|  |  |  |
| --- | --- | --- |
| **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 10.1.2015 | | 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 7](#_Toc408762857)

[2 Systémy pro správu obsahu – CMS 7](#_Toc408762858)

[3 Portál a portlety 7](#_Toc408762859)

[3.1 Portál 7](#_Toc408762860)

[3.2 Portletový kontejner 8](#_Toc408762861)

[3.3 Portlet 8](#_Toc408762862)

[3.4 JSR 168 9](#_Toc408762863)

[3.5 JSR 286 10](#_Toc408762864)

[4 Funkční popis aplikace 10](#_Toc408762865)

[5 Výběr JS frameworku 11](#_Toc408762866)

[5.1 ReactJS 12](#_Toc408762867)

[5.2 Flux 14](#_Toc408762868)

[6 Architektura aplikace Evrem 16](#_Toc408762869)

[6.1 Frontend 17](#_Toc408762870)

[6.2 Backend 21](#_Toc408762871)

[6.2.1 Apache Maven 25](#_Toc408762872)

[6.3 Administrátorské nastavení 28](#_Toc408762873)

[7 Ukázky aplikace 28](#_Toc408762874)

[8 Zdroje 28](#_Toc408762875)

[9 TODO 29](#_Toc408762876)

Seznam obrázků

**Nenalezena položka seznamu obrázků.**

Seznam tabulek

**Nenalezena položka seznamu obrázků.**

# Úvod

Portálové aplikace zastávají již delší dobu klíčovou roli ve především ve sféře enterprise systémů. Rozsáhlé korporace obsahují stovky malých, středních, ale i velice komplexních webových aplikací, ke kterým musí zaměstnanec denně přistupovat. Ke každé aplikaci bylo nutné zvláštní přihlášení a jejich kooperace byla řešena pomocí rozmanitých řešení. Uživatel musel přistupovat na různé adresy a pracovat na často odlišných platformách. Vývoj probíhal v různých jazycích, kde znalosti nebyly přenositelné. To mnohdy vedlo po odchodu vývojáře ke ztrátě cenného know how, jež se jen velmi těžko jeho nástupcům získávalo. Rozmach v oblasti softwaru musel být postupně korigován do větších integrovaných celků, jež poskytovaly odpovědi na společné často řešené problémy. Portletová architektura odpovídá na tyto otázky. Na trhu se nachází kvalitní řešení typu open source, ale především systémy s většinou nemalými licenčními poplatky. Téměř vždy se jedná o velice komplexní aplikace s velkou učící křivkou, z důvodů potřeby mnoha technologií a konceptů, které se v tomto segmentu využívají. Dokumentace těchto systémů pokrývá pouze základní část, a to ať už z důvodů udržení know how a nabídnutí svých expertních služeb za poplatek nebo z důvodů rychlého vývoje i těchto produktů pro udržení tempa s aktuálními trendy. 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

# Portál a portlety

## Portál

Portál je webový systém poskytující centrální bod uživateli pro přístup 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, které dané aplikace umožňují.

Zajišťuje autorizační či autentizační mechanismy jakým je například Single Sign On, integrace se sociálními sítěmi, adresářovými službami LDAP, openID nebo OAuth. Poskytují kompletní správu uživatelů, rolí a jejich hierarchické uspořádání do organizací nebo logických skupin. Nechybí ani základní komponenty pro sdílení dynamického obsahu. Výsledkem portálové stránky je sloučení několika různých značkovacích fragmentů generovaných portletovými aplikacemi. Portál komunikuje s portlet kontejnerem.

## 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 persistovat 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 tasků. PortletConfig obsahuje statické informace specifikované v deployment descriptor.
* **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, jejíž 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() pro AJAX komunikaci volanou javascriptem na klientovi

## 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. Na základě toho v Java Community Process (JCP) vznikl požadavek 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 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
* Definuje strukturu pro nasazení do prostředí webového kontejneru
* Umožňuje portabilitu portletů napříč portály dodržující JSR 168
* Agreguje portlety do portálové aplikace ve formě WAR
* Definuje popisný soubor portlet.xml rozšiřující standardní deployment descriptor web.xml u servletových aplikací

