VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MONITOROVÁNÍ VÝKONNOSTI SYSTÉMU MES PHARIS

MONITORING THE PERFORMANCE OF THE MES PHARIS SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ALEŠ ONDRÁČEK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Ondráček Aleš, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Softwarové inženýrství
Název: **Monitorování výkonnosti systému MES PHARIS**
Performance Monitoring of MES PHARIS
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte informační systém MES PHARIS. Nastudujte problematiku monitorování výkonnostních parametrů. Seznamte se s možnostmi rámce ELK Stack pro sběr, analýzu a vizualizaci výkonnostních metrik.
2. Analyzujte požadavky firmy UNIS na monitorování výkonnostních dat. Navrhněte rozšíření stávajícího výrobního informačního systému i rozšíření procesu jeho vývoje umožňující monitorování výkonnostních metrik. Zaměřte se monitorování celkových i dílčích částí provádění úloh. Navrhněte vhodný způsob reportování a vizualizace dat.
3. Implementujte monitorování výkonnostních metrik částečně jako rozšíření stávajícího systému MES PHARIS a částečně s využitím ELK Stack.
4. Vytvořte automatické testy pro získání referenčních výkonnostních parametrů a porovnání aktuálního stavu s referenčními hodnotami.

Literatura:

- Peter Farrell-Vinay. "Manage Software Testing," *Auerbach Publications*. 2008. ISBN-13: 978-0-8493-9383-9.
- T. C. Chieu and L. Zeng, "Real-time Performance Monitoring for an Enterprise Information Management System," *2008 IEEE International Conference on e-Business Engineering*, 2008, pp. 429-434, doi: 10.1109/ICEBE.2008.93.
- Domovská stránka projektu ELK Stack. https://www.elastic.co/elastic-stack/

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Tato diplomová práce se zabývá monitorováním výkonnosti automatizovaných procesů vývoje a výkonnostním testováním systému MES PHARIS. Hlavní náplní práce je sběr dat o úlohách prováděných na automatizačních serverech DevOps a Jenkins, zpracování těchto dat a jejich následná vizualizace. V druhé části diplomové práce je pak řešeno zpracování dat z výkonnostního testování a jejich vhodná reprezentace pomocí vizualizací. Stěžejní technologie, která je využita k řešení této problematiky, je ELK Stack.

Abstract

This diploma thesis deals with the performance monitoring of automated development processes and performance testing of the MES PHARIS system. The main scope of thesis is the collection of data on tasks performed on automation servers DevOps and Jenkins, processing of this data and their subsequent visualization. The second part of the diploma thesis deals with the processing of data from performance testing and their appropriate representation using visualization. The core technology that is used is ELK Stack.

Klíčová slova

výrobní informační systémy, výkonnostní testování, monitoring automatizovaných procesů, vizualizace výsledků testů, automatizační servery, Jenkins, DevOps, ELK Stack, ElasticSearch, Logstash, Kibana, Beats, Vega, Docker, Ruby, MES Pharis

Keywords

Manufacturing Execution Systems, performance testing, automated process monitoring, test result visualization, automation servers, Jenkins, DevOps, ELK Stack, ElasticSearch, Logstash, Kibana, Beats, Vega, Docker, Ruby, MES Pharis

Citace

ONDRÁČEK, Aleš. *Monitorování výkonnosti systému MES PHARIS*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Monitorování výkonnosti systému MES PHARIS

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Aleše Smrčky. Další informace mi poskytli Ing. Martin Švéda, Ing. Josef Lola, Mgr. Martin Kosmák a Ing. Josef Konečný. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Aleš Ondráček
15. května 2022

Poděkování

V první řadě bych chtěl poděkovat panu doktorovi Aleši Smrčkovi za úžasný přístup při vedení práce a za všechny jeho rady, které mi byly k užitku. Dále pak všem zaměstnancům firmy UNIS, kteří mi poskytli důležité informace a zpětnou vazbu k mé práci.

Obsah

Kapitola 1

Úvod

Zadavatelem této diplomové práce je firma UNIS, jež mimo jiné vyvíjí informační systém MES PHARIS. Jedná se o výrobní informační systém (anglicky Manufacturing Execution System, zkráceně pak MES). Požadavkem firmy je navrhnout řešení monitorování výkonnosti MES PHARIS a automatických procesů, které jsou používány v rámci vývoje tohoto systému. Dále pak má být tento návrh realizován. Toto zadání je řešeno ve dvou souběžných diplomových pracích, jejichž části na sebe navazují. Autorem druhé diplomové práce je Bc. Martin Oháňka. Ačkoliv s Martinem Oháňkou řešíme každý jinou část celkového zadání, jsou spolu práce úzce spjaty a navzájem se o sebe opírají, proto je pravděpodobná jistá podobnost, například při obeznámení s problematikou MES systémů či při analýze požadavků firmy UNIS.

Část zadání firmy, jež se zabývá sběrem dat, která jsou vhodná pro analýzu výkonnosti systému MES PHARIS a automatizovaných procesů, je hlavní náplní diplomové práce Martina Oháňky. Mým hlavním úkolem pak byla analýza samotných dat, jejich processing a následná vizualizace.

V první části diplomové práce (kapitola 2) se čtenář seznámí s výrobními informačními systémy (anglicky Manufacturing Execution System, zkráceně MES) a jinými podnikovými systémy, jež jsou důležité pro pochopení, jak MES systémy fungují a co je jejich cílem. Další kapitola 3 v krátkosti představí některé základní pojmy, jež se používají ve výkonnostním testování. Dále tato část nastíní, jak je výkonnostní testování prováděno a jaké jsou jeho cíle. V další kapitole 4.1 se čtenář dozví o open-source projektu ELK Stack, jenž je použit pro ukládání naměřených výkonnostních metrik, analýzu nad naměřenými daty a jejich vizualizaci. Kromě nástroje ELK Stack jsou pro tuto diplomovou práci důležité i jiné technologie, které budou taktéž představeny v této kapitole.

Další kapitola 5 pak představí čtenáři požadavky firmy a mnou navržené řešení. Přímo na tuto kapitolu navazuje další část 6, ve které jsou uvedeny stěžejní problémy, se kterými jsem se potýkal při realizaci řešení. Součástí zadání bylo řešení alespoň částečně pokrýt automatickými testy. Samotné vizualizace jsou poněkud složité pro automatické testování, neboť výstupem jsou především grafy. Automatické testy tak pokrývají zpracování dat pomocí nástroje Logstash. O sadě testů která byla použita se dozvíte v kapitole 7.

Kapitola 2

Informační systém MES PHARIS

V této kapitole bude čtenář seznámen s informačním systémem MES PHARIS, jenž je vyvíjen společností UNIS, a.s. Jedná se o společnost, jež se zabývá především realizací projektů v investiční výstavbě v oborech jako je zpracovávání ropy a zemního plynu, petrochemie, chemie či energetika. Mimo jiné se ale zabývá i vývojem softwaru, např. výrobního informačního systému MES PHARIS, jehož výkonnostní testování a pozorování výkonnosti automatizovaných procesů, které jsou používány při vývoji tohoto systému, je náplní mé diplomové práce.

Účelem této kapitoly není seznámit čtenáře jen s MES PHARIS, ale také s MES systémy obecně. K uvedení do problematiky MES systému je nutné čtenáře v krátkosti obeznámit i s pojmy jako jsou MIS (manažerské informační systémy) a ERP (podnikové informační systémy).

2.1 MIS – manažerské informační systémy

MIS [23] (Management Information System) jsou informační systémy sloužící ke zpracovávání dat z různých zdrojů za účelem usnadnění rozhodování managementu podniku. Toho dosahují pomocí konverze dat na poznatky. Ty mohou být využity v procesu rozhodování, plánování, vytváření strategií či monitorování. Správně fungující systém by pak měl včasně informovat management o výkonnosti podniku, nových trendech a pohybu na trhu či v okolí firmy. Základní struktura manažerského systému je následující:

1. **Extrakční zdroje** – mají za úkol provést extrakci dat z externích a interních zdrojů. Data jsou převedena a uložena ve vhodném formátu pro pozdější manipulaci a využití do databáze.

2. **Databáze** – zde jsou uchována data získaná pomocí extrakčních zdrojů.

3. **Analytické nástroje** – používají se k analýze a transformaci dat do vhodného formátu pro vizualizaci. Příkladem mohou být různé statistické metody. Dále zde bývá často využíváno vhodných ukazatelů. Nad těmito ukazateli jsou sledovány například vývojové tendence, které mohou být dále analyzovány pomocí různých algoritmů.

4. **Prezentační nástroje** – nástroje, kterými jsou data vhodně zobrazena uživateli. Jedná se například o grafy, tabulky, data v textové podobě atd. Ty jsou často organizovány do ucelených vizualizačních obrazovek (anglicky dashboard). Na základě takto vizualizovaných dat, může provádět management různá rozhodnutí.

2.2 ERP – plánování podnikových zdrojů

Zkratka ERP [21] je odvozena z anglického Enterprise Resource Planning. Do českého jazyka se pak tento pojem překládá jako plánování podnikových zdrojů nebo také podnikový informační systém. Druhé z uvedených označení však není zcela běžné a používá se zřídka. Přesto se můžeme v textech setkat s termínem podnikový informační systém jako s ekvivalentem ERP, častěji je však tento termín používán ve smyslu množiny informačních systémů, do které spadají veškeré systémy používané v podnicích.

ERP za pomoci počítačových systémů řídí a integruje všechny nebo alespoň většinu oblastí činností podniku. Mezi tyto oblasti typicky spadá plánování, marketing, nákup materiálů, finance, personalistika atd. Tyto oblasti jsou většinou v rámci podniku spravovány různými organizačními útvary. ERP pak přináší pro tyto jednotlivé organizační útvary aplikaci, jež zvládá plnit jejich potřeby. Tyto aplikace pak spolu dovedou komunikovat v rámci celého podniku, což je jednou z hlavních podstat ERP systémů. Jako učebnicový příklad se často uvádí ERP systém, který pokrývá čtyři základní oblasti, a to:

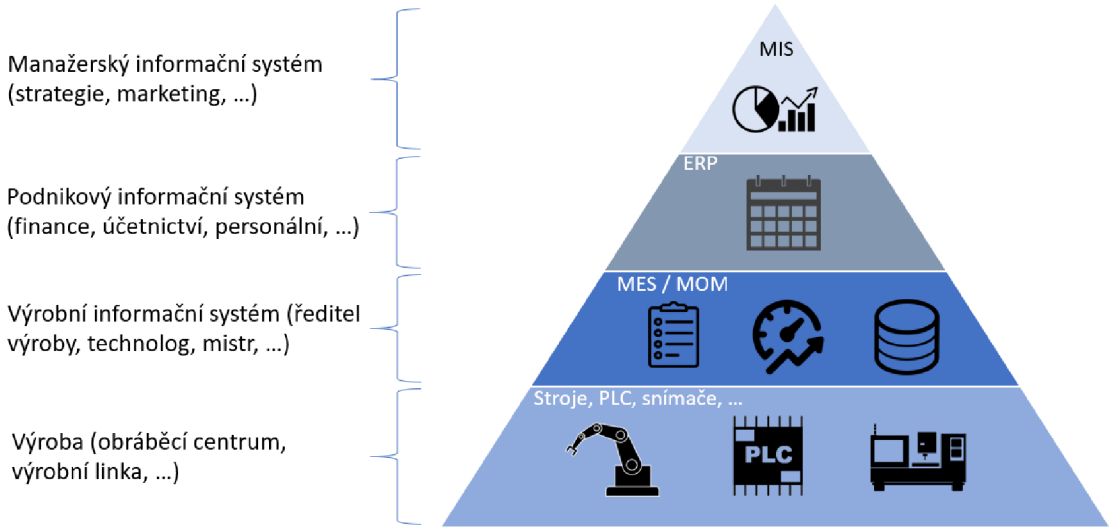
- finance,
- personalistika,
- výroba a logistika,
- marketing.

Tyto oblasti jsou pak navzájem propojené a všechny mohou využívat základních funkcionalit, jež jsou pro celý ERP systém společné. Ke každé oblasti je přiřazeno několik modulů. Jednotlivé moduly jsou schopny vykonávat konkrétnější funkcionality, které jsou pro danou oblast typické. Pokud například budeme uvažovat oblast výroby a logistiky, může se skládat z následujících modulů: Plánování výroby, Nákup a příjem zboží, Údržba strojů, Hodnocení dodavatelů, Monitoring zásob materiálu atd. Modul Plánování výroby by pak mohl poskytovat následující funkcionalitu: sběr dat ze strojů, zaplánování výroby, výpočet potřebného materiálu na určité období plánu atd. ERP systémy se využívají v různých odvětvích, a tak se jejich struktura liší na základě cílového podniku. Výše uvedená struktura tedy nemusí být nutně dodržena, žádné univerzální ERP neexistuje.

2.3 MES – výrobní informační systémy

Zkratka MES [20] je odvozena z anglického výrazu Manufacturing Execution Systems (česky se překládá jako výrobní informační systémy). Jak už název napovídá, jsou tyto počítačové systémy využívány v různých výrobních podnicích. Jejich hlavním úkolem je zvýšit efektivitu výroby. Toho dosahují prostřednictvím řízení a monitoringu výrobních procesů[1] v reálném čase, což vede k tomu, že rozhodující pracovníci (např. ředitel výroby, technolog, mistr) mohou rychle přijímat důležitá rozhodnutí či odhalit případné problémy.

MES systémy pomáhají podnikům v mnoha oblastech. Typickými příklady jsou správa výrobních postupů a příkazů, plánování a rozvrhování výroby, správa prostojů na strojích, řízení procesu kvality, dohledatelnost materiálů či OEE.[2] z těchto oblastí, do nichž MES zasahuje, lze odvodit hlavní přínos takového systému. Přínosy tedy jsou: zlepšení výrobních procesů, zajištění přesných výrobních dat a zprostředkování jednotného pohledu na ně, dohledatelnost výroby, snížení prostojů a zkrácení seřizovacích časů, zvýšení efektivity zařízení, snížení skladových zásob (je naskladněn jen nutný materiál k zaplánované výrobě), bezpapírová výroba či zhodnocení ekonomické stránky výroby. Na obrázku 2.1 je znázorněno zasazení MES systému do struktury podnikových systémů.



Obrázek 2.1: Začlenění jednotlivých systémů do struktury podnikových systémů
Zdroj: http://www.mescenter.org/cz/clanky/5-co-je-to-mes-system

[1] Jedná se o procesy, které vedou k přetvoření vstupních surovin na meziprodukty či finální výrobky.
[2] OEE jsou ukazatele efektivity zařízení.

2.3.1 Základní funkcionalita MES systémů

Níže je uvedeno sedm základních funkcionalit MES systémů dle MESA modelu[3]. Tento seznam funkcionalit opět není podmínkou a každý MES nemusí obsahovat pouze tyto funkcionality. MES systémy se využívají v různých odvětvích průmyslu, což znamená, že jsou funkcionality často implementovány na základě požadavků podniku, jenž MES využívá, a tak se často stává, že je MES o mnohé další funkcionality rozšířen, případně mohou být některé funkcionality obsažené v MESA modelu odebrány.

- **Správa výrobních zdrojů**[4] zprostředkovává přidělování a sledování zdrojů potřebných pro výrobní proces. Zda je zdroj volný se rozhoduje na základě aktuálního stavu a existující rezervace tohoto zdroje. Taktéž se musí rozhodnout, jestli je zdroj pro daný úkol možno použít (například dostatečné proškolení pracovníka).
- **Správa výrobních postupů** zprostředkovává evidenci, verzování a definici výrobních postupů (definici tvorby finálních produktů).
- **Plánování a rozvrhování výroby** je proces, během něhož se zakázky řadí do takzvané fronty práce, která definuje čas, během kterého se má na zakázce pracovat (začátek a konec), tzn. v jakém pořadí budou přiděleny výrobní zdroje zakázce za účelem jejího zpracování. Tato fronta může být sestavena různými způsoby. Nejběžnějším způsobem je dopředné a zpětné plánování, jež je založeno na jednoduchém algoritmu, který počítá s prioritami jednotlivých zakázek, podle nichž je řadí. Dnes se však již hojně využívají genetické algoritmy, popřípadě různé techniky strojového učení.
- **Řízení a monitoring výroby** spočívá v řízení toku výroby. Jedná se tedy o aktivity jako je přidělování práce jednotlivým strojům a osobám, přidělování a zajišťování materiálů, odhalování problémů při výrobě a jejich řešení (například výpadky). Taktéž se musí monitorovat stavy zakázek, aktuální stav výroby atd. Tato část je velice úzce svázána s ERP systémy.
- **Sběr dat** se výrazně liší na základě druhu podniku, pro nějž je MES implementován. Typicky se jedná například o stavy výrobních zařízení, jejich cykly, výrobní data atd.
- Každý MES by měl zaštiťovat **sledování výrobků a jejich rodokmenu**, a to nejen z důvodu legislativy, ale například při reklamacích produktu (dohledávání pracovníka) nebo auditů. Tato funkcionalita obnáší shromažďování a poskytování informací o výrobních zdrojích, jež byly u daného finálního produktu využity pro jeho dokončení. Shromažďují se tedy například data o použitém materiálu, jeho spotřeba, stroj, na němž byl produkt vytvořen, použité meziprodukty, zaměstnanci, již se na výrobě podíleli atd.
- **Výkonnostní analýzy** jsou používány k zhodnocení úspěšnosti podniku v jednotlivých oblastech celého výrobního procesu za pomoci různých ukazatelů, které se běžně u různých podniků liší. Jedním z nejběžnějších ukazatelů je již výše zmíněné OEE.

[3] Jedná se o model funkcionalit představený organizací MESA International roku 2006, je nástupcem MESA-11 modelu.

[4] Výrobní zdroje jsou například materiály, osoby, nástroje, zařízení atd.

2.4 MES PHARIS z pohledu cílového uživatele

MES PHARIS je systém vyvíjený společností UNIS. Jedná se o MES systém, který je využíván především v odvětví kovoobrábění, plastikářství a gumárenství. To znamená, že tvoří rozhraní mezi podnikovými ERP systémy (např. HELIOS, SAP, K2, NAVISION atd.) a vstřikovacími lisy či CNC stroji (viz. 2.1). Systém obsahuje veškerou funkcionalitu dle MESA modelu, který byl zmíněn v předcházející kapitole. Jeho úkolem je optimalizovat výrobní procesy počínaje zadáním a zaplánováním výroby a následným řízením výroby, dále pak sledováním jednotlivých zakázek a detailním evidováním jejich průběhu, až po závěrečné propouštění finálních výrobků či polotovarů. Dále pak elektronizace výrobní dokumentace a zajištění dokladování výrobního procesu ve formě elektronického záznamu o zakázce. Výstupy MES PHARIS se dají využít k prokazování kvality finálních výrobků, jednotlivých operací rozpracované výroby včetně dokladování použitých materiálů a zaměstnanců podílejících se na dané zakázce.

MES PHARIS je webový systém, který se skládá z modulů plnících určitou funkcionalitu. Tyto moduly si pak může zákazník zvolit dle potřeby. Nekupuje tedy pro něj zbytečné funkcionality. Zákazník si tedy může systém nakonfigurovat a zakoupit například takový systém, který zaštiťuje pouze sběr dat a nahrávání řídicích programů, nebo naopak při vyšších požadavcích na MES systém může zakoupit komplexní konfiguraci, která zaštiťuje celý výrobní proces. Moduly jsou následující:

- sběr dat z výroby,
- sledování rozpracované výroby,
- řízení výroby, odvody práce,
- KPI – klíčové výrobní ukazatele, reporty,
- vizualizace výroby,
- řízení údržby,
- kapacitní plánování a rozvrhování,
- správa a evidence nástrojů,
- centrální správa řídicích programů, DNT,
- alarmy a eskalační systém,
- řízení kvality,
- výrobní logistika, kanban, just-in-sequence.

U většiny zákazníků se pak MES PHARIS skládá ze dvou uživatelských aplikací. Hlavní aplikací, která je nedílnou součástí systému, je webový klient, který slouží k administrativě nad výrobními procesy, modely atd. Jeho částí může být i KPI, kde se dá hodnotit efektivita a kvalita výroby a produktů. Další částí jsou pak terminálové aplikace. Ty běží přímo u přístrojů ve výrobě a jsou ovládány operátory výroby. Pomocí tohoto terminálu pak operátor může například:

- přihlásit se k naplánovaným operacím,
- odvést práci (počet kompletních výrobků či polotovarů, evidují se i neshodné výrobky a typ neshody),
- zobrazit si výrobní dokumentaci potřebnou k právě probíhající operaci zakázky,
- přihlásit se k prostoji,
- zobrazit si zde predikovanou dobu na dokončení operace a další jiné aktivity.

V případě zájmu je pak možné v rámci výroby používat i mobilní WiFi zařízení (PDA) s mobilní verzí výrobního terminálu.

2.5 Vybrané technické detaily systému MES PHARIS

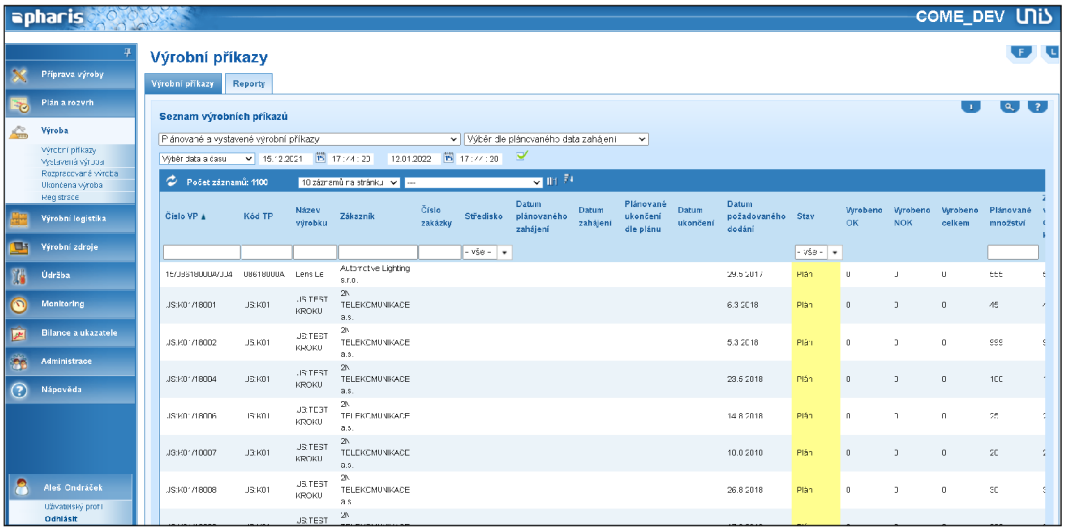
MES PHARIS je vyvíjen především v programovacím jazyce C# s využitím technologie .NET Framework. Při vývoji se používají převážně technologie firmy Microsoft. MES PHARIS je systém využívající třívrstvou architekturu.

Část MES systému, jež je určena pro vedení výroby (tzn. správu výrobních předpisů, zadávání zakázek atd.) je tvořena webovou aplikací. Webový server je hostován prostřednictvím IIS a samotná aplikace je vyvíjena pomocí technologie ASP.NET. Tato část systému MES PHARIS je implementována jako webová aplikace hned z několika důvodů. Hlavní výhodou je ale bezesporu to, že uživatel nemusí mít žádnou aplikaci a k interakci se systémem mu stačí pouze běžný webový prohlížeč. Taktéž aktualizace této části u zákazníka jsou díky tomu poměrně snadné, jelikož stačí aktualizovat aplikaci pouze na serveru, kde je MES PHARIS hostován.

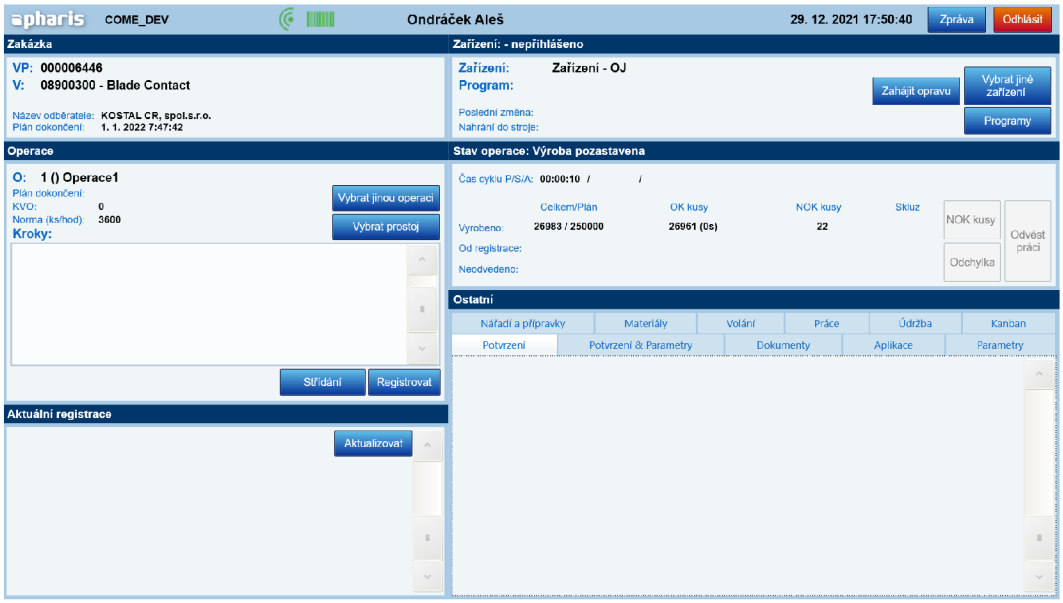
Výrobní terminálová aplikace, která je využívána operátory výroby, je desktop aplikace. Ta je navržena jak pro klasické ovládání pomocí periferních zařízení, tak i pro ovládání za pomoci dotykové obrazovky. Aktualizace těchto aplikací pak probíhá z jednoho místa, běžně ze serveru, kde je hostována i webová aplikace. Nejaktuálnějším mobilním výrobním klientem je aplikace pro Android, která se drží moderního material designu a je vyvíjená v Xamarinu. Všechny tyto aplikace jsou poměrně tenké a drtivá většina výpočtů se pak odehrává na serveru, se kterým tyto aplikace komunikují.

