Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

# BAKALÁŘSKÁ PRÁCE

Zpracování a vizualizace informací generovaných vývojovým procesem softwarového produktu

2019

Vojtěch Skopal

Vedoucí práce: doc. Mgr. Jan Outrata, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

**Bibliografické údaje**

| | |
|---|---|
| Autor: | Vojtěch Skopal |
| Název práce: | Zpracování a vizualizace informací generovaných vývojovým procesem softwarového produktu |
| Typ práce: | bakalářská práce |
| Pracoviště: | Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci |
| Rok obhajoby: | 2019 |
| Studijní obor: | Aplikovaná informatika, prezenční forma |
| Vedoucí práce: | doc. Mgr. Jan Outrata, Ph.D. |
| Počet stran: | 45 |
| Přílohy: | 1 CD/DVD |
| Jazyk práce: | český |

**Bibliograhic info**

| | |
|---|---|
| Author: | Vojtěch Skopal |
| Title: | Processing and visualization of information generated by the software product development process |
| Thesis type: | bachelor thesis |
| Department: | Department of Computer Science, Faculty of Science, Palacký University Olomouc |
| Year of defense: | 2019 |
| Study field: | Applied Computer Science, full-time form |
| Supervisor: | doc. Mgr. Jan Outrata, Ph.D. |
| Page count: | 45 |
| Supplements: | 1 CD/DVD |
| Thesis language: | Czech |

## Anotace

*Práce pojednává o aplikaci sloužící ke zpracování a následné vizualizaci dat, které jsou generovány vývojovým procesem softwarového produktu společnosti Thermo Fisher Scientific. Práce popisuje analýzu dat pro zpracování, architekturu aplikace a detaily její implementace. Práce obsahuje i uživatelskou dokumentaci k vytvořené aplikaci.*

## Synopsis

*This thesis is focused on the application for processing and subsequent visualization of data generated by the development process of the software product of Thermo Fisher Scientific. The thesis describes the analysis of data processing, application architecture and details of its implementation. The thesis also includes user documentation for the created application.*



**Klíčová slova:** zpracování a vizualizace dat; vývojový proces; softwarový produkt

**Keywords:** data processing and visualization; development process; software product

Na tomto místě bych chtěl poděkovat svému vedoucímu bakalářské práce doc. Mgr. Janu Outratovi, Ph.D. za poskytnuté konzultace a věnovaný čas. Ve firmě Edhouse s.r.o. bych rád poděkoval Mgr. Ondřeji Kašparovi za podporu a cenné rady. Děkuji i Mgr. Michalu Běčákovi za připomínky a konzultace technických požadavků ze strany společnosti Thermo Fisher Scientific. Nakonec bych chtěl poděkovat svojí rodině a přítelkyni za podporu během psaní práce a mého studia

*Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.*

datum odevzdání práce podpis autora

# Obsah

# Seznam obrázků



# Seznam zdrojových kódů

# 1 Úvod

V úvodu své bakalářské práce bych rád zmínil, jak jsem se dostal k výběru daného tématu. Chtěl jsem rozvinout své programátorské dovednosti a poznat jak to chodí v praxi, proto jsem hledal stáž pro studenty. Po zvážení všech nabídek jsem se rozhodl pro stáž ve firmě Edhouse. Firma Edhouse se specializuje na vyvíjení softwaru pro outsourcingové společnosti. Jedna z jejich hlavních zákaznických společností je společnost Thermo Fisher Scientific, která se primárně specializuje na vývoj elektronových mikroskopů. K obojím společnostem se více dostanu v samostatné kapitole.

Na stáži jsem pokračoval i při škole a mohl jsem tak získat mnoho zkušeností a více proniknout do vývoje softwaru v praxi. Během tohoto období jsem přemýšlel, jaké téma bakalářské práce si zvolit. Chtěl jsem takové téma, kde převážná část práce bude o vytvoření nástroje, který bude mít využití. Zanedlouho se mi otevřela možnost dělat bakalářskou práci pro společnost Thermo Fisher Scientific. Věděl jsem, že je to přesně takové téma, které jsem hledal. Byl jsem moc rád, že budu moci pracovat na bakalářské práci, která bude zaměřena přímo do praxe.

Nejdříve bych chtěl přiblížit o jakou práci konkrétně půjde a co mě do ní nejvíce motivovalo. Společnost Thermo Fisher Scientific generuje při svém vývojovém procesu velké množství dat. Generovaná data představují pro společnost velmi cenné informace, které jsou potřebné pro vývoj a testování jejich produktů. Tyto informace často nejsou nijak dále zpracovány a jsou uloženy v těžko přístupné podobě na různých místech. Pro společnost Thermo Fisher Scientific by bylo velkým přínosem, kdyby se tato data zpracovala a byla přehledně zobrazena.

Bylo pro mě velkou motivací vytvořit aplikaci, která usnadní dostupnost a přehlednost informací, které jsou potřebné pro vývoj softwaru. Chtěl jsem vytvořit aplikaci, která bude denně používaná jako interní software společnosti a bude sloužit pro vyhledávání všech důležitých informací.

Nejprve bylo nutné analyzovat data, která se budou zpracovávat, potom zjistit v jaké struktuře jsou uloženy a jáká je mezi nimi spojitost. Dále navrhnout vhodné modulové schéma celé aplikace. Posléze si rozmyslet rozmístění a logické propojení jednotlivých modulů. Následně bylo nutné rozčlenit zpracování dat do jednotlivých pluginů, které budou mít na starost jen úseky dat. Bylo zapotřebí navrhnout a vytvořit vhodný společný modul, ke kterému budou tyto pluginy připojeny, a který je bude spouštět. Modul, který musí být dostatečně obecný natolik, aby se k němu mohli v budoucnu připojit nové pluginy. Kladl jsem tedy důraz na snadnou rozšiřitelnost celé aplikace. Velmi důležitou částí byl i výběr vhodné databáze, která musí pojmout velké množství dat a dokázat v nich rychle vyhledávat.

Druhá polovina této práce se zabývá vizualizací zpracovaných dat. Bylo nutné vybrat vhodné technologie a vytvořit uživatelsky přívětivou webovou aplikaci. Poslední částí bylo vytvoření modulu pro prohledávání nashromážděných dat a posílání oznamovacích e-mailových zpráv daným uživatelům.

# 2 Představení společností

V této kapitole bych chtěl dát prostor pro krátké seznámení s oběma společnostmi, o kterých byla řeč už v úvodu práce. Představím vám, na co se obě společnosti specializují a krátce vám je charakterizuji.

## 2.1 Společnost Edhouse

Společnost Edhouse je českou softwarovou společností, která má celkem čtyři pobočky na Moravě, kde soustřeďuje svůj růst. Společnost Edhouse pracuje pro tzv. outsourcingové společnosti. To znamená, že nevlastní svoje produkty, na kterých by pracovala a nabízela je svým zákazníkům. Ale naopak jejich zákazníci jsou softwarové společnosti, pro které vyvíjí software a společně s nimi se na tomto vývoji podílí. Jedním z jejich hlavních zákazníků je společnost Thermo Fisher Scientific, kterou přiblížím v další kapitole. Zde je odkaz na webové stránky této společnosti: https://www.edhouse.cz/


## 2.2 Společnost Thermo Fisher Scientific

Společnost Thermo Fisher Scientific patří k světovým špičkám v oblasti mikroskopie. Jde o mezinárodní společnost, která má i jednu ze svých poboček v České republice. Hlavním produktem, který dodává svým zákazníkům, jsou tzv. elektronové mikroskopy. Tyto mikroskopy mají využití hlavně v oblasti vědy, například i pro výrobu mikročipů. Obecně se dá říci, že jde o všechny oblasti, kde se pracuje s nanotechnologiemi. Společnost Thermo Fisher Scientific zajišťuje celkový vývoj těchto mikroskopů a to po hardwarové i softwarové stránce.

Ve společnosti je definován vývojový proces, pomocí kterého se pracuje na vývoji softwaru pro elektronové mikroskopy. Na web společnosti se můžete dostat pomocí tohoto odkazu: https://www.thermofisher.com/cz/en/home.html

# 3 Analýza struktury dat

V této části práce se budu snažit popsat a analyzovat data, které generuje vývojový proces softwarového produktu společnosti Thermo Fisher Scientific. Bude zde ve stručnosti rozebráno, jakým způsobem se data generují. Dále analyzuji strukturu dat a popíši důležitou terminologii, která se používá u jednotlivých typů generovaných dat. Z této terminologie se bude vycházet v dalších částech této práce. Rozeberu postupně čtyři kategorie generovaných informací.

## 3.1 Build data

Představme si velmi rozsáhlý projekt aplikace, který obsahuje reference na další projekty aplikací a knihovny třetích stran. Pro sestavení takového projektu, bychom potřebovali mít všechny tyto knihovny aplikací na jednom místě. To by bylo značně nepraktické, pokud bychom kromě svého projektu museli manuálně dokopírovat i všechny ostatní projekty, který má náš projekt ve svých závislostech. Druhý problém by byl s aktualizací knihoven aplikací. Když bychom například chtěli používat ve svém projektu knihovnu jiné aplikace s novější verzí, museli bychom jednak změnit ručně závislosti, ale i manuálně aktualizovat závislý projekt na novější verzi.

Společnost Thermo Fisher Scientific tento problém s lokalizací a aktualizací závislých projektů řeší pomocí své interní platformy pro sestavování projektů. Jedná se o způsob, kdy všechny závislosti na druhých projektech jsou definované ve speciálním souboru u projektu. Pokud uživatel potřebuje sestavit svůj projekt, který má závislosti, platforma automatizuje potřebné kroky pro jeho sestavení. Nejprve synchronizuje všechny projekty, které jsou v závislostech a potom provede sestavení hlavního projektu. Uživatel nemusí svůj projekt sestavovat jen lokálně, ale může požít tzv. automatizované sestavování. K tomu se využívá nástroj *Jenkins*, přes který se tato akce spouští. Nástroj *Jenkins* obsahuje sekce pro sestavování projektů. Každý *build* (sestavení projektu), který je takto spuštěn, se nazývá *jenkins job*. Sestavování projektu se pak neprovádí lokálně u uživatele, ale jsou k tomu použity externí virtuální servery. Výsledná data sestavení projektu se zapisují do společného úložiště.

V tomto společném úložišti jsou přítomny data společně s verzemi projektů a jejími závislostmi. Jsou zde uložena data sestavení všech projektů společnosti Thermo Fisher Scientific. Tyto data budou jedním z hlavních zdrojů informací, které se budou shromažďovat. Pro pojmenování těchto dat se bude dále používat název *build data*.

*Build data* jsou uložena ve stromové struktuře složek. Počet zanoření této stromové struktury je pouze do třetí úrovně stromu. Kořenem stromu je společná složka pro všechny *build data*. Než se dostaneme k jednotlivým úrovním stromu, musíme nejprve znát následující pojmy s nimi spojené.

### 3.1.1 Component

*Component* (komponenta) definuje určitou kategorii, do které můžeme zařadit náš projekt. Tato kategorie vetšinou označuje typ projektu a zařazuje projekt do obecnější skupiny, který projekt charakterizuje. Komponenta je jednoznačně určena svým jménem.

Komponenta určuje také zařazení projektu z hlediska použití aplikace na mikroskopu. Například je komponenta pro práci s posuvným stolkem u mikroskopu nebo komponenta, která má na starost práci s vakuem na mikroskopu. Dále pak mohou být ještě aplikační komponenty, které integrují vyšší vrstvu na mikroskopu. Je to například zpracování a úprava nasnímaných vzorků z mikroskopu, grafické obarvení nebo jasové úpravy vzorků.

### 3.1.2 Package

*Package* (balíček) je jeden konkrétní výsledek sestavení našeho projektu. Je jednoznačně určen tzv. *package label* a názvem komponenty, do které spadá. *Package label* může obsahovat název komponenty, vždy obsahuje tzv. *package version* a další částí, které jej specifikují.

Balíček definuje tyto základní informace:

- *Component* - Definuje název komponenty, do které balíček patří.
- *Package label* - Udává konkrétní název balíčku, který jej specifikuje.
- *Version* - Určuje verzi balíčku, kterou obsahuje *package label*. Při každém novém sestavení projektu se verze balíčku zvyšuje v závislosti na předchozí verzi. Verze balíčku bývá často označována jako *build number*, protože udává právě číslo sestavení daného balíčku.
- *Dependencies* - Definuje závislosti balíčku, který může být závislý na dalších komponentách a jejich konkrétních balíčcích. Dále může mít i závislosti na komponentách třetích stran, které se budou označovat jako *scot* komponenty nebo pouze zkratkou *SCOTS*.

Závislosti balíčku jsou tvořeny stromem o neomezené hloubce zanoření. Každá závislost může mít tedy své vlastní závislosti na jiných komponentách nebo *scot* komponentách. Důležitým předpokladem je, aby v závislostech nevznikal kruh. Jednalo by se tak o cyklickou závislost, kterou by nešlo popsat stromem závislostí. Tuto problematiku, jak správně propojovat závislosti, si řeší společnost Thermo Fisher Scientific interně.

