korigován do větších integrovaných celků, jež poskytovaly odpovědi na společné často řešené problémy a poskytovaly jednotné rozhraní.

Portletová architektura odpovídá na tyto otázky. Na trhu můžeme nalézt jak kvalitní řešení typu open source, tak především systémy s většinou nemalými licenčními poplatky. Téměř vždy se jedná o velice komplexní systémy s velkou učící křivkou, z důvodů potřeby mnoha technologií a konceptů, které se v tomto segmentu využívají. Dokumentace pokrývá pouze základní část, a to z důvodů udržení know how, nabídnutí svých expertních služeb za poplatek nebo z důvodů rychlého vývoje těchto produktů a zanedbání souběžného zpracování znalostí. V případě platformy Liferay tomu není jinak a vývojář je tak často odkázán na reverse engineering kódu portálu, jakožto jediný spolehlivý, dostupný zdroj informací.

# Systémy pro správu obsahu – CMS

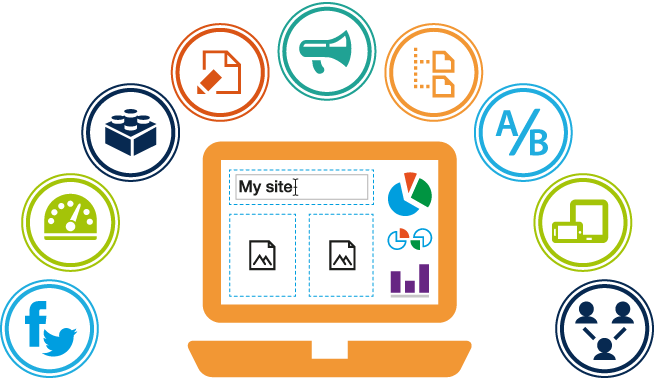
Content management system či specifičtěji web content management system je nástroj umožňující flexibilní a uživatelsky přívětivou správu obsahu na webu. Obsahem může být myšleno téměř cokoliv – text, obrázky, videa, hudba nebo dokumenty. V dřívějších dobách probíhal proces publikování nových informací následovně. Vlastník přišel s požadavkem, co by potřeboval umístit na své stránky. Sepsal text včetně obrázků a dalších médií do emailu a zaslal ho svému správci webu. Ten převedl obsah do HTML podoby a s trochou štěstí zaslal zformátovaný fragment ke schválení. Poté aktualizoval stránky nahráním nových souborů přes ftp připojení na web server. Celý proces vyžadoval úzké propojení vývojáře a zadavatele a ne vždy se představy o finální podobě střetávaly. Rychlost zpracování nebyla dostatečně pružná a vyžadovala dodatečné náklady.

Jednou z odpovědí na tuto problematiku byly WYSIWYG editory neboli software pracující na principu „What You See Is What You Get“. Nástroje jako MS FrontPage nebo Adobe Dreamweaver umožnily mírně pokročilým uživatelům tvořit vlastní stránky za pomocí jejich vizuálního aranžování. Uživatel snadno uchopil element, který naskládal na svoji stránku a editor se postaral o překlad do HTML kódu. Jejich nevýhodou bylo vygenerování mnoha nepotřebných fragmentů, které často způsobovaly problémy s kompatibilitou, a byly umísťovány hůře při indexování roboty. Ruku v ruce s vývojem obdobných nástrojů, šel i vývoj nových standardů pro HTML, CSS a JavaScript. Tím se vývoj začal stávat složitější ve všech ohledech, jelikož nové trendy vyžadovaly weby dynamičtější, responzivní a především uživatelsky přívětivé a zajímavé.

Tento pokrok přispěl ke vzniku WCMS nástrojů, jež většinu základních problémů řeší a nechávají prostor pro uživatelskou správu, tak aby nebyla narušena konzistence a celková funkčnost. WCMS umožňují vysokou modularitu s možností spravovat nejen obsah, ale i rozšiřovat aplikaci o zásuvné moduly, poskytující nové funkcionality. Běžný uživatel mohl bez asistence vývojáře vložit jednoduchý obsah s určitými formátovacími možnostmi do předem vydefinovaných bloků na stránkách, spojených navigací. Tento pokrok přispěl k decentralizované tvorbě nových dokumentů a obsah začal vznikat flexibilněji a byl sdílen mezi zainteresovanými stranami. To byly počátky WCMS systémů. S rozvojem sociální sítí a dynamický webů obecně, si uživatelé postupně zvykli na vyšší standard v oblasti interaktivity, UX a možností poskytovaných webovou aplikací. Adaptování na tyto trendy dospělo do stavu, kdy lze WCMS definovat několika body, jež většina z nich implementuje.

* Tvorba, editování, mazání a **sdílení rozmanitého obsahu** pro běžného uživatele.
* **Autentizační a autorizační** přístupové bariéry s managementem uživatelů, vrstvení do rolí a skupin.
* **Spolupráce** a komunikace mezi uživateli.
* **Škálovatelnost** za běhu nebo s relativní jednoduchostí. Lze přidávat jak nové stránky, tak nové aplikace.
* **Stylování, šablonování** a celková změna uživatelského designu skrze nastavení.
* **Workflow** tvorby obsahu. Nový obsah mohou zakládat určití uživatelé, podléhající schválení zodpovědnou osobou, která obsah zreviduje a vypublikuje nebo vrátí k dopracování.
* **Staging** umožňující náhled před samotným vypuštěním obsahu do světa.
* Obsah lze snadno **internacionalizovat** do více jazyků.
* **Verzování** jednotlivých změn.
* **Vyhledávání** v obsahu, indexování.
* **Emailování** uživatelům na základě vydefinovaných šablon.

Následující diagram výstižně popisuje možnosti moderních webových systémů pro správu obsahu. Propojení se sociálními sítěmi zaštiťuje autentizaci přes sociální účty nebo provázanost s firemními stránky, založenými například na Facebooku, Twitteru či Pinterestu. Aplikace jsou optimalizovány pro vysoký výkon a poskytují možnosti rozšíření. Obsah lze nejen tvořit, upravovat, ale i sdílet a seskupovat do vlastních struktur. Mnoho CMS poskytuje vlastní wiki, blogy a řešení pro nahrávání a sdílení dokumentů. Zásuvné moduly poskytují doplňkové funkce a často zde nechybí ani propojení s ostatními uživateli pro spolupráci na společném úkolu. A/B testování umožňuje neinvazivní experiment pro získávání informací od uživatelů o úspěšnosti nového obsahu, designu či funkcionality zobrazené pouze části uživatelů z demografického spektra. Trh mobilních aplikací se neustále rozvíjí a odhaduje se, že každý třetí návštěvník je mobilní. Z tohoto důvodu je brán velký ohled na optimalizaci pro tato zařízení, aby přinesla uživateli co nejlepší zážitek z prohlížení. Celé rozvržení si může uživatel, případně administrátor upravit dle vlastního uvážení do vydefinovaného layoutu bez zásahu do kódu.



Základní funkce a moduly dodávané s produktem většinou nedostačují. Uživatel může hledat možnosti rozšíření skrze obchody se zásuvnými moduly, kam široká veřejnost může přispívat a rozšiřovat tak působnost platformy nebo může požadavek zadat vývojáři, který takový modul vytvoří. Pro snadnější modifikace jsou tyto systémy dodávány se sadou vývojových nástrojů připravených pro tvorbu nových rozšíření nebo úpravu stávajících řešení. Všechny podpůrné komponenty, API, IDE mají souhrnný název Content management frameworks. Při výběru je důležitý především programovací jazyk v jakém je produkt napsaný a jaký je dále nutné použít pro samotný vývoj. Hned poté jsou to technologie, které daná platforma umožňuje nebo je přímo v sobě integruje.

Produktem stojícím vedle WCMS jsou portály. Rozdíl bychom mohli spatřit v zaměření, které je v prvním případě orientováno na bohatý obsah zobrazovaný koncovému uživateli tvořený správcem. Druhá varianta je orientována na sdružení rozmanitých zdrojů do uceleného systému zobrazovaných dle oprávnění a rolí. Většina portálů však zahrnuje integrovaná WCMS řešení, proto nelze zcela vymezit působnost těchto dvou pojmů.

