**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 7.2.2015 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

# Seznam obrázků



# Seznam tabulek

# 1 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.

# 2 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.


Moderní web content management systémy

Převzato z http://www.episerver.com/web-content-management/

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.

# 3 Portál a portlety

## 3.1 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.

## 3.2 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.

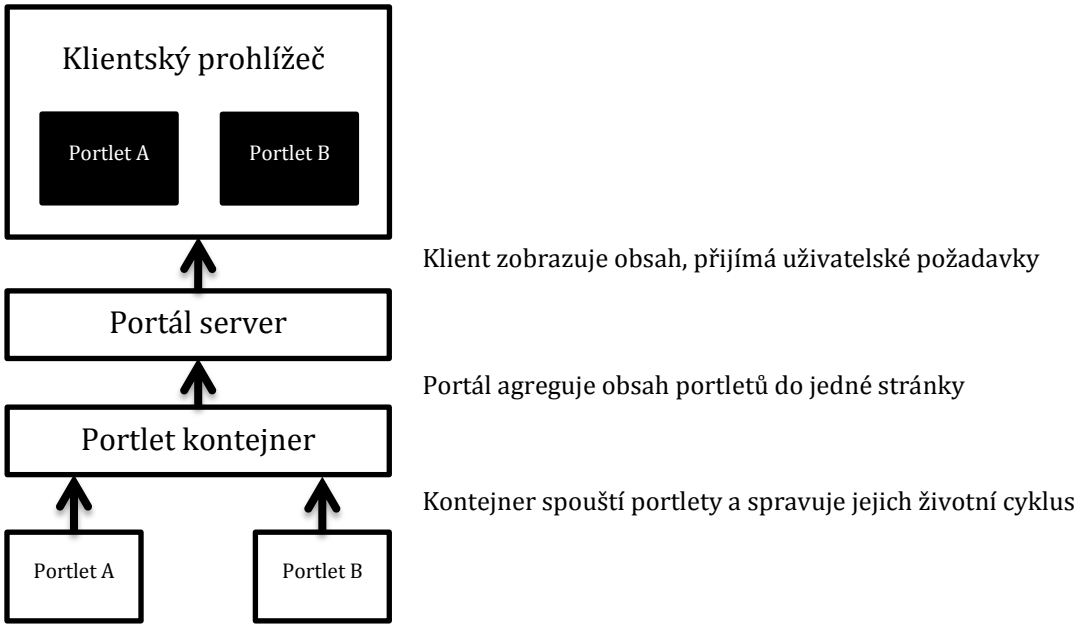



Diagram spolupráce portálu a portletů

## 3.3 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.

### 3.3.1 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.

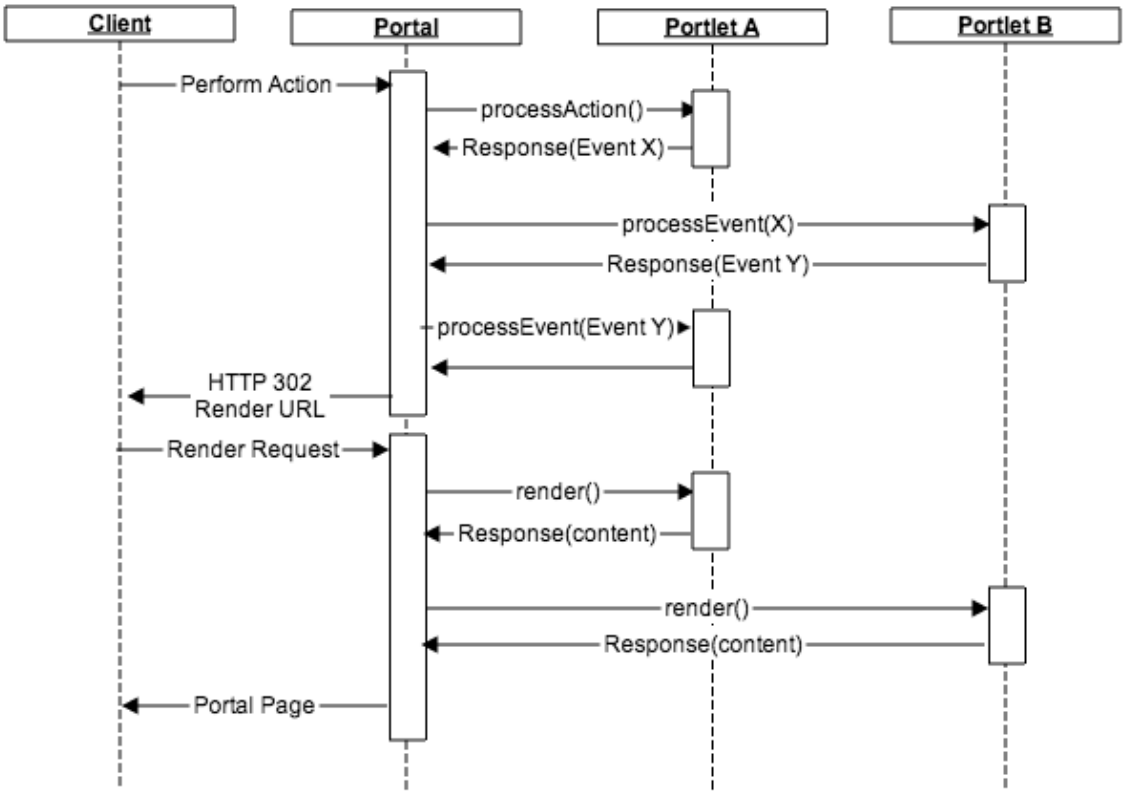
Rozšiřující operace životního cyklu:

- **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.




Životní cyklus portletů

Převzato z https://wiki.jasig.org/display/PLT/Portlet+Request+Lifecycle

## 3.4 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.

### 3.4.1 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.
- Zavádí módy pro správu, nápovědu a zobrazování
- Definuje stavy pro určení rozsahu zobrazení fragmentu na stránce

### 3.4.2 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 možností rozšíření pro vlastní reakce, modifikující či obalující určité chování našimi funkcemi.
    - **Filtry** – akce vykonávané před nebo po dané události.
    - **URL Listener** pro globální modifikaci adres a parametrů.
- **Přidání parametrů** běhovému kontejneru.
- **Zpětná kompatibilita** s přechozím standardem.
- Nové možnosti **kešování** pro zvýšení výkonu.
- Přiblížení **PortletSession** servletové variantě HttpSession umožňující pamatování uživatelského stavu po dobu prohlížení.

# 4 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.


Srovnání horizontálních portálů

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

## 4.1 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.

## 4.2 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.

## 4.3 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.

# 5 Liferay portál

Technologie ve světě programování se každým okamžikem mění. Neustále probíhá vývoj nových funkcionalit a frameworky jsou transformovány dle aktuálních trendů. U portálu toto platí dvojnásob. Jejich široká působnost vyžaduje zapojení mnoha knihoven, systémů nebo principů. Není snadné integrovat stále nejnovější verze technologií současně s moderními trendy. Liferay poskytuje mimo open source také enterprise edici, díky které musí být ve svém release cyklu mnohem opatrnější, jelikož si nemůže dovolit ztrátu kompatibility pro své platící zákazníky. Tento fakt často způsobuje nutnost využití starších frameworků a uzamknutí se v historických verzích. V této práci se dočtete o současně stabilní verzi 6.2, ačkoliv je verze 7 již na obzoru (milestone), přinášející například některé kýžené změny v oblasti frontendu.