### JSR 286

Po několika letech uvedení do provozu první standardu, bylo otestováno a zjištěno mnoho nedostatku. Jelikož neexistoval žádný standard, kterého by se dodavatelé portálových řešení měli držet, tvořili svá vlastní řešení, kterým kompenzovali nedostatky ve standardu. Toto konání způsobilo opět ztrátu portability, ze které standard 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 mezi portlety
* Poskytnutí zdrojů (souborů) pomocí metody serverResource
* Podpora AJAX požadavků
* Rozšíření oblasti působnosti o přístup ke cookies, hlavičky dokumentu nebo jiného obsahu vně portletu
* Zapracování další 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 a funkce portletů pro naše potřeby
  + Filtry – pre a post akce vykonané 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

# 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 aplikace. V této aplikaci jsou tyto funkce zdarma, pokud by se uživatel rozhodl stáhnout svá data a aplikace dále nepoužívat, provede export jednoduše do formátu \*.xlsx tedy Excel. 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 používat 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 v celé své klientské vrstvě využívá frameworku YUI verze 3 od Yahoo, který tvoří základnu pro Alloy UI. Koncem srpna 2014 Yahoo oznámilo 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 taskovacích 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 existovat, aby splňovala konkurenceschopnost a byla 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 clienta, 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 a 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 a není tedy nutné využití dalších nástrojů jako je CommonsJS pro vymezení oddělených scope. Angular však také nabízí velice účinnou, ale zároveň nebezpečnou zbraň, která se prolíná 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 zcela nesporné a jde zejména o rychlou reakci na konkrétní změny zadanou funkcí. Problém nastává v okamžiku, kdy se ve scope nachází desítky watchers, kdy v callbacku jsou měněny hodnoty, na kterých jsou opět navěšeny watchers. 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 scopes 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 watchers 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 dalších 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, na němž také běží a rychle sklouzává do obliby a povědomí dalších velkých firem. Díky jeho architektuře poskytující vysoký výkon, který je demonstrován například na staré hře Wolfenstein 3D, který je jako html stránka vykreslována přes ReactJS skládající se z mnoha divů s obrázky, generových jádrem Meteor. 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 tak stává základním stavebním kamenem, což přispívá k vysoké modularitě a znovupoužitelnosti. Pro naše potřeby využíváme nadstavbovou syntaxi JSX, umožňující psaní html tagů přímo v javascriptu, což opět usnadňuje 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 je generován na základě modelu, zjistíme, že se z naší JSP template stává více javascript než značkovací jazyk poskytující HTML kostru aplikace s vykreslování modelových dat.

Základem ReactJS je využití virtuálního DOM, který tvoří jakousi nadstavbu nad skutečným DOM. Jedná se o pouhé javascriptové 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 v aktualizaci pouze nezbytné části, která se skutečně změnila, jelikož jakákoliv úprava skutečného DOM je velice náročná záležitost a tak je nutné, aby zásahy byly co nejmenší.

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 jsou překresleny potřebné části DOM. Toto překreslení je však provedeno pouze jednou, díky virtuálnímu DOM, kde jsou provedeny všechny nezbytné kroky pro zjištění finálního stavu. Mechanismus se nazývá Reconciliation používající 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 by bylo zapotřebí miliarda porovnání pro finální transformaci. Takové množství operací nejsou nynější běžné procesory schopné zvládnout pod sekundu. 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. Ve finále je na nás, jak aplikaci napíšeme a 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 aplikačního dispatcheru.