### 3.1.3 Struktura uložení build dat

*Build data* jsou uložena ve stromové struktuře složek. V první úrovni stromu jsou složky, které představují skupiny komponent. Jedna z možností rozdělení

komponent do skupin je podle toho, k jakému typu mikroskopu se dané komponenty používají. O dalších metodách rozdělení zde nebudu více pojednávat. Druhou úrovní stromu jsou složky, které představují jednotlivé komponenty. Poslední úrovní stromové hierarchie jsou složky balíčků. Tyto složky jsou pojmenované podle *package label* balíčku, kterého v sobě obsahují.

Ve složce balíčku, která je v třetí úrovni stromu, jsou všechny data o balíčku. Uvnitř složky balíčku je složka s názvem *build*. Tato složka obsahuje soubory, které definují všechny informace o balíčku a jeho sestavení. Důležité jsou tři soubory, z kterých se budou shromažďovat data. Jsou to soubory *build_info.xml*, *package.build.xml* a *project.build.xml*.

### 3.1.4 build_info.xml

V souboru *build_info.xml* jsou obsaženy tyto informace:

- Typ verzovacího systému - Ve společnosti Thermo Fisher Scientific se primárně používají dva typy verzovacích systémů. Prvních z nich je *Rational Team Concert*, pro který se používá zkratku RTC. Druhým typem je verzovací systém GIT. Podrobnější terminologie pojmů k těmto verzovacím systémům zde nebude uvedena, bude se vycházet z toho, že ji už čtenář zná.

- *Stream* - Určuje název *streamu*, do kterého balíček patří. Má smysl jej uvažovat jen pro balíček používající RTC.

- *Workspace* - Udává název *workspace*, pro kterou byl proveden *build*. Opět je tato informace obsažena jen pro balíček používající RTC.

- *Repository* - Určuje url *repository*, která se pro balíček používá. Je obsažena jen pro balíček používající GIT.

- *Branch* - Určuje název *branch*, pro kterou byl proveden *build*. Tato položka je opět obsažena jen pro balíček používající GIT.

- *Jenkins url* - Určuje url adresu pro *jenkins job*, přes který byl *build* spuštěn. *Jenkins job* obsahuje parametry spuštění pro sestavování projektu. Důležité jsou tyto dva hlavní parametry. Pro RTC je to název *workspace*, který obsahuje náš projekt. Pro GIT je to *repository* a název *branche*, pro kterou chceme provést *build*. Druhý parametr nám udává informaci o architektuře systému, pro kterou je projekt určen. Tento parametr se bude dále označovat jako *target*. *Target* může být buď *debug* nebo *release*, pro 64 bitovou architekturu je to *debug64* nebo *release64*.

### 3.1.5 package.build.xml

Hlavním souborem, který obsahuje ty nejdůležitější informace o balíčku, je soubor *package.build.xml* obsahující tyto položky:

- *Name* - Udává název komponenty, do které je balíček zaintegrován.
- *Label* - Zde je uložen *package label* balíčku
- *Version* - Tato položka nese informaci o verzi balíčku.
- *PackageArea* - Definuje jaké je místo uložení balíčku. Pro *build data* je to jejich společné úložiště, které se nazývá *HoldingArea*. Komponenty třetích stran neboli *SCOTS* mají své úložiště a pro jeho označení se používá název *libraries*. Atribut *PackageArea* má smysl uvažovat jen pro komponenty, které jsou v závislostech balíčku. V primárních informacích o balíčku se bude vždy jednat o úložiště *HoldingArea*.
- *TimeStamp* - Udává přesný datum a čas vygenerování těchto dat pro daný balíček.
- *Dependencies* - Definuje stromovou strukturu XML elementů, která udává závislosti balíčku. Každý tento element je v XML struktuře pojmenován jako *Component* a obsahuje výše zmíněné atributy, které jsou *Name*, *Label*, *Version* a *PackageArea*.

### 3.1.6 project.build.xml

Posledním z trojice souborů je soubor *project.build.xml*. Shrnuje informace o projektu, do kterého je balíček zařazen. V této fázi je na místě objasnit vztah projektu k balíčku. Projekt obsahuje tzv. *source package* (zdrojový balíček), ze kterého je primárně složen. Projekt může obsahovat i více zdrojových balíčků. Jaké jsou tyto zdrojové balíčky pro daný projekt, definuje právě soubor *project.build.xml*.

Tento soubor má podobnou strukturu jako soubor *package.build.xml*. Nedefinuje však jen závislosti pro jeden balíček, ale komponentovou strukturu závislostí celého projektu. V této struktuře najdeme jak závislosti projektu tak i definici zdrojových balíčků. Poznáme je tak, že mají navíc v elementu *Component* atribut *Source* nastavený na pravdivostní hodnotu *true*. Je třeba zdůraznit, že každý balíček má svoje závislosti definované v souboru *package.build.xml* a k tomu navíc má definované i projektové závislosti uvedené v souboru *project.build.xml*.

## 3.2 Package binaries

Druhou kategorií generovaných dat je kategorie *package binaries* (binární soubory balíčku). Předchozí kapitola pojednávala o informacích, které se váží přímo

k balíčku. Nyní budou důležitá výsledná data, která jsou očekávána po sestavení daného balíčku neboli po sestavení projektu, který je v daném balíčku zahrnut.

Jedná se o binární soubory balíčku, které se označují i jako *package binaries*. Balíček obsahuje své binární soubory i všechny binární soubory každého balíčku, kterého má ve svých závislostech. Dále zahrnuje i všechny binární soubory *scot* komponent.

### 3.2.1 Struktura uložení package binaries

Znovu se vraťme k stromové struktuře uložení komponent a jejich balíčků, která popisuje přesnou hierarchii pro *build data*. Připomeňme si, že v třetí úrovni tohoto stromu se nachází složky, které zahrnují všechny informace o balíčcích. Tyto složky obsahují podsložku *build*, která pro nás byla důležitá v předchozí kapitole. Dále zde najdeme i podsložku *sdk*, kterou se teď budeme zabývat.

Složka *sdk* obsahuje nejméně dvě podsložky. První z nich je složka *bin*, ve které jsou uloženy všechny binární soubory balíčku. Tyto binární soubory jsou dále rozděleny maximálně do čtyř skupin, které představují další podsložky. Rozdělení je podle *target*, pro který byl *build* proveden. To znamená, že ve složce *bin* najdeme adresáře *Debug*, *Release* nebo *Debug64* a *Release64*. Binární soubory v těchto adresářích jsou na první pohled totožné, ale liší se architekturou operačního systému, pro které jsou určeny. Druhou složkou v adresáři *sdk* je složka *installations*, ve které jsou specifické binární soubory používané pro instalaci softwaru daného balíčku.

Jako binární soubory balíčku se budou zaznamenávat jen soubory s příponou *exe* nebo s příponou *dll*. Pro každý binární soubor se bude ukládat jeho název a jeho detailní informace o něm. Jedná se o informace, které můžeme najít například v operačním systému Windows v detailech u binárních souborů.

## 3.3 SCOTS

Dostali jsme se k další skupině informací, které se budou shromažďovat. Jsou to komponenty třetích stran, které se označují jako *scot* komponenty nebo jen zkratkou *SCOTS*. *Scot* komponenty jsou například obsaženy v závislostech balíčku.

Společnost Thermo Fisher Scientific všechny komponenty třetích stran, které se používají v balíčcích, ukládá do společného úložiště. Nejedná se tak přímo o generovaná data, ale jde spíše o proměnlivé úložiště pro *scot* komponenty. Do tohoto úložiště přibývají podle potřeby nové komponenty třetích stran.

### 3.3.1 Struktura uložení SCOTS

Strukturu uložení *SCOTS* definuje podobná hierarchie jako pro *build data*. Jedná se tedy o stromovou hierarchii. První úroveň stromu obsahuje název *scot* komponenty. Ve druhé úrovni jsou jednotlivé *scot* komponenty. Složky, které tyto komponenty obsahují, mají název složený z názvu *scot* komponenty a její verze.

*Scot* komponenta obsahuje hlavně své binární soubory, které potřebujeme jako externí knihovny v našich projektech. Názvy těchto binárních souborů společně s detailními informacemi o těchto souborech se budou zaznamenávat pro každou *scot* komponentu.

## 3.4 SmokeTest data

V dnešní době vývoje softwaru téměř neexistuje software, ke kterému by se zároveň při vývoji nepsaly i testy. Jednotlivé vývojové úseky určitého softwaru by se neměly považovat za dokončené, pokud k nim zatím testy neexistují. Může se jednat o automatizované testy, které nám aplikaci otestují hned při jejím spuštění nebo může jít i o tzv. *UI* testy. Ty nám testují *UI* desktopové nebo webové aplikace. Důkladné testování softwaru nám zaručuje předejití neočekávaných chyb v naší aplikaci a zkvalitňuje tak samotný vývoj.

Ve společnosti Thermo Fisher Scientific se právě tyto pravidla vývoje softwaru striktně dodržují. To znamená, že pro každou komponentu se při jejím vývoji píší i testy. Zvláště pro produkční komponenty, které se potom reálně používají na mikroskopu a jdou směrem k cílovým zákazníkům, jsou pevně nastavena pravidla testování. Některé komponenty mají vytvořeny ke svým aplikacím sady automatizovaných testů. Tyto sady testů dokáží po svém spuštění postupně otestovat celou aplikaci a v případě desktopových aplikací vytvořit i snímky aplikace v momentě jejího selhání. Výsledky tohoto testování jsou pak ukládány do výsledného souboru nebo jsou dostupné v jiné podobě. Chtěl bych doplnit, že jsou i testovací nástroje, které dovedou tyto testy spustit a přehledně zobrazit jejich výsledky.

Pro celou řadu komponent je také nutné nejprve připravit testovací prostředí. Musejí se nainstalovat všechny potřebné nástroje, které umožňují testovat danou aplikaci. Testovací prostředí se obvykle instaluje na virtuální servery, na kterých se pak provádí veškeré testování dané aplikace. Můžeme si všimnout, že tu máme spoustu kroků, které jsou potřeba pro dobré otestování aplikace, ale tyto kroky nejsou nijak automatizované. Tímto zjištěním se dostáváme k bodu, kdy je nutné popsat, jakým způsobem se tyto kroky ve společnosti Thermo Fisher Scientific automatizují.

### 3.4.1 SmokeTesty

Testy, které mají všechny tyto testovací kroky automatizované, se nazývají SmokeTesty. Ve společnosti Thermo Fisher Scientific je vyvíjen framework, který se stará o kompletní správu SmokeTestů. Tento framework je pojmenován jako *Component Test Framework*, zkráceně se pro něj používá zkratka *CTF*.

Pokud jsou definovány pro danou komponentu automatizované testy, mohou se pro ni vytvořit SmokeTesty. *CTF* mechanizuje všechny potřebné kroky pro spuštění testování. Vytvoří čisté virtuální servery, které se označují jako *virtual machines* a jejich počet je stanoven podle toho na kolika konfiguracích chceme aplikaci otestovat. Dále se na tyto servery nainstalují potřebné nástroje

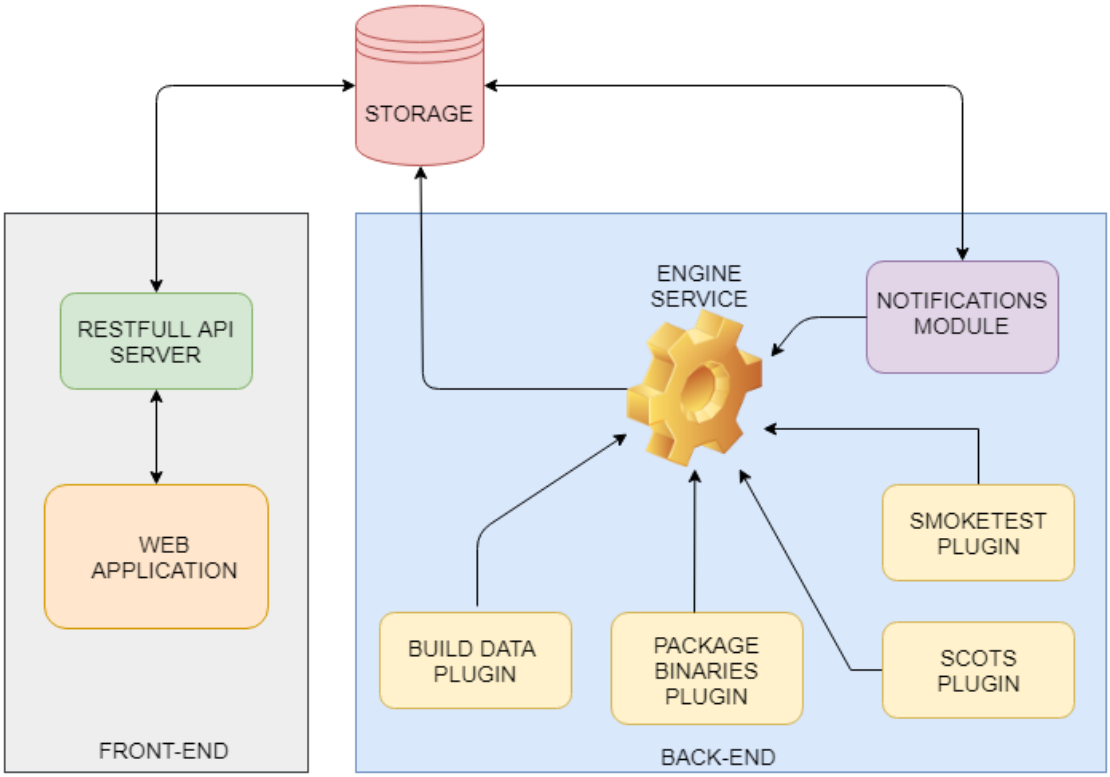
pro testování produktu, které jsou definované u komponenty. Po dokončení instalace jsou postupně spouštěny nadefinované testy. Výsledky těchto testů jsou následně ukládány do společné databáze. Pro zobrazení výsledků testování dané komponenty slouží webová aplikace, kde jsou dostupné všechny detailní informace o testování. Výsledky SmokeTestů se vyhledávají podle názvu *streamu*, který je často totožný jako v *RTC* a dále se testy filtrují podle parametru *build number*.