## *5.1 Architektura*

Platforma Liferay implementuje a podporuje řadu standardů a technologií, kterými se tato kapitola bude zabývat. Integrace mnoha klíčových enterprise nástrojů díky tomu bývá snadná a vývojář se může soustředit na tvorbu business logiky.

Základním předpokladem funkční instalace je operační systém, jenž je možné využít. Seznam v aktuální verzi čítá 19 položek, kde nechybí populární Windows, Unixový systém Mac, Solaris nebo Linuxový Ubuntu či Red Hat v kombinaci s nainstalovaným běhovým prostředím Java Virtual Machine – JDK od Oracle nebo IBM.

Instanci Liferay portálu lze provozovat na několika aplikačních serverech či jednodušších servletových kontejnerech, jež musí obsahovat několik knihoven a nastavení. Na oficiálních stránkách lze stáhnout hotové balíčky s předpřipravenými prostředími. Jedná se o servery Tomcat, Glassfish, Geronimo, JBoss, Jetty, JOnAs + Tomcat nebo Resin. Podpora dalších však není vyloučena. Liferay se snaží o maximální odstínění serverově založených technologií, jako je EJB, jež je podporováno jenom částí zmíněných kontejnerů a nahrazuje tuto sadu komponent, serverově nezávislou technologií Spring. Pomocí ní, je vybudovaná komplexní servisní vrstva, zajišťující transakce, low-level logiku portálu a spolu s ORM frameworkem Hibernate datovou vrstvu. Liferay dále využívá některé Java EE komponenty v konjunkci se serverem, mezi něž patří:

- **JNDI** (Java Naming and Directory Interface) – získávání objektů a dat prostřednictvím jména přes jednotné rozhraní (SPI)

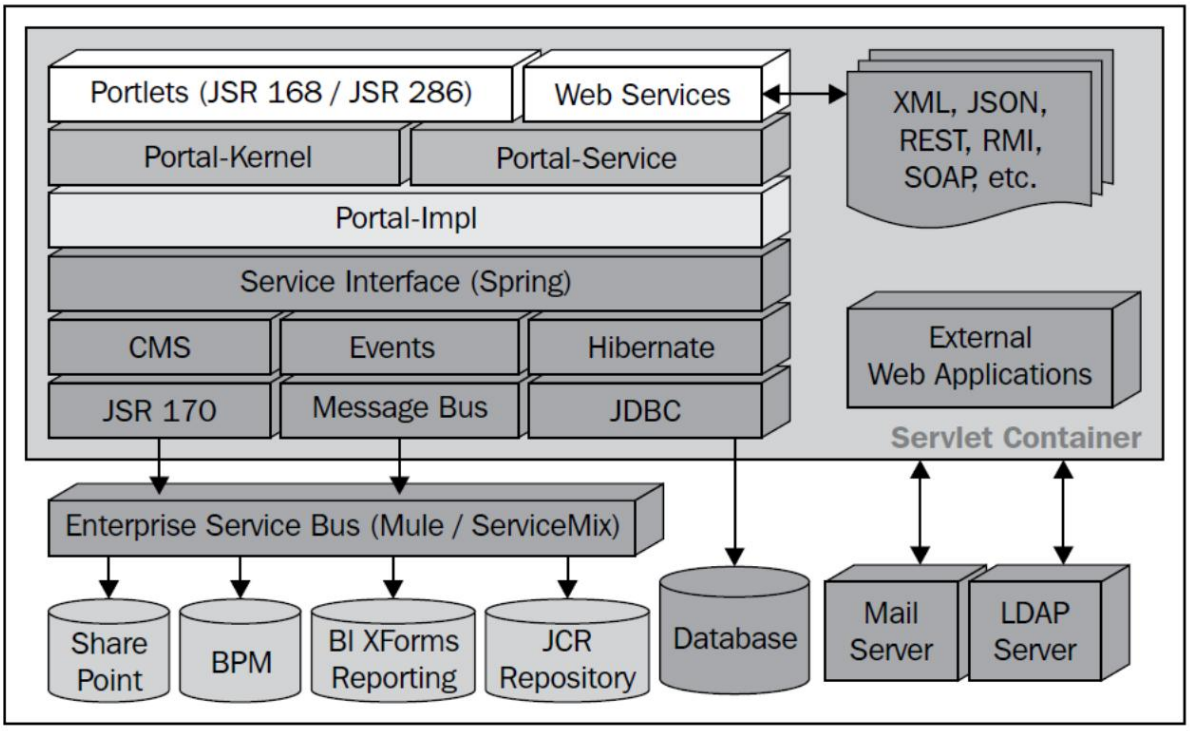
- **JDBC** (Java Database Connectivity) – standardní způsob pro práci s databází
- **JTS** (Java Transaction Service) – transakční management splňující ACID (Atomicity, Consistency, Izolation a Durability)
- **JAAS** (Java Authentication and Authorization Service) – API pro zabezpečení aplikace, jež popisuje autentizaci, autorizaci, šifrování nebo infrastrukturu s veřejných klíčů.
- **JWS** (Java Web Service) – komunikace s externími službami pomocí webových služeb
- **JSP/Servlet** (Java Server Pages) – dynamická značkovací view vrstva překládaná za běhu do Java servletů
- **JavaMail** – sada nástrojů pro emailovou komunikaci
- **JMS** (Java Messaging Services) – nepřímá výměna zpráv či naplněných objektů mezi různými klienty

Pro vyhledávání a indexaci dat lze využít Apache Solr postavený na knihovně Lucene od stejného dodavatele. Mezi jejich přednosti patří především rychlost, škálovatelnost a pokročilé funkce při fulltextovém vyhledávání s možností tvorby speciálních dotazů. Solr je samostatný vyhledávací server komunikující prostřednictvím REST API. Přijímá data různých formátů přes protokol http, jež si zaindexuje a vystaví webové služby pro vyhledávání vracející výsledky na dotaz. Obsahuje nástroje pro správu, monitorování, dolování dat, optimalizaci serveru nebo i rozšíření v podobě vyhledávání v pokročilých formátech Word či PDF.

Nasazení do korporátního světa vyžaduje leckdy komunikaci mezi různými službami, kde formát zpráv a jejich vyřizování může nabývat různých podob. Jednotný prostředkem pro komunikaci je zabudovaná ESB neboli Enterprise Service Bus vrstva, starající se o interakci s vnějšími aplikacemi. Díky ESB je možné zapojovat rozmanité systémy, aniž bychom se museli starat o jejich integraci ze strany portálu. ESB se chová jako jakási vstupně výstupní výhybka, umožňující využití jednotného API pro komunikaci s různými službami. Na výběr máme mezi Mule nebo Apache ServiceMix. Reálným příkladem je komunikace s různými Workflow systémy jako je Intalio nebo jBPM. Dále je možná integrace s portálem Microsoft SharePoint nebo propojení s JCR (Java Content Repository) Apache JackRabbit.

Dalšími prostředky pro komunikaci jsou webové služby. Liferay poskytuje servisně orientovanou architekturu (SOA), jež umožňuje zasílat zprávy v rozličných formátech.

Lze tak zasílat zprávy ve formátu XML, XML-RPC, JSON, REST, SOAP, RMI, Hessian, Burlap nebo tvořit vlastní tunely.