# Portál a portlety

## Portál

Tento název je do IT světa přejat z definice vstupní brány do jiného světa či ozdobný vstup do budovy. Výběr tohoto jména začne dávat větší smysl po přesnějším vydefinování, kdy lze spatřit analogii mezi různými interpretacemi. Portál je webový systém poskytující centrální přístupový bod uživateli k rozmanitému obsahu a službám. Agreguje obsah podle uživatelských preferencí a nabízí řadu možností pro personalizování prvků na stránce, rozložení, jejich vzhled nebo bližší nastavení vztahující se k uživateli. Lze je dělit podle působnosti na horizontální a vertikální. Vertikální ucelují zdroje a služby na úzce specifikovanou oblast například pojišťovna nebo mobilní operátor. Horizontální nemají definované hranice a mohou tak sdružovat zdroje a služby z celého spektra témat od počasí, politiku, financí přes automobily, vědu či módu. Příkladem je seznam.cz či yahoo.com. Nemusí se vždy jednat o giganty, překlenující všechna odvětví. Horizontálním portálem může být i aplikace zaměřená na jeden trh, využívaná více společnostmi nebo pouze čerpá z rozmanitých zdrojů či platforem.

Zajišťuje autorizační či autentizační mechanismy jakými jsou například Single Sign On, integrace se sociálními sítěmi, adresářovými službami LDAP, openID nebo OAuth. Poskytují kompletní správu uživatelů, rolí a hierarchicky je uspořádává do organizací nebo logických skupin. Nechybí ani základní komponenty pro sdílení a úpravu dynamického obsahu či přímo integrovaného CMS. Výsledkem portálové stránky je sloučení několika různých HTML fragmentů generovaných portletovými aplikacemi. Portál přímá požadavky od klienta a deleguje je do portlet kontejneru.

## Portlet kontejner

Portlet kontejner poskytuje běhové prostředí pro portlety, podobně jako servletový kontejner zaštiťuje servlety. Spravuje jejich životní cyklus a stará se o přijímání požadavků od hostujícího portálového serveru, které dále deleguje konkrétním portletům. Jak servlety tak portlety generují obsah, který může být statický i dynamický.

Portlet A

Portlet B

Klientský prohlížeč

Portál server

Portlet kontejner

Portlet A

Portlet B

Klient zobrazuje obsah, přijímá uživatelské požadavky

Portál agreguje obsah portletů do jedné stránky

Kontejner spouští portlety a spravuje jejich životní cyklus

## Portlet

Portlety jsou webové komponenty zodpovědné za specifické funkcionality, služby či určitý obsah. Jejich klíčovým prvkem je možnost shlukování na jednu stránku jako nezávislé a soběstačné celky. Oproti servletové architektuře kdy je na danou adresu namapován příslušný servlet, vyřizující příchozí požadavky a vykreslující celou stránku, portletová architektura pracuje s konceptem mapování jedné URL více portletům, obsluhovaných jejich kontejnerem a vykreslující pouze část – fragment HTML.

Odeslání formuláře uvnitř portletu zpravidla nekončí obnovením celé stránky, ale pouze rámcem ve kterém je umístěn, data mohou pocházet z několika zdrojů, aniž by uživatel poznal rozdíl a osoby s vyšším oprávněním mohou mít přístup k více portletům.

Vyznačují se také rozšiřujícími vlastnostmi typu Window state nabývající hodnot maximalizovaný, normální či minimalizovaný určující jakou část stránky zabírají. Mód portletu mimo výchozího View podporuje také Edit, jež je vhodný pro administrátorské úpravy a Help, který slouží pro sdílení informací o portletu a o tom jak s ním pracovat. Mnohdy je zapotřebí perzistovat dodatečné informace a nastavení, dostupná i po restartu serveru, a proto je zde mechanismus umožňující nastavovat a získávat jednotlivé položky Preferences. Ty je možné nastavit staticky do souboru portlet.xml nebo dynamicky do databáze spravované portálem.

### Životní cyklus portletu

Životní cyklus portletu vychází z interface **javax.portlet.Portlet,** jež nám definuje čtyři základní metody zajišťující interakci s portlet kontejnerem. Ve stejném API nalezneme výchozí implementaci tohoto rozhraní v podobě třídy **GenericPortlet** dostačující většině užitých případů. S rozšiřováním standardu přibyly další funkce, které tato třídy řeší. Jedná se o interface **PortletConfig, EventPortlet** a **ResourceServingPortlet.**

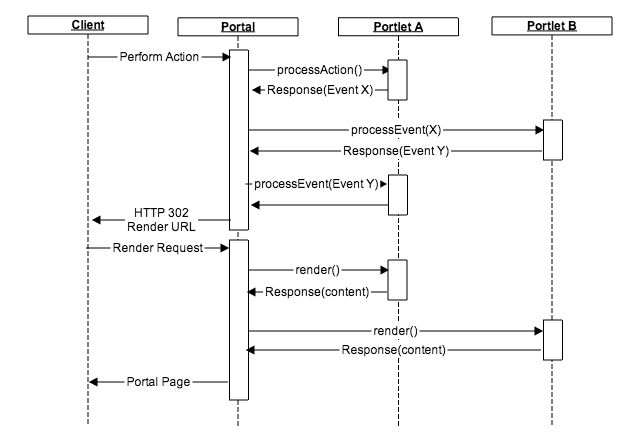
Základní metody životního cyklu:

* **init(PortletConfig config) –** je volána jednou ihned po vytvoření instance portletu. Může sloužit pro spouštění inicializačních úkonů. PortletConfig obsahuje statické informace specifikované v popisném souboru (portlet.xml).
* **processAction(ActionRequest request, ActionResponse response) –** metoda volána po odeslání formuláře nebo kliknutí na odkaz. Úkony v této lokaci slouží pro vykonávání business logiky či modifikaci dat. Je zde umožněno měnit stav portletu.
* **render(RenderRequest request, RenderResponse response) –** metoda následující po action fázi v životním cyklu. Slouží pro vykreslení fragmentu obsahu, který je také závislý na aktuálním stavu portletu.
* **destroy() –** posledním úkonem před spuštěním garbage collectoru je vykonání této metody, jejímž úkolem je uvolnění všech rozpracovaných zdrojů.

Rozšiřující operace životního cyklu:

* **EventPortlet –** po vykonání action fáze je před render fází, v případě nadefinování, zavoláno vykonávání události. To umožňuje meziportletovou komunikaci, která je popsána jako loosely coupled. Portlet zavolá setEvent() během vykonávání action fáze a konzumující strany poslouchající na danou událost, vykonají processEvent(), v němž je dostupný odesílaný objekt, metodou getEvent() pod daným EventRequest. Takto spolu mohou jednosměrně interagovat vzájemně propojené strany.
* **ResourceServingPortlet –** obsahuje jedinou metodu serveResource() sloužící pro AJAX komunikaci volanou JavaScriptem na klientovi.

Provázanost základního životního cyklu s událostmi je znázorněn na diagramu níže.



## Portletové specifikace

V době kdy přicházely na trh portálová řešení, nebyl specifikován žádný standard. Díky tomu vznikala dodavatelsky závislá řešení a uživatelé byli odkázáni na použití jediné platformy. Netrvalo dlouho a velcí lídři na trhu s portály vznesly požadavek do Java Community Process (JCP) na standardizaci těchto aspektů, pro zajištění kompatibility, přenositelnosti a jednotného přístupu k této problematice. Reakcí bylo v roce 2003 vytvoření specifikace JSR 168 a později JSR 286.

### JSR 168

Tato první specifikace definuje portlety jako webové komponenty založené na jazyce JAVA spravující nadřazeným kontejnerem. V tomto kontextu se jedná o zásuvné moduly rozšiřující presentační vrstvu portálu.

Specifikace standardizuje následující aspekty:

* Definuje běhové prostředí portletů – kontejner a API pro jejich komunikaci.
* Zajišťuje mechanismus ukládání dočasných a trvalých dat.
* Poskytuje způsob integrace se servlety, potažmo JavaServer Pages (JSP).
* Definuje strukturu pro nasazení do prostředí webového kontejneru.
* Umožňuje portabilitu napříč portály dodržující JSR 168.
* Agreguje portlety do portálové aplikace ve formě WAR.
* Definuje popisný soubor portlet.xml, rozšiřující standardní popisný soubor web.xml u servletových aplikací.

### JSR 286

Po několika letech uvedení do provozu prvního standardu, bylo otestováno a zjištěno mnoho nedostatků. Jelikož neexistoval žádný standard, kterého by se dodavatelé portálových řešení měli držet, tvořili svá vlastní řešení, kterým kompenzovali mezery ve specifikaci. Toto konání způsobilo opět ztrátu portability, ze které portlety těží. V reakci na to, vznikl v roce 2008 standard JSR 286, který tyto nedostatky řeší.