Když se zamyslíme nad jistou skutečnosti, která spojuje SmokeTesty a *build data*, dojdeme k závěru, že se jedná právě o parametr *build number*. *Build number* představuje číslo provedeného sestavení projektu, toto sestavení se nazývá jako *build*. Pod pojmem *build number* si také můžeme vybavit parametr *version* u struktury balíčku, a to protože jsou tyto dva parametry téměř totožné.

Právě přes parametr *build number* se budou pro každou komponentu spojovat generovaná *build data* s výsledky SmokeTestů, které se váží k téže komponentě. Pro jeden *build number* může existovat více výsledků SmokeTestů. Znamená to jen, že pro jednu konkrétní verzi komponenty byly SmokeTesty vícekrát spuštěny. Pro každý SmokeTest je spočítána celková procentuální úspěšnost, podle toho zda testy doběhly v pořádku či nikoli. Pokud tedy máme více SmokeTestů pro jeden *build number*, bude se zaznamenávat průměr těchto procentuálních úspěšností. Všechna *SmokeTest data* se budou získávat z databáze, kam se výsledky SmokeTestů ukládají.

# 4 Architektura aplikace

V této části práce chci pojednat o celkové architektuře aplikace, která je předmětem této práce. Rozeberu modulové schéma aplikace, které je zobrazeno pod tímto úvodním textem na obr. 1. Následně konkrétně popíši jednotlivé moduly aplikace. Vysvětlím jejich činnost, logické uspořádání a propojení všech modulů mezi sebou.




Obrázek 1: Modulové schéma architektury aplikace

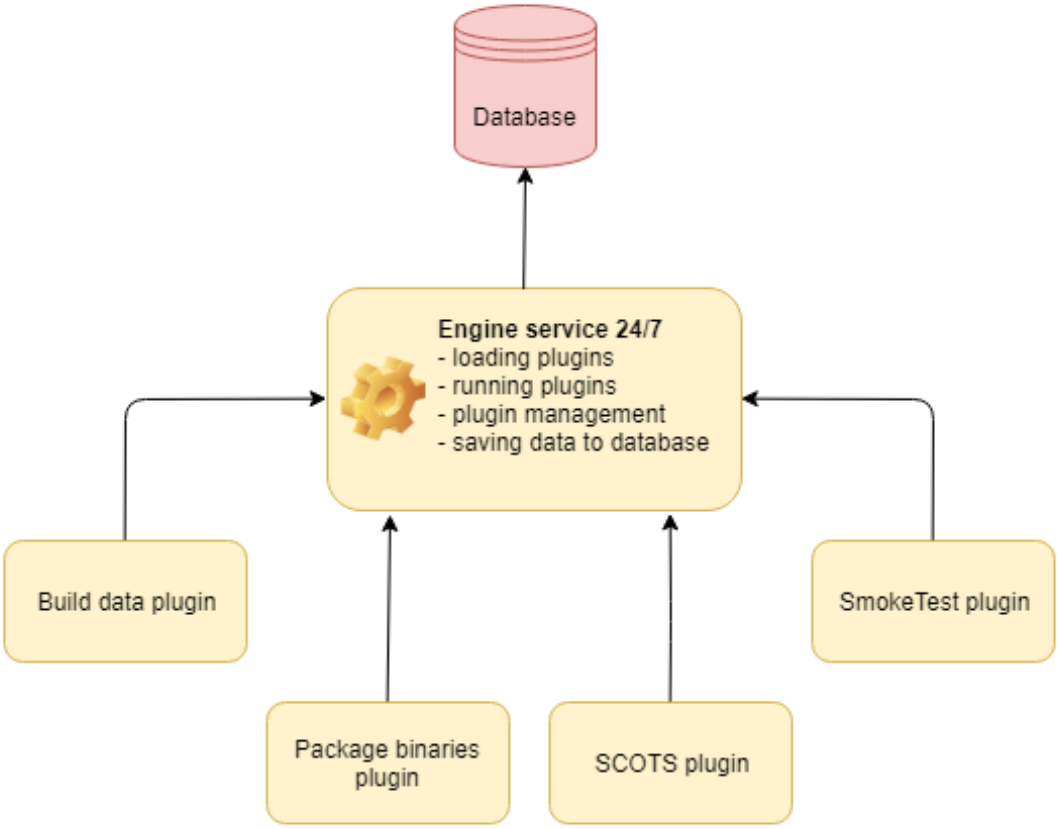
Podívejme se na modulové schéma na obr. 1, které zobrazuje architekturu aplikace. Jak jsem uvedl v úvodu práce, aplikace se skládá ze dvou hlavních částí. První část aplikace má na starost zpracování generovaných dat, které jsem přiblížil v předchozí kapitole. Druhá část aplikace má za úkol nashromážděná data vizualizovat a umožnit všechny tyto informace přehledně zobrazit uživateli. Zpracování dat v modulovém schématu definuje tzv. *back-end* aplikace a vizualizace těchto dat představuje *front-end* aplikace. Z této definice můžeme obě části chápat jako oddělené celky.

## 4.1 Zpracování generovaných informací

Všechny generované informace jsou zpracovávány pomocí pluginů. Každý plugin je určen ke zpracování dat pro jednotlivé kategorie informací, které jsem detailně popsal v předchozí kapitole o analýze dat. Pod pojmem plugin si můžeme

vybavit oddělený modul, který pro své spuštění a funkčnost potřebuje připojení ke společnému modulu všech pluginů. O spuštění a správu těchto pluginů se stará tzv. *Engine service*.

Další úlohou *Engine service* je správa a umožnění ukládání dat do společné databáze. Na modulovém schématu můžeme vidět ještě speciální modul, který se jmenuje *Notifications modul*. K tomuto modulu se dostaneme později v tomto textu. Nejdříve si postupně přiblížíme funkcionalitu všech modulů, která tvoří *back-end* aplikace a má na starost shromažďování generovaných dat. Pro ještě lepší představu o fungování zpracování dat nám může pomoci modulové schéma uvedené na obr. 2.




Obrázek 2: Modulové schéma zpracování dat

### 4.1.1 Engine service

Hlavním modulem, který řídí zpracování dat je modul *Engine service*. Je to služba, která je neustále spuštěna na serveru, kde je aplikace nasazena. K této službě jsou připojeny pluginy, které se starají o zpracovávání dat.

*Engine service* obsahuje následující funkcionalitu.

1. Načtení a inicializování všech pluginů.
2. Spuštění pluginů.
3. Správa pluginů a informování o jejich aktuálním stavu práce.
4. Průběžné ukládání zpracovaných dat ze všech pluginů do databáze.

Klíčovým předpokladem pro *Engine service* je udržování všech pluginů v konzistentním stavu, aby nedocházelo k výpadkům pluginů z důvodu neočekávaných chyb. Pokud se tak stane, mělo by dojít k automatickému restartu běhu daného pluginu a pokusu o jeho obnovení. V žádném případě by nemělo docházet k nutnosti restartovat celou *Engine service*.

Další vlastností tohoto modulu je jeho odstínění od všech pluginů, které se k němu připojují. *Engine service* nemá žádné konkrétní informace o pluginech. Pohlíží na ně pouze jako na obecné moduly, kterým přiděluje stejnou funkcionalitu. Touto vlastností je docíleno toho, že k *Engine service* mohou být do budoucna připojeny nové pluginy, které budou mít na starost zpracování nových dat. Pokud bude potřeba přidat určitou funkcionalitu pro tyto nově připojené pluginy, bude zaintegrována i pro všechny současné pluginy. Stejné odstínění platí i pro ukládání dat do databáze. Každý plugin si sám řídí kam a v jaké podobě svá zpracovaná data do databáze uloží. *Engine service* mu k tomu pouze poskytuje potřebnou funkcionalitu a zajišťuje připojení do databáze.

### 4.1.2 Build data plugin

První z pluginů je *Build data plugin*. Tento plugin zpracovává všechny data o balíčcích, která se generují pro každou komponentu, pro kterou je proveden *build*. *Build data plugin* je spuštěn ve smyčce a v každém novém běhu prohledává zdrojové úložiště pro *build data*, které se jmenuje *Holding area*. Plugin prochází složky, které mají stromové uspořádání a snaží se najít nově vzniklé balíčky, které ještě nejsou uloženy v databázi. U každé složky balíčku parsuje soubory *build_info.xml*, *package.build.xml* a *project.build.xml*. Z těchto souborů plugin získává všechny informace, které charakterizují balíček.

Po dokončení průchodu celého stromu složek následuje část procesu, ve kterém jsou doplněny projektové závislosti pro balíčky, ve kterých chybějí. Tento nedostatek je důsledkem toho, že u některých balíčků není přítomný soubor *project.build.xml*. Tento soubor je obsažen jen u balíčků, které jsou zároveň označeny jako zdrojové balíčky pro daný projekt.

Posledním krokem jednoho běhu tohoto pluginu je uložení všech nově zpracovaných balíčků do databáze. Po každém běhu se hlavní smyčka pluginu uspí na nastavitelnou dobu. Toto opatření je z důvodu šetření hardwarových prostředků a také slouží k odlehčení úložiště *Holding area*, ze kterého se data čtou. Uspávání hlavní smyčky pluginu se používá u všech čtyř pluginů.

### 4.1.3 Package binaries plugin

Jak už název napovídá, tento plugin slouží ke zpracování binárních souborů balíčků. Informace o binárních souborech jsou brány ze stejného úložiště jako pro *Buid data plugin*. Opět je procházena totožná struktura složek s tím rozdílem, že tentokrát jsou zpracovávány data o binárních souborech. Pro tento plugin je definována jedna závislost, která udává, že jsou zpracovávány jen binární soubory balíčků, pro které už existuje záznam v databázi. Záznamem v databázi jsou myšleny uložené informace o balíčku. *Package binaries plugin* má tedy závislost na datech z *Build data pluginu*.

### 4.1.4 SCOTS plugin

*SCOTS plugin* shromažďuje informace o *scot* komponentách. *SCOTS* najdeme ve speciálním úložišti pro komponenty třetích stran. Toto úložiště má podobnou stromovou strukturu jako v případě úložiště pro *build data*. Plugin prochází tuto strukturu a ukládá názvy *scot* komponent společně s informacemi o jejich binárních souborech. Tento plugin není závislý na jiném pluginu. Je pro něj definována pouze kontrola dosud uložených *scot* komponent v databázi. Hlavní účel tohoto pluginu je udržování všech aktuálních *scot* komponent, které jsou používány v závislostech balíčku.

### 4.1.5 SmokeTest plugin

Posledním pluginem pro zpracovávání generovaných informací je *SmokeTest plugin*. Rozdílem pro tento plugin je, že nepracuje se žádnou stromovou strukturou složek, nýbrž přistupuje k datům z externí databáze a tyto data potom dále zpracovává. Jedná se o databázi, ve které jsou pro každou komponentu uloženy výsledné informace ze SmokeTestů.

Plugin pracuje v několika krocích. Nejprve si z databáze, kde jsou ukládány zpracovaná data, vybere všechny balíčky. Pro rozlišení se někdy tato databáze bude označovat jako *Storage*. V dalším kroku plugin přistupuje do *SmokeTest* databáze a vyhledává *SmokeTest data*, která jsou vázána k těmto balíčkům. Po vrácení výsledků z databáze jsou *SmokeTest data* ukládána podle následujících pravidel.

Pokud již jsou *SmokeTest data* nějakého balíčku uložena ve *Strorage* dojde jen k aktualizaci parametru *average percentage* (průměr procentuální úspěšnosti SmokeTestů). Tento parametr je potřeba aktualizovat z toho důvodu, že pro daný balíček mohly od doby uložení *SmokeTest dat* přibít nově provedené SmokeTesty a parametr *average percentage* mohl být změněn. Naopak pokud *SmokeTest data* nejsou ve *Storage*, jsou do ní pro daný balíček uložena. Tento plugin je tedy přímo závislý na zpracovaných datech z *Build data pluginu*, podle nich získává *SmokeTest data* a aktualizuje parametr *average percentage*.