Architektura komponent

Převzato z knihy Liferay portal 5.2 systems development

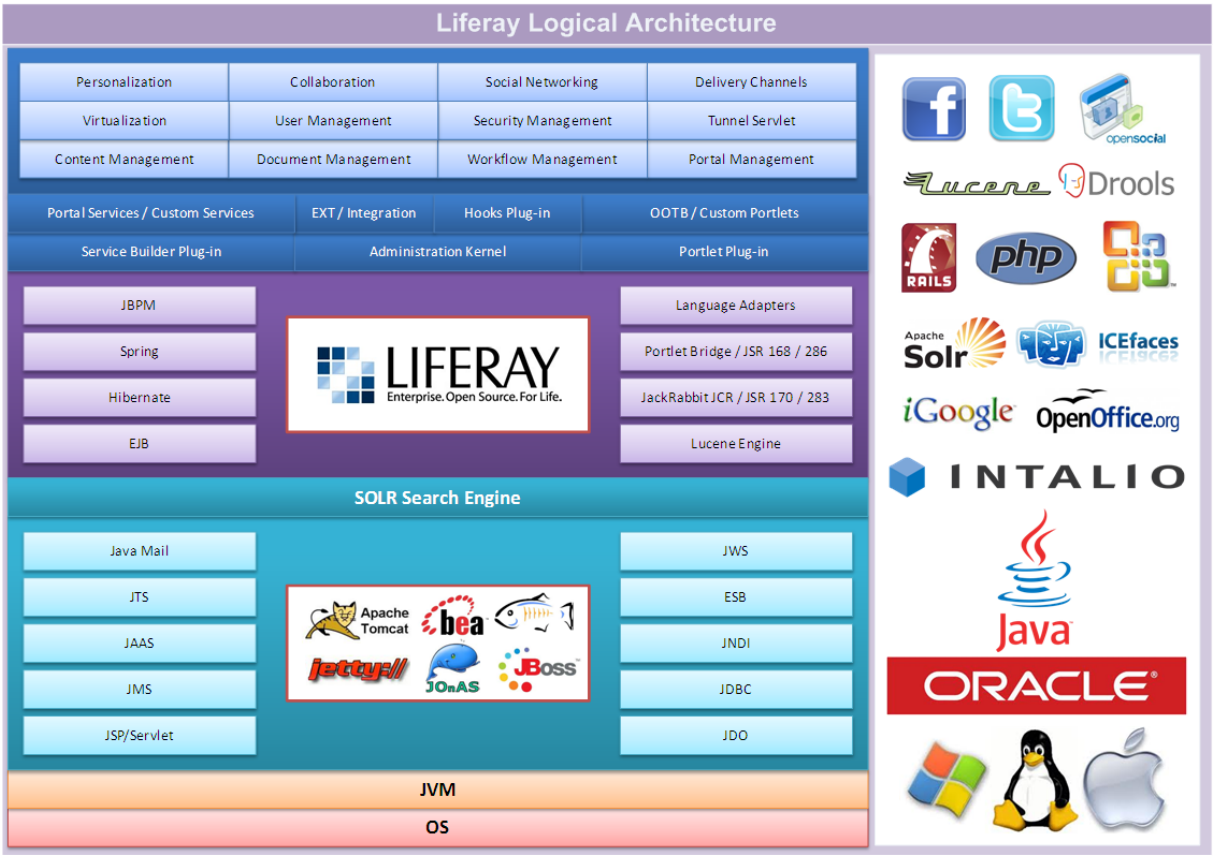
Liferay umožňuje psaní portletů také v dalších skriptovacích jazycích jako je PHP, Ruby on Rails, Python, Groovy a dalších. Díky tomu je klientská základna rapidně rozšířena a vývojáři bez znalosti Javy mohou těžit z výhod, jež portál přináší.

Zvyšování výkonu v kriticky důležitých oblastech je zajišťováno prostřednictvím klastrování, správného rozvrstvování požadavků přes load balancer, kešování či replikaci. V tomto ohledu lze využít širokou škálu zabudovaných nástrojů a nastavení.

Ve své nejvyšší vrstvě Liferay nabízí nástroje pro personalizaci, spolupráci, propojení se sociálními sítěmi, správu uživatelů, kompletní zabezpečení, tunelování (WebDAV – správa souborů přes http, vlastní servlety) a virtualizaci.

Mnoho položek nastavení je uživatelsky přístupné přes administrační rozhraní, což umožňuje rychlou reakci na změny bez nutnosti zásahu do kódu a přenasazování. Liferay integruje Content Management System propojený se standardem JSR-170 (JCR), Document Management System, ale také již zmíněné nástroje pro modelaci business proces modelů a jejich workflow.

V následujících kapitolách bude představena pro vývojáře nejdůležitější vrstva, představující tvorbu rozšíření v podobě portletů, modifikování v podobě modulu Hook, úpravu jádra přes Ext, sestavení vlastní šablony Theme nebo rozvržení obsahu do vlastního layoutu.




Technologická základna Liferay portálu

Převzato z http://www.liferay.com/community/wiki/-/wiki/1071674/Logical+Architecture

## 5.2 Integrované nástroje

## 5.3 Nastavení portálu

Portal.properties, portal-ext.properties…

Nastavení parametrů pro vývoj include-and-override.

## 5.4 Vývoj plugin modulů

Rozšíření portálu lze realizovat několika způsoby. Nejčastějším způsobem jsou portlety. Stejně esenciální bývá nutnost vlastního designu v podobě theme. Následující kapitoly popíší některé klíčové části tvorby těchto zásuvných modulů pro platformu

Liferay. Neexistuje jediný správný postup, a proto si každý vývojář musí zvolit vyhovující variantu a vzít si pouze to co ve své aplikaci potřebuje. Kapitoly se nebudou zabývat elementárními tvorbou modulů od začátku až do konce, ale budou zaměřeny na důležité, zajímavé, problematické či špatně popsané části v oficiální dokumentaci. V následujících příkladech budeme používat nástroj pro správu projektu Maven a vývojové prostředí Eclipse s integrovaným Liferay IDE.

# 6 Portlet plugin Liferay

V předchozích kapitolách jsme si vydefinovali dvě specifikace popisující vlastnosti a funkčnosti portletů. Liferay tyto specifikace implementuje a mimo standardních funkcí nabízí i řadu rozšíření, kompatibilních pouze s touto platformou. Jak již bylo zmíněno, jedná se o klasickou webovou aplikaci nasazovanou do webového kontejneru. Co však taková aplikace musí splňovat, aby se jednalo o Liferay portlet?

## *6.1 Konfigurační soubory*

Odpovědí jsou popisné a konfigurační soubory ve složce WEB-INF. Skrze ně lze celkově ovlivnit chování, vzhled, typ, název portletu a mnoho dalších. V základním projektu vygenerovaném přes archetypy máme následující soubory.

**web.xml** – standardní servletový deskriptor. Ve výchozím nastavení je tento soubor prázdný, avšak pro využití Spring portletu, je nutné, zaregistrovat zde ViewRendererServlet.

**portlet.xml** – analogie k web.xml v portletovém světě dle standardů JSR. Patří sem název, obslužná třída pro příchozí požadavky, módy, stavy, podporované MIME typy, definice veřejných parametrů a událostí pro meziportletovou komunikaci, oprávnění definované rolemi a další. V jednom souboru lze nadefinovat více než jeden portlet a celý projekt tak může sloužit pro několik aplikací zároveň.

**liferay-portlet.xml** – rozšíření předchozího souboru o Liferay specifika. Mezi některé z nich patří definice CSS či JS souborů, které lze pomocí tohoto nastavení deklarovat v hlavičce nebo patičce portálu, tudíž do umístění mimo rozsah portletu. Některé parametry slouží jako mapovací položky ze souboru portlet.xml na Liferay portál. Příkladem může být mapování rolí z portlet.xml na role uložené v portálové DB. Dále zde můžeme najít časovače spouštějící rutiny, definice URL, konfigurace sociálních, dokumentově či obsahově založených aplikací, položky pro přidání portletu do

administračního rozhraní, zdali je možné vložit více instancí na stránku, viditelnost, globální nastavení portletu nebo parametry vzhledu.