Nově zpracované a rozšířené oblasti:

* Meziportletová komunikace prostřednictvím **událostí.**
* **Veřejné render parametry** umožňující sdílení dat mezi portlety.
* **Poskytování zdrojů** (souborů) pomocí metody serveResource.
* Podpora **AJAX** požadavků.
* Rozšíření oblasti působnosti o přístup ke **cookies a hlavičky dokumentu**.
* Zapracování **dalších typů životního cyklu** ze specifikace servletů, umožňující více přístupových bodů pro vlastní reakce na dané události.
* Přidání možnosti rozšíření o **extension points**, které modifikují či obalují určité chování našimi funkcemi.
  + **Filtry** – akce vykonávané před nebo po dané události.
  + **URL Listeners** pro globální modifikaci adres a parametrů.
* **Přidání parametrů** běhovému kontejneru.
* **Zpětná kompatibilita** s přechozím standardem.
* Nové možnosti **cachování** pro zvýšení výkonu.
* Přiblížení **PortletSession** servletové variantě HttpSession umožňující pamatování uživatelského stavu po dobu prohlížení.

# Konkurence na trhu

Agentura Gartner každoročně srovnává a vyhodnocuje enterprise nástroje, zařazené ve stejných kategoriích. Mezi lídry v oblasti horizontálních portálů patří IBM, Liferay, Microsoft, Oracle a SAP. Tato pětice patří v roce 2014 k jedničkám na trhu, které jsou svoji stabilností a perspektivitou vhodné pro nejnáročnější portálová řešení. Ne náhodou je tématem této práce platforma Liferay, která již po páté v roce obsadila pozici lídra v magickém kvadrantu. Pro srovnání jsem si vybral kandidáty IBM WebSphere, eXo platform a Backbase. Všechny portály jsou postavené na Java EE standardech pro objektivnější srovnání.



## Backbase

Backbase portal se pyšní titulem vizionář roku. Důvodem je využití moderních technologií a inovativního přístupu ve vztahu k zákazníkům. Díky tomu byl označen jako Customer Experience Platform, vhodný pro použití především ve scénářích kdy je v roli uživatele zákazník. Nabízí řadu nástrojů pro podporu marketingu. Obsah může být zobrazován na základě demografického zařazení, zdali je uživatel student, podnikatel, senior, atp. Integrovaný WYSIWYG editor a nástroje pro analýzu návštěvníků. Snaží se o jednotný uživatelský zážitek, který není narušován skoky mezi různými aplikacemi. Jeho prezentační vrstva je založena na dialozích a widgetech, odstraňujících potřebu obnovovat stránku. Svoji sílu demonstruje např. na bankovní aplikaci, jež připomíná sociální síť s nástroji pro komunikaci, inteligentní kalkulačkou, správou svých financí v předdefinovaných kategoriích, ale i snadnou platbu a grafické vyhodnocení cashflow.

Jeho čerstvost na trhu má ovšem své nevýhody, v podobě nedostatečné historie v určitých odvětvích, což může odradit potencionální zákazníky. Nejedná se o open source licenci a svoje know-how si velice dobře chrání. Není proto snadné s touto platformou začít a řádně odzkoušet její možnosti.

Funkční fragmenty v případě Backbase nazýváme widgety. Jedná se o standardní portlety implementující oba dva standard JSR 168 a 286, od čehož se odvíjí i základní dodávané parametry popsané v předchozí kapitole.

## IBM WebSphere

IBM WebSphere představuje dlouhodobou jedničku na trhu s portálovými řešení. Díky své dlouholeté zkušenosti si vybudoval pověst tradičního portálu. Mnoho zákazníků na tyto zkušenosti slyší, což podporuje i faktická implementace snad každého případu B2C, B2B a B2E. Je dodávám s nejpestřejší škálou zabudovaných nástrojů, které jsou dále specializované na konkrétní použití ve vztahu k zákazníkovi nebo zaměstnanci.

IBM usiluje o dodání výjimečného zážitku uživateli používajícího portál, tyto komponenty a principy označuje pojmem Digital Experience. Sem spadají nastavení pro responzivitu, propojení se sociálními sítěmi, tvorbu formulářů během okamžiku přímo na webu, cloudová řešení, ale i bohatý obsah a analýzu uživatelů.

Velkou bariérou je cena, za kterou je možné portál provozovat. Roční licence jsou velice nákladné a mohou být odrazující. Je nutné hledět na celkové náklady na provoz IT, kam spadá integrace se stávajícími službami, hardware, nový vývoj, penále za výpadky či chyby, které ve většině případů několikanásobně převyšují výdaje za software. Analýza, školení, testování a mnoho dalších položek v rozpočtu, související s provozem a vývojem portálu, jež mohou být součástí již hotového řešení.

## eXo platform

Exo platform spadá do kategorie intranetových sociálně založených enterprise portálů. Z tohoto důvodů není zařazen mezi horizontální portály. Jeho licence je však nabízena i v modelu open source, a proto se jeví jako ideální kandidát vedle Liferay portálu. Ačkoliv je tato technologie na trhu od roku 2003, prošla si několika zásadními změnami a stejně jako ostatní portály představené výše se zaměřuje na úzkou vazbu na potřeby uživatele a personalizovaný zážitek. Dokumentace však nabízí pouze základní případy užití a příspěvky na komunitním fóru zůstávají často nezodpovězené. Exo platform bych volil jako vhodnou alternativu vůči Liferay. V mnoha ohledech, především ve využití frontendových technologií se eXo jeví lépe. Bohužel ještě nějakou dobu potrvá, než se vytvoří dostatečná znalostní základna pro plynulý vývoj.

Byly představeny 3 vedoucí portálová řešení, excelující vždy ve specifickém ohledu. Na trhu je bohužel velice málo open source produktů s kvalitní výbavou a informační základnou. Společným atributem je vize jakou všechny produkty sdílejí. **Orientace na uživatele a dokonalý zážitek z prohlížení**. Tento princip nalezneme v každém z nich.  Portály se postupně začínají podobat sociálním sítím a je jen těžké odhadnout, zdali bude tento faktor dostatečně využit a neodradí potenciální zákazníky. V následujících kapitolách si představíme platformu Liferay, která dominuje v open source řešeních a díky tomu má nejrozšířenější znalostní základu, která začíná u oficiální dokumentace, vydaných knih, příspěvcích na fórech, pokračující přes vypracovaná řešení na GitHub nebo popsané postupy na blogách či wiki. Tento aspekt shledávám klíčovým pro vývoj, jelikož výrazně zkracuje dobu řešení požadavku.

# Funkční popis aplikace

Aplikace Evrem slouží k upomínkování, organizování, tvorbu TODO listů, sticky note nebo čistě jen jako kategorizovatelný zápisník. Zaměřuje se na uživatelskou přívětivost a moderní frontendové technologie. Segment trhu je zacílený na mladé osoby co hledají jednoduchost, pestrost a interaktivitu. Hlavní motivací je nezapomenout. Uživatel vloží upomínku s roční platností, a přestože aplikaci nebude aktivně využívat, v den události obdrží email s vlastnoručně nadefinovanými parametry. Ačkoliv podobné aplikace existují a to převážně na mobilních platformách, velmi často si data udržují pouze v interní paměti a v okamžiku ztráty či reinstalace telefonu jsou data ztracena. Vyšší funkce jako export dat bývá velice často podmíněn zakoupením licence. V této aplikaci jsou všechny funkce zdarma, pokud se uživatel rozhodne stáhnout svá data a aplikaci dále nepoužívat, provede export jednoduše do formátu \*.xlsx tedy Excelu. Silnou stránkou aplikace je interaktivní zeď poznámek, kterou si uživatel může mechanismem Drag-And-Drop spravovat a zeď se mu transformuje za pomocí detekce kolize jednotlivých bloků. Díky portletové architektuře lze znovu využívat jednotlivé komponenty a v budoucí verzi nebude problém zajistit prémiovým uživatelům různé možnosti či uspořádání dle svých preferencí. V dnešní době již byla velká část nápadů realizována. Klíčem k úspěchu však zůstává finální provedení produktu a jeho propagace.

# Výběr JS frameworku

Liferay ve své klientské vrstvě využívá frameworku YUI verze 3 od Yahoo, který tvoří základnu pro komponentovou knihovnu Alloy UI. Koncem srpna 2014 oznámilo Yahoo ukončení vývoje frameworku YUI. Důvodem bylo odchýlení od současných trendů, které jsou tvořeny single page aplikacemi, izomorfními aplikacemi tvořené pomocí Node.js, ale i nástroji pro správu závislostí jakou jsou NPM či Bower, díky kterým lze celá klientská aplikace sestavit za pomocí úkolovacích nástrojů jako Grunt či Gulp a dalšími nezbytnými nástroji, které jsou nezbytné pro transpilaci kódu, správu závislostí, oddělování scope a mnoho dalších funkcí, které lze řešit nástroji jako Browserify nebo Webpack.  Vývoj YUI trval téměř 10 let a přestože se jednalo o velice komplexní knihovnu, bylo jasné, že v současné podobě nemůže splňovat konkurenceschopnost a být kompatibilní se zmíněnými nástroji.