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

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

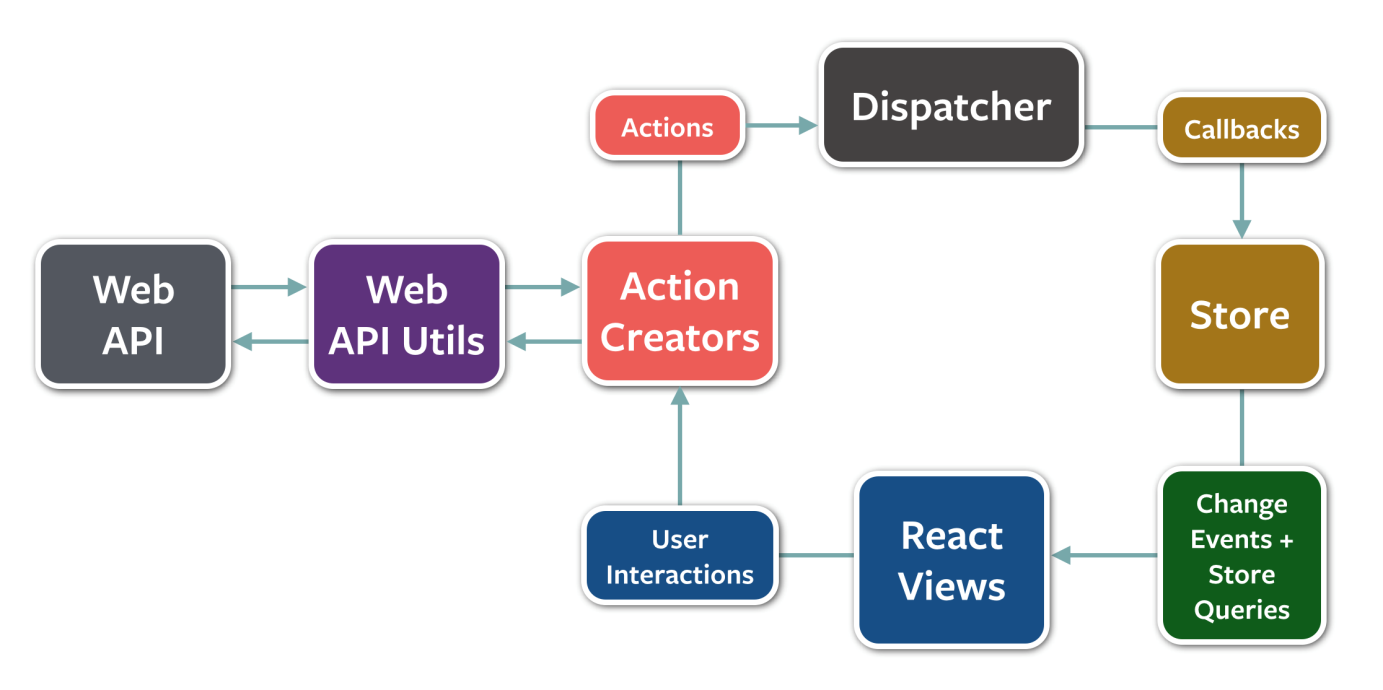
Store je jakési úložiště dat a zároveň vrstva operující nad těmito daty či komunikující se serverem, což nám zajišťuje konzistenci a jednotný obraz dat v celé aplikaci. Po vykonání callbacku je emitnuta změna a komponenty, poslouchající na danou událost jsou překresleny s aktuálními daty. Do Store vede 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 get metod odkazující na privátní proměnné uvnitř js souboru 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 zpravidla 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-databinding 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 portlety 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

Je nutné si položit otázky, zda tvořím portlety pro Marketplace tedy širokou veřejnost, na kolik budu využívat zabudované funkce portálu a hlavně jak budou vypadat a co mají umět. V neposlední řadě 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 nová verze Liferay již tyto technologie využívá a nám by ujel vlak, jelikož migrace našeho projektu na jinou technologii by pravděpodobně byla již téměř 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, které v aktuální verzi nepodporují mnoho funkcí. Dalším prvkem může být zvolení serverových knihoven jako Spring web mvc, které nám velice usnadňuje práci s requesty/responsy v prostředí portálového kontejneru. Umožňuje nám použití populárních controllerů, kde lze snadno mapovat, requesty a parametry prostřednictvím anotací, ale i využit 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 příslušných Stores daných portletů, kam patří i URL generované přes taglib <portlet:resourceURL id=*"saveCoordinates"* /> .

Toto je 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 javascriptový soubor obsahuje veškerou logiku a template 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, jejiž 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 balíčků pro každý portlet. 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 jsou vkládány uživatelské styly. Do theme jsou balíky kopírovány 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 webpackem, jehož velkou výhodou je integrovaný CommonsJS jsou obstarávány balíky pro JS část. Webpack lze nastavit pro práci s různými typy souborů, čemuž se říká loadery. Pomocí expression 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.