Součástí systému je pak také Microsoft SQL databázový server, kde jsou typicky dvě databáze, a to hlavní a historizační databáze. Hlavní databáze ukládá většinu dat, především data týkající se výroby. Historizační je zde kvůli ukládání některých dat, jež musí být historizována (např. auditrail data či procesní data zařízení ve výrobě).

Pro vývoj se využívá taktéž soubor vývojářských služeb Azure DevOps, jenž je poskytován firmou Microsoft. Azure Devops poskytuje vývojářům služby, jež mají za úkol umožnit týmu například plánovat práci, spolupracovat na vývoji kódu či sestavovat a nasazovat aplikace. Mezi využívané služby patří například Azure Repos, jež poskytuje Git, který je využíván pro správu a verzování kódu. Dále pak Azure Pipelines (pro anglický výraz pipeline v tomto smyslu bude používán výraz úloha), pomocí kterých se sestaví aplikace a vytvoří balíčky (pomocí těchto úloh se také provádí ověření sestavitelnosti při pullrequestu). Azure Boards se pak při vývoji Pharisu používá pro plánování práce, reportování chyb atd.



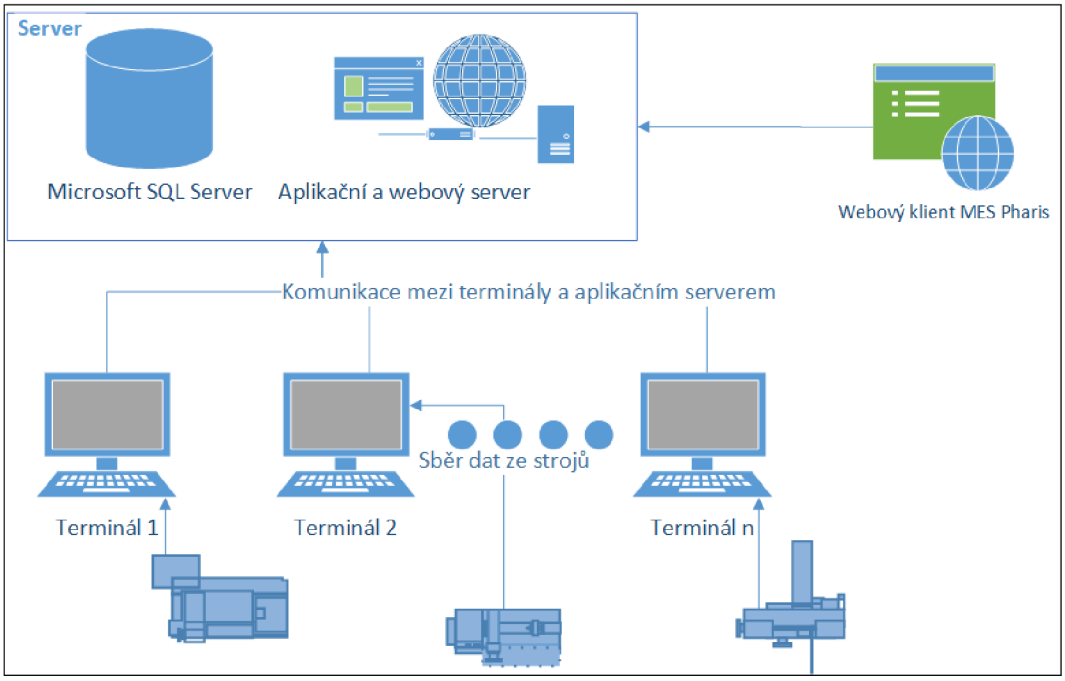
(a) MES PHARIS Webový klient



(b) MES PHARIS Terminálový klient

Obrázek 2.2: **(a)** ukázka grafického rozhraní webového klienta. Konkrétně se jedná o datagrid s výrobními příkazy. Při dostatečných oprávněních zde může uživatel provádět CRUD nad jednotlivými výrobními příkazy. **(b)** ukázka grafického rozhraní terminálového klienta. Zobrazena je obrazovka, kdy je operátor výroby přihlášen ke konkrétní operaci.

Poslední důležitou technologií, jež je pro zadání této práce dle mého pohledu důležité zmínit, je Jenkins. Jenkins je open-source projektem. Jedná se o automatizační server, pro nějž existují stovky rozšíření (angl. plugin), což ho činí použitelným při vývoji a nasazení téměř jakéhokoliv projektu. Díky tomu je tento nástroj v dnešní době jedním z nejpoužívanějších automatizačních serverů. Při vývoji MES PHARIS je používán například pro spouštění nočního sestavování aplikace a taktéž spouští automatické testy.



Obrázek 2.3: Jednoduché schéma architektury MES PHARIS

Kapitola 3

Výkonnostní testování

V česky psané literatuře se často setkáme se dvěma způsoby, jak je chápán výraz zátěžové testování. Často se používá jako podmnožina výkonnostního testování, ale stejně tak se můžeme setkat s tím, že jsou tyto dva termíny používány jako synonyma. V této práci jsem se rozhodl přistupovat k těmto výrazům právě jako k synonymům, jelikož v česky psané literatuře, jež jsem použil při studiu této problematiky, jsem se s takovým použitím setkal nejčastěji. Pro podmnožiny výkonnostních testů pak budu používat terminologii anglickou.

Výkonnostní testování [18, 19, 22, 25, 15] používá množinu testů, pomocí které například testujeme, jaká zátěž je pro systém únosná. Často se stává, že tento typ testů je firmami, jež pracují na vývoji softwaru, vnímán jako něco navíc, a není mu proto často věnována velká pozornost. Proto se v praxi stává, že výkonnostní testování se většinou aplikuje až na produkt, jenž je připraven k vydání a je nad ním dokončeno funkční testování. V dnešní době jsou tyto testy však důležité a jejich včasné zavedení může pro vývoj znamenat důležitý krok pro razantní zvýšení kvality vyvíjeného produktu. Testování výkonu se běžně zaměřuje na tyto cíle:

1. doba odezvy při volání příkazu,
2. vyhodnocení výstupu aplikace (tzn. nezáleží jen na výkonu, ale musí být kontrolován i správný chod aplikace),
3. rychlost zpracování dat,
4. využití šíře pásma sítě,
5. maximální počet uživatelů systému,
6. využití CPU,
7. využití paměti.

Ze zmíněných cílů výkonnostního testování pak lze odvodit účel takových testů. Ten se dá shrnout do následujících pěti bodů (tyto body jsou uváděny jako typický důvod k zavedení zátěžových testů, tzn. mohou existovat i jiné důvody, tyto jsou však nejběžnější):

1. měření stability systému během provozní špičky,
2. detekce úzkých míst v rámci systému,
3. ověření, zda systém splňuje výkonnostní požadavky (např. zvládá obsloužit zadaný počet uživatelů),
4. určení hardware požadavků, které jsou optimální pro očekávanou zátěž systému,
5. porovnávání mezi verzemi systémů (při implementaci nových funkcionalit či odstraňování chyb může dojít k zanesení zúženého místa).

Tyto cíle jsou však povětšinou u všech firem stejné nebo podobné natolik, že je možno standardizovat proces výkonnostního testování do několika po sobě jdoucích aktivit.

1. Prvním důležitým krokem je **určení testovacích nástrojů a prostředí.** v tomto kroku je důležité vybrat vhodné nástroje pro testování a měření. Výběr se pak odvíjí od prostředí, ve kterém budou testy spuštěny. To znamená, že je nutné udělat v rámci tohoto kroku kvalitní analýzu hardwaru, softwaru a infrastruktury testovaného systému. Dalším omezením pro výběr vhodných testovacích nástrojů mohou být již používané technologie v rámci jiných projektů, které je vhodné znovu použít kvůli jednotnosti napříč testy těchto projektů.
2. Dalším krokem je **definování přijatelných kritérií výkonu**, tzn. stanovení omezení prahových hodnot a cílů, na základě kterých budeme považovat test za úspěšný. Jednoduchým příkladem může být například přesáhnutí dané hranice používané operační paměti v průběhu testu. Pokud bude tato hranice přesažena, test bude vyhodnocen jako neúspěšný. V této fázi jsou odvozena především kritéria, jež plynou přímo ze specifikace projektu. Některé tyto hodnoty jsou pak často odvozeny až v pozdější fázi na základě referenčních hodnot odvozených z běhu testů.
3. **Plánování a design testů** je důležitý z různých důvodů. Typicky je dobré navrhnout strukturu testů tak, aby se sada mohla lehce rozšiřovat o další scénáře a aby byla přehledná.
4. **Příprava testovacího prostředí** je krok, při němž probíhá konfigurace nástrojů a prostředí, v němž budou testy vykonávány pro první běhy testů. Často je pro výkonnostní testování vymezen i vlastní hardware.
5. **Spuštění výkonnostního testování** může nastat, až je vše připraveno. Tato fáze obsahuje důležitou část, již je analýza výsledků chodu testů a vizualizace výsledků. Během této fáze se často objeví různé nedostatky v návrhu a realizaci výkonnostního testování, které by měly být opraveny.
6. **Reakce na výsledky testů** je fází, která nás vede k zavedení výkonnostního testování. V této fázi se v závislosti na výstupech testů vývojáři snaží ladit testovaný produkt tak, aby bylo dosaženo co nejlepších výkonnostních výsledků a aby byly odstraněny objevené nedostatky.

3.1 Výkonnostní parametry softwaru

V této podkapitole bude čtenář seznámen s hlavními výkonnostními parametry softwaru, tedy parametry, jež jsou během výkonnostního testování měřeny. Tyto parametry jsou důležité i pro pochopení rozdílů mezi různými typy testů, které jsou popsány v následující podkapitole (3.2).

- **Počet požadavků za periodu** typicky za jednu sekundu, je parametr, jenž nám říká, kolik požadavků uživatelů systému je systém schopen za tuto periodu přijmout a zpracovat.
- **Doba odezvy** je parametr, jenž nám říká, jak dlouhý je časový interval mezi odesláním požadavku nebo dotazu a počátkem odezvy na tuto událost.
- **Propustnost** nám říká, kolik transakcí může být provedeno v systému v daném časovém intervalu.
- **Dostupnost** nám říká, jak dostupný systém je během daného časového úseku, tedy jaké procento času je v chodu a použitelný pro uživatele.
- **Doba obrátky** je podobným parametrem jako doba odezvy, s tím rozdílem, že měřeným intervalem není pouze interval od odeslání požadavku po počátek odezvy, nýbrž až po přijmutí všech výsledků.
- **Střední doba mezi selháním** je střední dobou mezi selháním systému v jeho správné činnosti.
- **Využití zdrojů** nám říká, kolik je systémem využito zdrojů.

3.2 Rozdělení výkonnostních testů

Výkonnostní testování se dělí do několika základních skupin testů [18], a to load testů, stress testů, failover testů, soak testů, benchmark testů, volume testů a network sensitivity testů. Tyto pojmy jsou často zaměňovány. V této kapitole jsou tyto skupiny testů popsány a taktéž jsou uvedeny základní rozdíly mezi nimi.

3.2.1 Load testing

Load testing spočívá v testování chování systému při různých zátěžích. Zátěž (load) může být navyšována pomocí různých automatických toolů (více uživatelů, více požadavků atd.). Hladiny této zátěže jsou pak předem definovány. Často se používá například zátěž odpovídající průměru a špičce běžného provozu či nejvyšší zátěž, při které je zaručeno správné chování systému, což je nejzajímavější varianta a taktéž pravděpodobně nevíce využívaná. Pomocí scénářů pak můžeme testovat jen některé, pro nás zajímavé části systému. Z toho plyne, že load testy se nesnaží zahltit systém nezvladatelnou zátěží, ale pouze takovou, se kterou se vyrovná a dokáže během ní validně fungovat. Zátěž je pak pro každý běh testu generována stejně, což znamená, že i požadavky na systém by měly být totožné. To přináší poměrně velkou náročnost na přípravu takových testů, a to především při generování dat

potřebných k testu. Pomocí těchto testů lze odhalit chyby, které by jinak odhaleny být nemohly. Mezi ně patří například chyby v memory managementu, memory leaky, či přetečení bufferů. Mimo jiné se mohou tyto testy využít k výkonnostní analýze, jež nám může například poskytnout informace o počtu možných souběžných uživatelů či k určení provozních kapacit aplikace. Typicky se využívá k otestování různých infrastruktur, jestli jsou vhodné pro fungování aplikace.

Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.2 Stress testing

Stress testing je technika testování, při které se test snaží zahltit systém takovou zátěží, která může způsobit pád systému (tzn. velikost zátěže je vyšší než běžně očekávaná zátěž). Pomocí toho můžeme zjistit hranici množství výkonnostních požadavků, jak se při překročení této hranice systém zachová, popřípadě jestli je schopný se správně obnovit. Pro testery může být zajímavé například, jestli se systém zvládne obnovit do posledního správného stavu, jestli se zobrazí uživateli správný error, jestli před pádem byla uložena data, se kterými systém pracoval, zda pád nevede k bezpečnostním rizikům, jestli aplikace zamrzne či spadne atd. Toho stress testing dosáhne pomocí vysoké zátěže, již na systém klade. Na rozdíl od load testingu, kde se vyžaduje mít zátěž plně pod kontrolou a být schopen opakovat měření se stejnou zátěží, stress testing nic takového nevyžaduje, a zátěž tak může být generována náhodně. Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.3 Negative testing

Negative testing má stejné cíle jako stress testing, tzn. snažíme se dosáhnout taktéž maximálního vytížení, ale ne zvyšováním zátěže, nýbrž odebráním zdrojů systému (např. RAM, diskové úložiště, jádra procesoru atd.). To může být často podstatně jednodušší než generování zátěže. Hlavní výhodou je ale poměrně věrné navodění situace, k jaké může dojít při poruchách hardwaru. Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.4 Soak testing

Je obdobou load testingu s tím rozdílem, že test probíhá po delší časový úsek (den až desítky dnů). Tento test tak odhaluje stejné chyby, ale zvládne odhalit i ty, které se kumulují déle, než začnou způsobovat nějaký problém (např. memory leaky menšího objemu). Velkou výhodu oproti load a stress testům je pokrytí výkonnostních parametrů jako dostupnost a střední doba mezi selháním. Tyto parametry se totiž dají spolehlivě určit až při delším běhu testu. Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.5 Fail-over testing

Failover testing je výraz pro techniku testování, jež má za úkol ověřit, jak se chová záložní řešení při zhroucení systému. Tato technika není použitelná pro všechny systémy. Typicky se s tímto typem testů můžeme setkat například v oboru telekomunikací či bankovnictví, kde je potřeba mít záložní řešení.

3.2.6 Benchmark testing

Jedná se o techniku, jež má za úkol objevit úzká hrdla, tedy místa, která mohou za omezování výkonu systému. Typicky se můžou testovat jen konkrétní komponenty nebo různé podcelky systému, což vede právě k odhalení komponent, jež úzká hrdla obsahují. Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.7 Volume testing

Tato technika bývá často zaměňována s load testingem. V některé literatuře se můžeme setkat s tím, že se k volume testům přistupuje jako k podmnožině load testů. Typickým příkladem je generování vyššího množství dat, jež má testovaný systém zpracovat (například velké množství textových souborů). Díky tomu můžeme pozorovat, jak se při takovém zatížení systém chová jak z pohledu zpracovávání dat, tak i z pohledu velkého vytížení paměťové kapacity. Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.2.8 Network sensitivity testing

Network sensitivity testing má za úkol otestovat parametry síťové infrastruktury, a to především rychlost, propustnost a spolehlivost. To se nám opět může hodit při testování konkrétní infrastruktury systému (např. před nasazením systému ke konkrétnímu zákazníkovi). Tato skupina testů pak typicky pokrývá následující výkonnostní parametry:

- počet požadavků za periodu,
- dobu odezvy,
- propustnost,
- dobu obrátky,
- využití zdrojů.

3.3 Požadavky na výkonnostní testy

Požadavky na výkonnostní testy [27] se skládají z dvou základních částí, jež nemohou být nikdy vynechány. První částí je tzv. specifikace provozních podmínek během testu. Ta nám říká, jak má být nastaveno prostředí pro testování softwaru, infrastruktura systému, počet přihlášených uživatelů atd. Druhou částí je konkrétní hodnota výkonnostních parametrů softwaru. Tato hodnota by pak měla být dosažena za daných provozních podmínek (viz. první část). Jako příklad může posloužit následující zjednodušený požadavek na zátěžový test: Doba obrátky pro přihlášení uživatele při počtu dvaceti aktuálně přihlášených uživatelů je rovna 10 milisekundám. Výkonnostním parametrem je pak doba obrátky, stanovenou hodnotou výkonnostního parametru je 10 milisekund a specifikací provozních podmínek je počet přihlášených uživatelů.

Výše zmíněný požadavek je, jak jsem zmínil, pouze demonstrativní pro pochopení terminologie. V praxi je dobré se řídit tzv. SMART[1] kritérii, která nám říkají, jak by takový požadavek měl vypadat.

- **S**imple - jednoduché, krátké a výstižné
- **M**esurable - měřitelné
- **A**cceptable - přijatelné
- **R**ealistic - realistické
- **T**imeable - načasované

Z těchto kritérií lze pak odvodit charakteristiky, jenž by měly požadavky na výkonnostní testy splňovat.

- Detailní popis komponent, které mají dané testy pokrýt.
- Definování počtu uživatelů, jenž budou testem simulováni (tzn. kolik uživatelů je během testu obsluhováno).
- Definice hardwarové infrastruktury, tedy přesný popis systému z pohledu jeho zapojení a hardwarových komponent.
- Definice aplikačního transakčního mixu každé systémové komponenty (např. 20% přihlášení uživatelů, 60% odvodů, 20% odhlášení od výrobního terminálu).
- Definice systémového transakčního mixu (v rámci jednoho výkonnostního testu softwaru může být simulováno vícero zařízení, např. Terminál 1 40%, Terminál 2 40%, Terminál 3 20%).
- Definování časových požadavků pro všechny jednotlivé backendové procesy (definice maximálního přípustného času a nominálního času).

[1] Autorem tzv. SMART je George T. Doran, který definoval pětici kritérií, jimiž by se měl řídit management při definici cílů a záměrů.

Kapitola 4

Využité technologie

V této kapitole jsou stručně představeny stěžejní technologie a produkty, jenž byly použity při realizaci této diplomové práce. Každá technologie je popsána do takové hloubky, aby byl čtenář obeznámen se základními principy a využitím dané technologie. Konkrétní příklady, jak byly technologie k vypracování diplomové práce využity, jsou uvedeny v kapitole popisující implementaci (kapitola 6).

4.1 ELK Stack

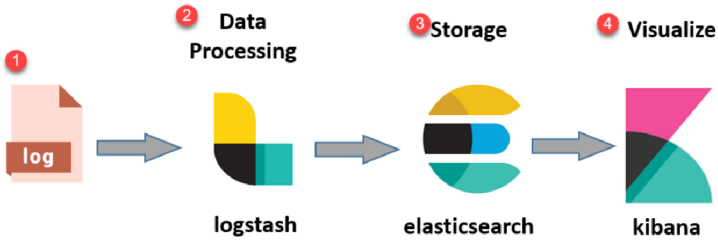
ELK stack [2] je kolekcí tří open-source nástrojů. Těmito nástroji jsou ElasticSearch, Logstash a Kinabna. Tato trojice open-source nástrojů je vyvíjena firmou nazývající se Elastic. Hlavním účelem ELK je poskytování centralizovaného záznamu událostí (dále bude používán výraz logování) za účelem identifikace problémů na straně serveru či aplikací. Hlavní výhodou je, že jsou záznamy událostí centralizovány na jedno místo, ze kterého uživatel může k těmto záznamům poměrně jednoduše přistoupit a analyzovat je. Do jedné instance ELK tak můžeme centralizovat záznamy událostí z vícero aplikací a serverů. Uživatel může jednoduše přistupovat k záznamům z pro něj zajímavých zdrojů v konkrétních časových rámcích. ELK stack uživateli také umožňuje získávat data z jakéhokoliv zdroje v libovolném formátu, nad těmito daty pak provádět analýzu a vyhledávání či různé vizualizace, a to v reálném čase.

4.1.1 Architektura ELK Stacku

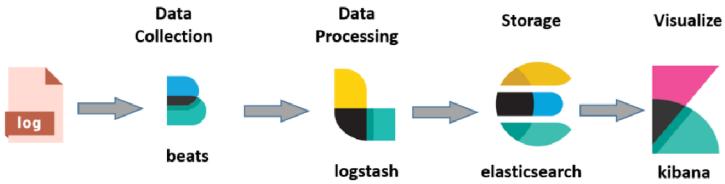
Architektura ELK Stacku se stává ze tří hlavních komponent, kterými jsou:

- **E**lastickSearch NoSQL databáze, jež slouží jako uložiště záznamů událostí
- **L**ogStash se používá pro přenos, zpracování a následné uložení záznamů událostí do ElasticSearch databáze. Záznamy, které zpracovává, mohou být libovolného formátu. Po zpracování jsou pak ve formátu, jenž podporuje ElasticSearch.
- **K**ibana je vizualizační nástroj hostovaný přes Nginx či Apache. Slouží jako rozhraní pro uživatele k vizualizaci logů, popřípadě vizualizačních obrazovek (anglicky dashboard), které mohou obsahovat různé komponenty jako grafy nad daty těchto záznamů událostí. Kibana také slouží jako GUI pro správu ELK.

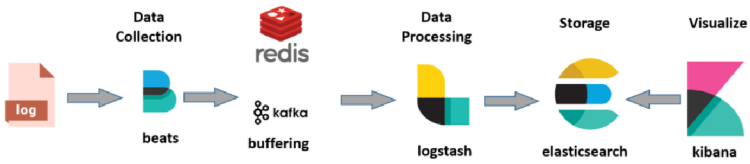
Níže jsou na obrázcích 4.1, 4.2 a 4.4 zobrazeny nejtypičtější architektury a krátké popisy, jak takové architektury fungují.



Obrázek 4.1: nejednoduší architektura, kde logy (1), které mají být zpracovány pomocí nástroje ELK Stack, jsou zaslány do Logstashe (2). Logstash pak záznamy událostí ukládá s časovou známkou a provádí transformaci a další úpravy dat. ElasticSearch (3) uloží transformovaná data, jež mu jsou předána nástrojem Logstash. Data jsou indexována. Kibana (4) pak zpřístupní pomocí GUI data v ElasticSearch NoSQL databázi (můžou být provedeny různé vizualizace, vyhledávání atd.).
Zdroj: https://www.guru99.com/elk-stack-tutorial.html



Obrázek 4.2: Tento diagram znázorňuje využití komponenty Beats. Ta je používána jako data shipper (česky by se dalo volně přeložit jako odesilatel dat) mezi zaznamenanými událostmi a Logstashem. Tato komponenta je taktéž open-source a těchto data shipperů existuje hned několik (viz. níže).
Zdroj: https://www.guru99.com/elk-stack-tutorial.html



Obrázek 4.3: Pokud je velký tok dat z Beatu do Logstashe, je vhodné použít vyrovnávací paměť (buffering). Jako vyrovnávací uložiště může posloužit například Redis či Kafka. Zdroj: https://www.guru99.com/elk-stack-tutorial.html

ElasticSearch

ElasticSearch je NoSQL databází, která funguje jako úložiště pro ELK Stack. Základem je vyhledávací engine Lucene[1] a využívá plně REST API, jehož výstup je ve formátu JSON. Jedná se o centrální úložiště záznamů událostí, nad kterým mohou být prováděny pokročilé vyhledávací dotazy, jež mohou být využity k podrobné analýze dat. Tyto dotazy jsou velice rychlé a vyhledávání se blíží k reálnému času (tzv. Near Real Time Searching). Podporuje i fulltextové vyhledávání. ElasticSearch dokáže ukládat data, jež mají určitou strukturu, ale také data, která žádné schéma nemají (tzn. celé soubory).

Důležitou vlastností nástroje ElasticSearch je, že dokáže ukládat obrovské objemy dat, nad nimiž dokáže i přes jejich velikost poměrně rychle vykonávat vyhledávací dotazy. To ho činí použitelným jako základní engine různých aplikací, které mají potřebu nad velkými daty dotazy provádět. ElasticSearch například používá i streamovací platforma Netflix.

Pro tuto práci je nutno vysvětlit základní termíny, s nimiž se můžeme při používání produktu ElasticSearch setkat:

- **Cluster** - Cluster je kolekcí uzlů (node), jež dohromady drží data. Cluster poskytuje společné indexování a vyhledávání nad daty, které jsou v něm uložena.
- **Node** - Node (uzel) je instancí ElasticSearch.
- **Index** - Index je souborem dokumentů, jež mají podobné nebo stejné vlastnosti. Dělení podle indexů je užitečné při vyhledávání, aktualizacích a mazání těchto dokumentů. Pomocí API se dá například smazat celý index nebo vyhledávat dokumenty nad jedním z indexů. Analogií indexu v klasických SQL databázových systémech pak může být tabulka. Uživatel může nadefinovat libovolné množství indexů v rámci jednoho clusteru.
- **Document** - Základní jednotka informace, jíž lze indexovat. Tato jednotka je reprezentována dvojicí klíč hodnota v JSON formátu, tedy např. `{"key":"value"}`. Každý takový dokument má pak svůj jedinečný identifikátor a typ.
- **Shard** - Za účelem distribuce dat může být každý index rozdělen do několika částí, tzv. shards. Shards jsou pak atomické části indexu, které lze rozmístit do libovolných uzlů v rámci clusteru.

[1] Jedná se o svobodný vyhledávač, jenž je vyvíjen pod záštitou Apache Software Foundation. Tento vyhledávač je pak dostupný jako knihovna a vykytuje se v mnoha jiných projektech.