Velice populárním JS frameworkem v komunitě LFR je zcela bezkonkurenčně AngularJS. Je velice jednoduchý pro prvotní inicializaci a téměř hned dokážeme vytvořit model komunikující s template/view vrstvou, jelikož zde funguje mechanismus Two-way data binding. Tento mechanismus zajišťuje propagování změn z view do modelu na straně klienta, ale i opačným směrem z modelu do view. Na počátku vývoje šetří tento přístup mnoho času a technických prostředků. Velice jednoduše lze přidávat nové controllery, hierarchicky je zanořovat a následně z nich dědit. Každý controller má svůj vlastní scope, který nám odděluje jednotlivé jmenné prostory, čímž se nám uchová čistota v globálním scope. Není proto nutné využívat dalších nástrojů jako je CommonsJS pro vymezení oddělených scope a správu závislostí. Angular také nabízí velice účinnou, ale zároveň nebezpečnou zbraň, prolínající se jak JS vrstvou tak template vrstvou. Jedná se o direktivu scope.watch(angularExpression, callback). Tato funkce zajišťuje kontrolu zadaného expression, kde při každé změně je zavolán callback se starou a novou hodnotou ve vstupních parametrech. Výhody jsou jasné, jde zejména o rychlou reakci na konkrétní změny zadanou funkcí v typickém příkladu validací. Problém nastává v okamžiku, kdy scope obsahuje desítky instancí watcher a v callbacku jsou měněny hodnoty, na kterých jsou opět navěšeny objekty watcher. Situace začne být velice nepřehledná a debug chyb v takovémto systému může zabrat hodiny i dny. Two-way data binding nese další obrovskou daň v podobě dirty checking mechanismu, který kontroluje změny v celé hierarchii scope a reaguje na ně příslušnou aktualizací. Celý cyklus se volá vždy minimálně dvakrát pro kontrolu, že byly všechny změny zpropagovány a že jiný event mezitím nezměnil hodnotu již ověřené proměnné. Pro případ, že by se někde vytvořila smyčka v našich watcher je nastaven limit opakování nad danou proměnou na 10 iterací. Výsledkem je velká výkonová náročnost, která způsobuje nemalé problémy i moderně vybaveným PC, především u větších aplikací. Dalším problémem je velká učící křivka při tvorbě direktiv a klíčových prvků nezbytných při vývoji.

## ReactJS

Na problémy zmíněné v předchozím odstavci výborně odpovídá ReactJS s architektonickým vzorem Flux. Jedná se o relativně novou technologie pocházející z dílny Facebook. Nejen Faceboo, ale i Instagram přešli na tuto technologii s relativně rychlým přepsáním. Díky jednoduchosti rychle sklouzává do obliby a povědomí dalších velkých firem. Jeho architektura poskytuje vysoký výkon, který je demonstrován například na staré hře Wolfenstein 3D. ReactJS skládá obrazovku z mnoha bloků s obrázky, generových jádrem Meteor a za běhu vyměňuje pouze potřebné segmenty. Další praktickou ukázkou může být desktopový textový editor Atom, který běží v prohlížečovém enginu, poskytující stejné funkce jako populární editor Sublime text.

Nesporná výhoda je jeho jednoduchost a krátká učící křivka. Ačkoli se jedná o koncept poskytující možnosti složitých js frameworků, jeho implementace je přímočará a přehledná. Tvorba komponent se stává základním stavebním kamenem, což přispívá k vysoké modularitě a znovupoužitelnosti. Pro mé potřeby využívám nadstavbovou syntaxi JSX, umožňující psaní HTML tagů přímo v JavaScriptu, zajišťující srozumitelnost a čitelnost kódu. Po naučení několika málo funkcí a principů lze velice rychle tvořit komponenty, jejichž možnosti vysoce převyšují JSTL, kde lze jen velmi krkolomným způsobem vytvořit to samé co v ReactJS. Pokud navíc máme vysoce dynamickou aplikaci, kde je téměř každý element interaktivní nebo generován na základě modelu, zjistíme, že provázanost s JS stranou majoritním způsobem ovlivňuje výslednou stránku a statické tagy jsou zastíněny business logikou napsanou v JS. V takovémto případě dává smysl sjednotit oba koncepty do jedné roviny.

Základem ReactJS je využití virtuálního DOM, který tvoří jakousi nadstavbu nad skutečným DOM. Jedná se o pouhé JavaScript objekty, které si drží stav a referenci vůči skutečnému DOM. To nám mimo jiné překlenuje rozdíly napříč různými prohlížeči a jejich verzemi. Implementace virtuálního DOM odlaďuje některé chyby v event systému, které jsou známy ve starších prohlížečích. Hlavní výhoda je však ukryta v propagaci změn a aktualizaci pouze nezbytné změněné části, jelikož jakákoliv úprava skutečného DOM je v tomto ohledu velice náročná.

V každé render fázi je heuristicky spočítáno, jaké nejmenší kroky musí být provedeny, aby se zpropagoval aktuální stav vykreslením do DOM. Logika vyhodnotí všechny uzly a hierarchicky níže postavené komponenty a překreslí potřebné části DOM. Tento úkon je proveden pouze jednou, díky virtuálnímu DOM, zjišťující všechny nezbytné kroky pro finální stav. Mechanismus se nazývá Reconciliation neboli diff algoritmus. Minimum kroků potřebných pro transformaci jednoho stromu nodů je problém o složitosti O(n^3) kde n je počet nodů ve stromu. Při 1000 nodů vykreslených ve stromu je zapotřebí miliarda porovnání pro finální transformaci. Takové množství operací nejsou nynější běžné procesory schopné zvládnout pod 1 vteřinu. React implementoval heuristicky založený algoritmus o složitosti O(n), který vychází z následujících předpokladů:

1. Komponenty stejného typu generují podobné stromy a komponenty různých typů generují naprosto odlišné (<span> je různý od <div>, <DataTable> je různý od <Header>).
2. Je možné poskytnout unikátní klíč pro elementy stabilní napříč překreslením.

Díky těmto předpokladům je docíleno rapidního zrychlení. Porovnávací algoritmus u elementů ověří, zdali jsou stejného typu a pokud ne, odstraní starý prvek a vloží nový. Pokud jsou stejného typu, zkoumá odlišnosti ve vnitřní struktuře a nahradí tak například pouze atribut class místo překreslení celého stromu potomků. V případě React komponent, které jsou stavové, máme k dispozici events component[Will/Did]ReceiveProps, které jsou při překreslování volány a je již na nás jakým způsobem budeme interagovat s nově přijímanými daty. Ačkoliv se nejedná o stoprocentně kompatibilní algoritmus aplikovatelný na všechny případy, výrazně urychluje většinu změnových operací nad DOM. V případě nedostatečného výkonu, knihovna nabízí další nástroje a mechanismy pro jeho optimalizaci.

## Flux

Flux je architektonický vzor, jenž ve svém kódu využívá Facebook. Nejedná se o knihovnu, ale o pouhý přístup jakým způsobem psát klientský kód, tak aby byl přehledný a dobře strukturovaný. Stojí na principu Unidirectional flow, které těží z jednotného postupu akcí.

Všechny akce vznikají ve View nebo na serveru, ať už se jedná o uživatelskou interakci, počáteční inicializaci či jinou událost vyvolávající změnu. Při jakékoliv akci přesahující platnost komponenty volají vrstvu Actions. Ta představuje pomocnou vrstvu pro vydefinování všech dostupných akcí. Zde specifikujeme parametry, typ a unikátní označení konstantou v kontextu komponenty application dispatcher.

|  |
| --- |
| registerNotes: **function**(notes) {  AppDispatcher.handleViewAction({  actionType: FilterConstants.REGISTER\_NOTES,  notes: notes  });  } |

Dispatcher se stará o delegování všech akcí na příslušné callbacky zaregistrované ve Store vrstvě. V případě, že využíváme více komponent typu Store, dispatcher deleguje akci na všechny takové komponenty. Obslužná činnost bude zavolána, pouze pokud je zaregistrována prostřednictvím switch konstruktu rozlišující dle typu akce (unikátní konstanta). Dispatcher také umožňuje čekání na vykonání jiné akce. V některých případech potřebujeme zavolat akci teprve, až po dokončení jiné například ve více různých Store.