**liferay-plugin-package.properties** – soubor obsahuje popisná metadata pro Liferay katalog softwaru, závislosti na knihovny obsažené v portálu nebo kontexty, jež je nutné nastartovat před tímto samotným.

**liferay-display.xml** – zařazení a název zobrazený při přidávání portletu na stránku.

## 6.2 Spring MVC Portlet

Vyřizování požadavků je ve výchozím nastavení realizováno třídou GenericPortlet, jež pokrývá životní cyklus portletu. Pro snadnější práci existují knihovny třetích strach zajišťující snadné mapování metod, parametrů, validování a další funkce, které vývojáři usnadňují život a zpřehledňují kód. K nejvýznamnějším z nich patří JSF, Struts a aktuálně probíraný Spring MVC.

### 6.2.1 Nastavení Spring MVC portletu

Ukázkový controller (řadič) neboli vstupní bod mezi klientem a business vrstvou, je možné nalézt v kapitole 8.2 Backend. Správné fungování je podmíněno několika nastaveními.

1) Zaregistrování ViewRendererServlet do web.xml. Tento servlet slouží jako most z portletového světa do servletového. Převede požadavky do tvaru HTTPServlet a umožní tak přepoužít celou vrstvu použitou v servletech.

```
<servlet>
	<servlet-name>ViewRendererServlet</servlet-name>
	<servlet-class>
		org.springframework.web.servlet.ViewRendererServlet
	</servlet-class>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>ViewRendererServlet</servlet-name>
	<url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

Definice Spring servletu

2) Zaregistrování třídy DispatcherPortlet do portlet.xml spolu s propojením a inicializací aplikačního kontextu. Tato třída slouží jako vstupní bod, delegující požadavky na příslušné řadiče.

```
<portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
<init-param>
```

```
    <name>contextConfigLocation</name>
    <value>/WEB-INF/{portlet-name}-portlet.xml</value>
</init-param>
```

Definice Spring hlavního řadiče

3) Vytvoření aplikačního kontextu zmíněného v bodu 2. Ve složce WEB-INF založíme soubor ve tvaru {portlet-name}-portlet.xml. Název není náhodný a je nutné dodržet uvedenou konvenci malými písmeny. Každá aplikace využívající Spring, musí obsahovat, alespoň jeden kořenový kontext. Zde nadefinujeme použití anotací, rozlišování a mapování JSP dle zjednodušeného názvu a skenování balíčku s našimi třídy typu Controller.

```
<context:annotation-config />
<bean
class="org.springframework.web.portlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<bean id="jspViewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
        <property name="order" value="1" />
</bean>

<context:component-scan base-package="my.package.with.controller"/>
```

Aplikační kontext

4) Vypnutí přepojmenovávání (namespacing) parametrů v liferay-portlet.xml jednoduše přes položku **requires-namespaced-parameters** na hodnotu false.
5) Přidání závislostí na Spring knihovny. V případě Maven projektu se zavádějí do pom.xml. Potřebné knihovny **spring-** aop, aspects, beans, context, core, expression, test, webmvc-portlet a webmvc.

## 6.3 Meziportletová komunikace

Jedním hlavních z problémů řešených v počátcích portletů byl mechanismus, umožňující vzájemnou komunikaci. Modularita, jako jedna z velkých výhod portálu nemohla být bez tohoto aspektu splněna. S příchodem JSR 286 bylo na tuto otázku poskytnuto řešení v podobě veřejných render parametrů a komplexního mechanismu zasílání událostí. Méně vhodným způsobem je výměna proměnných skrze objekt PortletSession, jež lze nakonfigurovat pro sdílení dat mezi portlety či portálem.

### 6.3.1 Události

Jak již zaznělo v kapitole 3.4.2, jedná se o standardní funkcionalitu, portletových aplikací. Komunikovat lze jak na straně klienta tak serveru. Události na straně serveru je

nutné zaregistrovat do souboru portlet.xml, spolu s datovým typem, jmenným prostorem a lokálním názvem a to jak na straně odesílatele tak příjemce. Odesílatel nastaví do aktuální instance ActionResponse událost se zmíněnými parametry, čímž je odeslána událost a dané poslouchající strany, na danou událost zavolají obslužné metody. V příkladu níže je zobrazena definice odchozí a příchozí události a zaregistrování datového typu události. V případě výměny vlastních objektů je nutné, aby k dané třídě měly přístup obě strany a objekt byl serializovatelný.

```
<portlet>
	<!-- Odesílatel -->
	<supported-publishing-event>
		<qname xmlns:x="http://myapp.com/events">x:note</qname>
	</supported-publishing-event>
	<!-- Příjemce -->
	<supported-processing-event>
		<qname xmlns:x="http://myapp.com/events">x:note</qname>
	</supported-processing-event>
</portlet>
<!-- Odesílatel i příjemce -->
<event-definition>
    <qname xmlns:x="http://myapp.com/events">x:note</qname>
    <value-type>java.lang.String</value-type>
</event-definition>
```

Definice události

Vytvoření takového mostu je pouze nezbytný předpoklad, po kterém můžeme začít psát samotnou logiku odeslání události a přijetí.

```
//Odesílatel
..processAction metoda..
QName qName = new QName("http://myapp.com/events", "note", "x");
response.setEvent(qName, "Nová poznámka");
//Příjemce
@ProcessEvent(qname = "{http://myapp.com/events}note")
public void processNoteRefreshEvent(EventRequest request, EventResponse response){
	Event event = request.getEvent();
	String value = (String) event.getValue();
}
```

Odeslání události a její příjem

Komunikace prostřednictvím událostí je silnou zbraní, která by se neměla užívat bez rozmyšlení. Událostní propojování může při častém a nevhodném užití výrazně znepřehlednit kód aplikací, a proto se běžněji setkáváme s komunikací pouze na klientské vrstvě pomocí JS příkazů níže. Zde stačí jednoduše nadefinovat název a kdekoliv jinde na stránce v jiném portletu je odchycena událost pod stejným jménem a zavolán obslužný callback. Jelikož je JavaScript netypový jazyk, nemusíme se zatěžovat

s definicí třídy a rovnou lze posílat data v polích, složitých objektech nebo pouhých hodnotách.

```
//Odesílatel
Liferay.fire(eventName, data);
//Příjemce
Liferay.on(eventName, callback, [scope]);
```

Události na straně klienta

## 6.4 Více instancí portletu

V základním režimu můžeme umístit portlet na konkrétní stránku pouze jednou. V takovém případě dostane název ve tvaru **portletId_WAR_webapplicationcontext,** který je odvozen z názvu portletu dle souboru portlet.xml a názvu kontextu aplikace, pod nímž běží ve webovém kontejneru. Kdekoliv se na tento portlet odkazujeme, použijeme zmíněný identifikátor. V případě více instancí je za tento název přidán ještě unikátní identifikátor pro označení konkrétní instance **_INSTANCE_ABC123.** Výhodou je zobrazení různých dat a specifických nastavení pro dvě stejné aplikace. Nastavení provedeme v liferay-portlet.xml položkou instanceable na true.