Logstash

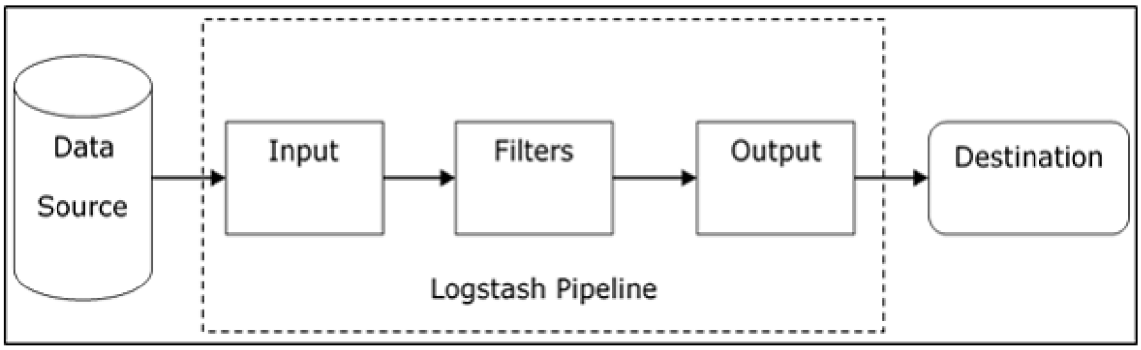
Logstash je nástrojem pro sběr dat. Prostřednictvím tohoto nástroje jsou shromažďována data z různých zdrojů, následně jsou normalizována a jejich formát upraven dle definovaných pravidel pro uložení a další používání v ElasticSearch databázi. ELK Logstash nabízí nespočet rozšíření pro zpracování různých vstupů, což ho činí použitelným téměř pro jakékoliv scénáře využití. Dokáže tedy zpracovat různě strukturovaná či nestrukturovaná data, díky čemuž může být ve většině případů použit jako jediný centralizovaný nástroj pro processing dat. Logstash může být taktéž využit k odfiltrování nepotřebných záznamů událostí. Největšími benefity nástroje Logstash je tedy možnost odfiltrovat nepotřebné záznamy, provádět zpracování záznamů událostí a možnost neřešit formát záznamů, jež jsou předávány do Logstashe, jelikož jejich formát je možno jednoduše přetvořit do podoby, která je pro další práci s záznamy událostí vhodná. Logstash se skládá ze tří hlavních komponent, kdy každá z těchto komponent má vlastní interní frontu pro vstupní data. Tyto komponenty jsou následující:

- **Input (Vstup)** – přijímá vstupní data a převádí je do formátu, se kterým dále pracuje (tzn. vytvoří takzvaný event, který je popsán dále v této podkapitole).
- **Filters (filtry)** – tato část má za úkol přetvořit data do definovaného formátu. V rámci této části jsou programátorovi dostupné různá rozšíření, která je možno pro tyto transformace použít. Nejčastěji používaným je například Grok, Prune, Mutate atd. Velice užitečné rozšíření, které jsem během psaní diplomové práce použil, je rozšíření Ruby, jenž umožní uživateli nadefinovat operace pro úpravu dat pomocí skriptovacího jazyka Ruby.
- **Output (výstup)** – tato část slouží k odeslání dat na požadovaný výstup. Výstupem může být například soubor, standardní výstup, odeslání výstupu pomocí TCP protokolu atd. Nejběžněji se však výstup posílá do ElasticSearch databáze, kde je uložen.

Průchod daty těmito komponentami je v terminologii Logstashe nazýván jako pipeline (česky zřetězená linka). Tato zřetězená linka pak slouží jako předpis, podle kterého Logstash zpracovává data. Jako hlavní objekt Logstashe je brán tzv. Event Object, který zapouzdřuje tok dat skrze jednotlivé komponenty (dále objekt události). Nejprve se do tohoto objektu události uloží vstupní data. Jeho struktura je pak měněna v závislosti na transformacích dat při průchodu filtry. Mohou být přidána například nová pole atd. Po průchodu filtry jsou pak data v rámci výstupu typicky odeslána službě ElasticSearch, kde jsou uložena. To ovšem není podmínkou. Výstup může být nastaven například i do souboru, na standardní výstup atd.

Kibana

Kibana je vizualizační nástroj, který poskytuje vývojářům GUI, v němž mohou snadno provádět analýzu a vyhledávání nad dokumenty uloženými v ElasticSearch databázi. Kibana nabízí možnost vytváření vizualizačních obrazovek (ukázka takové obrazovky je na obrázku 4.5), které mohou být sestaveny z různých komponent. Nejčastěji se setkáme s vizualizačními grafy, diagramy, mapami, tabulkami atd. Většina těchto komponent je interaktivní.

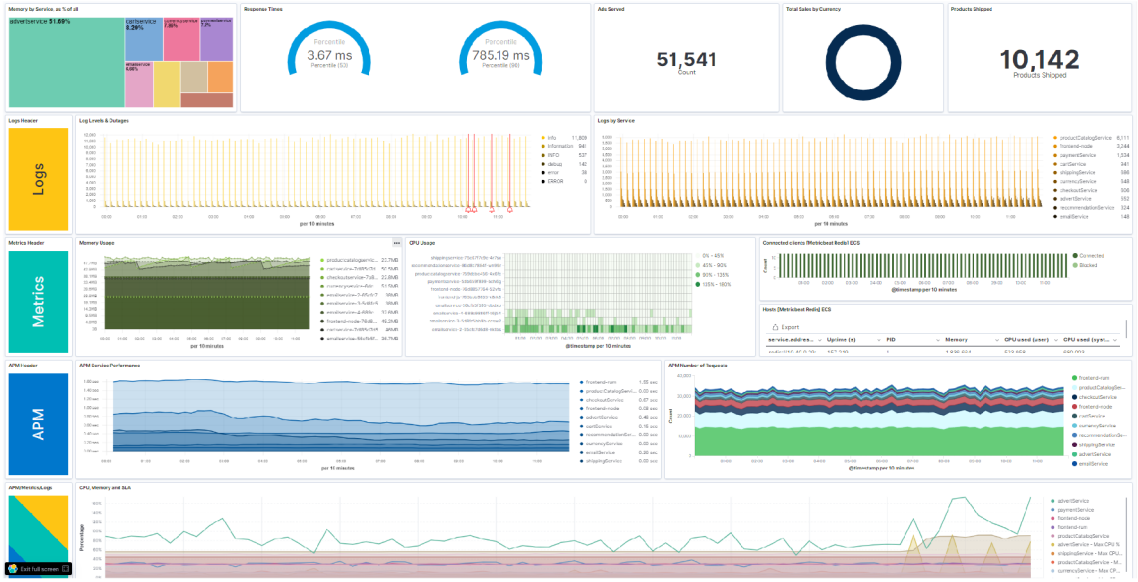


Obrázek 4.4: Logstash a schéma jeho interní struktury
Zdroj: https://www.tutorialspoint.com/Logstash/index.htm

Tyto vizualizace jsou pak prováděny nad indexovanými daty. Ačkoliv je dostupné velké množství komponent, z nichž je vizualizační obrazovky možno vytvářet, uživatel se může setkat s případy, kdy jeho data nejdou vhodně poskytnutými komponentami vizualizovat. Pro tento případ je v několika posledních verzích produktu Kibana dostupná možnost vytváření vlastních komponent prostřednictvím gramatik Vega a Vega-Lite. Tato gramatika je popsána v kapitole 4.2. Nejdůležitější součástí je ale vyhledávání, jež je důležité i pro vykreslování těchto komponent, ale i pro analýzu samostatných záznamů událostí. Kibana podporuje následující typy vyhledávání:

- **free text search** – vyhledává výskyt konkrétního textového řetězce,
- **field-level search** – vyhledává výskyt řetězce, ale je omezen na konkrétní pole (lze si představit jako sloupec tabulky),
- **proximity search** – vyhledávání řetězců či znaků, které jsou od sebe v určité vzdálenosti,
- **logical statement search** – kombinace předchozích vyhledávání, která využívá logických výrazů.

Kromě práce nad daty Kibana taktéž slouží jako GUI pro základní správu celého produktu ELK Stack. Uživatel zde například může spravovat nastavení a definice indexů, mazat indexy, nastavovat lifecycle indexu atd. Kibana taktéž poskytuje tzv. Dev Tools, jež jsou užitečné pro ladění API dotazů. Pomocí API může uživatel například mazat indexy, provádět vyhledávání nebo definovat tzv. index patterns. Index pattern je pak předpis, pomocí kterého je zpřístupněn daný index v rámci nástroje Kibana.



Obrázek 4.5: Příklad vizualizační obrazovky, jakou lze pomocí nástroje Kibana vytvořit
Zdroj: https://elastic-content-share.eu/

Beats

Beats se řadí mezi nástroje, které jsou nazývány jako data shippers. Jedná se o nástroj, který má pouze jediný úkol, a to sbírat a přeposílat data do nástroje Logstash či přímo do ElasticSearche databáze. V praxi se jedná o službu, která běží na počítači, kde má za úkol sbírat různá data (např. soubory se záznamy událostí). Podle konfigurace může zasílat tyto soubory například v časových intervalech či při změně souboru. Beats však není nástroj jako celek, ale jedná se o množinu hned několika nástrojů. Do rodiny těchto nástrojů patří následující:

- **Filebeat** – shipper pro záznamy událostí a jiná data, jež mohou být reprezentována v souborech;
- **Metricbeat** – shipper pro systémové metriky;
- **Packetbeat** – shipper pro data o síťovém připojení (například zdroje a cíle paketů, jejich obsah atd.);
- **Winlogbeat** – shipper pro záznamy událostí systému Windows;
- **Auditbeat** – shipper pro data auditního systému Linuxu;
- **Heartbeat** – shipper pro sledování provozuschopnosti (tento shipper se dotazuje na seznam zadaných URL adres, zda jsou dostupné a data o dostupnosti odesílá k dalšímu zpracování);
- **Functionbeat** – shipper pro sběr dat a monitoring cloudových služeb.

4.2 Vega

Vega [3] je vizualizační gramatika a deklarativní jazyk, pomocí kterého se dají vytvářet a sdílet interaktivní vizualizace. Tato gramatika pak využívá JavaScript knihovnu D3.js. Zdrojový kód gramatiky Vega je ve formátu JSON, prostřednictvím kterého uživatel definuje vzhled, funkcionalitu a data dané vizualizace. Jak již bylo zmíněno, Kibana podporuje psaní jednotlivých komponent právě pomocí této gramatiky. Mimo jiné je také možné použít gramatiku Vega-Lite, která je syntaxí podobná gramatice Vega, z níž je odvozena. Vega-Lite je podstatně jednodušší na psaní komponent, ale toto zjednodušení gramatiky vedlo také k tomu, že uživatel často nedocílí požadovaných výsledků. Proto je tato zjednodušená forma gramatiky vhodná spíše na jednoduché vizualizace a při vytváření složitějších interaktivních vizualizací je nutno použít gramatiku Vega.

Zdrojový kód gramatiky se skládá z několika základních bloků:

- **Data** – v tomto bloku se definují data, jež mají být prostřednictvím gramatiky vizualizována. Kibana nabízí rozšíření Vega gramatiky o možnost získávání dat prostřednictvím Rest API přímo z ElasticSearch databáze pomocí vyhledávacích dotazů (tzv. query). Více bude tato problematika vysvětlena v rámci kapitoly zabývající se realizací řešení (kapitola 6). Nad daty lze provádět různé transformace jako například výpočet statistických funkcí nad daným polem dat, řazení dat, agregace dat atd.
- **Signals** – prostřednictvím signálů lze vizualizaci udělat interaktivní. Signál může reagovat na různé události, jako je například kliknutí na různé elementy vizualizace, pohyb myši atd. Na událost může reagovat například změnou své interní hodnoty nebo v případě, pokud je Vega používána v rámci nástroje Kibana, provoláním dostupných funkcí jako je změna hodnoty filtru, změna časového rámce vizualizační obrazovky atd.
- **Scales** – v tomto bloku se definují měřítka os. Konkrétně například orientace měřítka (vertikální či horizontální), spojitost a nebo rozsah měřítka. Měřítko lze vždy provázat s konkrétními daty, a tím zajistit, že hodnoty dat budou vždy spadat do jeho rozsahu.
- **Axis** – blok, v němž jsou definovány osy. Každá osa je pak namapována na jedno měřítko z bloku scales. Mimo jiné se zde definuje například umístění osy, popis osy atd.
- **Marks** – tento blok je nejdůležitější částí definice vizualizace. V rámci bloku jsou definovány jednotlivé elementy, které většinou představují vizualizovaná data. Lze vizualizovat různé křivky, geometrické tvary, textová pole atd.

4.3 Ruby

Ruby [8] je plně objektově orientovaný skriptovací jazyk, který vytvořil vývojář japonského původu Jukihiro Macumoto už v roce 1995. Patří do skupiny interpretovaných jazyků (např. Python, Perl). Tento jazyk má poměrně snadnou syntaxi a i přes to disponuje dostatečným výkonem, se kterým je schopen dvěma výše zmíněným jazykům bez problému konkurovat. Jazyk podporuje modularitu zdrojového kódu, je multiplatformně přenositelný a datové typy jsou dynamické.

Jeho využití je poměrně široké od jednoduchých skriptů až po psaní grafických aplikací. Ruby se však nejčastěji používá pro zpracování dat v textové podobě, jelikož plně podporuje regulární výrazy. Jak již bylo zmíněno v kapitole 4.1.1, Logstash podporuje v rámci filtrů definování jednoduchého Ruby skriptu přímo do definice zřetězené linky, ale taktéž volání externího Ruby skriptu, kterému je možno předat vstupní argumenty včetně objektu události. Pomoci Ruby lze tedy v rámci nástroje Logstash poměrně snadno definovat filtry a zpracovat pomocí nich vstupní data.

4.4 Docker

Docker [6] je open-source projekt, jenž má za úkol izolovat aplikace či procesy v prostředí operačních systémů Linux, macOS a Windows. Tyto procesy izoluje do tzv. kontejnerů, které neobsahují operační systém, ale pouze požadované aplikace a soubory, které mohou být sdíleny s hostujícím OS. Z praktického hlediska se tedy jedná o formu virtualizace, při které však nevzniká žádný overhead v závislosti na OS. Výhodou oproti klasickým virtualizacím je tedy menší velikost, větší flexibilita a menší spotřeba CPU a RAM a téměř okamžitý start kontejneru. Nevýhodou je pak svázanost s hostujícím operačním systémem.

Definice toho, co má výsledný obraz (v anglické terminologii image), který se virtualizuje, obsahovat a jak má vypadat, je obsažena v textovém souboru, tzv. Dockerfile. Z každého obrazu pak současně můžeme vytvořit několik souběžně běžících virtuálních strojů, tedy kontejnerů, čímž se Docker liší od klasické virtualizace, kde je vztah obrazu a virtuálního stroje typicky v poměru jedna ku jedné. Docker je v řešení diplomové práce použit pro virtualizaci nástroje ELK Stack.

Kapitola 5

Analýza monitorování výkonnosti MES PHARIS

V této kapitole jsou shrnuty požadavky firmy UNIS na monitorování výkonnosti systému MES PHARIS, ale také na monitoring časové náročnosti automatizovaných procesů jako jsou úlohy (na serveru Jenkins je úloha nazývána anglicky, tzn. job) spouštěné na serveru Jenkins a DevOps Pipelines (dále DevOps pipelines budou označovány taktéž jako úlohy). Pro pochopení požadavků je ale důležité nejprve čtenáře seznámit se stávajícím řešením. Jedna z podkapitol se bude věnovat i konkrétním problémům, které ve firmě vznikaly kvůli absenci podobného pozorování a testování. V druhé části je pak návrh realizace tohoto monitorování, jenž je zaměřen především na analýzu a vizualizaci dat. Sběrem dat se zabývá již zmíněná diplomová práce Martina Oháňky. Požadavky a navrhované řešení jsou přehledně shrnuty v tabulkách 5.1 a 5.2. Tabulky jsou doplněny i o priority jednotlivých požadavků (číslem 1 je označena nejvyšší priorita, číslem 3 pak priorita nejnižší). Na základě těchto priorit pak bylo zvoleno, které požadavky budou v rámci diplomové práce implementovány. Tabulky zahrnují částečně i požadavky, které jsou náplní diplomové práce Martina Oháňky.

5.1 Požadavky firmy UNIS a stávající řešení

Firma UNIS má zájem o vytvoření sady výkonnostních testů a o vizualizaci výsledků těchto testů, a to především z toho důvodu, že doposud nejsou implementovány žádné výkonnostní testy. Dále pak má zájem o monitoring časové náročnosti automatizovaných procesů, které jsou spouštěny pomocí Azure DevOps Pipelines a serveru Jenkins. To zahrnuje i monitoring vytvořených artefaktů těmito procesy jako jsou například různé balíčky.

5.1.1 Monitoring časové náročnosti automatizovaných procesů

Při vývoji MES PHARIS se firma snaží co nejvíce využívat potenciál automatizačních serverů jako je například Jenkins či Pipelines, které nabízí přímo Azure Devops, jenž je využíván jak pro vedení týmu při vývoji, tak i jako distribuovaný systém správy verzí (tento produkt nabízí Git). Prostřednictvím těchto služeb jsou spouštěny automatické procesy, které se často váží na jednu z vývojových větví MES PHARIS. Hlavními vývojovými větvemi MES PHARIS jsou:

- COME_RELEASE – větev, která obsahuje poslední zveřejněnou verzi MES PHARIS. Verze systému, již tato větev obsahuje, je tedy již distribuována zákazníkům.
- COME_MAIN – větev, na které byl ukončen vývoj nových funkcionalit. Verze MES PHARIS obsažená v této větvi je ve stavu testování a opravy chyb před nasazením systému u zákazníka.
- COME_DEV – tato větev obsahuje nejvyšší verzi systému MES PHARIS a vyvíjí se na ní nové funkcionality.

Jenkins je využíván k nasazování instancí aplikací ve firmě (aplikace, na nichž je testována funkcionalita testery, popř automatickými testy), ke spouštění unit testů, GUI testů webové aplikace (psány pomocí Selenia), GUI testů terminálové aplikace (využívá Whitestack framework) či tvorbě nových instalačních balíčků. To vše zprostředkovává několik definovaných úloh, které mezi sebou mohou mít i návaznosti. Úlohy se pak spouštějí buďto na základě ručního spuštění přes GUI serveru Jenkins nebo automaticky na základě definovaných časů (popř. na základě dokončení úlohy, jež má ve svých návaznostech předdefinováno, které úlohy se mají spustit dále).

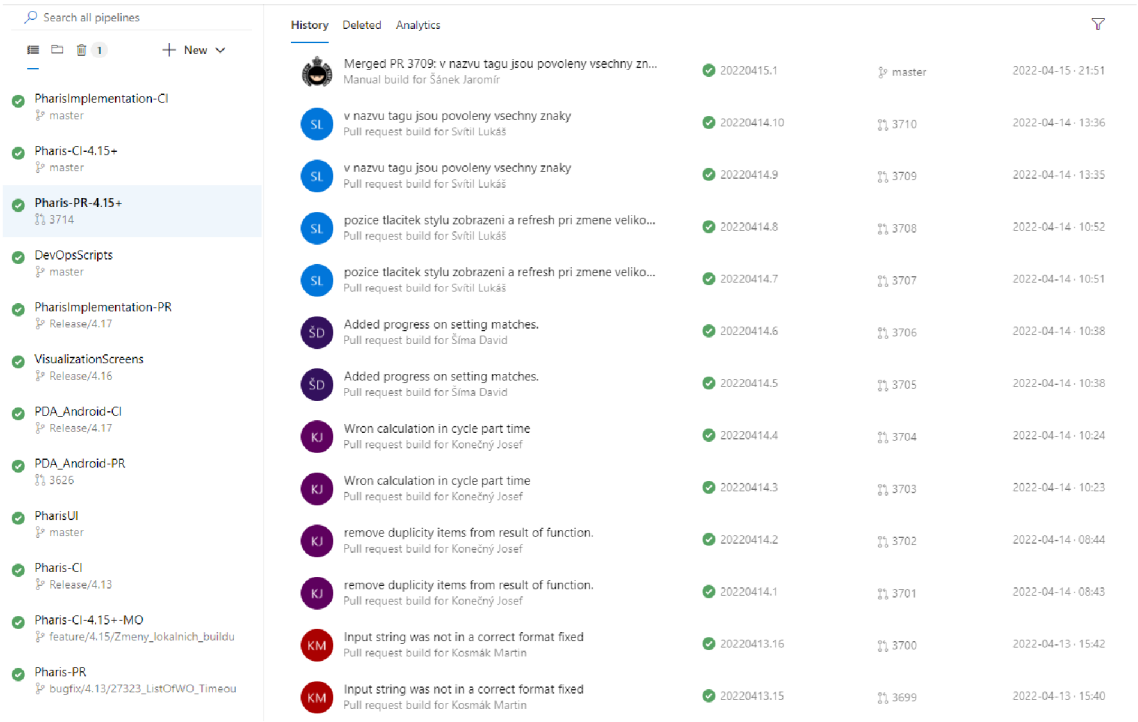
Podobně jako u úloh na serveru Jenkins je to i u zřetězených linek na serveru Devops. Tyto úlohy mají za úkol provádět sestavování aplikace při provedení příkazu `pull request` pro ověření přeložitelnosti změn v kódu. Taktéž se zde spouští unit testy, které musí projít, aby bylo možné propsat změny z vývojových větví do hlavních větví Gitu. Aktuálně je vývojář schopný zjistit délku jednotlivých běhů úlohy přímo z GUI Jenkins, popřípadě Azure Devops (tzn. na dvou rozdílných místech). GUI taktéž nenabízí možnost porovnat délku běhů jednotlivých úloh žádnou přívětivou formou, například ve formě grafu. Taktéž není možné rozumným způsobem kontrolovat překročení určité prahové hodnoty délky běhu úlohy a následně o tomto překročení prahové hodnoty informovat vývojáře.

Cílem firmy UNIS je tedy sledovat délku běhu úloh těchto procesů a alarmovat vývojáře v případě, že se doba běhu jednotlivých úloh začne zhoršovat, tedy překročí prahovou hodnotu. Další zajímavou vlastností, jež by měla být pozorována, je u některých těchto běhů velikost vytvořených artefaktů. Konkrétně se jedná o balíčky jednotlivých částí aplikace na serveru DevOps. Pro vývojáře jsou pak zajímavé níže uvedené úlohy:

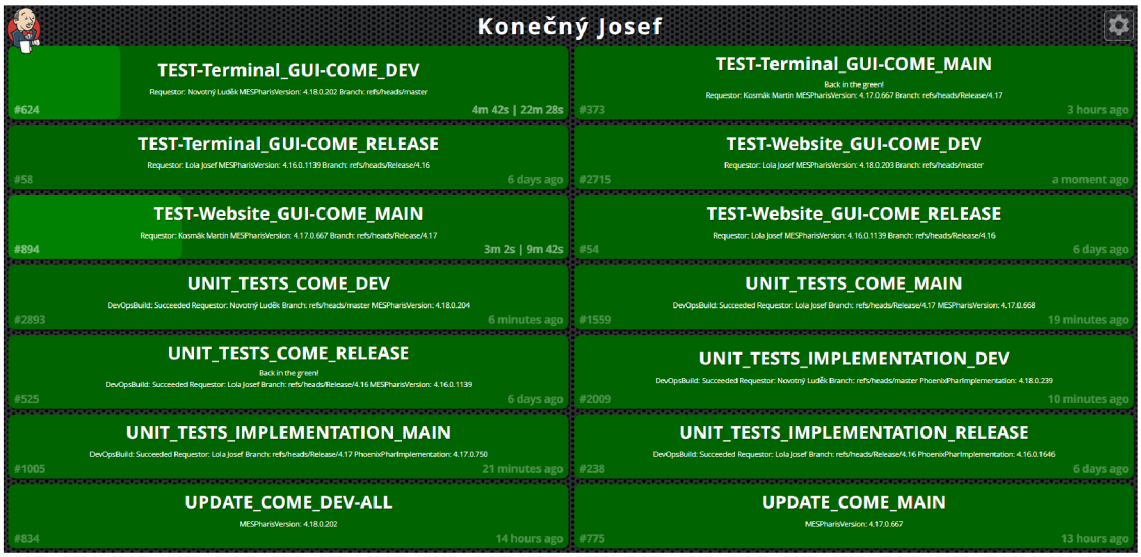
- **server DevOps:**
  - Pharis-CI-4.15+ – úloha zajišťující kontinuální integraci a vytvoření balíčků jednotlivých částí systému MES PHARIS.
  - Pharis-PR-4.15+ – tato úloha se spouští při provedení žádosti o změnu do cílové větve (pull request). Provedení změny nastane až po ověření přeložitelnosti kódu a pro úspěšném průchodu unit testy.

- **server Jenkins:**
  - TEST-Terminal_GUI-COME_[DEV|MAIN|RELEASE] – úlohy spouštějící GUI testy výrobního klienta, tedy terminálu.
  - TEST-Website_GUI-COME_[DEV|MAIN|RELEASE] – úlohy spouštějící Selenium testy webového klienta.
  - UNIT_TESTS_COME_[DEV|MAIN|RELEASE] – úlohy, které spouští unit testy jednotlivých částí systému MES PHARIS.
  - UNIT_TESTS_IMPLEMENTATION_[DEV|MAIN|RELEASE] – úlohy, které spouští unit testy implementačních balíčků.

Náplní této práce je pak zpracování získaných dat o časové náročnosti a vizualizace výše zmíněných úloh.



Obrázek 5.1: Ukázka GUI DevOps Pipelines, která zobrazuje přehled běhů konkrétní úlohy. Pokud by uživatel chtěl porovnat délku jednotlivých běhů, musel by vždy zobrazit detail konkrétního běhu, kde jsou však pouze statistiky daného běhu. Taktéž není možné pozorovat překročení hraniční hodnoty délky běhu a velikosti vytvářených artefaktů.