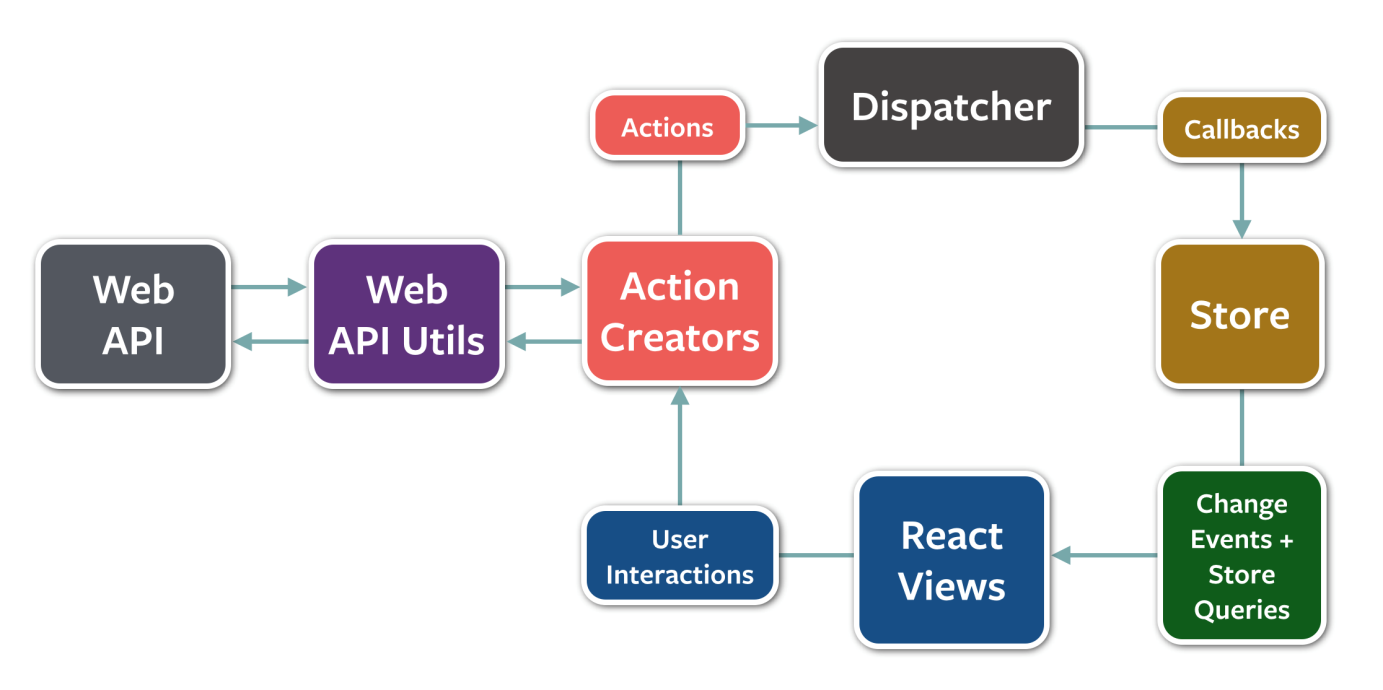
Store je jakési úložiště dat a zároveň vrstva operující nad těmito daty. Komunikuje se serverem a zajišťuje nám konzistenci a jednotný obraz dat v celé aplikaci. Po vykonání callbacku je odeslána změna a komponenty, poslouchající na danou událost jsou překresleny s aktuálními daty. Do Store vede cesta vždy přes Dispatcher respektive přes Actions. Není jiná možnost jak změnit data v této singleton struktuře. Naopak získávání dat ze Store je umožněno pomocí veřejných metod odkazující na privátní proměnné uvnitř JS souboru, na něž si drží referenci pouze objekt Store. Čerstvá data zpravidla konzumuje controller-view což je hierarchicky nadřazená komponenta poslouchající na globální change event, na jehož pokyn aktualizuje svůj vnitřní stav a překreslí všechny hierarchicky níže postavené komponenty s aktuálními daty předávanými skrze props (atributy tagu).

|  |
| --- |
| **function** getNotesState() {  **return** {notes: FilterStore.getNotes()};  }  **var** FilterContainer = React.createClass({  getInitialState: **function**() {  **return** getNotesState();  },  componentDidMount: **function**() {  FilterStore.addChangeListener(**this**.\_onChange);  },  componentWillUnmount: **function**() {  FilterStore.removeChangeListener(**this**.\_onChange);  },  render: **function**(){  **return** (<DataGrid notes={**this**.state.notes} />);  },  \_onChange: **function**() {  **this**.setState(getNotesState());  }  });  ..  //Nadřazená komponenta zavolá vykreslení controller view do příslušného DOM elementu  React.renderComponent(<FilterContainer />, document.querySelector('#filter-container')); |

|  |
| --- |
| //Privátní část - definice modelu, interních metod  **var** \_notes = {};  //Veřejný interface pro konzumaci modelu  **var** FilterStore = merge(EventEmitter.prototype, {  getNotes: **function**(){  **return** \_note;  },  emitChange: **function**() {  **this**.emit(GlobalConstants.CHANGE\_EVENT);  },  addChangeListener: **function**(callback) {  **this**.on(GlobalConstants.CHANGE\_EVENT, callback);  },  removeChangeListener: **function**(callback) {  **this**.removeListener(GlobalConstants.CHANGE\_EVENT, callback);  }  });  //Registrace obslužných metod pro změnu modelu  AppDispatcher.register(**function**(payload) {  **var** action = payload.action;  **switch**(action.actionType) {  **case** FilterConstants.REGISTER\_NOTES:  \_notes = action.notes;  **break**;  **default**:  **return** **true**;  }  FilterStore.emitChange();  **return** **true**;  }); |

Tento princip nám elegantně zpřehledňuje všechny události, které se v aplikaci dějí a je velice snadné v takovéto struktuře odlaďovat chyby a škálovat do větších celků. Dále se tímto vyvarujeme nepředvídatelnému sledů akcí a jejich provázanosti, které nám v případě two-data binding systému způsobují nemalé starosti.

Tento vzor jsem shledal velice vhodným v jakékoliv aplikaci pracující s modelem v klientské vrstvě. V portletové aplikaci má každý portlet svoji vlastní Flux architekturu, čímž jsou dokonale odděleny a jsou vůči sobě nezávislé. Mohou se rozvíjet do větších rozměrů, aniž by se z nich stal těžko udržovatelný kolos.

****

# Architektura aplikace Evrem

Před započetím vývoje je nutné položit si otázky, zda tvoříme portlety pro Marketplace tedy širokou veřejnost, na kolik budeme využívat zabudované funkce portálu a hlavně jak budou vypadat a co mají umět. Je důležité ujasnit si vyhlídky do budoucna, jelikož je i Liferay každým rokem technologicky dál a dál. Snažit se využívat pouze technologie, které nyní využívá stabilní verze, by nás mohlo omezovat ve využití plného potenciálu všech dostupných frameworků na trhu a za rok bychom mohli zjistit, že je nová verze Liferay již využívá a my bychom zůstali uzamčeni se zastaralými technologiemi, jelikož migrace našeho projektu na jiné by pravděpodobně byla již nemožná. Pokud víme, že je pro nás některá knihovna zcela klíčová a potřebujeme sledovat a pravidelně aktualizovat její verze, je na místě použít oficiální zdroj, který je možné kdykoliv vyměnit či aktualizovat.

Příkladem může být nevyužití SASS a Compass, jež v aktuální verzi Liferay nepodporují mnoho funkcí oproti oficiální verzi. Dalším prvkem může být zvolení serverových knihoven jako Spring web mvc, usnadňující práci s requesty/responsy v prostředí portálového kontejneru. Použití populárních controllerů, kde lze snadno mapovat, requesty a parametry prostřednictvím anotací, ale i využít aplikační kontext pro inicializaci bean a dalších prostředků. Počáteční čas pro nastavení a pochopení při integraci těchto nástrojů může být delší, jelikož chybí oficiální dokumentace, avšak v dlouhodobém horizontu, výhody silně převyšují nevýhody setrvání s výchozími nástroji.

## Frontend

V dnešní době již prakticky nenalezneme moderní aplikaci bez JavaScriptu. Není proto divu, že velká část aplikace je napsána právě v něm. JSP tak využíváme pouze při render fázi, na inicializaci počátečních dat do příslušného Store, kam patří i URL generované přes taglib <portlet:resourceURL id=*"saveCoordinates"* /> .