```
ThemeDisplay themeDisplay = (ThemeDisplay) request.getAttribute(WebKeys.THEME_DISPLAY);
themeDisplay.getPortletDisplay().getInstanceId();
```

Získání ID instance v řadiči nebo JSP

## 6.5 Friendly URL

Každá adresa v portálové aplikaci obsahuje mnoho informací. Důvodem je potřeba identifikace a další funkční parametry pro určení vzhledu, pozice v layoutu či módu v jakém portlet pracuje. Ačkoliv se jedná o potřebné informace, ve většině případů nás jejich nastavení nezajímá a vystačíme si s výchozími hodnotami.

**Tabulka 1 Základní parametry v URL**

| **Parametr** | **Funkce** |
|---|---|
| p_p_id | ID portletu viz kapitola [Více instancí portletu] |
| p_p_lifecycle | Životní fáze požadavku. Na výběr je ze 3 hodnot:<br>0 – Render fáze, 1 – Action fáze, 2 – Resource URL |
| p_p_state | Stav okna portletu dle JSR-168 - normal, maximized nebo minimized. |
| p_p_mode | Mód portletu dle JSR-168 - view, edit nebo help. |

| | |
|---|---|
| p_p_col_id | ID sloupce v použitém layoutu. |
| p_p_col_pos | Vertikální pozice ve sloupci. |
| p_p_col_count | Počet sloupců v layoutu. |
| [p_p_id]_jspPage | Další parametry lze specifikovat pomocí prefixu id portletu. V tomto případě určujeme jakou JSP stránku chceme použít. |

Parametry v URL [Liferay in Action]

Adresy v portálu lze generovat několika způsoby. Mezi nejspolehlivější a nejčastější patří generování přes JSP tag (viz kapitola 8.1) či vygenerování v řadiči přes aktuální objekt response pomocí metod createResource|Action|RenderUrl(). URL lze také generovat skrze JavaScript za využití objektu AUI.

```
AUI().ready("liferay-portlet-url", function(a) {
    var resourceUrl = Liferay.PortletURL.createResourceURL();
    resourceUrl.setPortletId("newnoteportlet_WAR_newnoteportlet");
    resourceUrl.setResourceId("saveNote");
    saveNoteUrl = resourceUrl.toString();
});
```

Tvorba URL adres přes JavaScript

Chceme-li zpřístupnit nějakou akci, data či obrazovku přes snáze zapamatovatelnou adresu, máme možnost využít tzv. Friendly URL. Díky tomu lze adresu s mnoha často nedůležitými parametry transformovat do jednotného formátu, s možností parsování parametrů do příslušných proměnných pomocí speciálních výrazů a není nutné pro její tvorbu použít komponenty zmíněné výše. Pro tato mapování vytvoříme soubor ve zdrojové složce /resources s názvem {název}-friendly-url-routes.xml a zaregistrujeme ho do liferay-portlet.xml.

```
<friendly-url-mapper-class>
       com.liferay.portal.kernel.portlet.DefaultFriendlyURLMapper
</friendly-url-mapper-class>
<friendly-url-mapping>{zkrácený-název}</friendly-url-mapping>
<friendly-url-routes>{název}-friendly-url-routes.xml</friendly-url-routes>
```

Registrace mapovacího souboru

Pokud bychom chtěli například získat produkty na základě jejich názvu a měli bychom v tagu friendly-url-mapping products, zavedeme do mapovacího souboru tag route níže. Po zavolání {aktuální-adresa}/-/products/getProduct/notebook, bude zavolána metoda typu resource s názvem getProduct s příchozím atributem productName, ve kterém bude hodnota notebook. Vrátil by se nám pravděpodobně

seznam laptopů například ve formátu JSON. Můžeme zde také nastavit řadu implicitních parametrů, jež budou napevno posílány při každém volání adresy.

```
<route>
	<pattern>/getProduct/{productName}</pattern>
	<ignored-parameter name="p_p_cacheability" />
	<implicit-parameter name="p_p_resource_id">getProduct</implicit-parameter>
	<implicit-parameter name="p_p_lifecycle">2</implicit-parameter>
</route>
```

Mapování URL

## 6.6 Alloy UI

Alloy je frontendový framework založený na JS knihovně YUI a CSS knihovně Twitter Bootstrap. Poskytuje velké množství často používaných komponent a usnadňuje tím tak práci. Nalezneme zde nápovědy, modální okna, tlačítka, vyjíždějící nabídky, palety pro výběr barev, ořezy obrázků a mnoho dalších. Liferay je dodáván s touto technologií a ve svém jádru ji hojně využívá. Díky ní lze generovat komponenty nejen pomocí JavaScriptu, ale také za použití JSP tagů, integrující tyto styly a funkce. Několikrát jsem se zmiňoval o rapidní rychlosti změny ve světě webových aplikací a právě na straně klientské a designové toto platí mnohanásobně. Liferay je v aktuální verzi dodáván s verzí Alloy UI 2, čímž je uzamčen s nyní již mrtvou knihovnou YUI a velice starou knihovnou Twitter Bootstrap 2. Ačkoliv v současné době existuje verze 3, povyšující alespoň CSS framework, v jejich prohlášení stojí kompletní přechod na jQuery a tvrdí, že aktuální verze není cílená na použití v Liferay 6.2 [Zeno Rocha, 2014]. Z mého pohledu se jedná o velice kontroverzní použít tuto technologii v moderních projektech, a proto bych ji použil pouze tam, kde nejsou kladeny velké nároky na UX nebo při prototypování aplikací.

## 6.7 Service Builder

Nástroj Service Builder slouží k usnadnění vytvoření low-level vrstvy. Hlavními technologiemi interně využitými jsou Spring a Hibernate. Pokud tvoříme nový projekt od základů, musíme se postarat o vytvoření doménového modelu s objektově relačním mapováním. Dalšími nezbytnými kroky je zajištění transakcí a správa session či navazování připojení k databázi. Postupem času optimalizujeme výkon a zapracováváme metody pro kešování výsledků. Ke každé entitě je nutné napsat CRUD operace, mnoho elementárních, ale i složitých vyhledávacích metod v našich DAO (Data Access Object)

vrstvách, jež nám vrací výsledky na dotazy. DAO vrstvu využívá zpravidla servisní vrstva, jež se stará o veškerou business logiku a sama je dostupná na mnoha místech aplikace pro pouhé injektování již zinicializovaného objektu. Těchto pár problémů pouze nastiňuje jaké možnosti Service Builder přináší v kontextu platformy Liferay. Ačkoliv je ve vývojovém prostředí označován jako další zásuvný modul, ve finále se jedná o běžný portlet s několik odlišnými konfiguračními soubory a předpřipravenou generovanou strukturou.

Celá komponenta je funkční pouze se správně nastaveným vývojovým prostředím, obsahující Liferay IDE, kterým je generování servisní vrstvy spouštěno. Po vytvoření service builder portletu dle průvodce dostaneme dva projekty. Při pojmenování projektu *service* dostanou název **service-portlet** a **service-portlet-service**.

Service-portlet-service je vygenerované JAR dle definice v service.xml a dalších pomocných xml v projektu service-portlet. 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 a další. 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-portlet modulu zvenčí projektu přistupujeme přes **{EntityName}LocalServiceUtil**. Ačkoliv zdrojový projekt obsahuje mnoho tříd vztahujících se k vytvořené entitě, modifikovatelné jsou pouze tři.

- **{EntityName}LocalServiceImpl** – implementace lokální servisní vrstvy dostupné ve stejné VM. Tato třída by měla sloužit pro práci se samotnou entitou – ukládání, modifikaci či mazání, ale měla by i obsahovat kompletní business logiku.
- **{EntityName}ServiceImpl** – implementace servisní vrstvy pro vzdálená volání, webové služby atd. Tato vrstva by měla pouze provolávat lokální servisu výše a zajišťovat dodatečné kontroly oprávnění.
- **{EntityName}ModelImpl** – rozšíření doménové entity o vlastní implementaci

Ostatní třídy jsou generovány na základě definice v service.xml a dalších konfiguračních souborech ze zdrojové složky /resources, kde ty důležité si nyní popíšeme.

- service.properties – obsahuje odkaz na externí service-ext.properties soubor, číslo, namespace, datum běhu nástroje service builder a důležitou položku pro automatické generování DB schématu. Dále obsahuje odkazy na konfigurační soubory pro Spring. Tento soubor by neměl být upravován.
- service-ext.properties – slouží pro vlastní modifikace původního nastavení.
- META-INF/portlet-model-hints.xml – nastavení dodatečného chování atributů v daných entitách. Lze zde nastavit jak chování z hlediska generování SQL, tak chování na straně klientské, při použití JSP tagu liferay-ui.

Nejdůležitější soubor se ukrývá ve složce WEB-INF a je to již několikrát zmiňovaný service.xml, jež řídí veškeré nastavení a generování. Konkrétní příklad vytvoření entity a servisy je k nalezení v kapitole [Serverová část aplikace 13.2**]. Tento soubor usnadňuje využívání principu servisně orientované architektury (SOA), jelikož nám kromě kompletního zaobalení doménových entit zprostředkuje i třídy pro webové služby SOAP. Je třeba dát si pozor na fakt, že schéma a modifikace nejsou zcela vždy generovány DDL skripty automaticky a je tak nutné, postarat se o své DB schéma vlastními silami. V některých případech lze využít Hibernate položky v portal-ext.properties **hibernate.hbm2ddl.auto=update,** avšak vazby pomocí cizích klíčů či integritní omezení jsou na databázovém administrátorovi.

Nejjednodušší metoda v lokální servisní třídě pro založení nové poznámky může vypadat následovně.