Obrázek 5.2: Ukázka GUI serveru Jenkins zobrazuje vizualizační obrazovku, která poskytuje informaci o výsledku posledního běhu úloh. Pro analýzu délky běhu úloh by uživatel opět musel zobrazit detail konkrétního běhu úloh, který zobrazí informace pouze o daném běhu.

5.1.2 Zátěžové testy

Jak již bylo zmíněno, MES PHARIS se testuje hned několika sadami testů, a to GUI testy webového klienta, GUI testy terminálové aplikace a pak unit testy, které mají za úkol pokrýt co nejvíce zdrojového kódu a jsou zaměřeny na dílčí části systému. Doposud však nebyly implementovány žádné zátěžové testy. V současné době firma pociťuje jejich absenci, a proto bylo vypsáno toto zadání diplomové práce. Hlavním požadavkem na sadu testů je snaha napodobit prostředí u zákazníka a simulovat tak situace z běžného používání MES PHARIS. Pro vývoj je zajímavých hned několik oblastí, které by tyto testy mohly pokrývat. Některé z nich mají však vyšší prioritu.

Nejdůležitější částí, kterou by testy měly pokrývat, je výkonnostní testování aplikace, která běží na serveru, k níž se připojují terminálové aplikace. Jedná se totiž o stěžejní část celého systému z pohledu výkonosti. To vyplývá z toho, že terminálových aplikací v běžném provozu může současně běžet až několik desítek a na server tak může chodit poměrně velké množství požadavků, které mohou vytvořit větší zátěž. Důležitost této části systému je také zřejmá ze samotné podstaty MES systémů. Výpadky či výrazné zhoršení výkonu může mít tedy následky, které pro zákazníka mohou představovat velké riziko. Testy by měly simulovat například registrace několika zařízení a provádění běžných operací, které operátoři ve výrobě vykonávají. Měla by se pak kontrolovat odezva serveru, jeho vytížení atd.

Jak jsme se již dozvěděli, MES systémy komunikují s ERP systémy, a tak pracují s daty generovanými stroji. Tyto stroje v reálné výrobě často generují velké objemy dat, s nimiž pak Pharis dále pracuje. Testy by tedy mohly pokrývat i takové scénáře, kdy je systém pod zátěží několika takových strojů a zpracovává jimi poskytované hodnoty jako jsou například cykly, stav stroje či teplota.

Podobně jako zátěž ze strany terminálu může vznikat i zátěž ze strany webové aplikace. Zde by se opět mohlo testovat přihlášení několika uživatelů současně a následné vykonávání běžných uživatelských operací. Pro firmu však tato varianta nemá takový přínos, jelikož v reálném provozu běžně nenastávají situace, kdy by webovou aplikaci souběžně využívalo větší množství uživatelů. Taktéž operace, jež se zde provádějí, nejsou tak frekventované jako operace u terminálového klienta.

Další oblastí, jež by mohla být užitečná, je měření výkonnosti během importu dat. Pro vývojáře může být zajímavý import dat přes SCI[1]. Dále pak import základních číselníků, technologických postupů či výrobních příkazů. Importy pak mohou být prováděny přímo z GUI, pomocí importačního nástroje, jenž je napsán jako rozšíření pro Excel či prostřednictvím API.

5.1.3 Vizualizace výsledků

Tato část požadavků je stěžejní částí mojí diplomové práce. Vývojáři požadují, aby data z výše zmíněného testování a měření byla ukládána nejlépe na jednom místě. Nad těmito daty by měla být prováděna analýza, popřípadě by měla být vizualizována ve vhodném formátu tak, aby mohl vývojář co nejsnadněji výsledek analyzovat a odhalit tak důvod, který vedl ke zhoršení výkonnosti. Cílem této analýzy a její následné vizualizace by mělo být včasné odhalování výkonnostních problémů jak samotného vyvíjeného systému, tak i automatizovaných procesů, které jsou během vývoje používány pro usnadnění práce. Hlavním problémem této části je vhodné zobrazení daných dat. Důležitým požadavkem pak je vhodné upozornění na to, že ke zhoršení výkonnosti došlo. Požadavkem vývojářů je možnost vizualizace jedné či dvou vizualizačních obrazovek, tedy aby byl vývojář schopný poznat, že nastal problém na základě těchto jednoduchých vizualizací, které budou neustále zobrazovány na vizualizačních monitorech v kanceláři vývoje. Pro další analýzu by pak měly být dostupné další výstupy, nejlépe ve formě podrobnější vizualizace.

V současnosti se výsledky testů vizualizují pomocí vizualizačních obrazovek, které jsou poskytovány prostřednictvím serveru Jenkins, který nabízí vizualizace nad daty získanými během běhu úloh. V aktuálním stavu jsou tedy zobrazovány například kolekce testů, jež prošly či neprošly úspěšně. Ve vývoji se aktuálně začíná používat ELK Stack k logování z aplikací. Do budoucna je plánováno možnosti ELK Stacku využívat co nejvíce, a to například i pro centralizaci záznamů událostí přímo od zákazníků. Jak již bylo zmíněno v kapitole 4.1.1, nástroj Kibana, jež je součástí ELK, lze využít k vizualizacím nad daty v databázi ElasticSearch. Proto se přímo nabízí možnost ELK pro tyto vizualizace využít.

[1] SCI neboli Standard Communication Interface je komunikační rozhraní mezi ERP a MES systémy.

ID	Název požadavku	Popis požadavku	Priorita
1	Monitoring délky běhu úloh serveru Jenkins	Sběr dat o jednotlivých běžích úloh, které jsou vykonávány na serveru Jenkins.	1
2	Monitoring délky běhu úloh serveru DevOps	Sběr dat o jednotlivých běžích úoh, které jsou vykonávány na serveru DevOps.	1
3	Monitoring velikosti vytvářených artefaktů (balíčků)	Sběr metrik, jež se týkají jednotlivých balíčků vytvářených jak serverem DevOps, tak i Jenkins. Zásadní metrikou je velikost vytvářených balíčků.	2
4	Centralizace sběru dat	Data získaná měřením by měla být ukládána na jednom místě.	1
5	Vizualizace pomocí přehledné vizualizační obrazovky	Vizualizační obrazovka by měla být přehledná a vývojář by z ní měl rychle zjistit, který běh a která úloha se zhoršila, nebo se chová nestandardně.	1
6	Vizualizace historických dat	Neměla by být vizualizována jen data aktuálních běhů, ale měla by být zobrazována i historie.	1
7	Upozornění vývojářů na výskyt neočekávaného chování	Vývojář by měl být upozorněn na výskyt neočekávaného chování co v nejkratším čase.	2

Tabulka 5.1: Tabulka požadavků a navrhovaného řešení pro monitoring délky běhu automatizovaných procesů.

ID	Název požadavku	Popis požadavku	Priorita
1	Zátěžové testování serverové aplikace – zatížení terminálovými aplikacemi	Server by měl být zatížen požadavky od několika terminálů. Pokusit se nasimulovat co nejvěrněji situace běžnému provozu.	1
2	Zátěžové testování serverové aplikace – data strojů	Testování chování serveru při zpracovávání dat ze strojů.	2
3	Zátěžové testování webové aplikace	Testování výkonnosti webové aplikace. Testování odezvy serveru na úkony požadavky zasílané z webového klienta.	3
4	Testování výkonnosti importu dat	Testování výkonnosti při importu dat.	3
5	Sběr naměřených dat na centralizovaném místě	Data z testů musí být sesbírána nejlépe na jedno úložiště, kde bude prováděna další analýza.	1
6	Vhodná vizualizace výstupu	Data by měla být vizualizována na vizualizační obrazovce. Je nutné navrhnout vhodný formát pro zobrazování v kanceláři vývoje.	1
7	Porovnávání s historickými běhy testů	Měly by být vizualizovány i předchozí běhy, popřípadě by měly být alespoň uchovávány jejich výsledky, a ty by měly být zohledněny při vizualizaci a analýze výsledků.	1
8	Detekce nestandardního chování	Mělo by být detekováno nestandardní chování a v případě detekce anomálie by měly být vizualizační obrazovky schopny na tento problém upozornit.	1
8	Analýza získaných dat	Nad daty by měla být provedena analýza, která by měla případně vývojáře dovést k jádru problému (tzn. pomoci nalézt chybu).	2

Tabulka 5.2: Tabulka požadavků a navrhovaného řešení výkonnostního testování MES PHARIS.

5.1.4 Detekce anomálií v měřených datech

Důležitou částí analýzy výsledků testů a pozorování délky běhu automatizovaných procesů je detekce anomálií, která vývojáře upozorní na nestandardní chování. Tato detekce by se dala provádět implementací tzv. detektorů anomálie. Detektory by mohly být následující:

- Konstantní prahová hodnota – jedná se o detektor, jenž detekuje anomálii na základě překročení zadané hodnoty. V tomto případě je parametr jen jeden:
  - Prahová hodnota – jedná se o hodnotu, jež nesmí být překročena
- Aproximace předchozích výsledků – tento detektor proloží X posledních výsledků měření funkcí. Tato funkce nám pak určí hranici, jež by neměla být překročena. Parametry jsou pak následující:
  - Počet posledních výsledků, pro něž bude provedena aproximace
  - Hodnota, o kterou může být vzniklá hranice aproximovanou funkcí překročena
- Aproximace referenční množiny dat – princip tohoto detektoru je stejný jako při aproximaci pomocí předchozích výsledků, s tím rozdílem, že aproximace bude probíhat nad referenční množinou výsledků, nikoliv nad X posledními výsledky. Parametry budou pak opět dva:
  - Množina referenčních výsledků
  - Hodnota, o kterou může být vzniklá hranice aproximovanou funkcí překročena
- Detekce strmosti – další zajímavou anomálií, již lze detekovat, je například strmost růstu metriky v čase. Detekce by pak mohla být prováděna například tak, že se pro každou metriku a měření určí směrnice, která bude představovat nejostřejší možný růst na základě předchozího pozorování či podle několika posledních měření. Naměřená data pak budou proložena přímkou (vždy mezi lokálními extrémy), pro kterou spočteme směrnici. Pokud bude strmost dána touto směrnicí vyšší než směrnice, jež je hraniční, bude detekována anomálie. Parametry jsou pak následující:
  - Data pro určení směrnice růstu
- Růst zátěže v rámci několika běhů testů – zajímavou anomálií může být zhoršování výkonnosti v průběhu několika měření. Během jednoho měření nemusí dojít k překročení prahové hodnoty, ale může být detekováno, že během X posledních běhů testů je pozorováno mírné zhoršení výkonnosti. Detekce této anomálie tak může předejít potencionálním větším problémům.
- Detekce pomocí neuronové sítě – tato varianta je poměrně experimentální a není zaručen její úspěch.

Anomálie v naměřených metrikách jsou v rámci diplomové práce detekovány pouze pomocí statických prahových hodnot. Taková detekce anomálií je poměrně přesná, má zaručené výsledky a tím i zaručený přínos pro firmu.

Prahové hodnoty jsou pak odvozeny z dříve naměřených metrik. Pro metriky pozorované u úloh spouštěných pomocí automatizačních serverů jsou odvozeny z padesáti posledních běhů, které byly úspěšně dokončeny. Horní prahová hodnota pak odpovídá kvantilu percentilu 97 a spodní percentilu 3. Tyto prahové hodnoty odpovídající percentilům jsou zavedeny pro každou pozorovanou metriku. S hodnotou percentilu může být experimentováno a může být měněna dle potřeby a charakteristiky metriky a úlohy.

5.2 Problémy z praxe řešitelné monitoringem automatizovaných procesů

Při vývoji systému MES PHARIS se již několikrát stalo, že nastaly různé problémy s automatizovanými procesy. Pokud jsou tyto problémy výkonnostního ražení (tzn. objeví se pouze zhoršení ve výkonnosti dané automatizované úlohy), je poměrně těžké takový problém odhalit. Jejich monitoring by měl takovým problémům předcházet. Mezi problémy, které se při vývoji MES PHARIS objevují patří:

- změna délky běhu úlohy kvůli vytížení stroje,
- změna délky běhu úlohy kvůli změně definice úlohy,
- změna délky běhu úlohy kvůli změně zdrojových kódů,
- změna velikosti vytvářených artefaktů (balíčků), typicky po přechodu na vyšší verze knihoven třetí strany.

5.3 Navrhované řešení

Získáním měřených metrik a dat z průběhů testů a sledováním časové náročnosti automatizovaných testů se zabývá diplomová práce Martina Oháňky. Je však poměrně důležitá i pro moji část, neboť se získanými daty dále pracuji, provádím nad nimi analýzu a vizualizuji je. V návrhu řešení je tedy okrajově zmíněno Martinovo navrhované řešení sběru dat, jelikož další práce na tento sběr přímo navazuje. Hlavní technologií, kterou moje část řešení využívá, je ELK Stack, a to hned z několika důvodů. Jedním z hlavních důvodů, jež jsem již zmínil, je fakt, že se firma snaží ELK Stack využívat jako centrální úložiště záznamů událostí z mnoha aplikací. ELK Stack je tedy ve firmě již používán a jeho základní či vyšší znalostí disponují všichni vývojáři. Dalším důvodem je, že tento nástroj nabízí řešení mnoha problémů, jako například balíček nástrojů Beats pro sběr a zasílání dat na jedno místo, Logstash pro dataprocessing a provádění základních analýz, a především pak nástroj Kibana, který nabízí vizualizační obrazovky pro vizualizaci výsledků. Za tímto účelem by se daly využít i jiné podobné nástroje jako je například Splunk [9], Graylog [4], Sumologic [10] či Loggly [5]. Některé z nich jsou však zpoplatněné a mají i různé jiné nevýhody (znázorněno v tabulce 5.3). Na základě těchto důvodů byl tedy zvolen jako hlavní technologie ELK Stack.

Pro odladění výsledného řešení byl vyčleněn počítač, na kterém je provozován celý ELK Stack. ELK Stack je na tomto počítači virtualizován pomocí technologie Docker. Pro každou z komponent (ElasticSearch, Logstash, Kibana) je vytvořen samostatný kontejner. Veškerá

data, jak z monitoringu automatizovaných procesů, tak i z výkonostního testování jsou tedy zpracována pomocí nástroje ELK Stack a uložena v jediné instanci databáze ElasticSearch, což splňuje požadavek č. 4 z tabulky 5.1 a č. 5 z tabulky 5.2 Přesnější detaily budou uvedeny v další kapitole 6.

	SPLUNK	ELK	GRAYLOG	SUMO LOGIC	LOGGLY
Rozšíření	Dostupné	Velké množství	Jen několik málo	Dostupné	Dostupné
Obtížnost na naučení	Vysoká	Vysoká	Nízká	Střední	Vysoká
Cena služby	Vysoké	Zdarma	Střední	Vysoká	Vysoká
Dokumentace	Dostupná	Dostupná	Dostupná	Dostupná	Dostupná
Vstupní formát dat	Jakýkoliv	Jakýkoliv	Závislost na rozšířeních	Závislost na rozšířeních	Jakýkoliv
Již využíván ve firmě UNIS	Ne	Ano	Ne	Ne	Ne

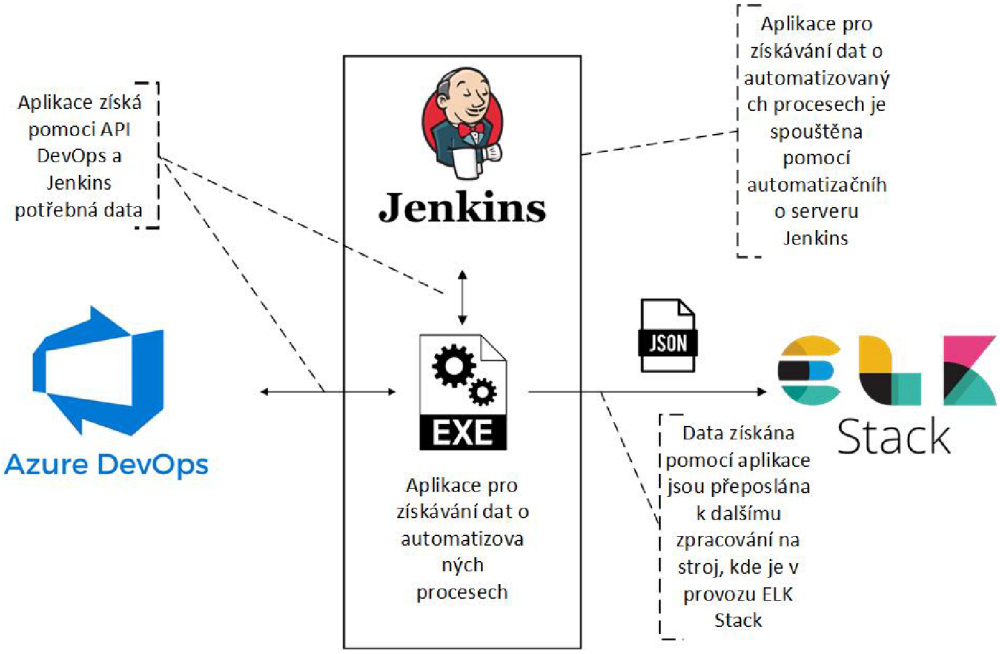
Tabulka 5.3: Tabulka zobrazující několik stěžejních vlastností logovacích nástrojů. Barevně jsou znázorněny pozitivnost či negativnost dané vlastnosti. Červená znázorňuje negativa, žlutá mírná negativa a zelená pak pozitiva.

5.3.1 Monitoring automatizovaných procesů vývoje – získání, zpracování a uložení potřebných dat

Potřebná data sledovaných procesů, jež jsou spouštěny na Jenkins serveru, jsou získávána pomocí konzolové aplikace, která využívá API, jež Jenkins a DevOps nabízí. Tato aplikace je psána v jazyce C# (viz. diplomová práce Martina Oháňky [24]). API poskytuje výstup v JSON formátu, což je i vhodný formát pro další zpracování pomocí nástroje Logstash. V rámci řešení diplomové práce Martina Oháňky je výstup z API těchto serverů ještě upraven a jsou přidány a dopočítány některé hodnoty, jež se v originálním výstupu nevyskytují, a to z důvodu usnadnění další práce při zpracování těchto dat. Finální formát získavaných dat pomocí této konzolové aplikace tedy vznikl na základě požadavků, které jsou od monitoringu automatizovaných procesů očekávány. Přidání hodnot, které jsou dopočítány za účelem usnadnění mé další práce, bude zdůvodněno v kapitole věnující se detailům implementace (6). Struktura tohoto JSON výstupu je pak popsána v přílohách pomocí tabulek (viz. příloha B).

Pro zasílání výstupů s potřebnými daty je využíván síťový protokol TCP, kterým jsou data zasílána do nástroje Logstash. Veškerá komunikace probíhá tzv. in-house, což znamená, že nemusí být řešeno zabezpečení. Aplikace, která získává potřebná data z automatizačních serverů, je spouštěna pomocí úlohy na serveru Jenkins. Frekvence spouštění této aplikace nemusí být příliš velká, jelikož běh pozorovaných úloh není taktéž příliš frekventovaný. Jako výchozí hodnota velikosti tohoto intervalu je zvoleno 20 minut. V případě potřeby se tento údaj dá jednoduše změnit v definici úlohy.

Po zaslání dat do ELK, konkrétně nástroje Logstash, je třeba data vhodně upravit pro další práci nad nimi, a to především pro vizualizace. Po této úpravě jsou data uložena do databáze ElasticSearch. Původní data jsou získávána ze dvou zdrojů (servery DevOps a Jen-



Obrázek 5.3: Zjednodušené schéma získávání potřebných dat pro další zpracování pomocí technologie ELK Stack

kins), a proto je nutné tato data odlišit. Za tímto účelem je použito několik indexů v rámci ElasticSearch. Záznam o každém běhu jednotlivých úloh, který je v rámci Logstash zpracováván, obsahuje i údaje o jednotlivých podúlohách (v terminologii DevOps a Jenkins tzv. stage) a v případě úloh, které spouštějí testy, jsou obsahem záznamu i údaje o jednotlivých testech. Konkrétně se jedná o klíče `stages`, `testRunResults` viz. B. Pro ukládání takové struktury dat je však ElasticSearch nevhodný, neboť při uložení pole objektů dojde k tzv. zploštění (anglicky flattened objects) a ztratí se tak asociace mezi jednotlivými vlastnostmi zanořených objektů. Z toho důvodu je nutné zavést i indexy pro jednotlivé podúlohy a testy. Tato problematika je detailněji popsána v části zabývající se implementací (6.2.1). Monitorované úlohy, které běží na serveru DevOps, se skládají z několika podúloh, ale neobsahují žádné testy. Lze je tedy ukládat do dvou indexů. U úloh ze serveru Jenkins se monitorují již i jimi obsažené testy, z toho důvodu je tedy nutné ukládat záznamy o těchto úlohách do tří indexů. Z toho vyplývá, že pro monitoring automatizovaných procesů je nutno vytvořit dohromady pět indexů. Tyto indexy jsou popsány v tabulce 5.4. Přehled monitorovaných úloh byl již zmíněn v kapitole 5.1.1.

Příchozí záznam je tedy pomocí Logstash filtrů rozdělen na jednotlivé části (záznam o běhu úlohy, záznamy o jednotlivých podúlohách a záznamy o spuštěných testech), které jsou odeslány na náležitý index do ElasticSearch databáze. U záznamů, jež obsahují informace o běhu úloh, je při zpracování v rámci filtrů doplněna horní a spodní prahová hodnota pro metriky. Toto přidání prahových hodnot je provedeno pomocí volání Ruby skriptu v rámci Logstash filtrů, jež načte konfigurační soubor ve formátu JSON, který obsahuje pro každou pozorovanou úlohu seznam metrik a jejich prahové hodnoty. Hodnoty z tohoto konfiguračního souboru jsou pak přidány do zpracovávaného záznamu. Ruby skript nepřidá pouze prahové hodnoty pro danou metriku, ale taktéž nastaví příznak, ze kterého je možné jednoduše zjistit, jestli tyto hodnoty byly v běhu překročeny. Příznaky jsou pak globální (ten

je nastaven jako true, pokud byla překročena prahová hodnota alespoň jedné metriky, nebo stav úlohy není roven hodnotě `"SUCCESS"`) a pak příznaky náležící přímo konkrétní metrice. Ty jsou zavedeny z toho důvodu, aby bylo možné dohledat, která metrika byla překročena. Konfigurace filtrů a detekce překročení prahových hodnot je podrobněji popsána v kapitole.

DevOps	
Index	Popis indexu
`logstahs-devopspipelines`	Do tohoto indexu jsou ukládány údaje o jednotlivých běžích úlohy
`logstash-devopsstages`	Do tohoto indexu jsou ukládány záznamy o jednotlivých podúlohách
Jenkins	
Index	Popis
`logstash-jenkinsjobs`	Do tohoto indexu jsou ukládány údaje o jednotlivých běžích úlohy
`logstash-jenkinsstages`	Do tohoto indexu jsou ukládány záznamy o jednotlivých podúlohách
`logstash-jenkinstests`	Do tohoto indexu jsou ukládány záznamy o jednotlivých testech, které jsou v rámci úlohy spuštěny

Tabulka 5.4: Soupis vytvořených indexů za účelem monitoringu automatizovaných procesů

5.3.2 Monitoring automatizovaných procesů vývoje – vizualizace výsledků

Nad daty, která jsou uložena v ElasticSearch databázi se již může provádět samotná vizualizace pomocí nástroje Kibana. Pro vytvoření vizualizace jsou použity vizualizační obrazovky, které Kibana nabízí (tzv. dashboards). Vizualizační obrazovky jsou interaktivní a dá se mezi nimi přecházet. Pro vizualizaci automatizovaných procesů bylo vytvořeno celkem sedm obrazovek. Tyto obrazovky jsou popsány v tabulce 5.5. Přechody mezi obrazovkami jsou popsány pomocí diagramu přechodů obrazovek (5.10). Každé obrazovce je pak věnována podkapitola s návrhem GUI a krátkým popisem funkcionality.

Společné obrazovky	
ID obrazovky	Popis obrazovky
`Jobs_Status_Monitoring`	Obrazovka zobrazovaná na vizualizačních monitorech v kanceláři vývojářů. Zobrazuje stav posledního běhu pro každou z monitorovaných úloh.
Obrazovky pro vizualizaci DevOps	
ID obrazovky	Popis obrazovky
`DevOps_Pipelines_pipeline85`	Tato obrazovka vizualizuje detailní informace o konkrétním běhu úlohy Pharis-PR-4.15+. Jsou zde zobrazeny všechny podúlohy vizualizované úlohy, historické délky běhu úlohy a základní informace o konkrétním běhu.
`DevOps_Pipelines_pipeline86`	Tato obrazovka vizualizuje detailní informace o konkrétním běhu úlohy Pharis-CI-4.15+. Jsou zde zobrazeny všechny podúlohy vizualizované úlohy, historické délky běhu úlohy a základní informace o konkrétním běhu. Na této obrazovce jsou také vizualizovány vlastnosti vytvořených balíčků a vývoj jejich velikosti v předešlých bězích.
`DevOps_Stage`	Zde jsou vizualizovány historické běhy dané podúlohy a informace o vybraném běhu podúlohy.
Obrazovky pro vizualizaci Jenkins	
ID obrazovky	Popis obrazovky
`Jenkins_Jobs_With_Tests`	Tato obrazovka vizualizuje detailní informace o konkrétním běhu úlohy. Jsou zde zobrazeny všechny podúlohy vizualizované úlohy, testy spouštěné v rámci úlohy, historické délky běhu úlohy a základní informace o konkrétním běhu.
`Jenkins_Stage`	Zde jsou vizualizovány historické běhy dané podúlohy a informace o vybraném běhu podúlohy.
`Jenkins_Tests`	Zde jsou vizualizovány historické běhy daného testu a informace o vybraném běhu testu.