Toto je v této aplikaci jediný účel JSP, ostatní funkce a tělo portletu je vygenerováno přes ReactJS do prázdného kontejneru ve view. Každý portlet má zpravidla pouze jedno view (view.jsp) a nejdůležitější soubor \*bundle.js. Tento JS soubor obsahuje veškerou logiku a šablonu daného portletu napsané v ReactJS. Toto odstínění od Liferay způsobu psaní, nám umožňuje vysokou variabilitu vzhledu, interaktivity, ale i propojení s moderními frontendovými knihovnami. Psaní logiky přes scriptlety v JSP neshledávám vhodné v aplikaci vykazující vysokou interaktivitu a dynamiku. Tento princip není u moderních aplikací považován za „best practice“, ačkoliv pro využití Liferay funkcionalit bývá velmi často nezbytným.

Pro účely aplikace, jejíž převážná část vyžaduje propojení s JavaScriptem je na místě použít technologii, která nepoužívá oddělenou šablonu od JS, nýbrž prolíná oba koncepty dohromady. Kód níže zobrazuje počáteční inicializaci portletu, kdy v okamžiku kdy je portlet načten a připraven, zaregistrujeme data do Store a vykreslíme hlavní komponentu, obsahující samotné view portletu.

|  |
| --- |
| /\*\* **@jsx** React.DOM \*/  **var** UpcomingActions = require('./actions/upcoming-actions');  **var** UpcomingContainer = require('./components/upcoming-container.jsx');  Liferay.Portlet.ready(  **function**(portletId, node) {  **if**(portletId.indexOf('upcomingportlet\_WAR\_upcomingportlet') !== -1){  **var** initialData = JSON.parse(jQuery('#data-upcoming').text());  UpcomingActions.registerUrls(initialData.urls);  UpcomingActions.registerNotes(initialData.notes);    React.renderComponent(<UpcomingContainer />,  document.querySelector('#upcoming-container'));  }  }  ); |

Jak již bylo zmíněno u architektury Flux, všechny akce prochází od View, Actions přes Dispatcher až do Store, kde v případě serverové akce je zaslán AJAX request, který je odchycen Spring web MVC controllerem, vracejícím response. Cílová URL adresa je použita z počáteční inicializace, tedy adresa obsahující všechny náležitosti potřebné k identifikaci portletu, metody, informaci o cachování, layoutu či módu portletu (VIEW, EDIT, HELP).

http://localhost:8080/web/evrem/login?**p\_auth**=s6JWEcZf&**p\_p\_id**=loginportlet\_WAR\_loginportlet&**p\_p\_lifecycle**=1&**p\_p\_state**=normal&**p\_p\_mode**=view&**p\_p\_col\_id**=column-1&**p\_p\_col\_count**=1

Portlety sdílejí společné JS knihovny a CSS styly prostřednictvím Theme. Základní proměnné obsahující reference na knihovny, jsou přístupné z globálního window scope, čímž je snížena velikost výsledných JS balíčků. Závislosti se neduplikují a bundle.js obsahuje pouze kód související s portletem.

Ačkoliv Liferay umožňuje použití SASS a Compass vestavěném přímo v Liferay IDE, využívám vlastní strukturu a kompilátor, který umožňuje použití nejnovějších funkcí v CSS. Toto odstínění nám umožňuje využívat vlastní adresář pro všechny frontendové soubory, které musejí projít zpracováním a teprve poté jsou rozkopírovány do příslušných umístění.

V případě CSS je veškerý vlastní kód zkompilován do souboru custom.css, což je ve výchozím nastavení jediný soubor, kam by měly být vkládány uživatelské styly. Do theme jsou balíky zkopírovány po provedení kompilace, minifikace a sloučení do jednoho souboru. Tyto úkony jsou zajišťovány pomocí správně nakonfigurovaného task managera Gulp, běžícího na Node.js.

Společně s Webpack, jehož velkou výhodou je integrovaný CommonsJS jsou obstarávány balíky pro JS část. Lze ho nastavit pro práci s různými typy souborů, čemuž se říká loadery. Pomocí výrazu řekneme, že soubor s danou koncovkou, má být předkompilován příslušným loaderem a teprve poté jsou zpracovány závislosti, které jsou v kódu využity a pokud se nejedná o globální závislosti speciálně vydefinované jako externals, jsou přibaleny do výsledného balíku a jsou pouze dostupné v daném rozsahu. Ve výsledku máme čistější globální scope s oddělenými prostory.

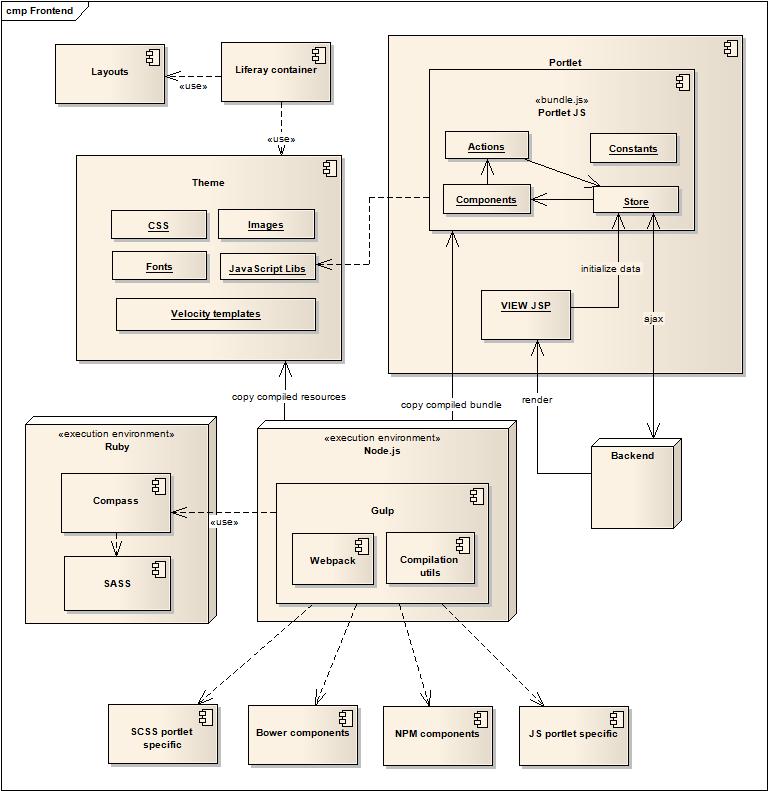
Gulp nám dále zpracovává CSS styly respektive nadstavbu SASS, která nám umožňuje dynamiku při psaní stylů, znovu použitelnost, definování proměnných, jednoduché funkce, ale i hierarchické zanořování. Ruku v ruce s ním jde Compass, což je funkční knihovna, pro psaní cross browser kompatibilních stylů. Nemusíme se tak starat o psaní moderních stylů pro všechna prohlížečová jádra. Jednoduše použijeme Compass funkci nebo mixin, který po kompilaci vygeneruje všechny styly pro dostupné prohlížečová jádra. Použití tohoto nástroje je podmíněno nainstalováním běhového prostředí Ruby s gemy Compass a Sass. Ukázkový kód níže znázorňuje dědičnost, použití funkcí, vlastních mixinů či proměnných.

|  |
| --- |
| *#isdone-input*{  @extend .main-input;  @include user-select(none);  @include hover-transition;  &*.isdone-checked*{  background-color: $*greenColor*;  }  &*.isdone-unchecked*{  background-color: $*rudeColor*;  }  } |
|  |

Gulp se nám v neposlední řadě stará o správu závislostí z Bower a NPM. Tyto populární package managery nám poskytují standardní způsob balíčkování knihoven, což je výhodné pro jednotné použití. Bower se liší od NPM hierarchií závislostí, která je plochá a optimalizovaná pro vysoký výkon, zatímco NPM má závislosti zanořené a může obsahovat několik verzí knihoven. Jak již z krátkého popisu vyplývá, oba dva jsou profilovány pro jiný účel, ačkoliv je mnoho modulů dostupných v obou repositářích.

V Theme využíváme šablonovací systém Apache Velocity, kde pracujeme s výchozí strukturu zděděnou z nadřazeného theme. Šablony tvoří kompletní strukturu webové stránky od hlavního HTML tagu až po kontejnery pro jednotlivé portlety. Nachází se zde mnoho předdefinovaných fragmentů, které lze změnit pouze za pomocí pluginu Hook nebo Ext. V aplikaci využíváme pouze Hook pro drobné modifikace původního kódu jako je například JSP s licenčními podmínkami. V šablonách je nutné být velice opatrný. Pokud smažeme nějaký zdánlivě nedůležitý prvek, můžeme se připravit o některé zabudované funkcionality, proto doporučuji co nejmenší zásahy v těchto souborech. V našich template souborech jsem se nevyhnul například úpravě navigačního panelu, o ikonu profilu s vlastní dropdown funkcí, přidání dalších knihoven do hlavičky (nová Google reCaptcha) nebo skrytí některých funkcí nepřihlášenému uživateli.