### 4.1.6 Notifications modul

Speciálním případem mezi všemi pluginy je *Notifications modul*. Nejedná se v tomto případě úplně tak o plugin, ale o modul, který však má odlišné vlastnosti. Modul se připojuje k *Engine service*, která ho spouští a kontroluje jeho běh, stejně jako to je u pluginu. Hlavní rozdíl oproti pluginu je ten, že modul neukládá data do *Storage* pomocí *Engine service*, ale obsahuje vlastní funkcionalitu a data do *Storage* ukládá na přímo. Tento rozdíl je zapříčiněn tím, že *Notifications modul* neslouží ke shromažďování dat, a proto potřebuje jinou funkcionalitu pro přístup do *Storage*. Smyslem tohoto modulu je neustálá kontrola všech balíčků ze *Storage* a posílání notifikací daným uživatelům.

V této části je na místě objasnit jak fungují notifikace. V první řade je důležité zmínit, že notifikace se budou definovat ve webové části aplikace. K vytváření notifikací se tedy více dostaneme v části práce, která bude pojednávat o webové aplikaci.

Když například máme komponentu, do které patří určitý balíček a chtěli bychom znát okamžik, kdy se nám tento balíček zaintegruje do závislostí jiného balíčku. Dále bychom chtěli být notifikováni právě v tento okamžik a znát všechny balíčky, do kterých byl náš balíček zaintegrován. Navíc můžeme požadovat, abychom sledovali jen určitou destinaci, do které se nám balíček může dostat. Tato destinace je buď samotná komponenta nebo pro balíček používající *RTC* je to *stream* či pro *GIT* balíček je to *repository* a název *branche*.

Činnost *Notifications modulu* je tedy následující. Nejprve jsou ze *Storage* získány všechny vytvořené notifikace. Tyto notifikace jsou procházeny a ze *Storage* jsou filtrovány balíčky, jejichž destinace je definována v právě procházené notifikaci. Mezi závislostmi těchto balíčků je vyhledáván balíček, který chceme notifikovat. Bude se někdy nazývat opět jako zdrojový balíček, ale tentokrát v kontextu s notifikacemi. V okamžiku, kdy je zdrojový balíček nalezený v závislostech prohledávaného balíčku, je tento balíček zaznamenán. Po dokončení prohledávání mezi balíčky dané destinace je prvně do *Storage* k dané notifikaci uložen seznam nalezených balíčků. Posléze je poslána oznamovací zpráva na e-mail uživatele, který notifikaci vytvořil. Zpráva obsahuje informace o notifikaci a seznam balíčků, do kterých se zdrojový balíček dostal.

## 4.2 Vizualizace zpracovaných dat

Jsme v polovině popisu architektury probírané aplikace. Přiblížil jsem jakým způsobem se všechny generované informace o balíčcích zpracovávají. V této sekci objasním, jak docílíme vizualizace těchto informací. Připomeňme si v jaké jsme fázi. Máme všechny data připravená v určité podobě ve *Storage*. Do této *Storage* neustále přibývají nová data, která potřebujeme vzápětí zobrazit ve webové aplikaci. Pro vizualizaci budou sloužit dva hlavní moduly.

Prvním z nich je *RESTfull API server*. Tento modul představuje server webové aplikace, který bude neustále běžet na serveru, kde bude celá aplikace nasazena. Jeho hlavním úkolem je přístup k datům ze *Storage*, které dále upravuje

a připravuje je webové aplikaci.

Druhým modulem v architektuře pro vizualizaci dat je webová aplikace. Ta prezentuje informace o balíčcích a ostatních zpracovaných datech uživateli. Je ve spojení s *RESTfull API serverem*, kterému zasílá své požadavky na získání dat ze *Storage* a nazpět očekává požadovaná data.

### 4.2.1 RESTfull API server

*RESTfull API server* představuje modul, který odstiňuje webovou aplikace od práce s databází. Tento modul obsahuje kompletní funkcionalitu pro přístup k datům ze *Storage* a metody pro zpracování a filtrování mezi těmito daty. Poskytuje tak webové aplikaci rychlí přístup k datům ze *Storage*, které potřebuje právě vizualizovat. K podrobnějšímu popisu jak tento modul konkrétně funguje se dostanu v dalších částech práce.

### 4.2.2 Webová aplikace

Výstupním modulem v modulovém schématu architektury aplikace na obr. 1 je modul označený jako *Web application*. Tento modul v sobě zahrnuje potřebnou funkcionalitu pro vizualizaci všech nashromážděných dat koncovému uživateli. Tato webová aplikace běží na straně klienta. Umožňuje uživateli vyhledávat informace o jednotlivých balíčcích, které přehledně uvidí na jednom místě. Dále pak poskytuje filtrování mezi těmito daty. Poslední částí, kterou webová aplikace nabízí, je vytváření notifikací a jejich správa. K webové aplikaci se více dostanu v části, která bude popisovat uživatelskou dokumentaci.

# 5 Implementace

V následující kapitole bych chtěl popsat implementaci zmiňované aplikace. Bude se jednat o technický popis implementace každého modulu aplikace. Proberu použité technologie, které byly pro tuto aplikaci vybrány. Uvedu příklady některých důležitých částí samotné implementace a objasním celkovou funkčnost aplikace. Dostanu se i k popisu funkcionality, kterou moduly obsahují. Popíši strukturu *Storage* a podobu dat, které se do ní ukládají. Pro přehlednost popis implementace opět rozdělím na dvě části. První z nich bude část o implementaci zpracovávání dat a další částí bude úsek, kde se seznámíme s implementací vizualizace zpracovaných informací.

## 5.1 Zpracovávání generovaných dat

Pro zpracování generovaných dat, které představuje *back-end* aplikace byl použit programovací jazyk C# verze .NET Framework 4.6.1. Rozebereme si jakým způsobem jsou implementovány *Engine service* a pluginy, které jsou do ní připojeny. Popíši co v implementaci spojuje jednotlivé pluginy. Dále přiblížím pomocí čeho se pluginy připojují do *Engine service* a jak funguje jejich načítání a spouštění. Nakonec popíši také práci pluginů a o jejich výsledných datech, které zpracovávají. Ukážeme si jak se pomocí *Engine service* tyto data ukládají do *Storage*. Jako poslední rozeberu implementaci *Notifications modulu*.

### 5.1.1 Rozhraní pluginu

Každý plugin implementuje společné rozhraní, které je pojmenováno jako *IPlugin.cs*. Toto rozhraní v sobě obsahuje základní předdefinované vlastnosti a metody. K těm hlavním vlastnostem patří název pluginu (*PluginName*) identifikující plugin a vlastnost udávající vlákno pluginu (*ThreadPlugin*), ve kterém bude plugin po spuštění běžet. Hlavní metoda, kterou rozhraní definuje, je metoda, ve které bude běžet hlavní smyčka pluginu, nazývá se *DoMainProcess*.

V rozhraní pluginu najdeme také řadu speciálních vlastností, které se nazývají události. Tyto události slouží ke komunikaci s *Engine service* a budou se primárně používat pro ukládání a vyhledávání dat ve *Storage*, které bude provádět *Engine service*. Ke všem těmto vlastnostem a metodám rozhraní pluginu se dostaneme později a vysvětlíme si k čemu se budou využívat. Pro nastínění jak *IPlugin.cs* vypadá, se na něj můžete podívat ve zdrojovém kódu 1.

```
1  public interface IPlugin
2  {
3    string PluginName { get; }
4    int TimeStamp { get; set; }
5    Thread ThreadPlugin { get; set; }
6    ThreadStart TreadStartPlugin { get; set; }
7    EventHandler<WriteLogLineArgs> WriteLineEvent { get; set; }
8    EventHandler<InsertDocumentArgs> InsertDocumentEvent { get; set; }
9    EventHandler<InsertDocumentWithCheckArgs>
        InsertDocumentWithCheckEvent { get; set; }
10   EventHandler<InsertDocumentWithDepentDbArgs>
        InsertDocumentWithCheckDepentCollectionEvent { get; set; }
11   SpecialEventHandler<bool, DbCollection> FindDocumentEvent { get;
        set; }
12   SpecialEventHandler<bool, DbCollection>
        FindDocumentWithOptionalElementEvent { get; set; }
13   SpecialEventHandler<List<string>, FindSingleDocumentsArgs>
        FindSingleDocumentsEvent { get; set; }
14   SpecialEventHandler<List<string>, DbCollection>
        FindAllElementValuesEvent { get; set; }
15   EventHandler<UpdateDocumentArgs> UpdateDocumentEvent { get; set; }
16   void DoMainProcess();
17 }
```

Zdrojový kód 1: Rozhraní pluginu (IPlugin.cs)

### 5.1.2 Načítání pluginů

*Engine service* nemá v sobě pluginy definované. Plugin představuje samostatný projekt, přičemž jeho hlavní třída implementuje rozhraní pluginu (*IPlugin.cs*). Po sestavení projektu s pluginem nám vznikne knihovna *DLL*. *Engine service* má definovanou pouze cestu ke složce, která obsahuje *DLL* pluginu. Postupně načítá každou *DLL* z této složky a snaží se zjistit jestli v této knihovně existuje třída, která obsahuje rozhraní pluginu (*IPlugin.cs*). Pokud jej najde, je vytvořena instance této třídy, se kterou bude *Engine service* pomocí rozhraní pluginu pracovat. Tímto způsobem se postupně načtou všechny pluginy.

Poslední částí načítání pluginů je parsování konfigurace pro jednotlivé pluginy a přiřazení získaných hodnot každé instanci pluginu. Konfiguraci pluginu definuje soubor *PluginsConfig.xml*. Tento konfigurační soubor obsahuje pro každý plugin tyto atributy.

- *name* - Udává název pluginu, který ho identifikuje a slouží hlavně jako označení pluginu pro sledování jeho průběhu.
- *dllpath* - Určuje plnou cestu ke složce, kde se nachází knihovna *DLL* pluginu.
- *enabled* - Definuje pravdivostí hodnotu, která udává jestli je povoleno načtení a používání daného pluginu či nikoli. Plugin s hodnotou atributu

*enabled* nastavenou na *false* nebude *Engine service* načítat neboli tento plugin bude vypnutý.

- *timestamp* - Tento atribut udává dobu čekání po skončení jednoho běhu pluginu v jeho hlavní smyčce. Hodnota se uvádí v sekundách.

Po načtení a přiřazení konfiguračních hodnot pro všechny pluginy jsou instance povolených pluginů předány *Engine service*. Načtení pluginů zobrazuje níže uvedený zdrojový kód 2. Část z načítání konfigurace pro pluginy je z důvody jednoduchosti v tomto zdrojovém kódu vynechána a celkově je tento kód zestručněn.

```
1  public List<IPlugin> LoadPlugins(string path)
2  {
3    string[] dllFileNames = Directory.GetFiles(path, "*.dll");
4    ICollection<Assembly> assemblies = new List<Assembly>(dllFileNames
         .Length);
5    foreach (string dllFileName in dllFileNames)
6      {
7        AssemblyName assemblyName = AssemblyName.GetAssemblyName(
             dllFileName);
8        Assembly assembly = Assembly.Load(assemblyName);
9        assemblies.Add(assembly);
10     }
11     Type pluginType = typeof(IPlugin);
12     ICollection<Type> pluginTypes = new List<Type>();
13     foreach (Assembly assembly in assemblies)
14     {
15       if (assembly != null)
16          foreach (Type type in assembly.GetTypes())
17             if (!type.IsInterface || !type.IsAbstract)
18                if (type.GetInterface(pluginType.FullName) != null)
19                   pluginTypes.Add(type);
20     }
21     List<IPlugin> plugins = new List<IPlugin>(pluginTypes.Count);
22     foreach (Type type in pluginTypes)
23     {
24       IPlugin plugin = (IPlugin)Activator.CreateInstance(type);
25       plugins.Add(plugin);
26     }
27     return plugins;
28 }
```

Zdrojový kód 2: Načítání pluginů (PluginLoader.cs)

### 5.1.3 Inicializace pluginů

Po načtení pluginů je třeba pluginy inicializovat. Ve třídě *MainEngine.cs*, která je hlavní třídou *Engine service*, jsou implementovány všechny potřebné

metody pro pluginy. Rozhraní pluginu (*IPlugin.cs*) obsahuje velké množství událostí, pomocí kterých pluginy vysílají požadavky a celkově komunikují s *Engine service*. Tyto události je nutné nastavit na konkrétní metody, které se budou v návaznosti na ně vykonávat. Inicializaci událostí z rozhraní pluginu vykonává *Engine service*, která k událostem nastavuje metody ze třídy *MainEngine.cs*.