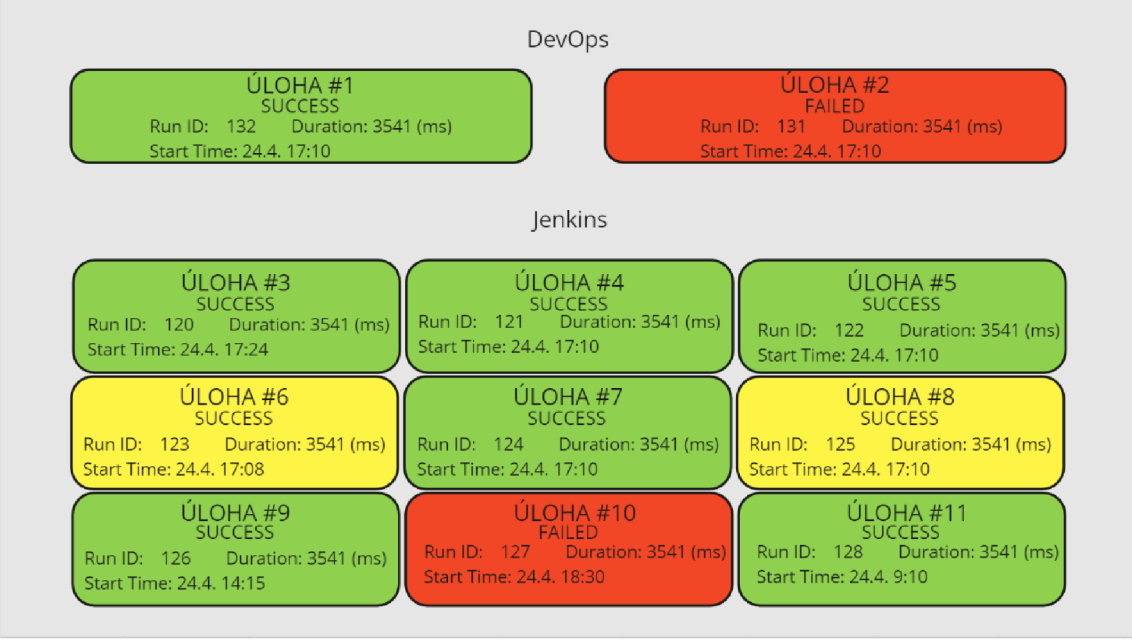
Tabulka 5.5: Tabulka vizualizačních obrazovek pro monitoring automatizovaných procesů a jejich stručný popis.

Obrazovka Jobs_Status_Monitoring

Jedná se o výchozí obrazovku, která je zobrazována v kanceláři vývojářů. Jejím hlavním účelem je vývojáři sdělit stav posledních běhů u všech pozorovaných úloh. Tato vizualizace je inspirována vizualizační obrazovkou, která je dostupná například i na serveru Jenkins pro monitoring sad testů (obrázek 5.2), který pomocí několika variant barevných dlaždic znázorňuje stav testů. Pro vizualizaci jednotlivých stavů byly zvoleny tři barvy, kterými mohou být dlaždice zbarveny:

- Zelená – symbolizuje stav, kdy poslední běh úlohy nezaznamenal žádnou chybu (tzn. jeho stav, jenž je uložený v ElasticSearch databázi, je roven hodnotě `"SUCCESS"`) a globální příznak překročení prahové hodnoty je roven `false`.
- Žlutá – symbolizuje stav, kdy poslední běh úlohy nezaznamenal žádnou chybu, ale globální říznak překročení prahové hodnoty je roven `true` (tzn. byla překročena definovaná prahová hodnota pro jednu z pozorovaných metrik pro danou úlohu).
- Červená – symbolizuje stav, kdy poslední běh úlohy zaznamenal chybu, tzn. jeho stav, jenž je uložený v ElasticSearch databázi, není roven hodnote `"SUCCESS"`.

Každá dlaždice je pak interaktivní a při kliknutí přesměruje uživatele na detail posledního běhu dané úlohy (pro úlohy serveru DevOps DevOps_Pipelines_pipeline86 v případe Pharis-CI-4.15+, DevOps_Pipelines_pipeline85 v případě Pharis-PR-4.15+ a pro úlohy serveru Jenkins Jenkins_Jobs_With_Tests).

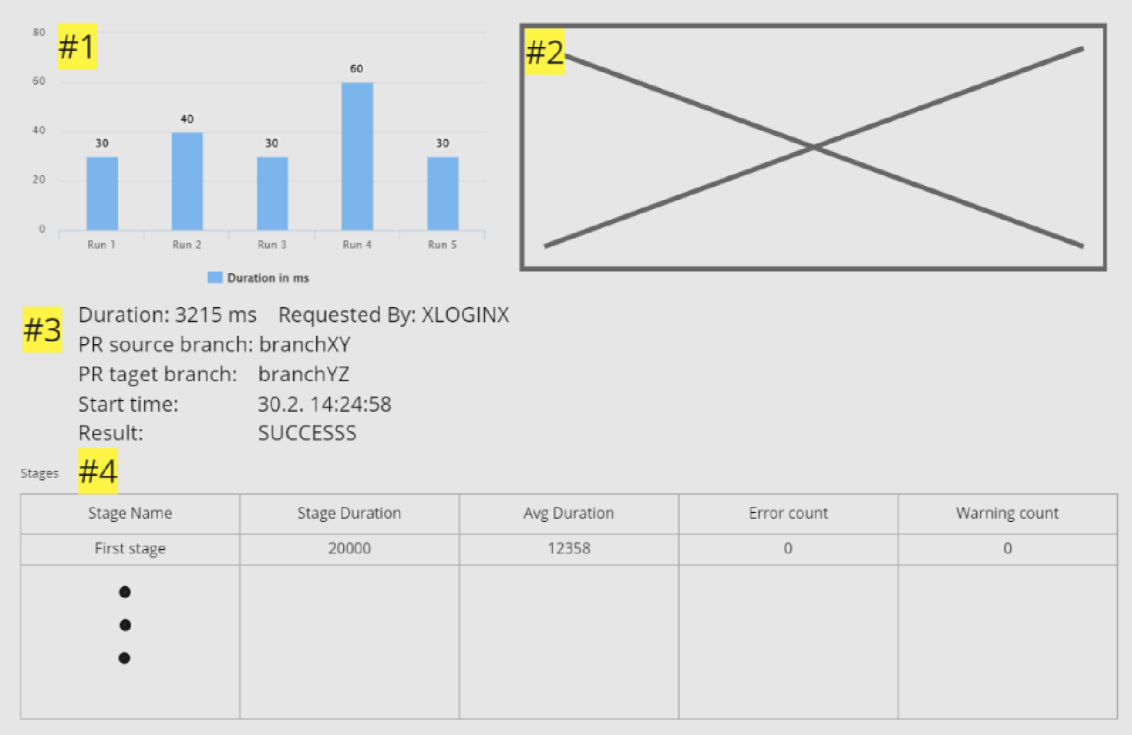


Obrázek 5.4: Návrh GUI pro obrazovku Jobs_Status_Monitoring

Obrazovka DevOps_Pipelines_pipeline85

Tato vizualizační obrazovka má za úkol vývojáři sdělit informace o jednotlivých bězích úlohy Pharis-PR-4.15+. Návrh GUI této obrazovky (obrázek 5.5) se skládá ze čtyř hlavních vizualizačních komponent, které jsou v návrhu GUI označeny žlutými symboly.

- **#1** – Sloupcový graf, který zobrazuje dobu trvání jednotlivých běhů úlohy. V grafu by taktéž měly být zobrazeny prahové hodnoty pro metriku udávající délku doby běhu. Tento graf je interaktivní a každý ze sloupců je možno kliknutím označit. Po označení se zbytek této vizualizační obrazovky překreslí a ostatní komponenty zobrazují data, jež náleží běhu, jehož sloupec byl označen.
- **#2** – Časová osa zobrazující současné běhy úloh spouštěných na serveru DevOps. Na serveru DevOps může být spuštěno několik úloh současně, což může způsobit vyšší zátěž na hardware, a tím pádem i prodloužení doby běhu. Na základě této časové osy tedy vývojář dokáže odvodit, zda prodloužení doby běhu (či dokonce překročení prahové hranice) nemohlo být způsobeno právě souběžnými běhy úloh.
- **#3** – Informace o vybraném běhu úlohy. Konkrétně u úlohy Pharis-PR-4.15+ jsou pro uživatele důležité například informace jako doba trvání úlohy, zdrojová větev PR (pull request), cílová větev PR, iniciátor PR, výsledek celé úlohy atd.
- **#4** – Tabulka, která obsahuje všechny podúlohy vybraného běhu úlohy. Po kliknutí na záznam je uživatel přesměrován na obrazovku DevOps_Stage, tedy obrazovku, jež vizualizuje detail zvolené podúlohy.



Obrázek 5.5: Návrh GUI pro obrazovku DevOps_Pipelines_pipeline85

Obrazovka DevOps_Pipelines_pipeline86

Tato obrazovka zprostředkovává vývojáři informace o jednotlivých bězích úlohy Pharis-CI-4.15+, u které nás zajímají i velikosti artefaktů, konkrétně velikosti balíčků jednotlivých částí systému MES PHARIS. Obrazovka se tím pádem skládá z vícero komponent než DevOps_Pipelines_pipeline85. Návrh GUI je k vidění na obrázku 5.6.

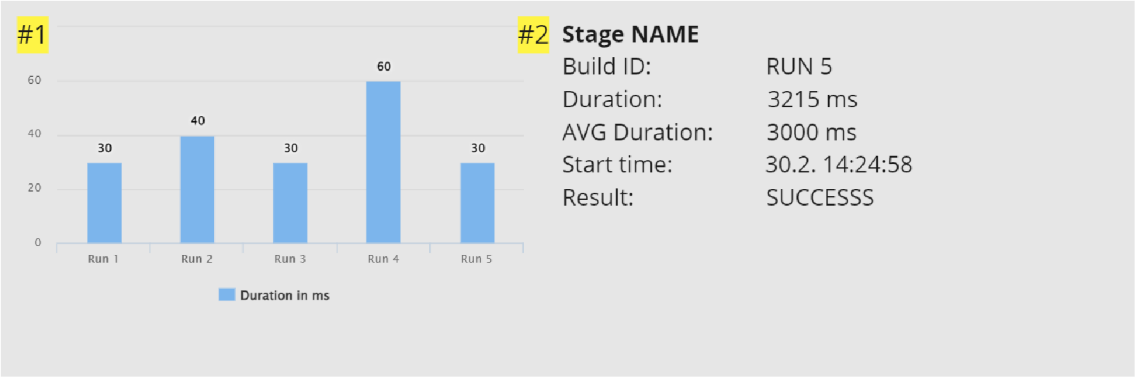
- **#1** – Shodné jako u DevOps_Pipelines_pipeline85.
- **#2** – Shodné jako u DevOps_Pipelines_pipeline85.
- **#3** – Informace o vybraném běhu úlohy jako například doba trvání úlohy, zdrojová větev, iniciátor úlohy, výsledek celé úlohy atd.
- **#4** – Shodné jako u DevOps_Pipelines_pipeline85.
- **#5** – Sloupcové grafy znázorňující velikosti vytvořených balíčků pro jednotlivé běhy. Tato komponenta může obsahovat i informace jako počet souborů balíčku atd.

Obrazovka DevOps_Stage

Tato vizualizační obrazovka slouží k zobrazení detailu konkrétní podúlohy, jež spadá pod úlohu spouštěnou pomocí serveru DevOps. Tato obrazovka obsahuje dvě hlavní komponenty (viz. obrázek 5.7).

- **#1** – Sloupcový graf, který zobrazuje dobu trvání jednotlivých běhů podúlohy. Tento graf je interaktivní a každý ze sloupců je možno kliknutím označit. Po označení se zbytek této vizualizační obrazovky překreslí a ostatní komponenty zobrazují data, která náleží běhu, jehož sloupec byl označen.
- **#2** – Textová reprezentace informací o vybrané podúloze. Měla by obsahovat informace jako identifikátor běhu, dobu trvání běhu podúlohy či výsledek podúlohy.

Obrázek 5.6: Návrh GUI pro obrazovku DevOps_Pipelines_pipeline86



Obrázek 5.7: Návrh GUI pro obrazovku DevOps_Stage. Komponenty, z nichž je toto GUI vytvořeno, se chovají stejně jako komponenty obrazovky Jenkins_Stage.

Obrazovka Jenkins_Jobs_With_Tests

Všechny úlohy spouštěné pomocí automatizačního serveru Jenkins jsou vizualizovány prostřednictvím této vizualizační obrazovky. K jejich vizualizaci není třeba více obrazovek jako pro úlohy spouštěné pomocí DevOps, a to z toho důvodu, že je struktura záznamu o úloze vždy stejná. Každá úloha obsahuje vždy několik podúloh a sadu testů, která je v rámci úlohy spouštěná. Pro tuto vizualizaci jsou potřebné následující komponenty (viz. obrázek 5.8):

- **#1** – Sloupcový graf, který zobrazuje dobu trvání jednotlivých běhů úlohy. V grafu by taktéž měly být zobrazeny prahové hodnoty pro metriku udávající délku doby běhu. Tento graf je interaktivní a každý ze sloupců je možno kliknutím označit. Po označení se zbytek této vizualizační obrazovky překreslí a ostatní komponenty zobrazují data, jež náleží běhu, jehož sloupec byl označen.
- **#2** – Informace o vybraném běhu úlohy. Konkrétně pro úlohy spouštěné na serveru Jenkins jsou pro uživatele důležité například informace jako doba trvání úlohy, počet spuštěných a úspěšných testů, verze systému MES PHARIS, proti které byly testy spuštěny, iniciátor úlohy, výsledek celé úlohy atd.
- **#3** – Tabulka, která obsahuje všechny podúlohy vybraného běhu úlohy. Po kliknutí na záznam je uživatel přesměrován na obrazovku Jenkins_Stage, tedy obrazovku, která vizualizuje detail zvolené podúlohy.
- **#4** – Tabulka obsahující všechny testy vybraného běhu úlohy. Po kliknutí na záznam je uživatel přesměrován na obrazovku Jenkins_Tests, tedy obrazovku, jež vizualizuje detail zvoleného testu.

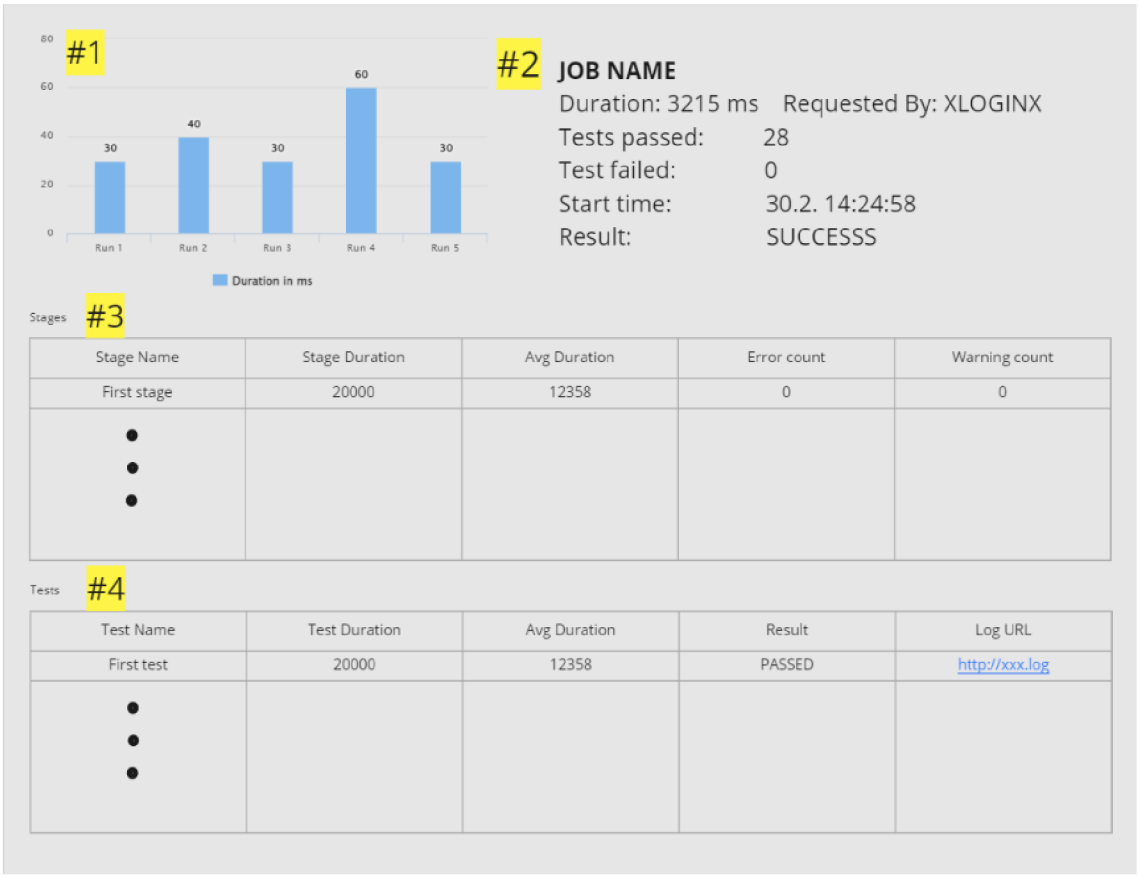
Obrazovka Jenkins_Stage

GUI této vizualizační obrazovky není nikterak odlišné od obrazovky Jenkins_Stage, jehož návrh je zaznamenán na obrázku 5.7.

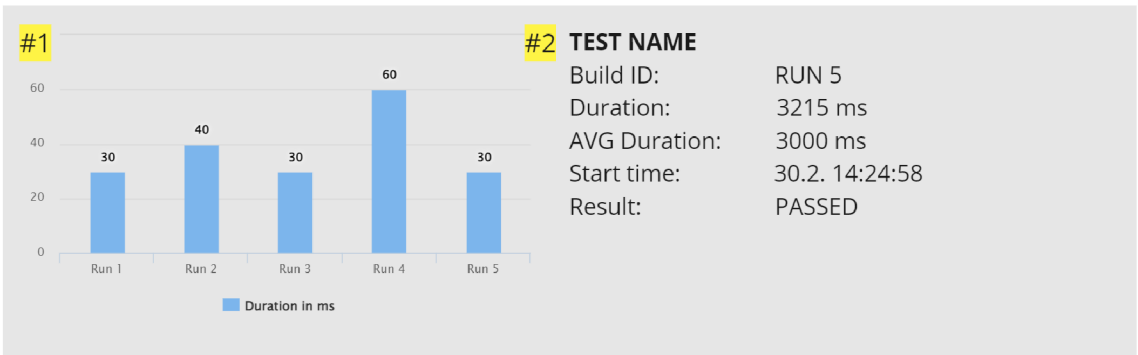
Obrazovka Jenkins_Tests

Tato vizualizační obrazovka slouží k zobrazení detailu konkrétního testu, jenž spadá pod úlohu spouštěnou pomocí serveru Jenkins. Tato Obrazovka obsahuje dvě hlavní komponenty (viz. obrázek 5.9).

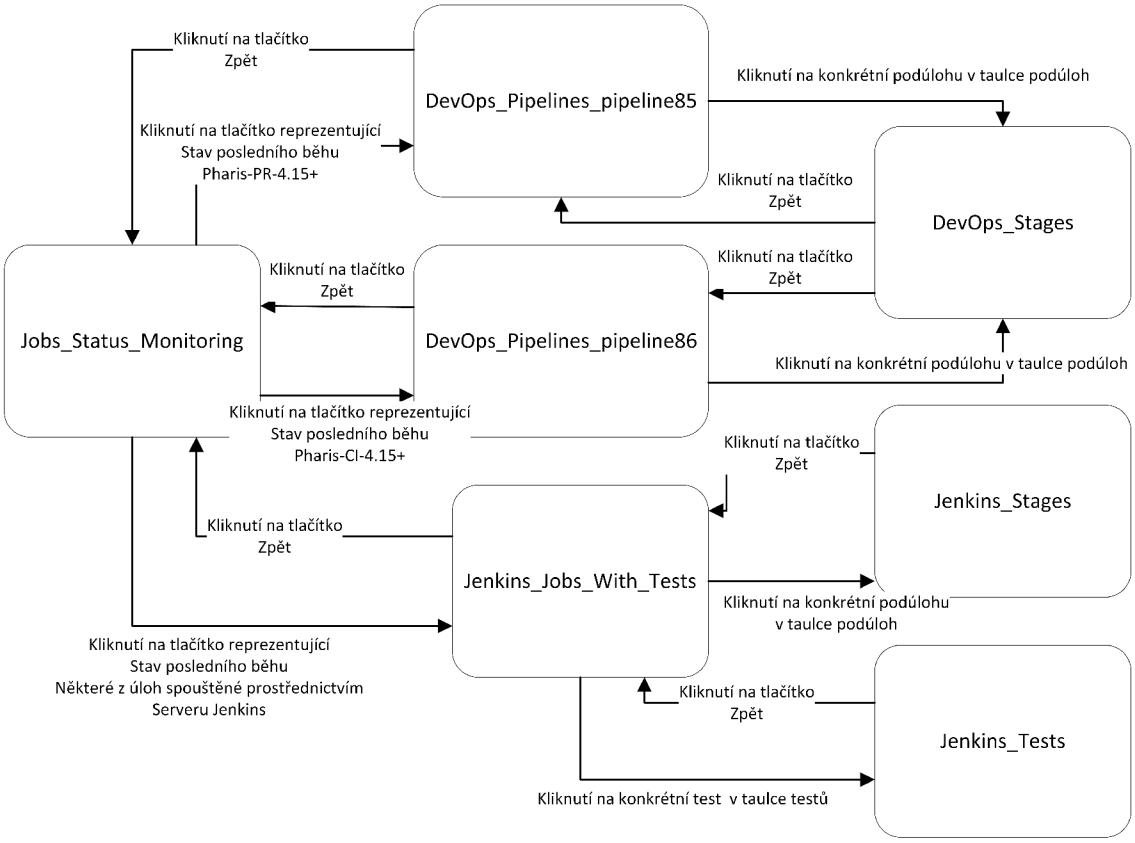
- **#1** – Sloupcový graf, který zobrazuje dobu trvání jednotlivých běhů trestů. Tento graf je interaktivní a každý ze sloupců je možno kliknutím označit. Po označení se zbytek této vizualizační obrazovky překreslí a ostatní komponenty zobrazují data, která náleží běhu, jehož sloupec byl označen.
- **#2** – Textová reprezentace informací o vybraném testu. Měla by obsahovat informace jako identifikátor běhu, dobu trvání běhu testu či výsledek testu.



Obrázek 5.8: Návrh GUI pro obrazovku Jenkins_Jobs_With_Tests



Obrázek 5.9: Návrh GUI pro obrazovku `Jenkins_Tests`



Obrázek 5.10: Diagram přechodů obrazovek

5.3.3 Zátěžové testy – zpracování naměřených hodnot

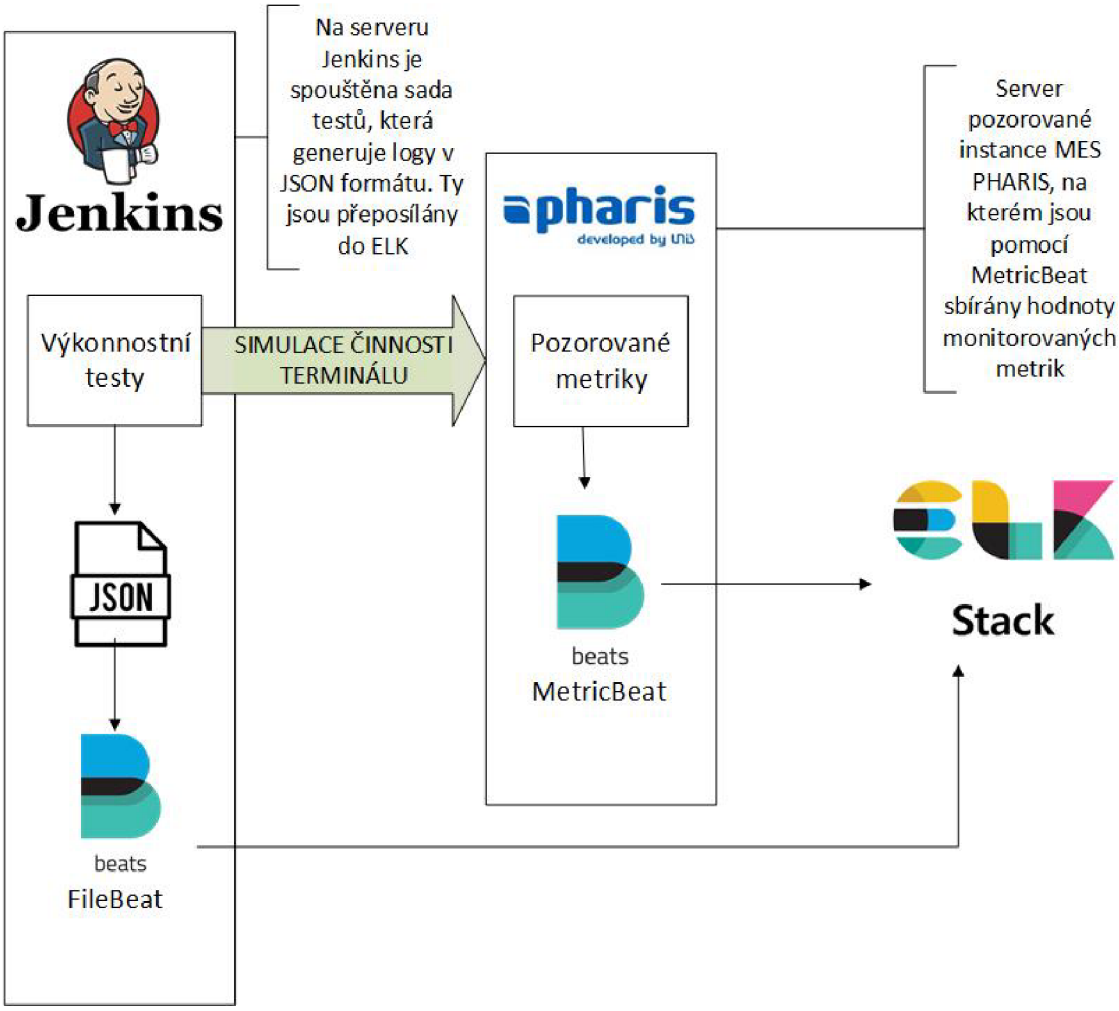
Jak již bylo zmíněno, systém MES PHARIS je poměrně velkým a komplikovaným systémem, jenž se skládá z různých komponent, na něž by se mohly zátěžové testy zaměřit. Nejvyšší prioritu má však testování, které by simulovalo zasílání požadavků na aplikační server výrobními terminály. Za tímto účelem je v rámci diplomové práce Martina Oháňky implementována `.dll` knihovna, jež simuluje chod několika terminálů a prostřednictvím API zasílá požadavky na aplikační server. Testy, které využívají knihovnu, fungují tak, že spustí několik procesů. Tyto procesy pak vykonávají operace, které jsou definovány pomocí scénářů (každý proces se může řídit jiným scénářem). Tímto chováním je tedy simulováno několik terminálových aplikací, které komunikují s aplikačním serverem, a vyvíjejí tak na něj zátěž. Každý proces loguje do složky záznamy o vykonaných akcích ve formátu JSON. Kromě záznamů jednotlivých událostí se taktéž vytvoří záznam obsahující informace o celkovém běhu testu obsahující například celkovou dobu běhu testu a informaci o tom, zda test proběhl úspěšně. Tyto záznamy jsou k další analýze zasílány pomoci nástroje FileBeat na server, kde běží ELK Stack. Struktura zaznamenaných událostí je popsána v příloze C. Instance MES PHARIS, proti které tyto testy běží, je pak provozována na počítači, jež je vyčleněn pouze k tomuto účelu.