Celé schéma frontedové části je znázorněno na diagramu níže.



## Backend

Liferay nabízí podporu několik aplikační či webových serverů. Hotové balíčky jsou ke stažení na oficiálních stránkách a tak je snadné rozchodit portál za několik minut. Vývojář si může zvolit variantu dle preferencí, infrastruktury a zkušeností. Má aplikace běží na serveru Apache Tomcat vybraného dle zmíněných kritérií.

Serverová část je rozdělena do několika modulů. Pro komunikaci s frontendem využíváme portlety, které obsahují logické celky zajišťující konkrétní činnosti. Portlety obsahující controllery, které jsou namapovány na dané metody. Ve většině případů je komunikace zajišťována prostřednictvím AJAX volání typu Resource. Typický portlet je vidět níže – při render fázi (refresh obrazovky) jsou dotažena data, jež jsou vložena do modelu a je vrácena hodnota odpovídající názvu jsp. Na frontendu je vykreslena obrazovka se získanými daty z modelu. Uživatel založí novou poznámku a po stisknutí uložit je zavolána metoda typu resource, která přijme data ve formátu JSON. Deserializuje je do DTO objektu a uloží prostřednictvím servisní vrstvy. Po dokončení vrátí v response aktuální poznámku, případně chybové hlášky v serializovaném stavu JSON.

|  |
| --- |
| @Controller  @RequestMapping(value = "VIEW")  **public** **class** NewNoteController {  @RenderMapping  **public** String view(RenderRequest request, ModelMap map) {  .. getData ..  map.addAttribute("periods", JsonSerializer.toJson(periods));  map.addAttribute("colors", JsonSerializer.toJson(colors));  **return** "view";  }  @ResourceMapping("saveNote")  **public** **void** saveNote(@RequestParam("jsonNote") String jsonNote, ResourceRequest request, ResourceResponse response) {  NoteFormModel note = convertToFormModel(jsonNote);  **try** {  note = NoteLocalServiceUtil  .saveNote(note,PortalUtil.getUserId(request));  } **catch** (Exception e) {  log.error("Error during saving note", e);  }  AjaxResponse<NoteFormModel> resObject = **new** AjaxResponse<NoteFormModel>();  resObject.setPayload()  response.getWriter().write(resObject.toJson());  }  } |

Pro sdílené akce, máme vytvořen commons-function-portlet, který je staticky umístěn, na každé stránce portálu ve skryté podobě. Nastavení statického portletu provedeme pomocí property **layout.static.portlets.all** portal-ext.properties souboru, kde jako hodnotu vložíme fully qualified portlet id (portletId\_WAR\_webapplicationcontext). Tento portlet dále obsahuje časované rutiny, kde využíváme zabudované řešení v Liferay. Jednoduchým přidáním položky liferay-portlet.xml spouštíme rutinu na základě vydefinované časové prodlevy nebo cron výrazu. Jediné co musí scheduler třída splňovat je implementace rozhraní MessageListener.

|  |
| --- |
| <scheduler-entry>  <scheduler-event-listener-class>  net.evrem.portlet.commonsfunction.scheduler.NotificationScheduler  </scheduler-event-listener-class>  <trigger>  <simple>  <simple-trigger-value>5</simple-trigger-value>  <time-unit>minute</time-unit>  </simple>  </trigger>  </scheduler-entry> |

V automaticky spouštěných rutinách potřebujeme téměř vždy přistupovat k veškerým datům. Liferay neobsahuje žádnou informaci o tom, že by měl mít scheduler práva do celé servisní vrstvy potažmo ke všem datům. Takovéto nastavení je nutné provést explicitně. UserId administrátora máme uloženo v portal-ext.properties, pomocí něhož si získáme uživatele a všechna jeho oprávnění nastavíme vláknu, pod kterým rutinu spustíme.

|  |
| --- |
| @Override  **public** **void** receive(Message message) **throws** MessageListenerException {  User user = UserLocalServiceUtil.getUserById(  Long.valueOf(PropsUtil.get("evrem.admin.userid")));  PermissionChecker permissionChecker =  PermissionCheckerFactoryUtil.create(user);  PermissionThreadLocal.setPermissionChecker(permissionChecker);  runMyJob();  } |

Pro snazší přístup ke sdíleným funkcionalitám má commons-function-portlet nastaveny friendly-url. Není proto nutné generovat adresy taglibem v jsp, ale lze jednoduše připojit za aktuální adresu **/-/common/resourceId?parametry.**

Další společný modul je tvořen projektem Commons, jež poskytuje util třídy, konstanty, convertery, třídy pro emailing spolu s Velocity šablonami a třídy pro exporty dat do Excelu skrze Apache POI.

Předávání dat mezi portlety a service portletem má na starosti projekt Dtos. Zde jsou nadefinovány data transfer objects pro přenos dat mezi různými vrstvami uvnitř aplikace. Z frontendu přijmeme data ve formátu JSON, která odpovídají strukturálně form modelu. Pomocí technologie Jackson deserializujeme řetězec do form modelu a ten předáme do servisní vrstvy. Je nutné, aby projekt respektive výsledné JAR s DTOs bylo umístěno pouze ve složce **/lib/ext** v Tomcat kontejneru pro zachování jediné definice třídy ve stejném kontextu jako service portlet. Pokud by každý portlet obsahoval závislost na tomto jar pak by předávání mezi různými ClassLoadery skončilo výjimkou ClassNotFoundException. Při volání service portletu, který je uložen taktéž v **/lib/ext**, příslušný ClassLoader nehledá závislosti uvnitř nadeployovaných aplikací (portletů), nýbrž v jeho vlastním kontextu webového kontejneru.

Service portlet je vygenerované jar dle definice v service.xml a další pomocných xml v projektu service portlet service. Po spuštění service builderu jsou vygenerovány předpisové třídy pro implementaci ve zdrojovém projektu a v cílovém projektu jsou vygenerovány všechny potřebné třídy pro volání, persistenci, atd. Vygenerované metody obalují naši implementaci pomocným kódem, zajišťují low level logiku, ale i samotnou inicializaci servis. Díky tomu máme v celé aplikaci přístup k naší servisní vrstvě. K metodám ze service portletu zvenčí projektu přistupujeme přes {EntityName}LocalServiceUtil.

Servisní vrstva generována přes Service Builder je defaultně nastavena na datový zdroj LiferayPool, který obsahuje všechna data o uživatelích, layoutech, společnostech, právech, rolích a mnoho dalších. Jedná se o obrovské schéma čítající přes 180 tabulek. Z výkonnostních důvodů, ale i lepší správy a separace využívám vlastní datový zdroj. Liferay tuto možnost nezakazuje, avšak primárně je uzpůsoben pro využití jednoho databázového schématu. Zapracování jiného datového zdroje je tak ne zcela přímočará záležitost. V service portlet projektu je nutné v ext-spring.xml nadefinovat vlastního transaction managera, session factory a v našem případě jndi datasource, který je vydefinován v context.xml webového kontejneru. Transaction manager je aspektem aplikován na naše servisní třídy. Zmíněné beany je nutné zapsat ke každé entitě generované service builderem.