### 5.1.4 Spouštění pluginů

Po dokončení počáteční části *Engine service*, kde probíhá načtení a inicializace pluginů, přichází na řadu spouštění pluginů. Je vytvořeno vlákno pro každý plugin, ve kterém je nastavena a následně i spuštěna hlavní metoda pluginu (*DoMainProcess*). Plugin tak vstoupí do své hlavní smyčky, ve které bude neustále běžet a vykonávat svoji práci. Příklad, který ukazuje vytvoření vlákna pluginu a jeho spuštění, najdete ve zdrojovém kódu 3.

```
1 public void StartEngine()
2 {
3   foreach (IPlugin plugin in Plugins)
4   {
5     plugin.TreadStartPlugin = plugin.DoMainProcess;
6     plugin.ThreadPlugin = new Thread(plugin.TreadStartPlugin);
7     plugin.ThreadPlugin.Start();
8   }
9 }
```

Zdrojový kód 3: Spouštění pluginů (MainEngine.cs)

### 5.1.5 Hlavní smyčka pluginu

Nyní bych chtěl věnovat pozornost hlavní smyčce pluginu. V předchozím textu jsem uvedl, že metoda rozhraní pluginu (*IPlugin.cs*), ve které bude běžet hlavní smyčka pluginu, se nazývá *DoMainProcess*. Smyčka pluginu je v této metodě konstruována jako nekonečný cyklus, který je ošetřen proti neočekávaným selháním funkcionality uvnitř tohoto cyklu. Pokud k takovému selhání dojde, nastane pouze zachycení těchto chyb. Následně je smyčka uspána tak jako při dokončení jednoho běhu smyčky. Posléze je smyčka obnovena a pokračuje se dalším během cyklu. Výhody tohoto opatření si můžeme snadno představit na následujícím příkladu.

Budou se získávat *build data* z úložiště *HoldingArea*, které je fyzicky definováno jako síťové úložiště. Například při problémech v připojení k tomuto úložišti nebo při nedostupnosti úložiště z důvodu jejího přetížení může docházet k neočekávaným chybám při získávání dat. Proto hlavní smyčka pluginu musí být připravena na vše a nikdy nesmí dojít k jejímu zastavení.

### 5.1.6 Práce pluginů a jejich výsledná data

O tom jak pracují a jaké data zpracovávají jednotlivé pluginy jsme si řekli už v kapitole, která popisuje architekturu aplikace. Proto zde nebudu opět detailně rozvádět každý plugin. Chtěl bych jen zdůraznit, že jakmile doběhne jeden běh smyčky pluginu a data jsou uložena do databáze, následuje další běh smyčky. V něm se zpracují nová data vygenerovaná vývojovým procesem.

Implementace první třech pluginů je v jistém smyslu podobná. Vnořenými cykly se iteruje ve stromové struktuře složek a zpracovávají se data. V případě *Build data pluginu* se parsují soubory *build_info.xml*, *package.build.xml* a *project.build.xml*. Záměrem je data získané z těchto souborů dostat do podoby modelové třídy. U *build dat* je to třída *Package.cs*, která reprezentuje balíček. Můžete se na ni podívat níže ve zdrojovém kódu 4. Třída *Package.cs* obsahuje pouze položky, které jsem popisoval u analýzy generovaných dat pro balíček a pro jeho soubory. Navíc je v ní jen položka *SourceType*, která udává zda jde o *RTC* nebo *GIT* balíček. Třída *Package.cs* v sobě definuje i třídu *Component.cs*, za pomoci které je zkonstruován strom závislostí balíčku.

Pro *SmokeTest plugin* je implementace trochu odlišná. Při každém běhu cyklu se plugin připojuje k *SmokeTest* databázi. Konkrétně jde o *MySQL* databázi. Z této databáze získává plugin ke každému balíčku *SmokeTest data*. Více nechci implementaci pluginů rozvádět, je však k nahlédnutí v přiložených zdrojových souborech.

```
1  public class Package
2  {
3    public string Name { get; set; }
4    public string Label { get; set; }
5    public string Version { get; set; }
6    public string SourceType { get; set; }
7    public string StreamName { get; set; }
8    public string GitRepositoryUrl { get; set; }
9    public string GitBranchName { get; set; }
10   public string JenkinsJobUrl { get; set; }
11   public DateTime TimeStamp { get; set; }
12   public string FullPath { get; set; }
13   public List<Component> Dependencies { get; set; }
14   public List<Component> ProjectDependencies { get; set; }
15 }
16 public class Component
17 {
18   public string Name { get; set; }
19   public string Label { get; set; }
20   public string Version { get; set; }
21   public string PackageArea { get; set; }
22   public List<Component> ChildComponents { get; set; }
23 }
```

Zdrojový kód 4: Modelová třída balíčku (Package.cs)

### 5.1.7 Databáze a ukládání dat z pluginů

V této podkapitole bych chtěl v první řadě více popsat samotnou databázi *Storage*. Poté rozeberu jak do ní pluginy ukládají data. Pro databázi *Storage* byl použit typ databáze *NoSQL* a v závislosti na tom byla zvolena *MongoDB* databáze. Technologii *MongoDB* zde nemám v plánu více popisovat. Pouze pro účely práce je dobré vědět, že místo tabulek používá tato databáze tzv. kolekce, do kterých se vkládají dokumenty. Dokumenty pak jsou jednotlivé datové objekty.

Databáze *Storage* obsahuje celkově 5 vytvořených kolekcí. Čtyři kolekce jsou pro data ze čtyř pluginů, jejich názvy jsou podle kategorií dat, které se do nich ukládají. Poslední kolekce je pro notifikace.

Většinu funkcionality pro práci s *MongoDB* databází implementuje *Engine service*. Přesněji se tato funkcionalita nachází ve třídě *Database.cs*. Každý plugin má u sebe definován název kolekce, do které bude ukládat výsledná data.

Pokud chce plugin uložit data do *Storage*, zavolá příslušnou událost z rozhraní pluginu. Například může zavolat událost *InsertDocumentEvent*. K tomuto volání se doplní argumenty. Pro tuto událost je prvním argumentem instance pluginu, ze kterého je událost volána. Druhým argumentem je objekt s daty modelové třídy, který se převede na dokument. Posledním argumentem je název kolekce ve *Storage*. *Engine service* následně reaguje na tuto událost a vykoná příslušnou metodu, která je k této události přiřazena. V tomto případě jde o událost, která bez kontroly přidá dokument do kolekce. Pro pluginy jsou implementovány ještě další události, které například kontrolují již uložená data v databázi, nebudu tu však všechny uvádět.

*Build data plugin* ukládá svoje data o balíčcích do kolekce *PackagesFDT*. Unikátnost dokumentů se kontroluje podle kombinace názvu komponenty a *package label* (název balíčku).

Kolekce pro *Package binaries plugin* se jmenuje *PackageBinaries*, kontrola dokumentů je totožná. Tento plugin je závislý na kolekci *PackagesFDT*. Je závislý kvůli tomu, že ukládá jen záznamy, jejichž *package label* je už obsažen v určitém dokumentu v kolekci *PackagesFDT*. *Package binaries plugin* tedy navíc obsahuje název závislé kolekce, která se při ukládání dat kontroluje.

Nezávislým pluginem je *SCOTS plugin*, který odesílá data do kolekce *ScotComponents*. Poslední plugin (*SmokeTest plugin*) je opět závislý na kolekci *PackagesFDT*. Je závislý na unikátnosti dokumentu v kolekci *PackageFDT* podle položky *Version* určitého balíčku.

### 5.1.8 Notifications modul

Posledním modulem, který je zahrnut do *back-endu* aplikace, je *Notifications modul*. Tento modul nepatří až tak úplně do zpracovávání dat. A to v tom smyslu, že neshromažďuje nová data, nýbrž pracuje s daty, která už jsou zpracována ve *Storage*. Z pohledu architektury ani implementace se na tento modul nedá dívat jako na plugin a používat pro něj jeho funkcionalitu. Proto je zavedené nové rozhraní pro tuto skupinu, které bude tento modul ohraničovat.

Jedná se o rozhraní *IModule.cs*. Toto rozhraní je velmi podobné jako rozhraní *IPlugin.cs*. Dokonce obsahuje vlastnosti a metody, které jsou stejné jako u rozhraní pro plugin. Hlavním rozdílem však je, že rozhraní *IModule.cs* v sobě neobsahuje události pro práci s databází. Proč tomu tak je, jsem už dříve vysvětlil. Načítání a spouštění modulů funguje stejně jako pro pluginy. *Engine service* však nejprve načte pluginy a až po nich načte i moduly. To stejné platí i pro inicializování a spouštění.

Hlavní třídou *Notifications modulu* je třída *NotificationsProvider.cs*, která implementuje rozhraní *IModule.cs*. Kolekce ve *Storage*, s kterou tento modul pracuje, nese název *NotificationsAccounts*. V kapitole o architektuře aplikace jsem pro notifikace záměrně neuvedl jednu podstatnou věc, která se týká vytváření těchto notifikací. Budu muset předběhnout svůj výklad, protože k vytváření notifikací se celkově dostanu až v popisu webové aplikace.

Každý uživatel, který si chce nastavit nějakou notifikaci, si nejprve musí vytvořit přes webovou aplikaci tzv. *notifications account* (účet notifikací). Tyto účty jsou ukládány do kolekce *NotificationsAccounts*. *Notifications modul* v každém běhu svojí smyčky iteruje mezi těmito účty. Ve vnořeném cyklu pak prochází všechny vytvořené a zároveň aktivní notifikace pro každý účet. Aktivní notifikací je myšlena notifikace, která ještě nebyla provedena. Z databáze *Storage* a kolekce *PackageFDT* modul získá balíčky, které obsahují destinaci definovanou v dané notifikaci. Dále rekurzivně prohledává strom závislostí každého získaného balíčku. Pokud v závislostech najde balíček, který obsahuje tzv. *source component* (zdrojovou komponentu) a *source package label* (název zdrojového balíčku) uvedené v notifikaci, uloží si *package label* právě prohledávaného balíčku.

Po dokončení prohledávání všech balíčků následuje posílání notifikace uživateli. O posílání notifikace se stará třída *EmailReporter.cs*. Tato třída vytvoří obsah zprávy, kde jsou údaje o notifikaci a seznam výsledných *package labels* (názvů balíčků), ve kterých byl zdrojový balíček nalezen. Pro posílání e-mailů byl použit volně přístupný nástroj *blat.exe*. K tomuto nástroji je třeba definovat e-mailový server, přes který má zprávy posílat. Pro tento účel jsem použil e-mailový server společnosti *Thermo Fisher Scientific*. Třída *EmailReporter.cs* pro odeslání e-mailu volá aplikaci *blat.exe* s příslušnými parametry.

Pokud se jedná o notifikaci definovanou jako *first notifications* (první notifikace), je po odeslání e-mailu notifikace označena jako neaktivní a je aktualizována i ve *Storage*. Následně jsou do *Storage* k dané notifikaci přidány *package labels*, které už byly notifikovány. Tento popsaný průběh se neustále opakuje v každém běhu hlavní smyčky tohoto modulu.

## 5.2 Vizualizace dat

Pro implementaci vizualizace dat byly zvoleny poměrně novodobé technologie. Jaké to jsou, uvedu následně v obou podkapitolách. Seznámíme se v nich jak přesně funguje výměna dat mezi modulem *RESTfull API server* a webovou aplikací. Proberu všechny funkce, které nám nabízí webová aplikace a krátce se zastavím i u jejich implementace.

### 5.2.1 RESTfull API server

*RESTfull API server* je implementován pomocí technologie *Node.js*, který se používá pro psaní webových serverů. Pro webový server byl dále použit *Node.js* framework, který se nazývá *Express*. Tento framework usnadňuje vytvoření *RESTfull API* serveru.

Celý *RESTfull API server* tvoří jediný soubor, který se jmenuje *server.js*. Je v něm definováno naslouchání na určitém portu. Dále jsou v souboru *server.js* implementovány tzv. *GET* a *POST* metody, přes které se budou získávat data. Přesněji budou tyto metody sloužit pro výměnu dat mezi webovou aplikací a *RESTfull API* serverem. Tyto metody se budou volat například pro *localhost*, a to pokud budeme server testovat přímo na místě, kde bude nasazen. K tomuto volání je nutné přidat název portu, přes který se serverem komunikujeme a dále příslušnou url adresu s jejími argumenty. V samotných *GET* nebo *POST* metodách se pak data na základě vstupních argumentů získávají nebo aktualizují v databázi *Storage*.