```
public void createNote(NoteFormModel form) throws SystemException {
	Long noteId = super.counterLocalService.increment(Note.class.toString());
	Note note = super.createNote(noteId);
	NoteFormModelToNoteConverter.convertToNote(form, note);
	super.addNote(note);
}
```

Vytvoření entity v servisní třídě

Zde máme dostupné základní persistenční metody zděděné z předka. V případě, že bychom chtěli využívat specifické vyhledávací metody, mazání dle parametrů a další rozšíření spojené s persistencí lze využít objektu notePersistence, kdekoliv v aktuální třídě. V příkladu výše vstupuje do metody naplněný objekt daty z view vrstvy v pomocné DTO třídě. Pomocí zabudované komponenty pro získávání ID vygenerujeme primární klíč a založíme novou entitu. V této chvíli ještě není persistována do DB. Převedeme data z DTO objektu do doménové entity a pomocí předpřipravené metody addNote uložíme

entitu do DB. Kdekoliv v našich portletech, obsahující potřebnou závislost, můžeme jednoduše zavolat NoteLocalServiceUtil.*createNote*(form); bez nutnosti starání se o transakce, kešování nebo správu session.

# 7 Theme plugin Liferay

# 8 Hook plugin Liferay

Rozšíření pomocí hook pluginu realizuje dodatečné akce na základě jistých událostí nebo přepisuje stávající prvky implementované v ROOT aplikaci Liferay jako jsou JSP, properties soubory. Ve složce WEB-INF se nachází soubor liferay-hook.xml, obsahující soubory, které chceme přepsat vlastními.

Následující úryvek přepisuje vybrané properties v lokalizačním souboru a vybrané položky z hlavního konfiguračního souboru. Dále definuje složku s vlastními JSP, které chceme přepsat. Do této složky vložíme soubory stejného názvu se stejnou adresářovou strukturou. Hook se v okamžiku deploy postará o nahrazení původního souboru s vytvořením záložní kopie pro vrácení do původního stavu po odebrání tohoto zásuvného modulu. Nemusíme se proto bát, ztracení originálního obrazu portálu. V případě, že bychom chtěli modifikovat portál v rámci konkrétního portletu, je možné konfigurační soubor liferay-hook.xml vytvořit přímo v portletu a dále postupovat jako v případě autonomního hook projektu.

```
<hook>
	<language-properties>Language.properties</language-properties>
	<language-properties>portal.properties</language-properties>
	<custom-jsp-dir>/META-INF/custom_jsps</custom-jsp-dir>
</hook>
```

Modifikace portálu pomocí hook pluginu

## *8.1 Přepsání zabudovaných servisních tříd*

Další možnosti úpravy Liferay portálu je reimplementace metod v zabudované servisní vrstvě spravující objekty dodávané s výchozí instalací. Patří k nim například uživatel, role, layout, organizace a další. Service-type z příkladu níže definuje servisu, kterou chceme přepsat a service-impl označuje naši implementační třídu dědící z wrapper třídy zdrojové servisy (UserLocalServiceWrapper). Jelikož obalující wrapper

třída pouze převolává metody z implementační vrstvy, lze v naší implementaci pouze přepsat potřebné metody, čímž efektivně upravíme velice často využíváné komponenty.

```
<service>
	<service-type>
		com.liferay.portal.service.UserLocalService
	</service-type>
	<service-impl>
		net.evrem.hook.service.CustomUserServiceLocalImpl
	</service-impl>
</service>
```

Přepis servisní třídy

## 8.2 Pre a post akce

Liferay poskytuje několik událostí, při jejichž volání lze před nebo po, zavolat vlastní akci. Toto se hodí například před uskutečněním přihlášení uživatele pro dodatečné kontroly oprávnění či po odhlášení, kdy lze například uvolnit některá data spojená s aktuálním uživatelem. Akci je nutné zaregistrovat do souboru portal.properties ve stylu `login.events.post=net.evrem.hook.LoginPostAction`, kde třída LoginPostAction dědí z `com.liferay.portal.kernel.events.Action` a implementuje jedinou metodu s dostupnými objekty HTTPServletRequest a HTTPServletResponse, se kterými lze právě na tomto místě pracovat.

## 8.3 Filtry

Další možností interakce s příchozími požadavky a odchozími odpovědi jsou servlet filtry. Ty lze namapovat na konkrétní URL, nadefinovat jaké filtry musí být zavolány před aktuálním a samotná reference na obslužnou třídu s výkonným kódem. Obslužná třída musí implementovat rozhraní `javax.servlet.Filter`, jež poskytuje metody `init`, `doFilter`, `destroy`. Pokud potřebujeme získat konfigurační statické parametry, využijeme metodu `init` s objektem `FilterConfig` obsahující námi nadefinované parametry ze servlet kontejneru. Metoda `doFilter` obsahuje hlavní logiku. Ve většině případů by zpracování mělo vypadat následovně.

- Analyzuj požadavek
- Libovolně doplň hlavičky a parametry požadavku/odpovědi
- Libovolně obal vlastním chováním požadavek/odpověď
- Pošli požadavek dále do dalšího filtru nebo ukonči celý řetězec následných filtrů a přeruš aktuální zpracování

Ve finále je zavolána metoda `destroy` pro vyčištění a uvolnění zdrojů před ukončením instance filtru.

# 9 Layout plugin Liferay

# 10 Ext plugin Liferay

# 11 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.

# 12 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.

## *12.1 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 Facebook, 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 ( je různý od , <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.

## 12.2 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
	});
}
```