V příkladu níže jsou znázorněny dvě entity, první z nich je plnohodnotná doménová entita s nadefinovaným db zdrojem, session factory a transaction managerem. Druhá entita je pouze servisní vrstvou a proto u ní service builder negeneruje třídy spojené s persistencí.

|  |
| --- |
| <!-- Doménová entita se servisní vrstvou -->  <entity name=*"Note"* local-service=*"true"* remote-service=*"false"*  table=*"note"* data-source=*"evremDataSource"*  session-factory=*"evremSessionFactory"* tx-manager=*"evremTransactionManager"*>    <column name=*"noteId"* type=*"long"* primary=*"true"* db-name=*"note\_id"* />  <column name=*"text"* type=*"String"* db-name=*"text"*/>  .. ostatní sloupce ..  <finder name=*"RemindDate"* return-type=*"Collection"*  where=*"hasReminder=1 AND isDeleted=0"*>  <finder-column name=*"userId"* />  </finder>  </entity>  <!-- Pouze lokální servisní vrstva -->  <entity name=*"SupportNote"* local-service=*"true"* remote-service=*"false"*></entity> |

Service builder se mimo jiné stará o cachování entit z databáze a výsledků vyhledávání. V některých případech se nám toto nastavení nehodí. Máme možnost na pevno nastavit cachování v definici entity pomocí atributu cache-enabled nebo lze toto nastavení explicitně přepsat service-ext.properties ve formátu:

value.object.entity.cache.enabled.net.evrem.service.model.Note=false

value.object.finder.cache.enabled.net.evrem.service.model.Note=false

Zasílání emailů je realizováno přes zabudované API, poskytující zjednodušené odesílání emailu na základě konfigurace spravované přes administrátorské rozhraní. Ačkoli mnoho návodů uvádí konfiguraci přes portal-ext.properties od Liferay verze 6.2, se nastavení z properties, nahraje pouze při počátečním spuštění. Poté je již veškerá správa možná pouze přes Control panel. Email lze odeslat pomocí následujících dvou řádků kódu. První dva parametry jsou objekty z javax.mail InternetAddress, textový řetězec předmět, text emailu a rozlišení zda se jedná o html formát či strohý text. Použití je tedy velice jednoduché. Velkou nevýhodou zůstává nemožnost zasílání obrázků z důvodů chybějící podpory pro MimeMultipart.

|  |
| --- |
| MailMessage mailMessage = **new** MailMessage(from, to, subject, body, **true**);  MailServiceUtil.*sendEmail*(mailMessage); |

### Apache Maven

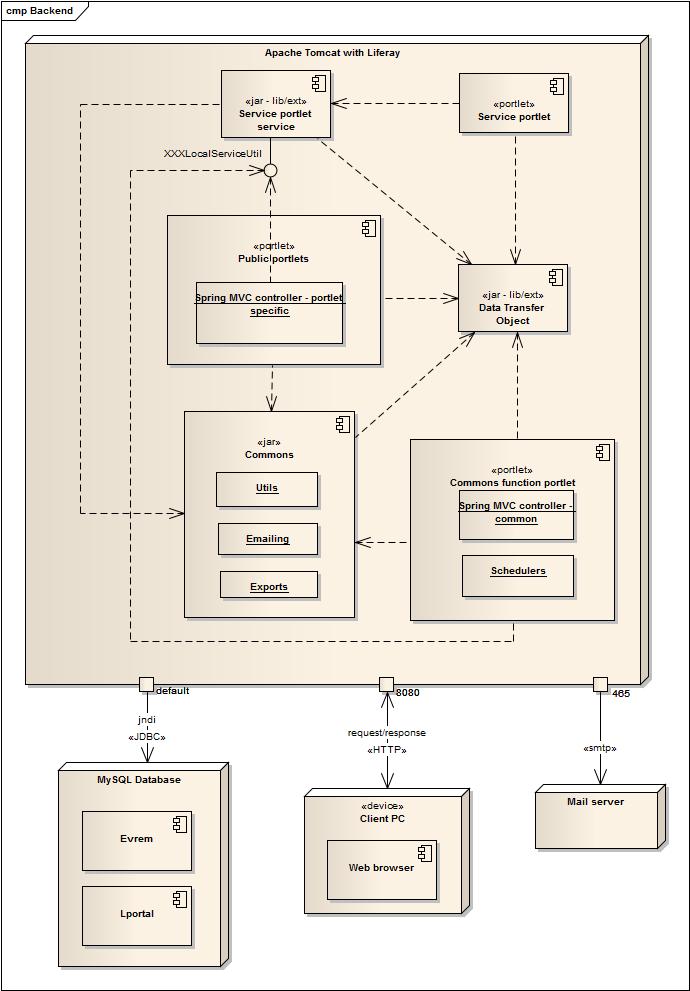
Nástroj pro správu projektu Apache Maven jsem si zvolil kvůli jeho popularitě a předchozí zkušenosti. Liferay je původně uzpůsoben pro buildovací nástroj Ant. Díky tomu je většina oficiální dokumentace psaná ve vztahu k tomuto softwaru, což občas způsobuje zavádějící informace a ve finále nefunkční návody. Začátečník naráží na mnoho problémů, při modifikaci theme, tvorbě nových plugin projektů nejsou nenalezené archetypes pro vytvoření projektu dle šablony v nastavených repositářích, chybějící závislosti, nenastavené cesty nebo nedostatek paměti.

Pro správné fungování je třeba nastavit cesty ke klíčovým složkám v adresáři staženého balíčku Liferay s vybraným webovým kontejnerem. Dále je nutné nastavit Maven repositáře, ve kterých se nachází potřebné závislosti. Tato nastavení náleží spíše k danému vývojáři než k projektu, proto jsem je zavedl do konfiguračního souboru settings.xml, který slouží jako globální nastavení napříč projekty a není tak nutné uvádět proměnné jako je lokace běhového prostředí do POM projektu.

|  |
| --- |
| <liferay.version>6.2.1</liferay.version>  <liferay.maven.plugin.version>6.2.1</liferay.maven.plugin.version>  <liferay.auto.deploy.dir>c:\liferay\deploy</liferay.auto.deploy.dir>  <liferay.app.server.dir>c:\liferay\tomcat</liferay.app.server.dir>  <liferay.app.server.deploy.dir>c:\liferay\tomcat\webapps</liferay.app.server.deploy.dir>  <liferay.app.server.lib.global.dir>c:\liferay\tomcat\lib\ext</liferay.app.server.lib.global.dir>  <liferay.app.server.portal.dir>c:\liferay\tomcat\webapps\ROOT</liferay.app.server.portal.dir> |

Téměř každý vývojář při prvním kontaktu s portálem musí zvýšit paměť pro build, což je nastaveno pomocí systémové proměnné **MAVEN\_OPTS** s adekvátním nastavením např. -Xmx1500m -XX:MaxPermSize=500m.

Schéma a rozvržení zmíněných backend modulů vidíme na diagramu níže.



## Administrátorské nastavení

Každý portlet může být na stránce obsažen pouze jednou, není však problém toto nastavení změnit. Nespornou výhodou je umístění stejného portletu variabilně na více stránek. Liferay umožňuje pouhým drag and drop vložit portlet na stránku. Tento prvek jsme využili v případě new-note-portlet, což nám umožňuje na každé stránce vytvářet novou poznámku.

# Ukázky aplikace

Obrázky..

# Zdroje

Flux architecture - <https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture>

<http://blogs.atlassian.com/2014/08/flux-architecture-step-by-step/>

<http://facebook.github.io/flux/docs/overview.html>

Yahoo anouncement - http://yahooeng.tumblr.com/post/96098168666/important-announcement-regarding-yui

AngularJS dirty checking - http://www.sitepoint.com/understanding-angulars-apply-digest/

ReactJS Diff mechanism - <http://calendar.perfplanet.com/2013/diff/>

Portal and portlets –

http://www.contentmanager.eu.com/portal.htm <https://docs.oracle.com/cd/E24705_01/doc.91/e21055/intro_to_portlets.htm>

<https://jcp.org/ja/jsr/detail?id=168>

<https://jcp.org/ja/jsr/detail?id=286>

<http://www.theserverside.com/tutorial/JSR-286-development-tutorial-An-introduction-to-portlet-programming>

<http://www.ibm.com/developerworks/websphere/library/techarticles/0803_hepper/0803_hepper.html>

<http://www.liferay.com/web/meera.success/blog/-/blogs/portlet-introduction-portlet-technology-introduction>

<http://www.theserverside.com/tutorial/JSR-286-development-tutorial-Understanding-the-PortletSession>

<http://jsr286tutorial.blogspot.cz/p/portlet-lifecycle.html>

<https://wiki.jasig.org/display/PLT/Portlet+Request+Lifecycle>

CMS - <http://cms-software-review.toptenreviews.com/>

http://www.contentmanager.eu.com/wcms.htm

Obrázek WCMS <http://www.episerver.com/web-content-management/>

http://www.cmswire.com/cms/web-cms/portals-vs-web-cms-whats-the-difference-013713.php

Srovnání

http://www.gartner.com/technology/reprints.do?id=1-22PHCII&ct=141002&st=sg

Backbase

<http://www.backbase.com/portal-software/technology>

WebSphere

http://whywebsphere.com/2013/09/26/software-costs/

http://www-01.ibm.com/software/collaboration/digitalexperience/analytics.html

<http://www-01.ibm.com/software/webservers/appserv/was/library/>

Exo Platform

http://community.exoplatform.com/portal/intranet/documentation-public

# TODO

1. Generování db schématu by nemělo být řízeno aplikací ale DBA, hibernate update property