Měření výkonnostních metrik je zprostředkováno pomocí MetricBeat, který funguje jako služba na monitorovaném serveru (v našem případě aplikační server hostující MES PHARIS). Tyto metriky jsou taktéž zasílány do nástroje ELK Stack. Sada takto vytvořených testů je opět spuštěna pomocí automatizačního serveru Jenkins. V rámci této úlohy se taktéž spouští služba MetricBeat. Tím je zaručeno, že výkonostní metriky jsou zaznamenávány pouze během vykonávání testu. Průběh výkonnostního testování je znázorněn na obrázku 5.11.

V rámci architektury ELK Stack jsou data posílána prostřednictvím Beats nástrojů konkrétně do komponenty Logstash. V této komponentě jsou provedeny nad daty některé změny (k jednotlivým logovaným událostem jsou přiřazeny jednoznačné identifikátory běhů testů atd.) a data jsou poté rozdělena do tří indexů v rámci databáze ElasticSearch. Tyto indexy jsou stručně popsány v tabulce 5.6. Při zpracovávání dat, která mají být uložena do indexu `logstash-performance-events-testresult`, se v rámci filtrů v komponentě pro požadované metriky opět nastaví příznaky překročení prahových hranic pomocí Ruby skriptu stejně jako u monitoringu automatizovaných procesů. Podrobněji je konfigurace komponenty Logstash pro toto použití popsána v kapitole 6.2.1.

Index	Popis indexu
`logstash-performance-events-testresult`	Index, v němž jsou ukládány logy popisující celkový výsledek testu
`logstash-performance-events`	Index pro ukládání událostí, které jsou logovány jednotlivými procesy, které vytvářejí zátěž na serveru.
`logstash-performance-metrics`	Index pro ukládání měřených metrik získaných pomocí MetricBeat.

Tabulka 5.6: Soupis vytvořených indexů za účelem monitoringu výsledků výkonnostních testů



Obrázek 5.11: Znázornění průběhu výkonnostních testů

5.3.4 Zátěžové testy – vizualizace výsledků

Nad uloženými daty je prováděna vizualizace pomocí nástroje Kibana prostřednictvím vizualizačních obrazovek, které tento nástroj poskytuje. Testů bude do budoucna přibývat dle potřeby vývojářů ověřovat výkonost systému u konkrétních scénářů, které jsou z hlediska výkonu zajímavé. Z tohoto důvodu bylo nutné hlavní vizualizační obrazovku, která bude k vidění na vizualizačním monitoru v kanceláři vývojářů, navrhnout tak, aby se pomocí ní daly vizualizovat výsledky i většího počtu testů. Proto není vhodné podobné rozložení vizualizace jako tomu bylo u obrazovky Jobs_Status_Monitoring. Výsledná vizualizace se skládá ze dvou vizualizačních obrazovek, které jsou popsány v tabulce 5.7.

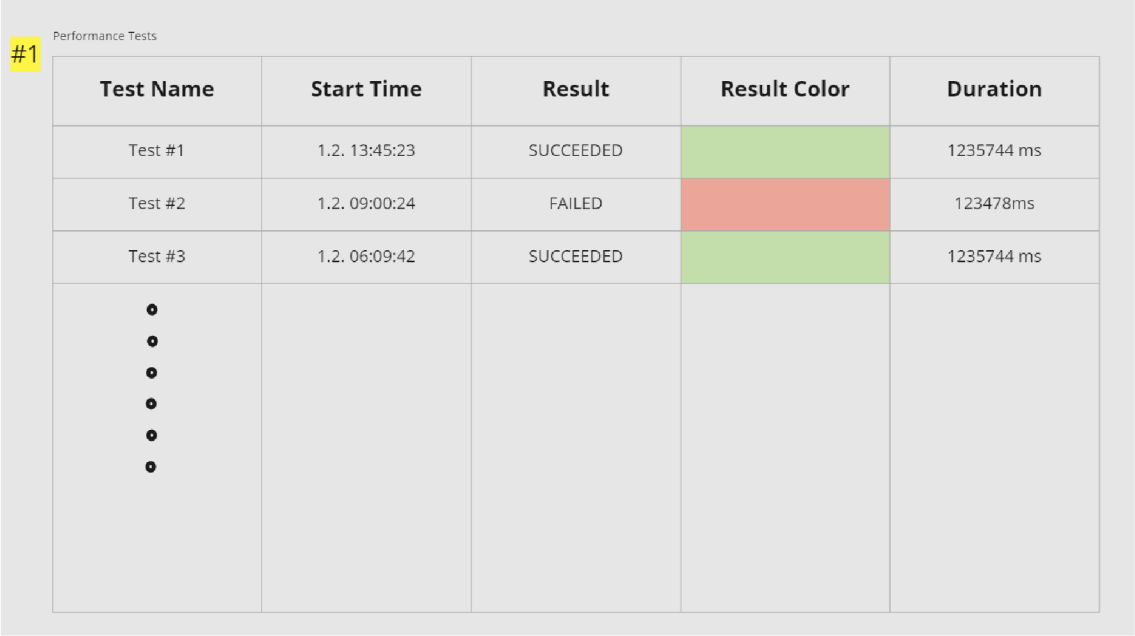
ID obrazovky	Popis obrazovky
`Perf_Tests_Main_Dashboard`	Obrazovka zobrazovaná na vizualizačních monitorech v kanceláři vývojářů. Zobrazuje stav posledních běhů pro každý z výkonnostních testů.
`Performance_Test_Result`	Tato obrazovka vizualizuje detailní informace o konkrétním běhu výkonnostního testu. Jsou zde vizualizovány jednotlivé metriky, které byly během testu měřeny. Na základě vizualizace může vývojář zjistit, kdy a jak se měnilo vytížení monitorované instance MES PHARIS.

Tabulka 5.7: Tabulka vizualizačních obrazovek pro monitoring výsledků výkonnostních testů.

Obrazovka Perf_Tests_Main_Dashboard

Jedná se o vizualizační obrazovku, která je neustále zobrazena v kanceláři vývojářů. Jejím hlavním účelem je zobrazovat stav posledních běhů jednotlivých testů a upozornit vývojáře v situaci, kdy by některý z testů nedopadl úspěšně. Z této obrazovky se pak uživatel může podívat na detail konkrétního testu viz. obrázek 5.14.

Obrázek 5.12 pak představuje návrh GUI. Tato obrazovka se pak skládá pouze z jednoho elementu, který je označen v obrázku symbolem **#1**. Tento element je tabulka, jejímž obsahem pro jednotlivé řádky jsou poslední běhy testů. Na každém řádku je vizualizováno jméno testu, stav testu slovně i barvou a doba trvání testu.

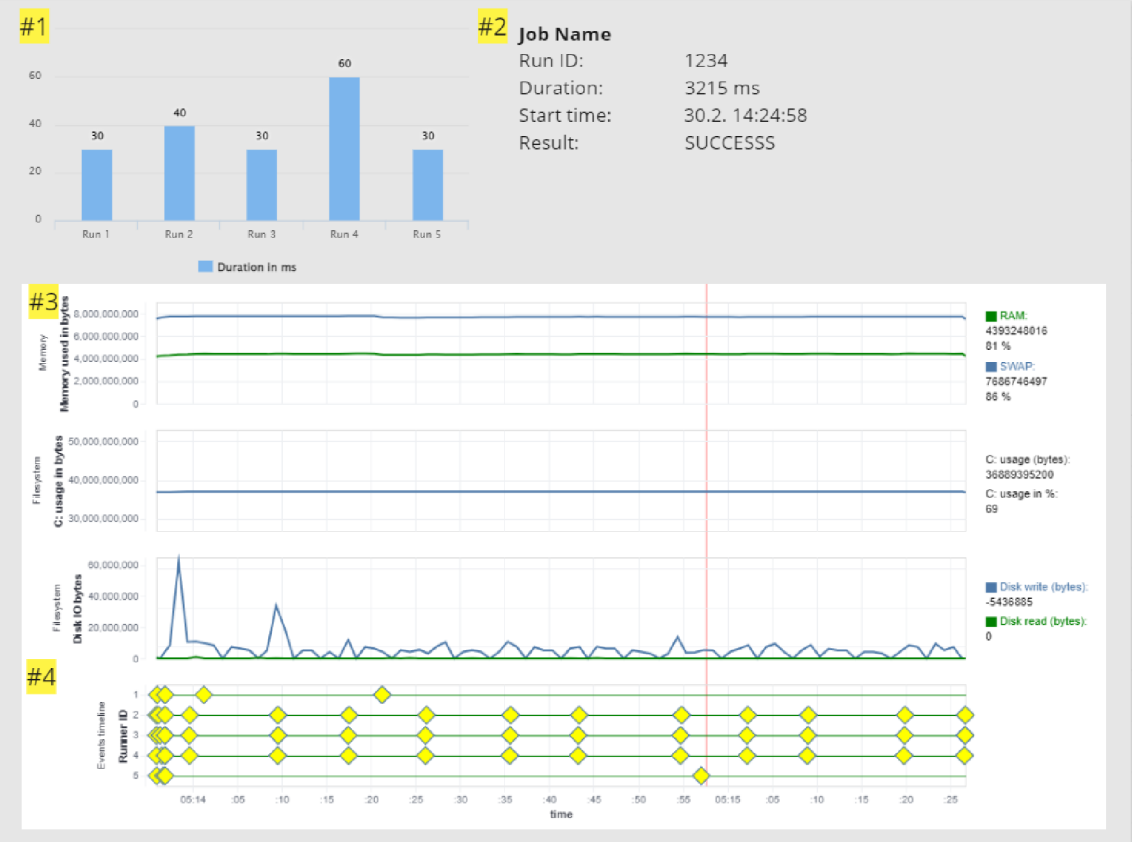


Obrázek 5.12: Návrh GUI pro obrazovku Perf_Tests_Main_Dashboard

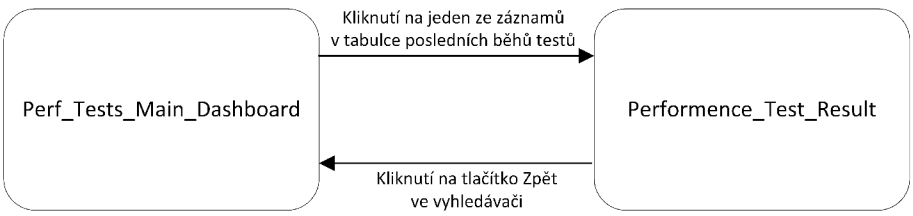
Obrazovka Performance_Test_Result

Tato vizualizační obrazovka má za úkol prezentovat vývojáři výsledky výkonnostních testů. Vždy pak vizualizuje výsledek právě jednoho běhu (vybraného) pro konkrétní test. Této vizualizace je docíleno pomocí čtyř komponent, které jsou popsány níže.

- **#1** – Sloupcový graf, který zobrazuje dobu trvání jednotlivých běhů testů. V grafu by taktéž měly být zobrazeny prahové hodnoty pro metriku udávající délku doby běhu. Tento graf je interaktivní a každý ze sloupců je možno kliknutím označit. Po označení se zbytek této vizualizační obrazovky překreslí a ostatní komponenty zobrazují data, která náleží běhu, jehož sloupec byl označen.
- **#2** – Informace o vybraném běhu testu. Konkrétně se jedná o výsledek testu, unikátní identifikátor běhu testu, délku běhu testu a dobu trvání.
- **#3** – V této části obrazovky jsou vykresleny grafy jednotlivých metrik v závislosti na čase. Definičním oborem je tedy časový interval, jenž je vymezen začátkem a koncem testu. Oborem hodnot jsou pak naměřené výsledky pro danou metriku. Grafů reprezentujících metriky pak může být v této části definováno hned několik podle potřeb uživatele. Zobrazované metriky jsou limitovány pouze těmi, které dokáže sesbírat nástroj MetricBeat.
- **#4** – Graf, který znázorňuje časovou osu s událostmi, které zalogovaly jednotlivé procesy v rámci běhu testu. Každá zelená čára symbolizuje jeden proces, v našem případě tedy simulovaný terminál. Pomocí žlutých kosočtverců jsou pak vyobrazeny jednotlivé události. Údaje o jednotlivých událostech se pak dají zobrazit pomocí nápovědy (tzv. tooltip) po najetí kurzorem nad vybranou událost.



Obrázek 5.13: Návrh GUI pro obrazovku Performance_Test_Result



Obrázek 5.14: Diagram přechodů obrazovek pro monitoring výkonnostního testování

Kapitola 6

Implementace a realizace řešení

V této kapitole je čtenář obeznámen s důležitými implementačními detaily. Účelem této diplomové práce bylo vytvořit pro firmu UNIS vhodné vizualizační obrazovky pro monitoring automatizovaných procesů a výsledků testů tak, aby byly použitelné při vývoji a opravdu se využívaly. Vzhledem k tomu, že se automatizované úlohy mohou časem měnit či přibývat nové, stejně tak jako testy, je možné, že i vizualizační obrazovky pro jejich monitoring přestanou být dostačující. Proto se snažím v této kapitole popsat co nejlépe jednotlivé vizualizační komponenty, nastavení produktu ELK Stack atd. tak, aby bylo možné kdykoliv za pomoci této technické zprávy upravit vizualizace pro nové účely, popřípadě provést různé změny, které by mohly vizualizace vylepšit. Kapitola tedy obsahuje i konkrétní ukázky zdrojových kódů a představení základů syntaxe gramatiky Vega či konfiguračních souborů Logstash a jednotlivých rozšíření, které byly využity.

6.1 Zprovoznění produktu ELK Stack

Jako první krok pro implementaci a zprovoznění vizualizací je třeba uvést do chodu ELK Stack. Jak již bylo zmíněno, aktuálně celé řešení vizualizací funguje na k tomuto účelu vyčleněném počítači, kde běží ELK Stack, který je kompletně dockerizován. Tzn. pro každou z komponent (ElasticSearch, Logstash, Kibana) je vytvořen obraz a spuštěn kontejner.

Pro zprovoznění řešení diplomové práce je tedy potřeba počítač s jakýmkoliv OS, ve kterém je možné provozovat Docker. Jako kostra projektu bylo využito řešení volně dostupné na GitHubu od autora Deviantony [17]. Toto řešení je poměrně kvalitně zdokumentované na stránkách projektu (viz. zdroje). Uvedení ELK Stacku do provozu se nezměnilo. Bylo provedeno jen několik změn v jednotlivých `Dockerfile` souborech a `docker-compose.yml` souboru.

Požadavky na hostitelský stroj

Níže jsou uvedeny požadavky na hostitelský stroj. Velikost operační paměti byla v začátcích řešení pouhých 8 GB. Tato velikost se však ukázala jako nedostačující a při větším zatížení jednotlivé kontejnery provozující komponenty ELK havarovaly kvůli nedostatku paměti. Z toho důvodu byla velikost postupně zvětšena na 16 GB. S takto velkou operační pamětí již nebyly pozorovány žádné problémy, a to ani ve špičkách zátěže.

- Docker Engine verze 18.06.0 nebo novější
- Docker Compose verze 1.26.0 nebo novější
- Ideálně 16 GB RAM

Nastavení portů jednotlivých služeb

Porty jednotlivých komponent jsou nastaveny následovně:

- 5044: Logstash Beats input
- 5000: Logstash TCP input
- 9600: Logstash monitoring API
- 9200: Elasticsearch HTTP
- 9300: Elasticsearch TCP transport
- 5601: Kibana.

V případě potřeby se tyto porty pro každou z komponent dají přenastavit v souboru `docker-compose.yml`.

Sdílené svazky

Jak již bylo zmíněno, jednotlivé kontejnery virtualizované pomocí Dockeru mohou sdílet adresáře s hostujícím OS. Takto sdílené adresáře a soubory jsou v terminologii Dockeru nazývány svazky. Svazky se taktéž definují v souboru `docker-compose.yml`. Pro naše účely jsou sdíleny například konfigurace zřetězených linek pro Logstash nebo Ruby skripty, které jsou v rámci filtrů používány. Ve svazku, který je definován pro ElasticSearch, jsou pak ukládána veškerá naměřená data. Data v těchto svazcích jsou perzistentní. V případě potřeby jde tato perzistentní data odstranit provoláním následujícího příkazu.

```
User:/ELKStack$ docker-compose down -v
```

Uvedení produktu ELK Stack do provozu

Pro uvedení dockerizovaného ELK do provozu stačí provolat následující příkazy v hlavní složce repozitáře.

```
User:/ELKStack$ docker-compose build
User:/ELKStack$ docker-compose up
```

Po prvním spuštění je třeba nastavit hesla tzv. build-in uživatelů. Návod, jak toho docílit, je dostupný na stránkách repozitáře, ze kterého byla převzata kostra projektu. Pro zprovoznění celkového řešení monitoringu automatizovaných procesů a výkonnostních testů je třeba také zprovoznit aplikace a služby z rodiny Beats, které posílají data do komponenty Logstash. Tato část je popsána v diplomové práci Martina Oháňky.

6.2 Monitoring automatizovaných procesů – zpracování dat pomocí nástroje Logstash

V této podkapitole je popsáno zpracování dat obsahujících informace o monitorovaných úlohách, jež jsou do ELK zasílány konzolovou aplikací implementovanou v rámci souběžné diplomové práce. Tato aplikace zasílá data prostřednictvím TCP protokolu v JSON formátu do komponenty Logstash. Formát zasílaných dat je popsán v příloze B pomocí tabulek, jejichž autorem je Martin Oháňka.

Tato data jsou zpracována pomocí jedné zřetězené linky. V rámci repozitáře se její konfigurace nachází v souboru `Logstash/pipeline/devOpsJenkinsMonitoring.conf`. Konfigurace této zřetězené linky je popsána v následujících podkapitolách, které ji rozdělují do dvou logických celků, a to části pro zpracování dat získaných ze serveru DevOps a ze serveru Jenkins. Jak již bylo zmíněno v kapitole 4.1.1, každá konfigurace zřetězené linky se skládá ze dvou povinných částí: vstupu (input), výstupu (output) a volitelné části, tzv. filtrů (filters).

6.2.1 Konfigurace zřetězené linky pro data ze serveru DevOps

Input

Vstupem je komunikace pomocí síťového protokolu TCP na port 5000 (viz. nastavení ELK Stack síťových portů). Pro přijetí dat tedy bude využito rozšíření `tcp`. Formát, ve kterém jsou vstupní data zasílána, je vhodné do objektu události převést pomocí `json_lines` kodeku, který po řádcích načítá JSON soubory.

```
1 input {
2   tcp {
3     id => "jenkins_devops_tcp_input"
4     mode => "server"
5     port => 5000
6     codec => "json_lines"
7   }
8 }
```

Filters

V terminologii nástroje Logstash se nazývají jednotlivé hodnoty jako pole. Hodnota pole se pak dá získat pomocí závorkové notace `[jmenoPole]`. V případě, že je v poli uložen objekt, dají se hodnoty jednotlivých vlastností zpřístupnit pomocí následujícího zápisu `[jmenoPole][jmenoVlastnosti]`. Objekty události vždy také obsahují pole s názvem `[@metadata]`, do kterého lze přidávat různé pomocné hodnoty. Toto pole se pak na konci fáze výstupu smaže a výsledný dokument, jenž je odeslán dál, ho již neobsahuje. Každý objekt události obsahuje po průchodu vstupem pole, která odpovídají struktuře definované pomocí tabulek v přílohách.

Na základě přítomnosti pole `[server]` se rozhodne, zda přijatá data obsahují informace o monitorovaných úlohách. Hodnota tohoto pole pak určí, kterými filtry objekt události bude zpracován. Pole je datového typu string a může nabývat hodnoty `"jenkins"` nebo `"devops"`.

```
filter{
    if ([server]){ #overeni pritomnosti pole
        if ( [server] == "jenkins" ) {
            #zde se zpracovava objekt udalosti obsahujici informace
            #o behu ulohy ze serveru Jenkins
        }
        if ( [server] == "devops" ) {
            #zde se zpracovava objekt udalosti obsahujici informace
            #o behu ulohy ze serveru DevOps
        }
    }
}
```

Výpis 6.1: Rozdělení průchodu filtry na základě serveru

Struktura objektu události je v této fázi prakticky totožná se strukturou, jež je definována v přílohách. Obsahuje tedy i seznam objektů, který je uložen v poli `"stages"`. Pokud by byl takový objekt události uložen do databáze ElasticSearch, došlo by ke zploštění jednotlivých vnořených objektů v tomto seznamu, a ztratila by se tak asociace mezi jejich vlastnostmi. Zploštění objektů lze demonstrovat na jednoduchém příkladu (výpis 6.2).

```
# na index1 je ulozen nasledujici dokument
PUT index1/_doc/1
{
  "group" : "fans",
  "user" : [
    {
      "first" : "John",
      "last" :  "Smith"
    },
    {
      "first" : "Alice",
      "last" :  "White"
    }
  ]
}
```

```
#dokument je transformovan do nasledujiciho tvaru
#asociace mezi jednotlivymi vlastnostmi jsou ztraceny, jelikoz jsou ulozeny do poli
{
  "group" :        "fans",
  "user.first" : [ "alice", "john" ],
  "user.last" :  [ "smith", "white" ]
}
```

Výpis 6.2: Příklad demonstrující zplošťování seznamu objektů

Tento problém je vyřešen následovně:

1. jsou vytvořeny dva klony původního objektu události (`devops_all` a `devops_stages`) za pomoci rozšíření `clone`.
2. z klonu `devops_all` je odebráno pole `[stages]`. Následně je vytvořeno pole `[@metadata][target_index]`. Tomuto poli je přiřazena hodnota `"devopspipelines"`. Pomocí tohoto pole bude ve fázi výstupu určeno, na který index v rámci ElasticSearch má být dokument odeslán. Dále jsou pak pro jednotlivé úlohy přidány prahové hodnoty pro definované metriky (je popsáno níže).
3. z klonu `devops_stages` jsou odebrána všechna pole kromě `[stages]`, `[buildId]` a `[pipelineId]` (využito rozšíření `prune`). Pole `[buildId]` a `[pipelineId]` jsou zachována z důvodu spárování s během úlohy, ke kterému jednotlivé podúlohy náleží.
4. Pomocí rozšíření `mutate` je do objektu události přidáno pole pro rozpoznání cílového indexu `[@metadata][target_index]`. Pomocí tohoto rozšíření je také přidáno pole `[uniqId]`, které je vytvořeno konkatenací pole `[pipelineId]` a `[buildId]`. Toto pole pak slouží jako jednoznačný identifikátor konkrétního běhu úlohy, ke kterému podúlohy patří.
5. Za využití rozšíření `split` se vytvoří pro každý záznam v poli `[stages]` nový objekt události, který reprezentuje konkrétní podúlohu.
6. Nakonec se vytvoří pole `[buildIdDate]` pomocí rozšíření `date`. Toto pole obsahuje hodnotu `[buildId]` převedenou do formátu časového údaje. Důvod, proč je takové pole potřeba, je popsán v kapitole věnující se vizualizacím.

```
clone {
    clones => ['devops_all', 'devops_stages']
}
#pokud jde o~klon s~nazvem devops_stages
if ([type] == 'devops_stages'){
  #procistit ode vseho krome stages
  prune {
    whitelist_names => [ "stages", "buildId", "pipelineId" ]
  }
  #nastaveni ciloveho indexu
  mutate {
        add_field => { "[@metadata][target_index]" => "devopsstages" }
        add_field => { "[uniqId]" => "%{[pipelineId]}_%{[buildId]}" }
    }
  #split pres vsechny stage
  split { field => "[stages]" }
  date {
```

```
18          match => ["buildId", "UNIX_MS"]
19      target => "buildIdDate"
20      }
21  }
22  #pokud jde o~klon s~nazvem devops_all
23  if ([type] == 'devops_all') {
24      mutate {
25          #nastaveni ciloveho indexu
26          add_field => { "[@metadata][target_index]" => "devopspipelines" }
27          #odstraneni pole s~nazvem klonu a~odstraneni pole, ktere obsahuje vsechny stage
28          remove_field => [ "stages", "type" ]
29    }
30      ruby {
31        #doplneni trasholdu pro metriky a~vyhodnoceni, zda se jedna
32        #o outliery (byly prekroceny prahove hodnoty)
33        path => "/usr/share/Logstash/tresholds/rubyScripts/addThresholdJenkinsDevOps.rb"
34        script_params => {
35              "server" => "devops"
36      }
37    }
38  }
```

Výpis 6.3: Konfigurace filtrů pro zpracování dat ze serveru DevOps

V rámci průchodu objektu události zřetězenou linkou jsou objektům, které reprezentují informace o běhu úloh, přidány k definovaným metrikám prahové hodnoty. Také jsou vytvořeny příznaky, které jsem pojmenoval jako `outlier`. Každý takový objekt obsahuje globální příznak a příznaky pro jednotlivé metriky. Tyto příznaky jsou datového typu `boolean`. Pokud příznak náležící metrice nabývá hodnoty `true`, znamená to, že metrika překročila jednu z prahových hodnot (prahové hodnoty jsou dvě – horní a spodní). Globální příznak je pak nastaven na hodnotu `true` v případě, že alespoň jeden z příznaků metrik nabývá hodnoty `true` a nebo v případě, že stav úlohy není roven hodnotě "SUCCEEDED.