Podívejme se nyní na uvedený zdrojový kód 5, který vrací názvy všech komponent z kolekce *PackagesFDT*. Označením *package name* je myšlen název komponenty. Dále si můžeme všimnout, že každá url adresa metody obsahuje nejprve rozlišovací uzel. Je to z toho důvodu, aby se rozlišilo s jakou kolekci ze *Storage* chceme pracovat. Obsahem metody je vždy připojení do *MongoDB* databáze a uvedení správné kolekce. Metoda v příkladu níže pouze získá jedinečné názvy komponent, které jsou v dokumentech kolekce *PackagesFDT* pod názvem položky *Name*. Na závěr jsou získaná data setříděna a odeslána jako odpověď serveru na toto volání metody. Data jsou na výstupu v JSON podobě.

```
app.get("/getpackagesfdt/getallpackagenames",
 (req, res) => {
   MongoClient.connect(url, (err, client) => {
      if (err) throw err;
      var db = client.db("Storage");
      var collection = db.collection("PackagesFDT");

      collection.distinct("Name", (err, result) => {
         if (err) throw err;
         res.send(sortArrayAscending(result));
         client.close();
      });
   });
});
```

Zdrojový kód 5: Získání názvů komponent (server.js)

### 5.2.2 Package portal

Název webové aplikace pro vizualizaci dat, jak může být již z pojmenování kapitoly zřejmé, nese název *Package portal*. Pro webovou aplikaci *Package portal* byla zvolena novodobá technologie pro vývoj webových aplikací. Jedná se o javascriptový framework, který se jmenuje *Angular*. Pro implementaci byla použita verze frameworku *Angular 6.1.7*. Tuto technologii opět nebudu popisovat dopodrobna. Pouze pro pochopení rozložení webové aplikace chci uvést, že *Angular* rozděluje jednotlivé části a pohledy webové aplikace do tzv. komponent. Komponenta potom obsahuje svůj *HTML* kód a logickou část, pro kterou se používá jazyk *typescript*.

Než se dostanu k jednotlivým komponentám a funkcionalitě webové aplikace, je nutné nejprve vysvětlit jak *Package portal* komunikuje s *RESTfull API* serverem. Tuto komunikaci má na starost tzv. *angular service*. Konkrétně je tato http služba implementována v souboru *http-rest.service.ts*. Tento soubor obsahuje metody, které pomocí tzv. http klientu komunikují se serverem. V této komunikaci dochází k výměně dat. V argumentech url se při volání serveru posílají vstupní hodnoty z webové aplikace. Většinou jsou to hodnoty, které zadal přímo uživatel. Tomuto volání se říká tzv. *http request*. Jako odpověď od serveru http služba dostane tzv. *http response*, který v sobě obsahuje výsledná JSON data. Tyto JSON data jsou předána komponentě, která o ně žádala. Ta je dále zpracuje do výsledné podoby pro uživatele.

*Package portal* je rozdělen na čtyři hlavní komponenty, které představují čtyři základní pohledy webové aplikace. Jak tyhle pohledy vizuálně vypadají bude ukázáno v kapitole, která bude popisovat uživatelskou dokumentaci. První komponenta zobrazuje všechny informace o balíčcích, které jsou uloženy v kolekci *PackagesFDT*. K těmto informacím komponenta připojuje *SmokeTest data* a informace o binárních souborech balíčku. Druhá komponenta implementuje funkcionalitu pro filtrování určitého balíčku mezi závislostmi všech ostatních balíčků.

Dále zobrazuje výsledky, kde je tento balíček zaintegrován. Další komponentou, která slouží také k filtrování, je komponenta pro vyhledávání binárních souborů podle jejich názvu a dalších parametrů.

Poslední komponenta slouží pro práci s notifikacemi. Obsahuje v sobě další čtyři komponenty. Ty implementují registraci uživatele, jeho přihlášení do svého účtu, vytváření notifikací a zobrazení tzv. *notifications inventory*, který uživateli poskytuje správu jeho notifikací.

Nakonec bych se ještě chtěl zastavit u bezpečnosti webové aplikace z pohledu url adresy. Pro testovací účely práce byl pro webovou aplikace použit pouze protokol http. Konečná verze webové aplikace, která se bude používat ve společnosti Thermo Fisher Scientific, však bude obsahovat zabezpečený protokol https. Tímto jsem shrnul hlavní body webové aplikace. Opět je k dispozici k nahlédnutí ve zdrojových kódech v přiloženém DVD k této práci.

# 6 Uživatelská dokumentace

V této kapitole bude popisována webová aplikace *Package portal* z uživatelského pohledu. Tato kapitola bude tedy zároveň uživatelským manuálem pro orientaci ve webové aplikaci. Webová aplikace bude popsána tak, aby uživatel pochopil všechny její funkce a možnosti. Každá důležitá část webové aplikace bude vysvětlena na ukázkových příkladech. Chtěl bych dodat, že pro všechny ukázky z webové aplikace v tomto textu byla použita testovací data. Toto opatření bylo nutné z důvodu ochrany vnitřních dat společnosti Thermo Fisher Scientific.

Když si zobrazíme aplikaci *Package portal* v prohlížeči, dostaneme se na úvodní stránku, která se jmenuje *Index*. Úvodní stránka *Index* slouží pouze jako navigace pro *Package portal*. Dostaneme se odtud na všechny hlavní sekce aplikace, o kterých se více dozvíme v následujících podkapitolách.

## 6.1 Package info

První z těchto hlavních sekcí je sekce s názvem *Package info*. Z pojmenování této sekce můžeme vytušit, že se bude jednat o hlavní část aplikace pro vizualizaci všech informací o balíčcích. Vstoupíme-li do této sekce, uvidíme, že máme na výběr seznam komponent. Když zvolíme jakou chceme komponentu, musíme dále vybrat konkrétní *package label* pro tuto komponentu. Výběr těchto dvou parametrů pro vyhledání balíčků ilustruje níže uvedený snímek z aplikace *Package portal* na obr. 3.


Obrázek 3: Výběr balíčku v aplikaci Package portal

Po kliknutí na tlačítko *generate* se zobrazí informace o příslušném balíčku. V horní části zobrazených informací najdeme název *package label* a pod ním přesný čas, kdy byl tento balíček vytvořen. Pod těmito informacemi následují čtyři sekce, které obsahují ostatní informace k danému balíčku. Všechny si je postupně projdeme a ukážeme na příkladech.

### 6.1.1 Build info

Jako první po vyhledání balíčku se zobrazí sekce s názvem *Build info*. Najdeme v ní základní informace o balíčku. V první řadě zde uvidíme rozlišení o jaký

typ balíčku se jedná z pohledu verzovacího systému. Připomenu, že rozlišujeme dva typy verzovacích systémů a to buď verzovací systém *RTC* nebo *GIT*. V příkladu níže na obr. 4 můžeme vidět, že jde o tzv. *RTC* balíček. Z toho důvodu je u něj informace o jeho *stream name*. Dále je zde uveden název komponenty, kterou jsme zadávali při vyhledávání. Následuje parametr *Version*, který udává přesnou verzi balíčku. Pod ním je uveden odkaz na *Jenkins job*. Jako poslední zde najdeme plnou cestu fyzického úložiště balíčku, odkud se všechny tyto informace zpracovaly. U obou předchozích položek jde opět o testovací data, takže uvedené cesty nevedou na reálné zdroje dat.

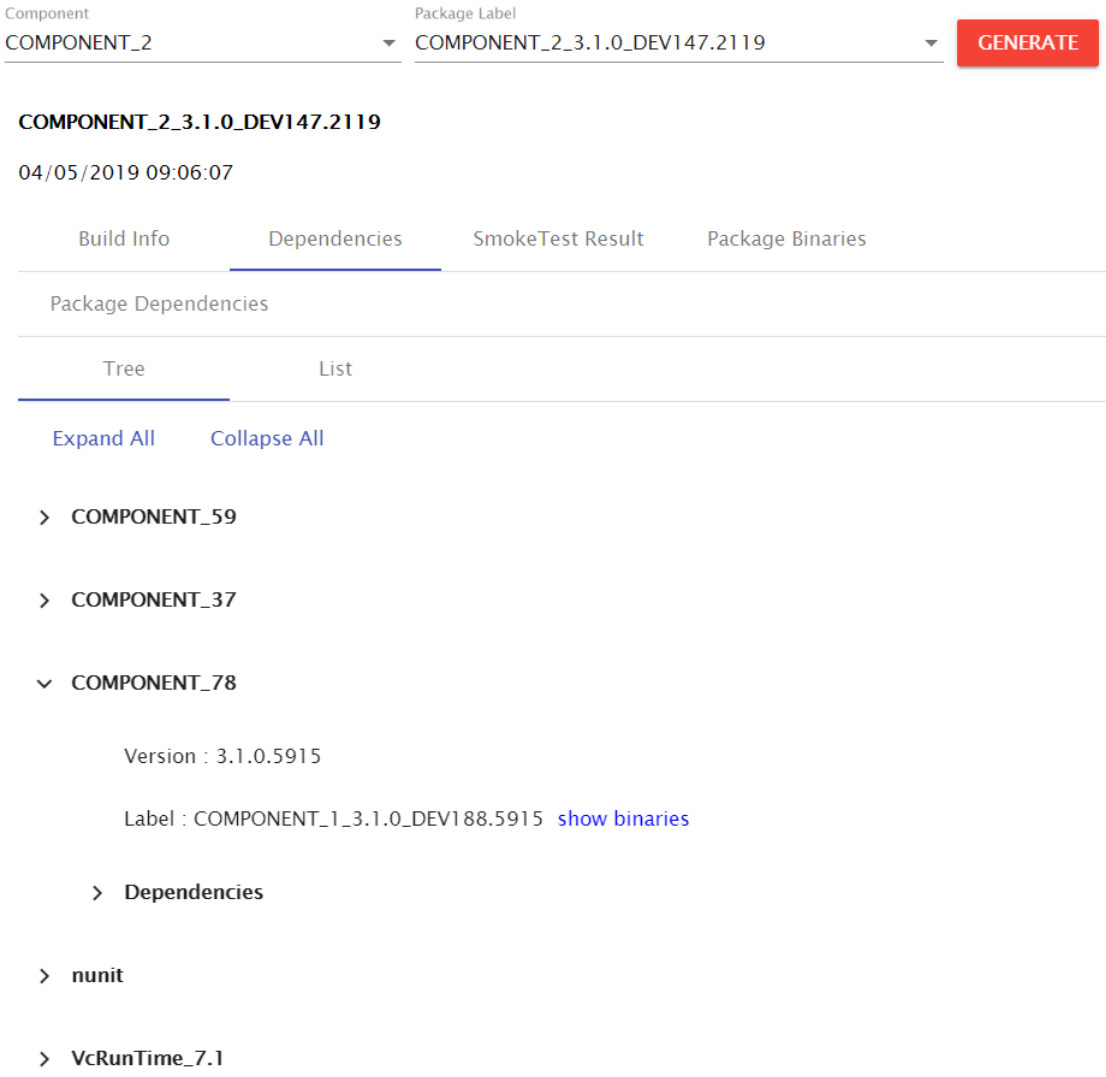

Obrázek 4: Základní informace o balíčku

### 6.1.2 Dependencies

Druhou sekcí, na kterou se můžeme přepnout, je sekce *Dependencies*. Tato sekce zobrazuje strom závislostí balíčku. V ukázce na obr. 5 můžeme vidět, že vyhledaný balíček má primárně závislosti na třech dalších komponentách a dvou *scot* komponentách. Každá ze závislých komponent obsahuje svoje další závislosti, které se zahrnují do závislostí daného balíčku. Celým tímto stromem závislostí se dá v aplikaci procházet. Jak můžeme vidět každý uzel stromu se jmenuje

podle názvu komponenty a obsahuje v sobě *version* a *package label*. Dále může každý uzel komponenty obsahovat podstrom svých závislostí.

Poslední funkcí, kterou najdeme pro všechny uzly, je možnost zobrazit si informace o binárních souborech balíčku ve stromu závislostí. Celý strom závislostí si může uživatel v kartě *List* nechat převést do seznamu všech závislých komponent. Stejná struktura je i pro projektové závislosti, které by byly uvedeny ve vedlejší podsekci jako *Project dependencies*. U většiny balíčků je však nenajdeme, proto tato podsekce většinou ani nebývá zobrazena.