Definice akce pro dispatcher

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'));
```

Flux controller-view

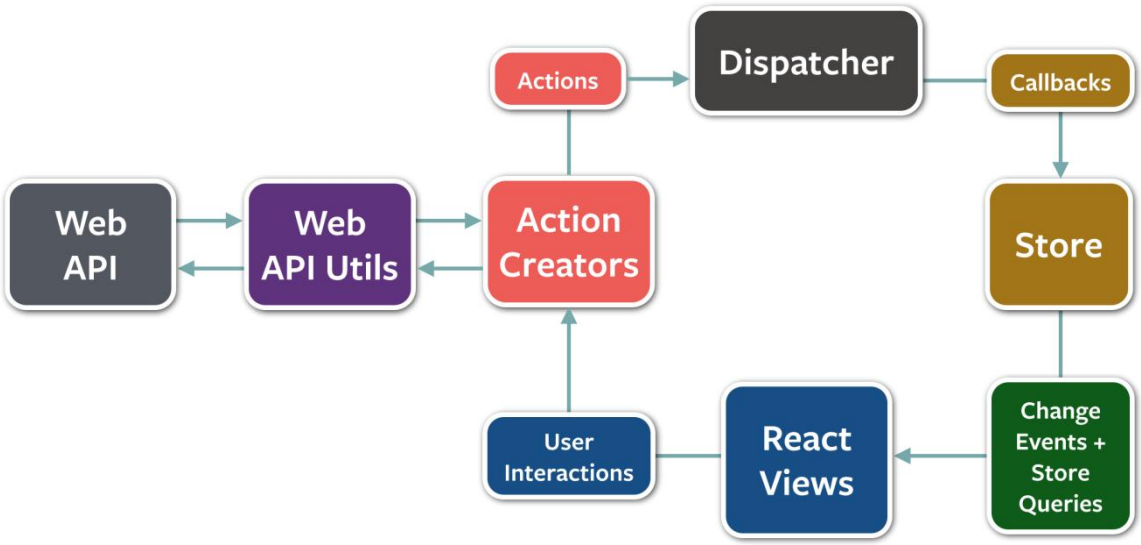
```
//Privátní část - definice modelu, interních metod
var _notes = {};
//Veřejný interface pro konzumaci modelu
var FilterStore = merge(EventEmitter.prototype, {
       getNotes: function(){
              return _note;
       },
       emitChange: function() {
              this.emit(GlobalConstants.CHANGE_EVENT);
       },
       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;
});
```

Flux store

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.




Unidirectional flow

Převzato z https://github.com/facebook/flux

# 13 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.

## *13.1 Klientská část aplikace*

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 obejktu 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'));
       }
    }
);
```

Inicializace view vrsty

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 kešování, 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;
                        }
}
```

Compass a SASS použití

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 klientské části je znázorněno na diagramu níže.

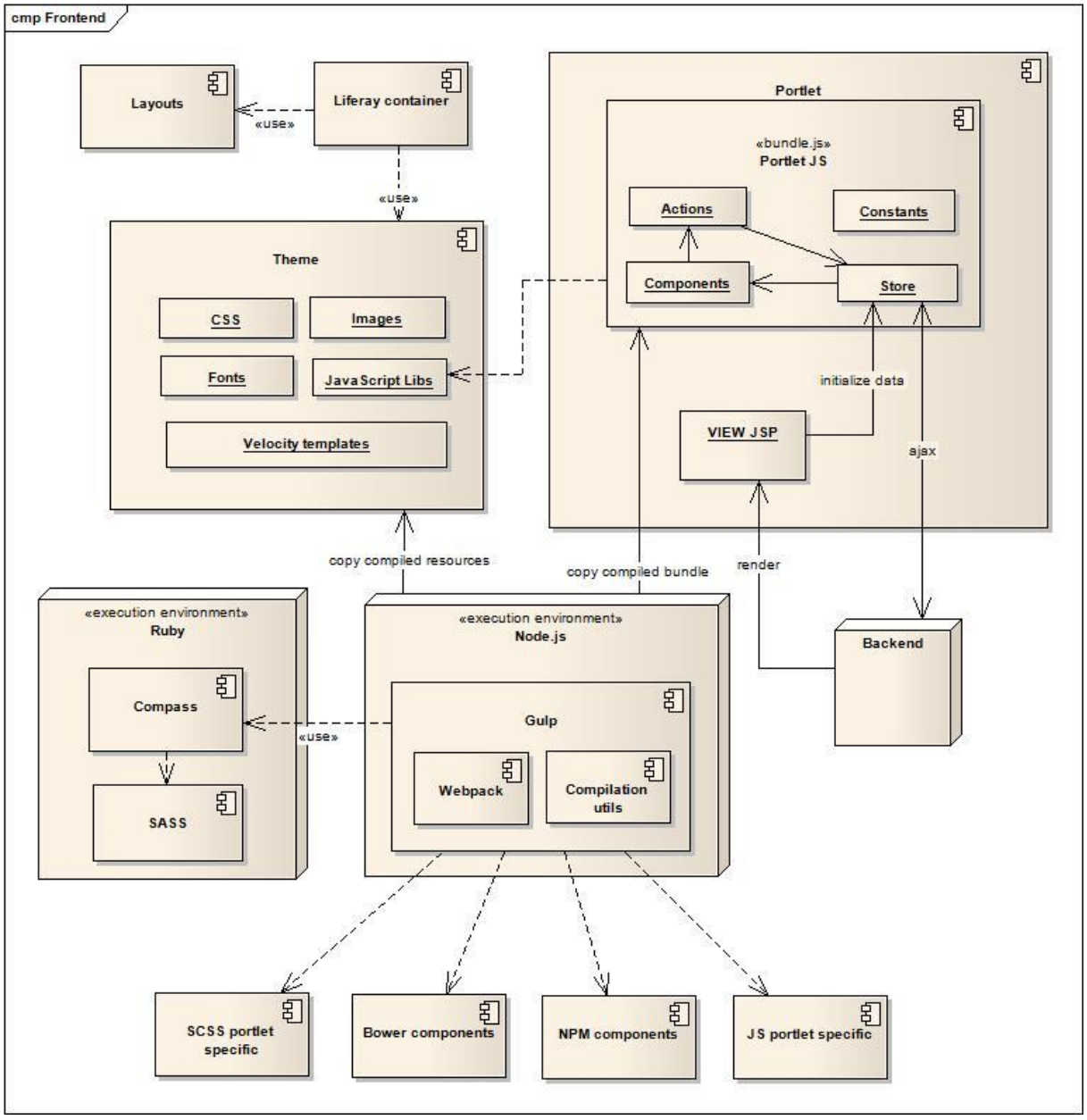



Schéma klientské části

## 13.2 Serverová část aplikace

Liferay nabízí podporu několika aplikační či webových serverů. Hotové balíčky jsou ke stažení na oficiálních stránkách a tak je možné rozchodit portál za několik minut. Vývojář si může zvolit variantu dle preferencí, infrastruktury, popularity či zkušeností. Má aplikace běží na serveru Apache Tomcat vybraném 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 mají namapované metody na ID příchozí akce. 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 payload 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(note);
	response.getWriter().write(resObject.toJson());
}
}
```

Spring MVC controller – render a resource request

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, s využitím zabudovaného řešení v Liferay. Jednoduchým přidáním položky v liferay-portlet.xml spouštíme rutinu dle vydefinované časové prodlevy nebo cron výrazu. Jediné co musí třída scheduler 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>
```

Nadefinovaný Liferay scheduler

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();
}
```

Programové přidání administrátorských práv danému vláknu

Pro snazší přístup ke sdíleným funkcionalitám má commons-function-portlet nastaveny friendly-url. Díky tomu není nutné generovat adresy taglibem v jsp, ale lze je jednoduše vytvořit připojením **/-/common/resourceId?parametry** za aktuální adresu**.**

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

Předávání dat mezi veřejnými 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 DTO, 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 čerpajícím knihovny právě ze zmíněné složky /lib/ext.