Pro toto přidání prahových hodnot a vyhodnocení jejich překročení je používán Ruby skript, jenž je volán během průchodu filtry za využití rozšíření `ruby` (viz. výpis 6.3, řádek 33). Pro spuštění skriptu je nutno zadat jeho cestu, popřípadě je možno předat skriptu parametry. V našem případě je předáván parametr `server`, který nabývá hodnoty `devops`. Na základě tohoto parametru skript rozliší, z kterého serveru data o úloze přišla, a jak s nimi má naložit. V rámci repozitáře se tento skript nachází ve složce `Logstash/thresholds/rubyScripts` a jeho název je `addThresholdJenkinsDevOps.rb`. V adresáři `Logstash/tresholds` se nachází soubor ve formátu JSON `devOpsPipelinesThresholds.json`. Tento soubor slouží jako konfigurace určující, které metriky jsou pro danou úlohu pozorovány, a vůči jakým prahovým hodnotám tyto metriky mají být porovnány. Aby byly tyto potřebné soubory dostupné i v rámci Logstash kontejneru, bylo třeba definovat svazky v `docker-compose.yml`.

```
1  {
2      "85": { //unikatni identifikator ulohy (pipelineId)
3          "metrics": [ //seznam obsahujici metriky, jez maji byt kontrolovany
4              {
5                  "name": "durationMillis", //jmeno metriky
6                  //pokud neni treba hlidat spodni nebo horni prahovou hodnotu
7                  //nastavi se prahova hodnota na null
8                  "upperLimit": 290000, //horni prahova hodnota metriky
9                  "lowerLimit": 200000  //spodni prahova hodnota metriky
10             }
11         ]
```

```
    },
    "86": {
        "metrics": [
            {
                "name": "durationMillis",
                "upperLimit": 700000,
                "lowerLimit": 550000
            },
            {
                //!!!!metrika v~zanorenem poli objektu udalosti (viz. odstavec nize)!!!!
                "name": "packagesContentInfo..Phoenix.Phar.EventSystem.Package..size",
                "upperLimit": 210000000,
                "lowerLimit": 138000000
            }
            //pro kazdou ulohu muze byt definovano vice pozorovanych metrik
        ]
    }
}
```

Výpis 6.4: Ukázka konfiguračního souboru `devOpsPipelinesThresholds.json`

Po průchodu tímto skriptem je objekt události přetvořen následovně:

```
//puvodni struktura objektu udalosti bez prahovych hodnot metrik
{
    /*
     * vlastnosti objektu
     */
     "metrika1": 350, //metrika cislo 1
     "metrika2": 420  //metrika cislo 2
}
//vysledna struktura po pruchodu Ruby skriptem
{
    /*
     * vlastnosti objektu
     */
     "metrika1.value": 350,
     "metrika1.upperLimit": 450,
     "metrika1.lowerLimit": 320,
     "metrika1.outlier": false,

     "metrika1.value": 420,
     "metrika1.upperLimit": 410,
     "metrika1.lowerLimit": 400,
     "metrika1.outlier": true,

     "globalOutlier": true

}
```

Jak je vidět z příkladu, definování nových úloh, kterým mají být přidělovány prahové hodnoty pro dané metriky, je poměrně jednoduché. Pozor si musíme dát, pokud je potřeba pozorovat metriku, jež je zanořená v některém z polí objektu události. V příkladu výše jsou u úlohy s identifikátorem 86 přidělovány prahové hodnoty metrice velikosti balíčku. Tato metrika je v rámci objektu události uložena v poli `[packagesContentInfo] [Phoenix.Phar.EventSystem.Package] [size]`. V konfiguračním souboru se jednotlivé úrovně zanoření pole oddělují pomocí `..` (dvou teček). V tomto případě tedy do klíče `name` přiřadíme hodnotu `packagesContentInfo..Phoenix.Phar.EventSystem.Package..size`.

Ruby skript je velmi detailně komentován ve zdrojovém kódu, a proto nepovažuji za nutnost seznamovat čtenáře detailněji s jeho implementací.

Output

V této podkapitole je popsána konfigurace výstupu pro data o bězích úloh spouštěných na serveru DevOps. Nejprve je na základě podmínky, která kontroluje přítomnost pole `[@metadata] [target_index]`, rozhodnuto, jestli na výstup budou data poslána. Tímto je ošetřeno, že do ElasticSearch databáze nebudou poslána žádná nevalidní data, která by neprošla filtry. Pro výstup je použit rozšíření `elasticsearch`. V rámci tohoto rozšíření je třeba definovat, kde je komponenta ElasticSearch hostována. V našem případě se jedná o kontejner `elasticsearch`, který poslouchá na portu 9200. Dále je pak třeba definovat uživatele a jeho heslo, které jsme nastavili během zprovozňování nástroje ELK Stack. Důležité je taktéž nastavit cílový index, do kterého bude dokument uložen. Název indexu je vytvořen z hodnoty, jež je obsahem pole `[@metadata][target_index]`

```
output {
  if ([@metadata][target_index]){
    elasticsearch {
      hosts => "elasticsearch:9200"
      user => "JMENO_UZIVATELE"
      password => "HESLO_UZIVATELE"
      index => "Logstash-%{[@metadata][target_index]}"
    }
  }
}
```

Výpis 6.5: Konfigurace výstupu

V průběhu ladění konfigurace zřetězené linky v nástroji Logstash se osvědčilo rozšíření `stdout`, který vypisuje výstupní dokument na standardní výstup. V rámci `stdout` je opět možné definovat kodek, jakým mají být data zpracována. Pro ladění je vhodný `json` a nebo `rubydebug`, který dokument interpretuje jako objekt skriptovacího jazyka Ruby.

```
output{
    stdout {
      codec => rubydebug
  }
}
```

Výpis 6.6: Konfigurace výstupu vhodná pro ladění zřetězené linky

6.2.2 Konfigurace zřetězené linky pro data ze serveru Jenkins

Jak již bylo zmíněno, pro zpracování dat pomocí nástroje Logstash je použita jedna konfigurace zřetězené linky, tedy `devOpsJeninsMonitoring.conf`. Konfigurace vstupu a výstupu je pak zcela totožná.

Input

Input pro data z obou serverů je stejný. Logstash tedy přijímá zprávy od aplikace, jež komunikuje pomocí TCP protokolu na portu 5000. Konfigurace již byla popsána v kapitole 6.2.1.

Filters

V předchozí kapitole, kde byla popsána implementace filtrů včetně hlavní podmínky, která další filtrování objektu události rozdělila do dvou hlavních větví, a to pro objekty události, jež představují data o úlohách ze serveru DevOps, a pak data o úlohách ze serveru Jenkins (toto větvení je popsáno ve výpisu 6.1).

Ze struktury dat (popsáno v příloze B), jež jsou v rámci vstupního bloku převedena z formátu JSON do reprezentace pomocí objektu události, je zřejmé, že pokud by byl objekt události předán v tomto formátu na výstup, došlo by v rámci databáze ElasticSearch opět ke zploštění seznamu objektů, a to u polí `[stages]` a `[testResult][testRunResults]`. Problém se zploštěním je vyřešen podobně jako u dat ze serveru DevOps. Zde však musí být vytvořeny klony objektu události tři. Výsledné objekty události jsou pak rozřazovány na tři různé indexy v rámci ElasticSearch databáze. Tyto indexy byly popsány pomocí tabulky 5.4. Průchod filtry tedy probíhá následovně:

1. jsou vytvořeny tři klony původního objektu události (`jenkins_all`, `jenkins_tests` a `jenkins_stages`) za pomoci rozšíření clone.
2. z klonu jenkins_all je odebráno pole `[stages]` (obsahuje informace o jednotlivých podúlohách), `[testsResult][testsRunResult]` (obsahuje výsledky testů) a `[type]` (pole obsahující název klonu, je vytvořeno pomocí rozšíření `clone`). Poté je vytvořeno pole `[@metadata][target_index]`. Tomuto poli je následně přiřazena hodnota `"jenkinsjobs"`. Pomocí tohoto pole bude ve fázi výstupu určeno, na který index v rámci ElasticSearch má být dokument odeslán. Dále jsou pak pro jednotlivé úlohy přidány prahové hodnoty pro definované metriky (je popsáno níže).
3. z klonu `jenkins_stages` jsou odebrána všechna pole kromě `[stages]`, `[buildId]` a `[pipelineId]` (využito rozšíření `prune`). Pole `[buildId]` a `[pipelineId]` jsou zachována z důvodu spárování s během úlohy, ke kterému jednotlivé podúlohy náleží.
4. Pomocí rozšíření `mutate` je do objektu události přidáno pole pro rozpoznání cílového indexu `[@metadata][target_index]` s hodnotou `"jenkinsstages"`.
5. Za využití rozšíření `split` se vytvoří pro každý záznam v poli `[stages]` nový objekt události, který reprezentuje konkrétní podúlohu.

6. Nakonec se vytvoří pole `[buildIdDate]` pomocí rozšíření `date`. Toto pole obsahuje hodnotu `[buildId]` převedenou do formátu časového údaje. Důvod, proč je takové pole potřeba, je popsán v kapitole věnující se vizualizacím.

7. Je třeba odstranit všechna přítomná pole kromě `[testsResult][testsRunResult]`, `[pipelineId]` a `[buildId]`. Pokud má být zachováno zanořené pole, v tomto případě `[testsResult][testsRunResult]`, je potřeba ho zkopírovat do nového pole, které zanořeno není. Toho je dosaženo pomocí rozšíření `mutate` a příkazu `copy`. Vytvořeno je nové pole `[testsRunResult]`, které obsahuje původní hodnotu pole jenž je smazáno (`[testsResult][testsRunResult]`).

8. Z klonu `jenkins_tests` jsou odebrána všechna stávající pole kromě `[testsRunResult]`, `[pipelineId]` a `[buildId]` (za využití `prune`). Pole `[buildId]` a `[pipelineId]` jsou zachována z důvodu spárování s během úlohy, ke kterému jednotlivé testy náleží.

9. Pomocí rozšíření `mutate` je do objektu události přidáno pole pro rozpoznání cílového indexu `[@metadata][target_index]`.

10. Za využití rozšíření `split` se vytvoří pro každý záznam v poli `[testsRunResult]` nový objekt události, který reprezentuje konkrétní běh testu.

11. Pomocí rozšíření `mutate` je přidáno pole `[testsRunResult][uniqId]`, které slouží jako unikátní identifikátor každého testu. Tento unikátní identifikátor vznikne konkatenací hodnot polí `[testsRunResult][testClass]` a `[testsRunResult][testName]`.

12. Nakonec se vytvoří pole `[buildIdDate]` pomocí rozšíření `date`. Toto pole obsahuje hodnotu `[buildId]` převedenou do formátu časového údaje. Důvod, proč je takové pole potřeba, je popsán v kapitole věnující se vizualizacím.

```
clone {
  clones => [ 'jenkins_all', 'jenkins_tests', 'jenkins_stages']
}
if ([type] == 'jenkins_stages'){
    #smaze vsechna pole krome vyjmenovanych
    prune {
        whitelist_names => [ "stages", "buildId", "pipelineId" ]
    }
    #nastaveni ciloveho indexu
    mutate {
        add_field => { "[@metadata][target_index]" => "jenkinsstages" }
    }
    #split pres vsechny podulohy
    split { field => "[stages]" }
    #dopocitani pomocneho pole
    date {
        match => ["buildId", "UNIX_MS"]
        target => "buildIdDate"
    }
}
if ([type] == 'jenkins_tests'){
    #procistit ode vseho krome testu, pipelineId a buildId
    mutate {
        copy => { "[testsResult][testsRunResult]" => "testsRunResult" }
    }

```

```
27      prune {
28        whitelist_names => [ "buildId", "pipelineId", "testsRunResult" ]
29      }
30      #nastaveni ciloveho indexu
31      mutate {
32      add_field => { "[@metadata][target_index]" => "jenkinstests" }
33      }
34      #split pres vsechny testy
35      split { field => "testsRunResult" }
36      #class+name testu = unikatni identifikator
37      mutate {
38      add_field => { "[testsRunResult][uniqId]" => "%{[testsRunResult][testClass]}.%{[
        testsRunResult][testName]}" }
39    }
40    date {
41          match => ["buildId", "UNIX_MS"]
42      target => "buildIdDate"
43    }
44  }
45  if ([type] == 'jenkins_all'){
46      mutate {
47          #nastaveni ciloveho indexu
48          add_field => { "[@metadata][target_index]" => "jenkinsjobs" }
49          #odstraneni pole s~nazvem klonu, odstraneni pole, ktere obsahuje vsechny stage a~
        pole s~testy
50          remove_field => [ "stages", "type", "[testsResult][testsRunResult]" ]
51    }
52    ruby {
53        #doplneni prahovych hodnot pro metriky a~vyhodnoceni, zda se jedna o~outliery
54        path => "/usr/share/Logstash/tresholds/rubyScripts/addThresholdJenkinsDevOps.rb"
55        script_params => {
56            "server" => "jenkins"
57        }
58    }
59  }
```

Výpis 6.7: Konfigurace výstupu vhodná pro ladění zřetězené linky

Podobně jako u dokumentů, jež nesou informace o jednotlivých bězích úloh spouštěných na serveru DevOps, lze i pro úlohy ze serveru Jenkins nastavit prahové hodnoty pro vybrané metriky. Toho je docíleno pomocí již zmíněného Ruby skriptu `addThresholdJenkinsDevOps`. Zde je volán skript pouze s jiným parametrem `server`. Hodnota tohoto parametru je rovna `"jenkins"`. Hodnoty prahových hodnot jednotlivých metrik pro konkrétní úlohy jsou pak definovány opět v souboru formátu JSON, `jenkinsJobsThresholds.json`. Soubor má totožnou strukturu, jak bylo ukázáno ve výpisu 6.4. Na rozdíl od úloh serveru DevOps není identifikátor úlohy datového typu integer, ale string, který představuje i jméno úlohy (např. UNIT_TESTS_COME_RELEASE).

Output

Konfigurace výstupu je totožná s předešlou konfigurací. Na základě hodnoty obsažené v poli `[@metadata][target_index]` je dokument vzniklý z objektu události zaslán na náležitý index. Její definice je zobrazena na výpisu 6.5 a popsána v kapitole 6.2.1.

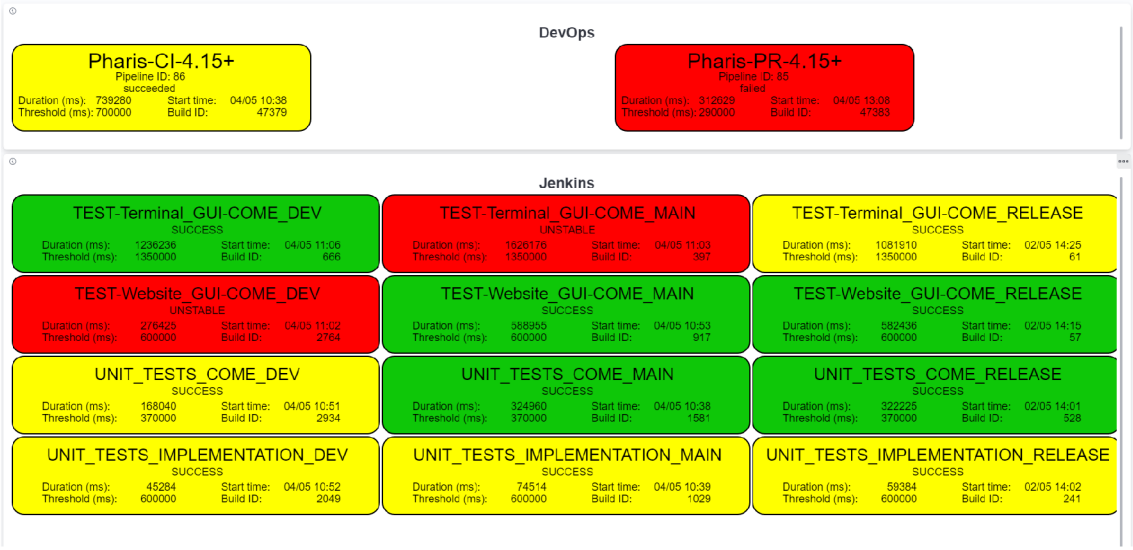
6.3 Monitoring automatizovaných procesů – vizualizace pomocí nástroje Kibana

Tato kapitola popisuje, jak jsou realizovány jednotlivé vizualizační obrazovky pro monitoring automatizovaných procesů. S hlavní myšlenkou jednotlivých obrazovek a s jejich účelem jsme se obeznámili v kapitole 5.3.2. Každé obrazovce je věnována krátká kapitola, v rámci které je popsána realizace jednotlivých komponent, z nichž se obrazovka skládá, a ukázáno výsledné GUI. V podkapitolách, které se věnují jednotlivým obrazovkám, jsou vždy popsány stěžejní problémy, které musely být vyřešeny.

Důležité je zmínit, že každá vizualizační obrazovka v nástroji Kibana má svoje filtry. V rámci obrazovky uživatel může filtry přidávat ručně nebo pomocí interaktivních vizualizačních komponent. Každý filtr je pak definovaný pro jeden tzv. index pattern (jeden index pattern může pokrývat i několik indexů). Podrobnou dokumentaci filtrů najdete na oficiálních stránkách nástroje Kibana. Důležitý je také časový filtr, který určuje, z jakého časového úseku nás zajímají dokumenty, na základě kterých jsou vizualizace vykresleny. Každé komponentě se dá tento časový filtr nastavit samostatně.

6.3.1 Realizace obrazovky Jobs_Status_Monitoring

Tato obrazovka se skládá ze dvou komponent napsaných pomocí gramatiky Vega. První komponenta zobrazuje stav úloh spouštěných pomocí serveru DevOps, druhá pak pomocí serveru Jenkins (viz. kapitola 5.1.1). Finální podoba obrazovky je zobrazena na obrázku 6.1.



Obrázek 6.1: Finální GUI obrazovky Jobs_Status_Monitoring

První komponenta na této vizualizační obrazovce je realizována pomocí gramatiky Vega. V kapitole 4.2 byl čtenář seznámen s hlavními vlastnostmi této gramatiky. Jak zde bylo popsáno, jedná se o tzv. data-driven vizualizační gramatiku. Veškeré vizualizace jsou tak založeny na datech, která mohou být různými způsoby zobrazena.

V rámci nástroje Kibana je tato gramatika plně podporována a jsou zde implementovány některé rozšiřující funkcionality. Nejdůležitější funkcionalitou je možnost provolat v bloku, kde jsou definována data, libovolný ElasticSearch dotaz a získat tak potřebná data. Této vlastnosti se dá dobře využít při vytváření složitějších vizualizací (například s daty z několika indexů), které by pomocí komponent poskytovaných nástrojem Kibana nebylo možné.

```
"data": [
    {
        "name": "NAME_OF_DATASET"
        "url": {
            "index": "TARGET-INDEX",
            "body": {
                //pozadovany ElasticSearch dotaz
            }
        }
        //vysledek dotazu je typicky nekolik zaznamu, ktere jsou ulozeny v poli
        //pomoci radku nize se definuje, ktere pole bude pouzito jako dataset
        "format": { "property": "aggregations.nameOfAgragation.buckets" }
    }
]
```

Výpis 6.8: Ukázka volání ElasticSearch dotazu uvnitř bloku `data` v gramatice Vega.

Tato komponenta má zobrazovat stav posledních běhů úloh serveru DevOps. Za účelem získání posledních běhů testů je v rámci bloku data vytvořen následující dataset, který pomocí dvojité agregace nejprve zařadí všechny běhy testů do skupin podle stejných názvů a poté tyto skupinky setřídí sestupně podle času, kdy tyto běhy byly dokončeny. Z každé skupinky je vybrán první běh (v rámci skupiny).

```
{
    "name": "pipelines"
    "url": {
        "index": "Logstash-devopspipelines",
        "body": {
            "_source": false,
            "aggs" : {
                "pipelinesLastRuns" : {
                    "terms" : {
                        "field" : "pipelineName.keyword"
                    },
                    "aggs":{
                        "last" : {
                            "top_hits": {
                                "size": 1,
                                "sort": {
                                    "finishTime": {
                                        "order" : "desc"
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    "format": { "property": "aggregations.pipelinesLastRuns.buckets" }
}
```

Každý záznam v tomto datasetu představuje jednu dlaždici vizualizace. Veškeré vizualizace napsané pomocí nástroje Vega musí mít definovaná měřítka. Pomocí těchto měřítek jsou pak spočteny pozice jednotlivé dlaždice. Pro pozicování je pro každý záznam v datasetu dopočítán sloupec (`col`) a řádek (`row`), ve kterém má být záznam zobrazen. Výpočty nových polí pro datasety jsou prováděny pomocí transformací [14]. Dále je pro každý záznam také vytvořena URL adresa, na kterou je uživatel přesměrován po kliknutí na záznamu náležící dlaždici. URL adresa má vždy následující formát:

```
../app/dashboards#/view/{ID OBRAZOVKY}?_a=(filters:!((query:{POZADOVANY FILTR})))
```

URL adresa tedy vede na požadovanou obrazovku, která zobrazuje detail vybrané úlohy. V rámci URL jsou pak předány i filtry, pomocí kterých se na další obrazovce budou zobrazovat pouze data pro vybranou úlohu. Tyto filtry jsou v rámci URL zapsány ve formátu Rison [12], což je formát velmi podobný JSON formátu, ale je podstatně vhodnější pro užití v URL adresách. V našem případě budou doplněny fitry nad poli `[pipelineId]` a `[buildId]`. Filtry v URL jsou tedy klasické ElasticSearch dotazy, které jsou převedeny do tohoto formátu. Pro tyto převody je vhodné použít například nástroj RISON online decoder [16].

Obdobně funguje i druhá komponenta obrazovky s tím rozdílem, že jsou získávána data o posledních bězích úloh které jsou spouštěny pomocí automatizačního serveru Jenkins. Hlavní rozdíl v implementaci je tedy v bloku data při definici ElasticSearch dotazu.

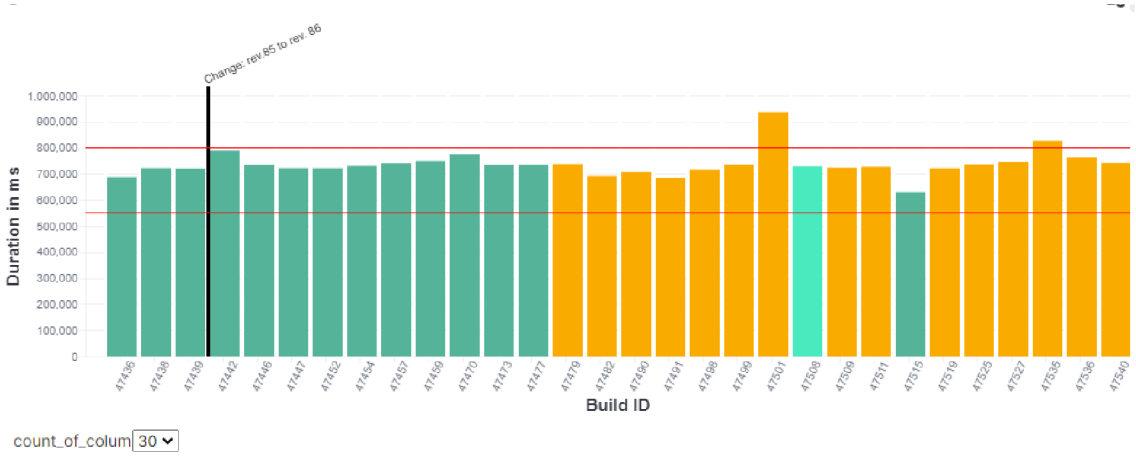
6.3.2 Realizace obrazovky DevOps_Pipelines_pipeline85

Tato obrazovka vizualizuje běhy úloh s unikátním identifikátorem 85. Konkrétně se jedná o úlohu Pharis-PR-4.15+. Finální GUI této vizualizační obrazovky se skládá z pěti komponent. V záhlaví obrazovky je komponenta zobrazující název úlohy. Pro tuto komponentu stačilo využít statické textové pole, neboť obrazovka slouží k zobrazování pouze jedné úlohy, a tak se obsah této komponenty nikdy nemění.

Další a současně nejzajímavější komponentou této obrazovky je sloupcový graf realizovaný pomocí gramatiky Vega, jenž slouží k zobrazování několika posledních běhů této úlohy (množství zobrazených běhů se dá změnit pomocí drop-down seznamu). Jednotlivé sloupce představují konkrétní běhy a výška sloupců pak dobu trvání náležitého běhu. Kromě sloupců jsou v grafu vykresleny prahové hodnoty pro délku běhu. Sloupce pak mohou nabývat tří barev.

- Zelená – úspěšný běh s očekávaným výsledkem a bez nastaveného globálního příznaku pro překročení metriky.
- Žlutá – úspěšný běh s očekávaným výsledkem, ale s nastaveným globálním příznakem pro překročení metriky.
- Červená – běh s neočekávaným výsledkem.