Obrázek 5: Strom závislostí balíčku

### 6.1.3 SmokeTest results

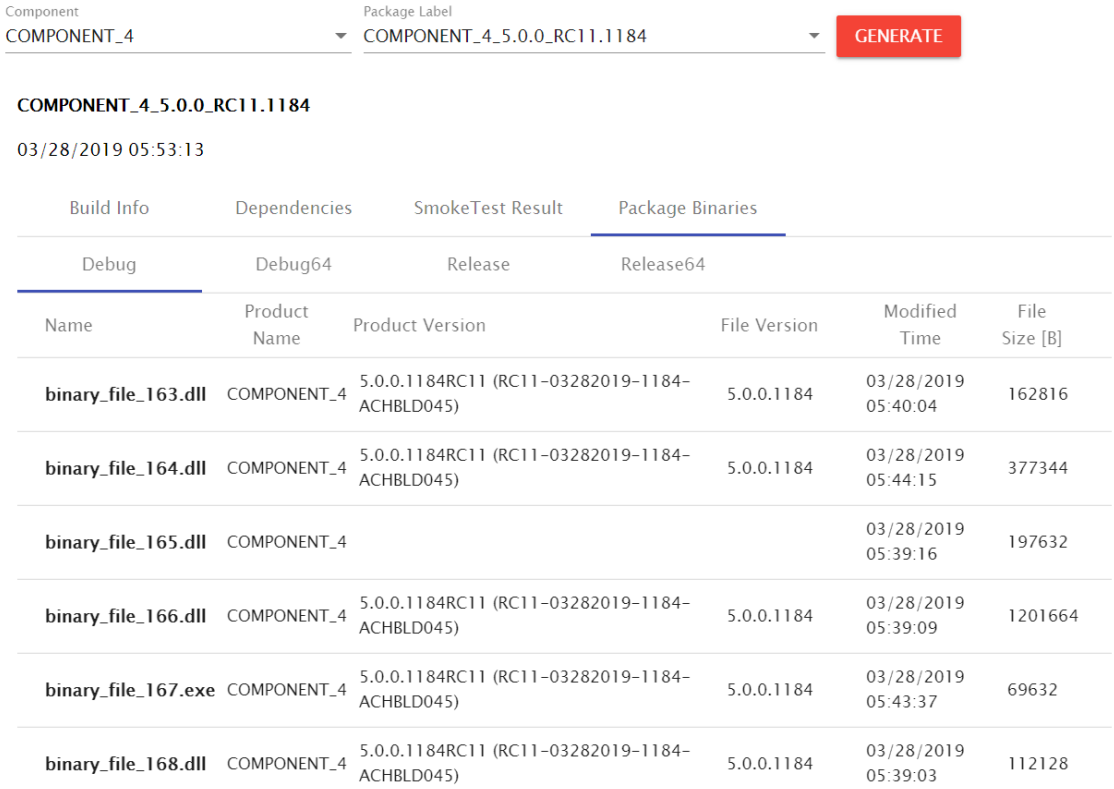
Další informací, kterou o balíčku máme, jsou *SmokeTest* výsledky. Najdeme je v další sekci informací k balíčku, jak to ukazuje obr. 6. Jsou tu pro úplnost

znovu uvedeny informace o *version* a *stream*. Potom je zde uvedena procentuální úspěšnost všech *SmokeTestů* pro daný balíček. Jako poslední je k dispozici odkaz, který vede do interní webové aplikace s názvem *Test report* společnosti Thermo Fisher Scientific. Tato webová aplikace obsahuje všechny detailní informace o *SmokeTestech*. Uživatel si tak snadno může zobrazit *SmokeTest* výsledky pro balíček, který si vyhledal. V tomto případě se jedná o testovací data, a proto odkaz na *Test report* nebude v testovací verzi aplikace fungovat.


Obrázek 6: SmokeTest výsledky pro balíček

### 6.1.4 Package binaries

Poslední skupinou informací, kterou pro balíček můžeme najít, jsou informace o jeho binárních souborech. Na snímku z aplikace, který je na obr. 7, jako první vidíme tabulku s informacemi o binárních souborech. Jsou v ní jména binárních souborů a další informace k těmto souborům. Po kliknutí na název sloupce, můžeme podle něj setřídit řádky v tabulce. Máme možnost přecházet v kartách s binárními soubory podle zvoleného *target*, pro který byl *build* balíčku proveden. Binární soubory v jednotlivých kartách jsou na první pohled totožné, mohou se však lišit například ve velikosti souborů.

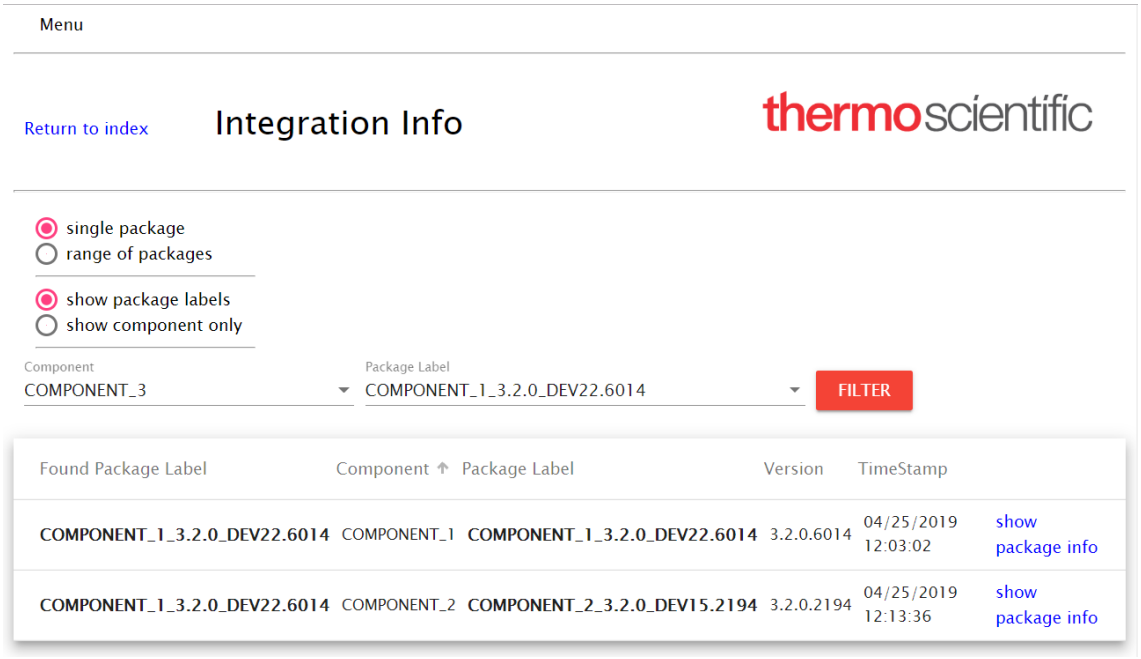



Obrázek 7: Informace o binárních souborech balíčku

## 6.2 Integration info

Jedna z dalších částí aplikace *Package portal* je sekce *Integration info*. Tato část aplikace slouží pro vyhledávání balíčků, ve kterých je zaintegrován jeden nebo více balíčků, jejichž integraci jsme chtěli nalézt. Slovem zaintegrován je myšleno, že daný balíček je obsažen v závislostech jiného balíčku. Je možné si vyhledat jen jeden určitý balíček, jako tomu je v příkladu níže na obr. 8, dále zadat i rozsah od jednoho balíčku ke druhému. Vyhledají se nám pak takové balíčky, ve kterých je zaintegrován balíček mezi zadaným rozsahem.

Ve výsledné tabulce najdeme *package label*, který označuje balíček, jehož integrace byla nalezena. Dalšími sloupci v tabule jsou informace o balíčku, ve kterém byla tato integrace nalezena. Každý řádek tabulky obsahuje i přepínací odkaz do *Package info*, kde si může uživatel zobrazit informace k balíčku s touto integrací. Uživatel si může zvolit nastavení v jakém chce dostávat výsledky vyhledávání. Učiní to pomocí přepínačů. Jedno z těchto nastavení je například to, že si uživatel zvolí jako výsledek vyhledávání jen zobrazení komponenty, ve které je balíček zaintegrován.

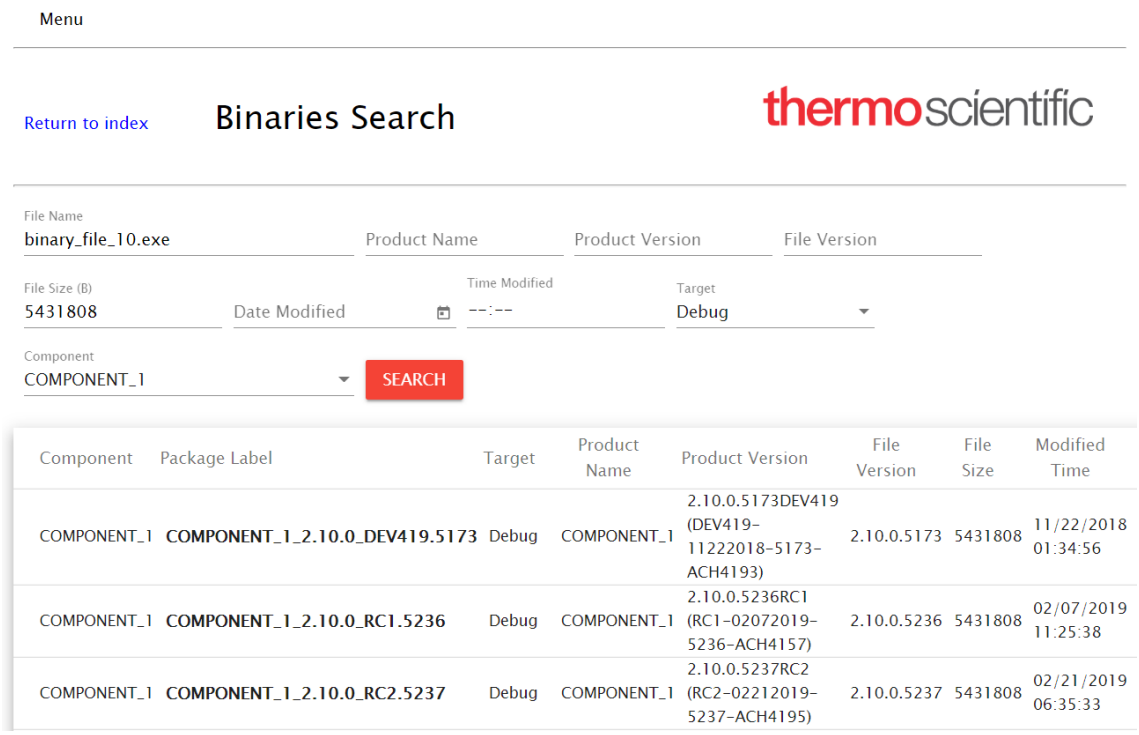



Obrázek 8: Vyhledávání integrací balíčku

## 6.3 Binaries search

Pokud uživatel zná jméno nějakého binárního souboru nebo tento soubor má fyzicky u sebe a vidí i jeho další vlastnosti, může si tento binární soubor vyhledat a zjistit tak, kde všude je zaintegrován. Pro tento účel slouží část aplikace, která se jmenuje *Binaries search*. Povinným parametrem pro vyhledávání je celé jméno binárního souboru a to i včetně jeho přípony. Ostatní vyhledávací parametry jsou volitelné. Tyto volitelné parametry pomáhají uživateli zpřesnit filtr vyhledávání. Můžeme je všechny vidět na příkladu na obr. 9.

Pokud požaduje uživatel vyhledat jen jeden konkrétní binární soubor, který má například i u sebe, nejlépe mu pomůžou parametry *File size*, *Date modified* nebo i *Time modified*. Druhým důležitým předpokladem pro vyhledávání je znalost názvu komponenty, ve které se binárních soubor nachází. Předpokládá to ale, že tuto komponentu uživatel zná, případně si pomocí ní zpřesňuje filtr pro opětovné hledání. Výsledkem tohoto vyhledávání binární souborů je opět tabulka. V ní jsou uvedeny nejprve informace o balíčku, kde byl soubor nalezen, a k nim i všechny ostatní informace k tomuto souboru.




Obrázek 9: Filtrování binárních souborů

## 6.4 Notifications

Poslední funkcí, kterou *Package portal* umožňuje, je vytváření a správa notifikací. Když se dostaneme do sekce *Notifications*, uvidíme jako první přihlašovací formulář k účtu s notifikacemi. Pokud ještě účet nemáme vytvořen, stačí se přepnout do formuláře pro založení účtu. Uvedeme v něm jen svoji e-mailovou adresu, na kterou chceme, aby se nám notifikace zasílaly, a heslo k svému účtu.

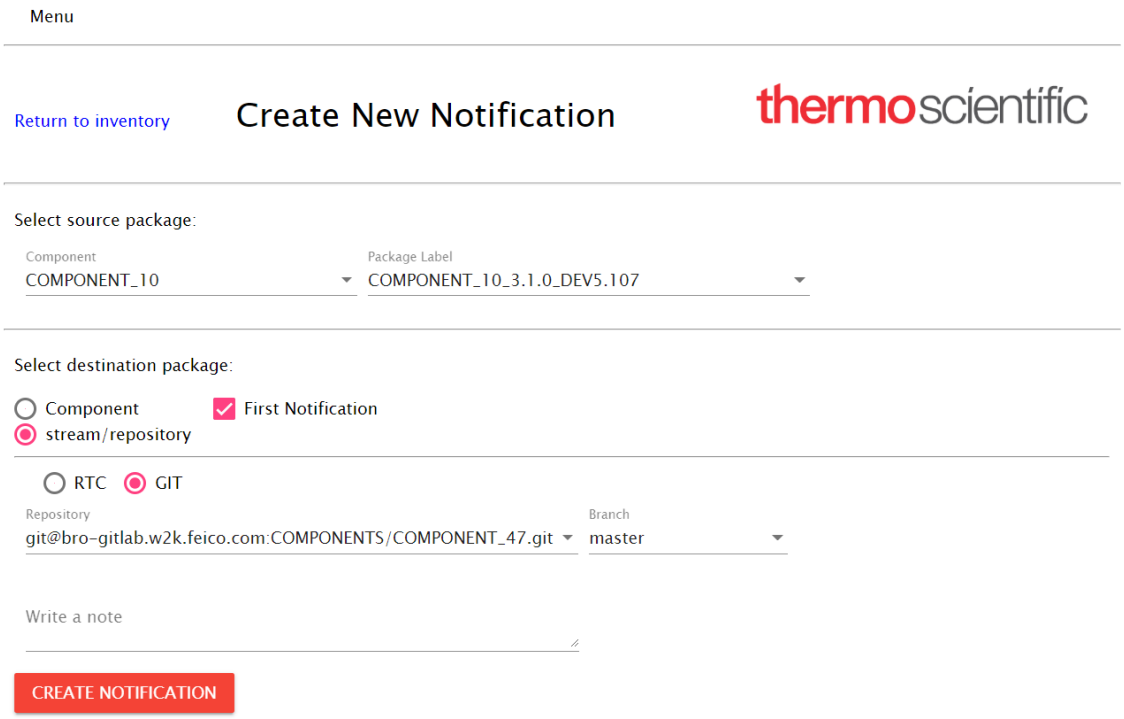
Jakmile se zaregistrujeme a přihlásíme, budeme přesměrováni do svého účtu s notifikacemi. Pokud jsme si založili nový účet nebo ještě nemáme žádné notifikace vytvořeny, dostaneme se jen do prázdné stránky s tlačítkem pro přidání nové notifikace a s naší e-mailovou adresou k účtu.