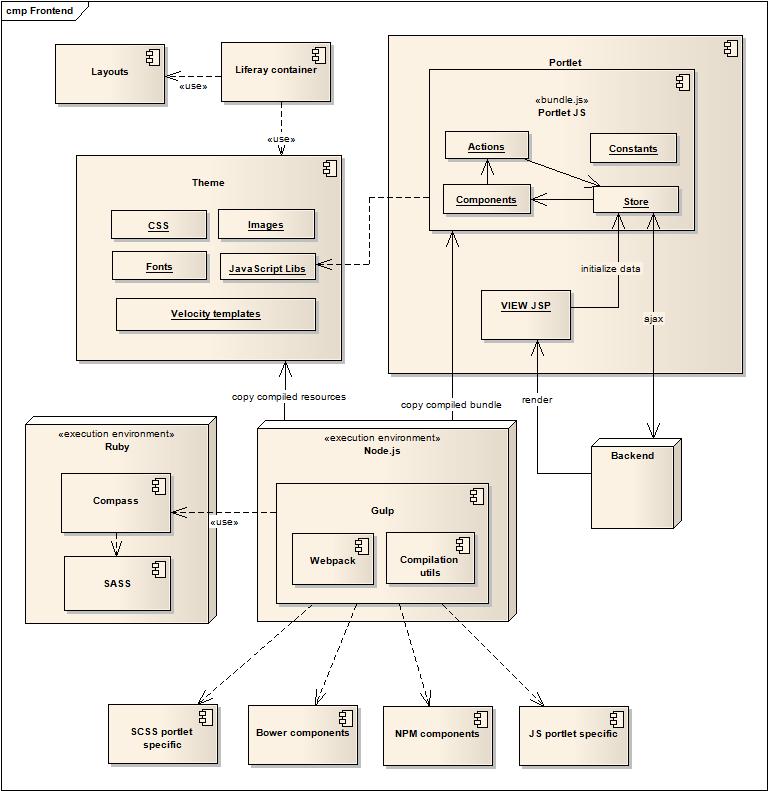
Gulp nám dále zpracovává CSS styly respektive nadstavbu SASS, která nám umožňuje dynamiku při psaní stylů, znovu použitelnost, definování proměnných, jednoduché funkce, ale i hierarchické zanořování. Ruku v ruce se SASS jde Compass což je funkční knihovna, pro psaní cross browser kompatibilních stylů. Nemusíme se tak starat o psaní moderních stylů pro všechna prohlížečová jádra, ale 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 lze vidět níže, znázorňující 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 hierarchí 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. Zde využíváme výchozí strukturu zděděnou z výchozího 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

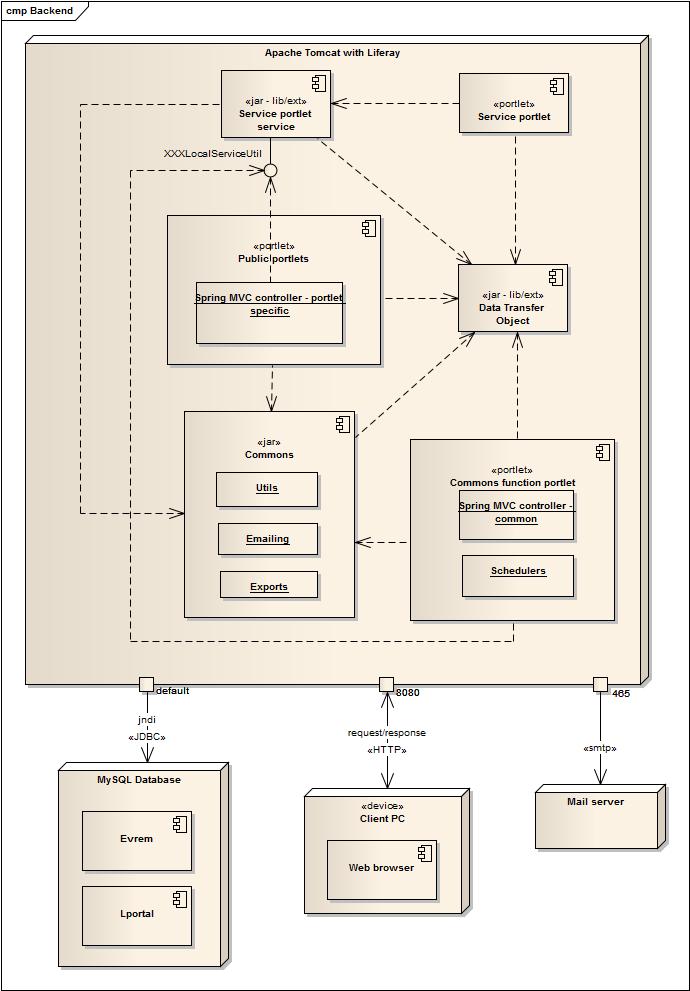
Nástroj pro správu projektu Apache Maven jsem si zvolil kvůli jeho popularitě a předchozí zkušenosti. Liferay je původně uzpůsoben pro buildovací nástroj Ant. Díky tomu je většina oficiální dokumentace psaná ve vztahu k tomuto softwaru, což občas způsobuje zavádějící informace a ve finále nefunkční návody. 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 - <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

# TODO

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