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. Využití jiného datového zdroje je ne zcela přímočarou záležitostí. V service portlet projektu je nutné v ext-spring.xml nadefinovat vlastní instance transaction manager, session factory a v našem případě JNDI datový zdroj, 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 databázovým 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>
```

Entity definované v service builder

Service builder se mimo jiné stará o kešování entit z databáze a výsledků vyhledávání. V některých případech se nám toto nastavení nehodí. Máme možnost napevno nastavit kešování v definici entity pomocí atributu cache-enabled nebo lze explicitně přepsat v souboru 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 zde velice jednoduché. Velkou nevýhodou je ovšem nemožnost zasílání obrázků z důvodů chybějící podpory pro MimeMultipart.

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

Zasílání emailů přes zabudované komponenty

### 13.2.1 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 mnoho zbytečných hodin konfigurace navíc. Začátečník naráží na mnoho problémů, při modifikaci theme, tvorbě nových plugin projektů nejsou nalezené archetypy (vytvoření projektu dle šablony), 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>
```

Properties nastavené v settings.xml v Maven

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 rozvržením zmíněných modulů na serveru vidíme na diagramu níže.

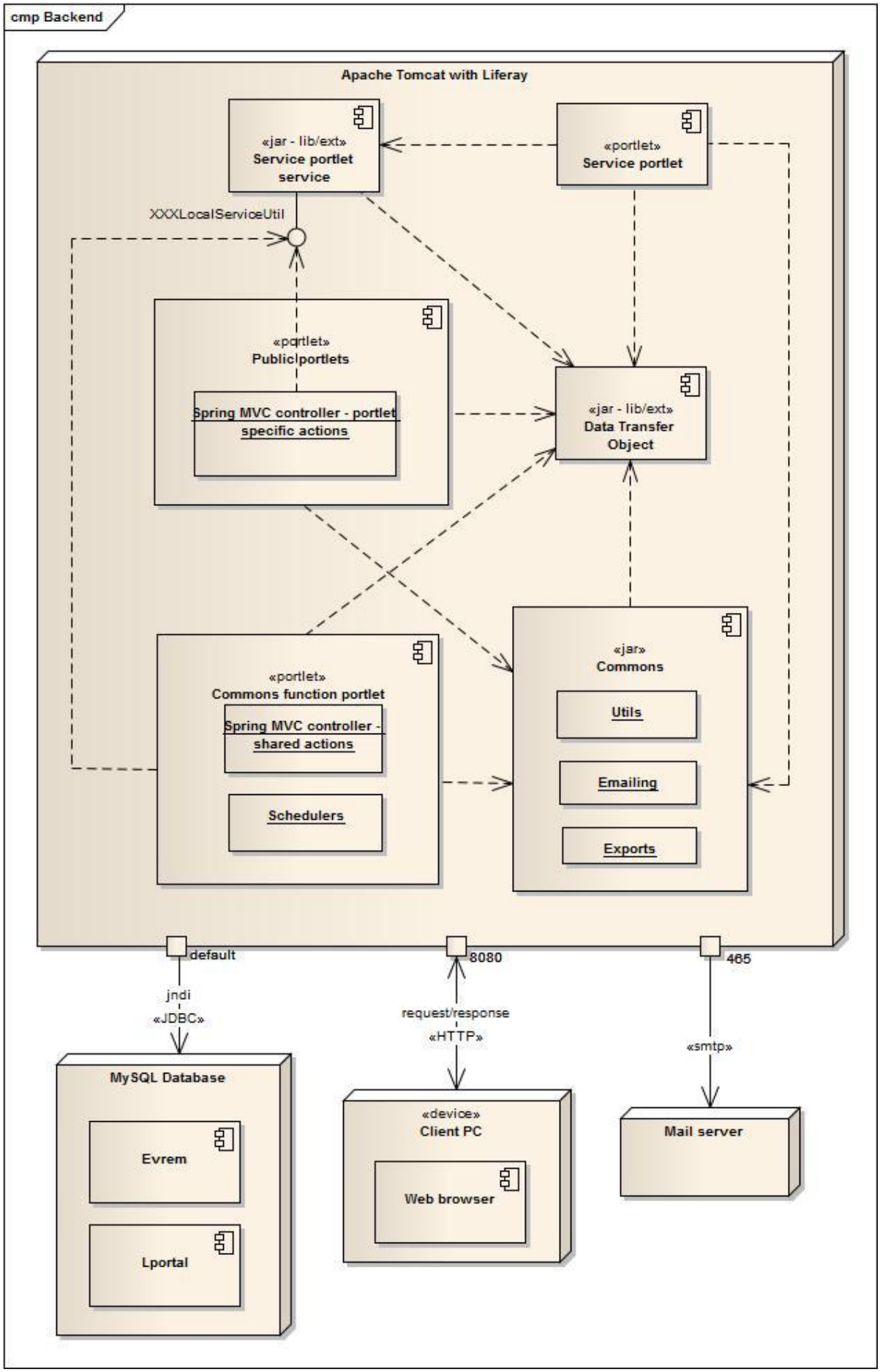

cmp Backend
Apache Tomcat with Liferay
«jar - lib/ext»
Service portlet service
«portlet»
Service portlet
XXXLocalServiceUtil
«portlet»
PublicIportlets
Spring MVC controller - portlet specific actions
«jar - lib/ext»
Data Transfer Object
«portlet»
Commons function portlet
Spring MVC controller - shared actions
Schedulers
«jar»
Commons
Utils
Emailing
Exports
default
8080
465
jndi
«JDBC»
request/response
«HTTP»
«smtp»
MySQL Database
Evrem
Lportal
«device»
Client PC
Web browser
Mail server

Schéma serverové části

### *13.3 Administrátorské nastavení*

Nespornou výhodou portletů je jejich znovupoužitelnost, kdy lze stejný portlet umístit na více stránek nebo vícekrát na jednu stránku (více instancí). Tento prvek jsme využili v případě new-note-portlet, což nám umožňuje na každé stránce vytvářet novou poznámku.

# 14 Ukázky aplikace

Obrázky..

# 15 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
https://github.com/facebook/flux
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

**Liferay**

http://www.liferay.com/community/wiki/-/wiki/1071674/Logical+Architecture
https://www.subbu.org/articles/jts/JTS.html
https://www.liferay.com/products/liferay-portal/tech-specs
https://www.liferay.com/community/wiki/-/wiki/Main/Portal+Architecture
http://www.liferay.com/web/jorge.ferrer/blog/-/blogs/liferay-s-architecture-the-beginning-of-a-blog-series
http://lucene.apache.org/solr/
https://www.liferay.com/community/wiki/-/wiki/Main/Scripting+languages+to+develop+portlets++
http://interval.cz/clanky/zaklinadlo-jmenem-webdav/

Portlety

http://docs.spring.io/spring-framework/docs/3.0.7.RELEASE/reference/portlet.html
http://www.opensource-techblog.com/2012/09/spring-mvc-portlet-in-liferay.html
https://www.liferay.com/community/wiki/-/wiki/Main/Portlet+to+Portlet+Communication
http://www.liferay.com/web/zeno.rocha/blog/-/blogs/alloyui-3-released-bye-bye-yui

# 16 TODO

1.