### 6.4.1 Create new notifications

Tlačítko pro přidání nové notifikace nás přesměruje do sekce pojmenované jako *Create new notifications*. Příklad pro tuto sekci můžeme vidět na obr. 10. Nejprve musíme vybrat tzv. *source package* (zdrojový balíček), který chceme sledovat. Přesněji musíme vybrat jeho komponentu a *package label*.

Na druhém místě musíme zvolit destinaci, do které očekáváme, že se nám balíček dostane a chceme být o tom notifikováni. Pomocí přepínačů si můžeme vybrat jestli půjde o destinaci zahrnující komponentu, *stream* nebo *repository* a název *branche*. Dále můžeme nastavit, zda se bude jednat o tzv. první notifikaci či nikoli. Na konci můžeme ještě volitelně vyplnit poznámku, kterou bude obsahovat

zpráva s notifikací. Pak už jen vše potvrdíme pomocí tlačítka *create notification* a budeme přesměrováni nazpět do našeho účtu, kde vytvořenou notifikaci uvidíme.




Obrázek 10: Vytvoření nové notifikace

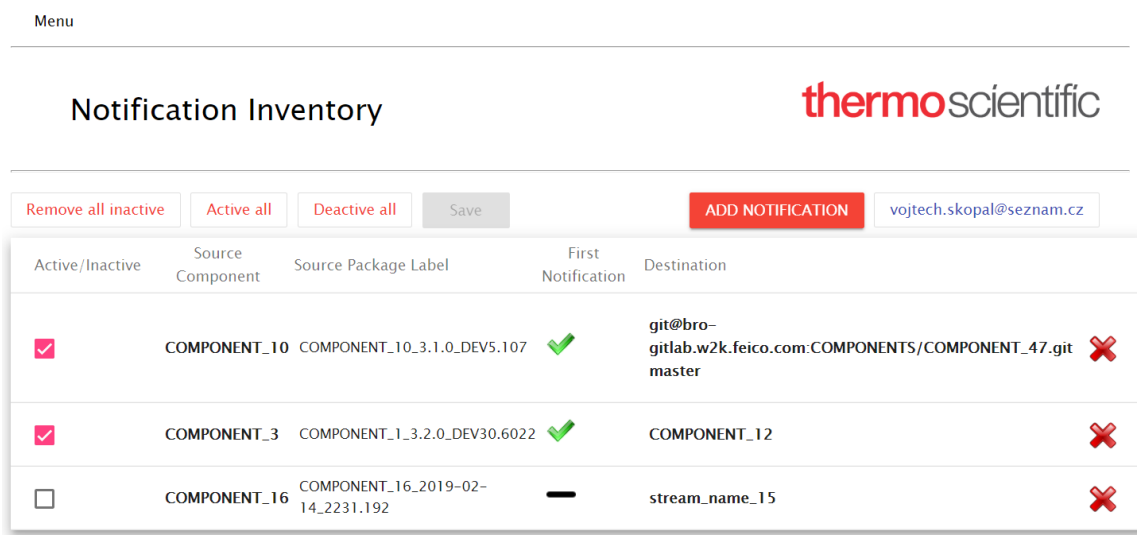
### 6.4.2 Notifications inventory

Do prostředí našeho účtu nám přibude nově vytvořená notifikace. Toto prostředí se nazývá *Notifications inventory*. Vidíme zde všechny naše doposud vytvořené notifikace. Můžeme pro ně používat následující operace.

Pro každou notifikaci jde nastavit zda bude v aktivním stavu či nikoli. Pro dokončení nastavení aktivního či neaktivního stavu je třeba vše uložit tlačítkem *save*. Nad tabulkou notifikací najdeme ještě další pomocné tlačítka pro správu našich notifikací. Každou notifikaci můžeme také smazat pomocí mazací ikony na konci každého řádku tabulky. Příklad *Notifications inventory* najdeme na obr. 11 pod tímto textem. Notifikace, které jsou pro tento příklad vytvořeny, nemají hlubší souvislost, takže je nemůžeme brát jako notifikace, které by mohly nastat.

Všechny vytvořené notifikace v *Notifications inventory* mají jen informativní charakter pro uživatele. Ten je může spravovat a mít tak přehled o notifikacích které vytvořil. Není požadavkem společnosti Thermo Fisher Scientific, aby mezi vytvořenými notifikacemi byly i informace o jejich provedení a o zaintegrovaných balíčcích, které v nich byly definovány. Tuto informativní vlastnost však

poskytují e-mailové zprávy s notifikacemi. Jednotlivé integrace balíčků, ke kterým došlo, si pak může uživatel dohledat v hlavních sekcí *Package info* nebo *Integration info*.




Obrázek 11: Přehled vytvořených notifikací

# Závěr

Cílem bakalářské práce bylo zpracovat a sjednotit data vývojového procesu softwarového produktu společnosti Thermo Fisher Scientific. Následně bylo třeba tyto data vizualizovat jako webovou aplikaci. Modul pro zpracování dat měla představovat běžící služba, která bude tyto data neustále procházet a ukládat je. Webová aplikace měla být navržena tak, aby přehledně zobrazovala všechny data patřící k sobě. Měl být kladen důraz na správný návrh celé aplikace pro její budoucí rozšíření. Výsledná aplikace obsahuje všechny výše zmíněné požadavky, podle kterých měla být vytvořena. Aplikace obsahuje i rozšiřující funkce, které byly přidány v průběhu jejího vývoje.

Seznámil jsem vás s problematikou, která vysvětluje základní pojmy potřebné pro pochopení vytvořené aplikace. Posléze jsem pojednal o architektuře aplikace a probral jsem hlavní části její implementace. V poslední části práce jsem popsal webovou část aplikace z pohledu uživatele. Nyní je třeba ještě doplnit některé informace k tvorbě, konečné podobě aplikace a jejím reálném použití.

Celkový vývoj popisované aplikace a všechny jednotlivé fáze tohoto vývoje byly konzultovány s odborníky jak ze strany společnosti Edhouse, tak i ze strany společnosti Thermo Fisher Scientific, pro kterou byla aplikace vytvořena. Nejprve musela být prokonzultována navržená architektura aplikace. Po té musely být schváleny technologie, které byly pro aplikaci vybrány. Poslední fází vývoje byly úpravy aplikace a to hlavně ve webové části. Bylo nutné webovou aplikaci přizpůsobit natolik, aby vyhovovala uživatelským potřebám, které byly kladeny od společnosti Thermo Fisher Scientific.

Po dokončení vývoje byla aplikace prezentována ve společnosti Thermo Fisher Scientific. Tato prezentace obsahovala seznámení s aplikací po uživatelské stránce a zdůraznění nových možností, které aplikace přináší. Setkal jsem se s pozitivními ohlasy v reakci na vytvořenou aplikaci. Byla vyzdvižena použitelnost aplikace a její přínos v oblasti usnadnění vývoje softwarového produktu a interních platforem ve společnosti. Celkově byla aplikace zhodnocena jako přínosná aplikace pro sledování a vyhledávání informací, které generuje vývojový proces softwarového produktu. Nakonec byla aplikace včetně jejích zdrojových kódů odevzdána společnosti Thermo Fisher Scientific. Aktuálně se připravuje její oficiální nasazení a publikování do celé společnosti.

Vytvořená aplikace bude mít reálné využití hlavně pro vývojáře, vedoucí vývoje, ale i pro managery společnosti Thermo Fisher Scientific, kteří pomocí ní budou moci sledovat informace generované vývojovým procesem. Dostanou tak přehled o průběhu vývoje, díky jemuž budou moci lépe plánovat a řídit zakázky celého vývoje. Aplikaci je také možné snadno rozšířit díky její modulárnosti. V budoucnu je možné ji doplnit nebo vylepšit o nové funkce a nabídnout tak ještě komplexnější řešení aplikace.

# Conclusions

The goal of this bachelor thesis was to process and unify the data of the development process of the software product of Thermo Fisher Scientific. Subsequently, it was necessary to visualize these data as a web application. The data processing module was supposed to be a running service that would constantly crawl and store the data. The web application should be designed to clearly display all the data belonging to each other. Emphasis should be placed on the correct design of the entire application for its future expansion. The resulting application contains all the above-mentioned requirements to be created. The app also includes additional features that were added during its development.

I introduced you to an issue that explains the basic concepts needed to understand the application. Then I discussed the architecture of the application and discussed the main parts of its implementation. In the last part of the thesis I described the web part of the application from the user's point of view. Now we need to add some information about the creation, the final application and its real use.

The overall development of the described application and all the different stages of this development have been consulted by experts from both Edhouse and Thermo Fisher Scientific for which the application was created. First, the proposed application architecture had to be consulted. Afterwards, the technologies that were selected for the application had to be approved. The last stage of the development was the modification of the application, mainly in the web part. It was necessary to customize the web application to meet the needs of Thermo Fisher Scientific.

After development, the application was presented at Thermo Fisher Scientific. This presentation featured a user-friendly application and highlighted the new features that the application brings. I met with positive feedback in response to the created application. The applicability of the application and its contribution to facilitating the development of software product and internal platforms in the company have been highlighted. Overall, the application has been evaluated as a beneficial information tracking and retrieval application generated by the software product development process. Finally, the application, including its source code, was handed over to Thermo Fisher Scientific. It is currently preparing its official deployment and publishing to the entire company.

The created application will be use mainly for developers, development leaders, but also for Thermo Fisher Scientific managers who will be able to track information generated by the development process. This gives them an overview of the progress of the development, allowing them to better plan and manage their development orders. The application can also be easily expanded due to its modularity. In the future, it can be supplemented or improved with new features to offer even more complex application solutions.

# A Obsah přiloženého DVD

**bin/**
: Kompletní adresářová struktura webové aplikace Package portal (v ZIP archivu) pro zkopírování na webový server. Adresář obsahuje i všechny runtime knihovny a další soubory potřebné pro bezproblémový provoz webové aplikace na webovém serveru.

**doc/**
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

**src/**
: Kompletní zdrojové texty webové aplikace Package portal (v ZIP archivu) se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření adresářové struktury pro zkopírování na webový server.

**readme.txt**
: Instrukce pro nasazení webové aplikace Package portal na webový server, včetně všech požadavků pro její bezproblémový provoz, a webová adresa, na které je aplikace nasazena pro účel testování při tvorbě posudků práce a pro účel obhajoby práce.

Navíc DVD obsahuje:

**data/**
: Testovací data pro webovou aplikaci Package portal (v ZIP archivu) pro zkopírování na webový server

**install/**
: Instalátory aplikací, runtime knihoven a jiných souborů potřebných pro provoz webové aplikace Package portal, které nejsou standardní součástí operačního systému určeného pro provoz webové aplikace.

# Bibliografie

[1] .NET DOCS, .NET Documentation [online]. [cit. 2019-05-08]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/

[2] MONGODB DOCS, MongoDB Documentation [online]. [cit. 2019-05-08]. Dostupné z: https://docs.mongodb.com/

[3] NODEJS DOCS, NodeJS Documentation [online]. [cit. 2019-05-08]. Dostupné z: https://nodejs.org/en/docs/

[4] NODEJS TUTORIAL, W3Schools Online Web Tutorials [online]. [cit. 2019-05-08]. Dostupné z: https://www.w3schools.com/nodejs/

[5] NODEJS EXPRESS FRAMEWORK TUTORIAL, TutorialsPoint Online Web Tutorials [online]. [cit. 2019-05-08]. Dostupné z: https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

[6] ANGULAR DOCS, Angular Documentation [online]. [cit. 2019-05-08]. Dostupné z: https://angular.io/docs

[7] ANGULAR MATERIAL, Angular Material Documentation [online] . [cit. 2019-05-08]. Dostupné z: https://material.angular.io/