Každý ze sloupců je interaktivní a při kliknutí na něj se změní filtry obrazovky. Konkrétně se jedná o filtr pro pole `buildId`. Změnou tohoto filtru se pak změní obsah zbylých komponent a detailnější informace jsou zobrazovány pro zvolený běh (na základě pole `buildId`).



Obrázek 6.2: Ukázka interaktivního sloupcového grafu, který zobrazuje délku běhu v milisekundách. Tučná černá vertikální čára uživatele upozorňuje na změnu revize úlohy, která může být taktéž příčinou zhoršení výkonnosti. Červené horizontální čáry jsou spodní a horní prahová hodnota.

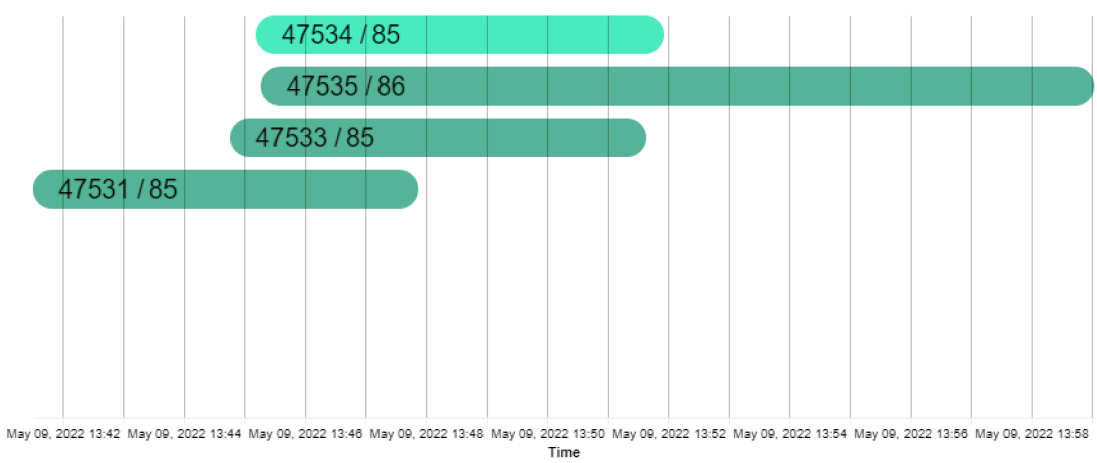
Pro nastavení filtrů je možno volat v rámci výrazů (`expr` [13]) gramatiky Vega tyto funkce:

```
/**
  * @param {object} query   => objekt predstavujici ElasticSearch dotaz
  * @param {string} [index] => cilovy index pro dotaz, pokud neni uveden,
  *                            je pouzit defaultni index
  * @param {string} [alias] => pokud ma byt v~GUI zobrazovan alias pro filtr, muze byt
  *                            definovan timto parametrem
  */
kibanaAddFilter(query, index, alias)

/**
  * @param {object} query   => objekt predstavujici ElasticSearch dotaz
  * @param {string} [index] => cilovy index pro dotaz, pokud neni uveden,
  *                            je pouzit defaultni index
  */
kibanaRemoveFilter(query, index)

/**
  * tato funkce nema zadny parametr a maze vsechny filtry vizualizacni obrazovky
  */
kibanaRemoveAllFilters()

/**
  * nastavi casovy filtr, tedy zacatek a konec intervalu, ktery nas zajima
  * @param {number|string|Date} start => zacatek intervalu
  * @param {number|string|Date} end   => konec intervalu
  */
kibanaSetTimeFilter(start, end)
```

Výše zmíněných funkcí je využito v rámci signálu, který reaguje na událost kliknutí myší na sloupec představující běh. Po kliknutí na sloupec se nejprve odstraní všechny stávající filtry. Na základě vybraného běhu se pak vytvoří filtry s novými hodnotami.

```
"signals":[
    {
        "name": "bar_click",
        "value": null,
        "on": [
            {
                "events": {
                    "source": "scope",
                    "type": "mousedown",
                    "markname": "bar"
                }
            "update": '''kibanaRemoveAllFilters()'''
            },
            {
                "events": {
                    "source": "scope",
                    "type": "mouseup",
                    "markname": "bar"
                }
                "update": '''kibanaAddFilter({"range": {"buildId": { "lte": datum.buildId,
    "gte": 0}}},'Logstash-devops*')'''
            },
            {
                "events": {
                    "source": "scope",
                    "type": "mouseup",
                    "markname": "bar"
                }
                "update": '''kibanaAddFilter({"match_phrase": {"pipelineId": datum._source.
    pipelineId}},'Logstash-devops*')'''
            }
        ]
    },
    /*
     * Dalsi signaly
    */
]
```

Jak již bylo zmíněno, délka běhu může být ovlivněna tím, že na automatizačním serveru je spuštěno více úloh souběžně. K tomu, aby měl uživatel přehled o tom, které úlohy běžely souběžně s vybraným během, slouží komponenta s časovou osou. Tato komponenta je opět realizována pomocí gramatiky Vega. Na časové ose se vždy zobrazují pouze souběžné úlohy k aktuálně vybranému běhu pomocí filtrů. V tomto případě je vhodné využít následujících proměnných, ve kterých jsou uloženy hodnoty filtru a časového filtru. Jedná se o `%timefilter%` a `%dashboard_context-filter_clause%`. Tyto proměnné mohou být vloženy do ElasticSearch dotazu, jenž je volán v rámci bloku `data`, čímž je zaručeno, že budou vybrány dokumenty, které odpovídají nastaveným filtrům.



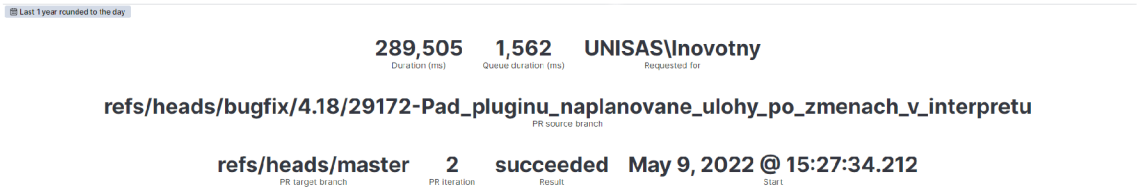
Obrázek 6.3: Časová osa zobrazující souběžné běhy. V tomto případě byly vykresleny souběžné běhy pro běh s `buildId` rovno 47534. Souběžné běhy jsou vyznačeny tmavším odstínem barvy.

Další část vizualizace byla vytvořena pomocí komponenty zobrazující hodnoty posledních dokumentů, které byly vyfiltrovány na základě filtrů vizualizační obrazovky (tedy hodnoty pro aktuálně vybraný běh v komponentě se sloupcovým grafem běhů). Pro tuto vizualizaci je využita jedna ze základních agregačních komponent zobrazujících metriky. Jsou zde zobrazeny poslední hodnoty následujících polí:

- `durationMillis.value` – délka běhu v milisekundách
- `queueDurationMillis` – délka čekání na spuštění běhu ve frontě úloh
- `requestedFor.uniqueName` – unikátní identifikátor uživatele, jenž zažádal o vložení kódu do větve
- `parameters.system.pullRequest.sourceBranch` – zdrojová větev, z níž mají být přeneseny změny
- `parameters.system.pullRequest.targetBranch` – cílová větev žádosti o vložení kódu
- `parameters.system.pullRequest.iteration` – iterace žádosti o vložení kódu
- `result` – výsledek běhu úlohy
- `startTime` – čas, kdy byla úloha spuštěna

Poslední komponenta je vytvořena pomocí nástroje Lens, který umožňuje uživateli vytvářet jednoduché grafy, vizualizace metrik, tabulky atd. pomocí GUI. Tato komponenta je tabulka obsahující informace o jednotlivých podúlohách, které byly spuštěny během vybraného běhu úlohy. V tabulce jsou tedy zobrazena data z indexu `Logstash-devopsstages`. Každý řádek představuje jednu podúlohu. Pro podúlohy je zobrazen čas trvání, průměrný čas trvání, počet chyb a URL adresa na server DevOps, kde je zobrazen záznam událostí konkrétní podúlohy. Aby pole mohlo být zobrazeno jako URL, je třeba nastavit tuto vlastnost ve vzoru

Indexu. V návrhu je zmíněno, že by tato komponenta měla být interaktivní a po kliknutí na konkrétní podúlohu by měl být uživatel přesměrován na obrazovku `DevOps_Stages`. Tento přechod na obrazovku je realizován pomocí tzv. Drilldowns [1]. Po přechodu na další obrazovku jsou zachovány stávající filtry a je doplněn filtr pro pole `stages.name`, aby na navazující vizualizační obrazovce byla zobrazena pouze data týkající se vybrané podúlohy.



Obrázek 6.4: Komponenta zobrazující důležité informace pro vybraný běh úlohy.

6.3.3 Realizace obrazovky DevOps_Pipelines_pipeline86

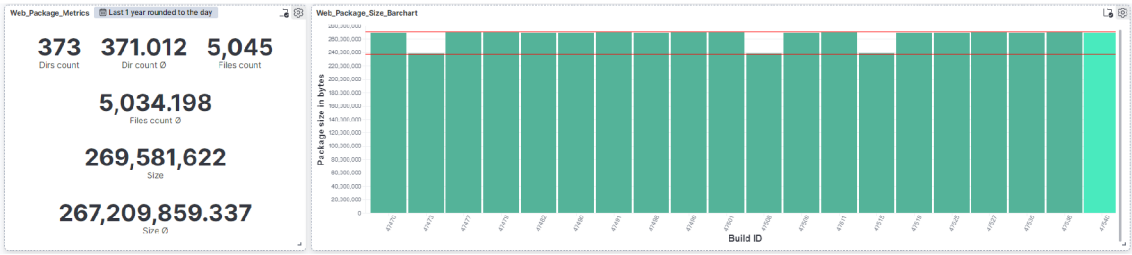
Tato obrazovka vizualizuje běhy úloh s unikátním identifikátorem 86. Jde tedy o úlohu Pharis-CI-4.15+. Základ obrazovky je stejný jako u `DevOps_Pipelines_pipeline85`. U obrazovky pro vizualizaci této úlohy bylo nutno upravit komponentu, která zobrazuje detaily vybraného běhu. V tomto případě jsou zobrazovány poslední hodnoty následujících polí:

- `durationMillis.value` – délka běhu v milisekundách
- `queueDurationMillis` – délka čekání na spuštění běhu ve frontě úloh
- `requestedFor.uniqueName` – unikátní identifikátor uživatele, jenž zažádal o vložení kódu do větve
- `sourceBranch` – větev, nad kterou byla úloha provedena
- `result` – výsledek běhu úlohy
- `startTime` – čas, kdy byla úloha spuštěna

Dále se také během CI sestavení vytváří balíčky jednotlivých částí aplikace. Konkrétně se jedná o balíček webového klienta, event–system balíček a balíček terminálové aplikace. Informace o těchto balíčcích jsou uloženy v následujících polích:

- `packagesContentInfo.[NAZEV_BALICKU].Package.dirs` – počet adresářů v balíčku;
- `packagesContentInfo.[NAZEV_BALICKU].Package.files` – počet souborů v balíčku;
- `packagesContentInfo.[NAZEV_BALICKU].Package.size.value` – velikost balíčku v bajtech;
- `packagesContentInfo.[NAZEV_BALICKU].Package.items` – počet jednotlivých položek (adresářů a souborů).

Hodnoty těchto polí pro jednotlivé balíčky jsou opět vizualizovány pomocí komponenty, která umožňuje zobrazovat poslední hodnoty polí. Pro každý balíček je také vytvořena komponenta, která zobrazuje velikosti balíčků vytvořených v historii jinými běhy úloh. Uživatel tak může sledovat vývoj velikosti balíčků. Tato komponenta je opět sloupcový graf, ve kterém jednotlivé sloupce představují velikost balíčku vytvořeného během konkrétního běhu v bajtech. V grafu jsou zvýrazněny i prahové hodnoty.



Obrázek 6.5: Vizualizace vlastností balíčku a historie velikosti balíčku.

6.3.4 Realizace obrazovky DevOps_Stage

Tato obrazovka má za úkol zobrazovat detail konkrétní podúlohy. Skládá se ze dvou komponent napsaných v gramatice Vega. Jedna z komponent zobrazuje detail o vybraném běhu podúlohy v textové podobě. Druhá komponenta je sloupcový graf, kde každý sloupec představuje délku běhu podúlohy v milisekundách pro jednotlivé běhy úlohy (tzn. vývoj délky běhu podúlohy v několika posledních bězích).

Již bylo zmíněno, že pokud chceme, aby byla komponenta psaná pomocí gramatiky Vega dynamická a měnila svůj obsah na základě filtrů nastavených na vizualizační obrazovce, můžeme využít proměnné `%dashboard_context-filter_clause%`. Tato proměnná obsahuje seznam filtrů v textové podobě tak, aby bylo možné ji použít v ElasticSearch dotazu. Velkou nevýhodou této proměnné je, že nemůže být upravena a v bloku data z ní uživatel nemůže odebrat filtry, které se mu nehodí. U této vizualizační obrazovky je v rámci jedné komponenty provedeno hned několik ElasticSearch dotazů. Některé z nich však potřebují pouze část filtrů vizualizační obrazovky. Konkrétně u některých není potřeba filtrovat na základě pole `buildId`. Nejprve jsem tento problém řešil tak, že jsem získal všechna data z indexu, a pak jsem z těchto dat pomocí transformací (transformace jsou kvalitně zdokumentované na oficiálních stránkách gramatiky Vega [14]) získal pouze ta potřebná. Pokud však ElasticSearch získává všechny dokumenty z indexu, je vykonání dotazu poměrně pomalé a neefektivní. Navíc je maximální počet takto získaných dokumentů omezen na 10 000. Rozhodl jsem se využít poněkud netradičního řešení.

1. Nejprve je odstraněn filtr nad polem `buildId`.
2. Hodnota z filtru nad polem `buildId` je převedena do časového údaje a je nastaven časový filtr s touto hodnotou.
3. Pokud je třeba filtrovat podle jednoznačného identifikátoru běhu (`buildId`), je využito již zmíněné pole `buildIdDate`, které obsahuje tento identifikátor převedený na časový záznam.

Výsledný ElasticSearch dotaz v bloku `data` pak může vypadat následovně:

```
"data": [
    {
        "name": "selected_by_filters",
        "%context%": true,
        "url": {
            "index": "Logstash-devopsstages",
            "body": {
                "query": {
                "bool": {
                "filter": ["%dashboard_context-filter_clause%",
                    {
                        "range": {
                            "buildId": {
                                "lte": {"%timefilter%": "max" },
                                "gte": null
                            }
                        }
                    }
                    ]
                }
                },
            "sort": [
                {
                    "stages.startTime": "desc"
                }
            ],
            "size": 1,
            }
        }
        "format": {"property": "hits.hits"},
    }
]
```

6.3.5 Realizace obrazovky Jenkins_Jobs_With_Tests

Tato vizualizační obrazovka má podobné rozložení a účel jako již zmíněné vizualizační obrazovky `DevOps_Pipelines_pipeline85` a `DevOps_Pipelines_pipeline86`. Hlavní rozdíl je ve zdroji dokumentů, které se zde vizualizují. V tomto případě se jedná o indexy, do nichž jsou ukládána data o běhu jednotlivých úloh automatizačního serveru Jenkins. Ke všem úlohám se váží kromě jednotlivých podúloh i testy, které jsou v rámci běhu spuštěny. Pro vizualizaci stavu jednotlivých testů, které byly spuštěny, je využito poskytované komponenty Lens. Pomocí této komponenty byla vytvořena tabulka, jejíž jednotlivé řádky představují spouštěné testy. Opět bylo využito interaktivního prokliku na navazující obrazovku, která zobrazuje detail vybraného testu (obrazovka `Jenkins_Tests`). Přechod na tuto obrazovku je realizován pomocí tzv. Drilldown. Po přechodu na další obrazovku jsou zachovány stávající filtry (filtrování konkrétního běhu pro danou úlohu) a je přidán filtr nad polem `testsRunResult.uniqId`, což je pole složené z třídy testu a názvu testu (unikátní identifikátor testu).

6.3.6 Realizace obrazovky Jenkins_Stage

Implementace této obrazovky je velice podobná jako tomu bylo u `DevOps_Stage`. Obrazovka je opět tvořena pomocí dvou vizualizací psaných v gramatice Vega. Opět bylo nutné vyřešit problém zmíněný v kapitole řešící implementaci vizualizační obrazovky `DevOps_Stage` 6.3.4. Musíme zde pouze přistupovat k datům z jiného indexu, ve kterém se mírně liší názvy jednotlivých polí a uložené informace o podúloze.

6.3.7 Realizace obrazovky Jenkins_Tests

Podobně jako `Jenkins_Stages` a `DevOps_Stages` je nutné vyřešit problém se získáváním dat pro vizualizace, které jsou psané pomocí gramatiky Vega. Obrazovka se příliš neliší od obrazovky zobrazující detail podúlohy. Rozdíl je opět pouze v indexu, ze kterého jsou data získávána, a ve struktuře tohoto indexu.

6.4 Výkonnostní testování – zpracování naměřených dat pomocí nástroje Logstash

Jak již bylo zmíněno v kapitole popisující návrh řešení (5.3.3), zdroje dat, jež jsou pro vizualizaci výsledků výkonnostních testů zpracovávány, jsou dva. Data jsou posílána pomocí nástroje Beats (tzv. shipper). Konkrétně se jedná o MetricBeat, prostřednictvím kterého je monitorována zátěž serveru hostujícího server systému MES PHARIS. Tento nástroj funguje jako služba, která pozoruje uživatelem definované metriky a sesbíraná data zasílá na definované místo, v našem případě Logstash. Druhým nástrojem je pak FileBeat, který odesílá soubory s logy vzniklé při běhu testů. Konfigurace těchto dvou nástrojů je popsána v diplomové práci Martina Oháňky.

Struktura logů, jež jsou odesílány prostřednictvím FileBeat, je uvedena v příloze pomocí tabulek C. Jedná se o logy ve formátu JSON, které představují jednotlivé události, které se odehrály během testu. V následujícím textu budou tyto logy označovány jako log události. Každý log události obsahuje klíč `RunnerId` datového typu `long`. Tento klíč představuje unikátní identifikátor běhu scénáře v rámci jednoho testu. Tzn. dá se říct, že pomocí tohoto identifikátoru lze určit, který simulovaný terminál zalogoval danou událost. Pokud je tento identifikátor roven `-1`, je tato zpráva logována přímo orchestrátorem. Takové zprávy obsahují informace o stavu celého testu jako jeho trvání či stav (úspěch/neúspěch).

Data získávaná pomocí těchto dvou nástrojů jsou zpracována pomocí jedné zřetězené linky v rámci nástroje Logstash. Tato zřetězená linka (`perfTestsBeats.conf`) je uložena ve stejném adresáři jako zřetězená linka určená ke zpracování dat o automatizovaných procesech. Jedna instance kontejneru, na kterém je provozován Logstash, dokáže spustit hned několik zřetězených linek současně. Pro tyto účely je ale třeba upravit konfiguraci. Konkrétně se jedná o dopsání dalších potřebných zřetězených linek do konfiguračního souboru `/Logstash/config/pipelines.yml`. Všechny zřetězené linky, které zde uživatel definuje, jsou pak spuštěny při startu kontejneru. V našem případě bude konfigurace vypadat následovně:

```
#soubezna linka pro zpracovani dat z aut. serveru Jenkins a DevOps
- pipeline.id: devOpsJenkinsMonitoring
  path.config: "/usr/share/Logstash/pipeline/devOpsJenkinsMonitoring.conf"

#soubezna linka pro zpracovani dat potrebnych pro vizualizaci vysledku vykonnostnich testu
- pipeline.id: perfTestBeats
  path.config: "/usr/share/Logstash/pipeline/perfTestBeats.conf"
```

Pokud by uživatel chtěl přidat nové zřetězené linky, je také třeba myslet na to, že pro její konfiguraci je nutné taktéž definovat nový svazek v `docker-compose.yml`.

6.4.1 Konfigurace zřetězené linky pro zpracování dat z výkonnostního testování

V této podkapitole je popsána konfigurace zřetězené linky pro zpracování dat potřebných k vizualizaci výsledků zátěžových testů. Popis konfigurace je opět rozdělen do tří částí, z nichž se zřetězená linka skládá. Zřetězená linka odesílá výsledné dokumenty na tři indexy, které byly popsány v tabulce 5.4.

Input

Obě služby, jenž posílají data nástroji Logstash, jsou z rodiny nástrojů Beats. Pro taková data je určeno rozšíření s názvem `beats`. Data jsou přijmána na portu 5044. Tento rozšíření implicitně počítá s předdefinovanou strukturou dat, jež je zasílána výše zmíněnými nástroji. Proto není třeba využít žádný speciální kodek.

```
input {
    beats {
        port => 5044
    }
}
```

Filters

Nejprve je třeba pomocí podmínky rozlišit, jestli je zpracovávaný objekt události z nástroje MetricBeat či FileBeat. Tato informace je uložena v poli `[agent][type]`. Dále je v podmínce ověřeno, jestli je služba zasílající data hostována na námi požadovaném serveru (`[agent][name]`). Tímto rozšířením podmínky eliminujeme cizí datové zdroje, které nás pro vyhodnocení testů nezajímají.

```
filter{
    if ([agent][type] == 'filebeat' and [agent][name] == 's3Performance'){
        #zde se zpracovava objekt udalosti, jenz byl prijat z filebeat
    }
    if ([agent][type] == 'metricbeat' and [agent][name] == 's3Performance'){
        #zde se zpracovava objekt udalosti, jenz byl prijat z metricbeat
    }
}
```

Ve větvi, která zpracovává logy událostí, je potřeba zpracovat data následovně:

1. Jednotlivé zprávy neobsahují žádný údaj, ze kterého by mohl být určen konkrétní běh testu. Každý záznam zaslaný pomocí FileBeat však obsahuje pole `[log][file][path]`, které obsahuje cestu k souboru, v němž byl záznam události uložen. Tento soubor obsahuje v názvu čas ve formátu `yyMMddHHmmss`. Tento čas je možno brát jako jednoznačný identifikátor testu, neboť testy jsou spouštěny po sobě, a tak lze vyloučit kolizi zapříčiněnou vznikem souboru se stejným názvem. Pomocí rozšíření `mutate` je vytvořeno nové pomocné pole `[@metadata][runId]`, které obsahuje cestu k původnímu souboru se záznamy událostí. Poté pomocí krátkého Ruby skriptu a rozšíření `ruby`, je vytvořeno pole `runId`, do kterého je z cesty vyjmut podřetězec obsahující čas v již zmíněném formátu.

2. Dále je potřeba zjistit, zda se jedná o záznam události od jednotlivých procesů, jež simulují chod terminálů (tzv. runner), nebo od orchestrátoru, tedy záznam události obsahující informace o celém průběhu testu. To je rozhodnuto na základě hodnoty v poli `[RunnerId]`. Pokud je hodnota rovna -1, jedná se o záznam události orchestrátoru.

3. V případě vyhodnocování objektu události, jenž náleží orchestrátoru, je ověřeno, zda se nejedná o log o zahájení testu. To je zjištěno díky obsahu pole `[Message][Text]`. Záznam o zahájení testu není pro vizualizace důležitý, a tak je možno jemu náležící objekt události pomocí volání `drop{}` odstranit.

4. Pokud se jedná o objekt události obsahující souhrn informací o výsledku testu, je mu přidáno pole `[@metadata][target_index]`, do kterého je vložena následující hodnota `"performance-events-testresult"`. Na základě této hodnoty bude opět rozhodováno, na který index bude výsledný dokument odeslán.

5. Do objektů událostí, které obsahují souhrn o výsledku testu, lze opět pro zvolené metriky přidat prahové hodnoty. To je opět provedeno pomocí rozšíření `ruby` a jím spouštěného Ruby skriptu. Vyhodnocení prahových hodnot funguje totožně jako u monitoringu automatizovaných procesů. Jak lze tyto hodnoty nakonfigurovat, je popsáno níže v této podkapitole.

6. Pokud se jedná o objekt události náležící logům jednotlivých procesů, není potřeba provádět žádné úpravy. Nutné je pouze vytvořit pole `[@metadata][target_index]` a nastavit jeho hodnotu na `performance-events`, které slouží k rozhodnutí, na který index budou data zaslána.

```
mutate {
    add_field => { "[@metadata][runId]" => "%{[log][file][path]}" }
}
ruby {
    code => "event.set('runId', event.get('[@metadata][runId]')[-16 .. -5])"
}
if ([RunnerId] == -1){
    if ([Message][Text] == "Test started"){
        drop{}
    }


```

```
    mutate {
        add_field => { "[@metadata][target_index]" => "performance-events-testresult" }
    }
    ruby {
      #doplneni prahovych hodnot pro metriky a vyhodnoceni, zda se jedna o outliery
        path => "/usr/share/Logstash/tresholds/rubyScripts/addThresholdPerfTests.rb"
    }
}
else {
    mutate {
        add_field => { "[@metadata][target_index]" => "performance-events" }
    }
}
```

Přiřazování prahových hodnot funguje obdobně jako u automatizovaných procesů. Za tímto účelem byl implementován druhý skript, s názvem `addThresholdPerfTest.rb`. Tomuto skriptu není předáván žádný argument na rozdíl od `addThresholdJenkinsDevOps.rb`, jelikož vždy přiděluje prahové hodnoty do objektů událostí se stejným formátem. Soubor, ve kterém jsou tyto prahové hodnoty definovány, je opět ve formátu JSON. Jedná se o `perfTestsThresholds.json`. Jako identifikátor jednotlivých testů se zde používá unikátní jméno pro každý test. Struktura tohoto souboru se pak neliší od již zmíněných JSON souborů, které se používají pro definici prahových hodnot u automatizovaných procesů.

V případě, kdy prochází objekt události druhou větví podmínky (jedná se o log o z nástroje MetricBeat), není potřeba do struktury objektu události zasahovat. Opět je jen nutné určit cílový index. To je zaručeno přidáním pole `[@metadata][target_index]`, do kterého je vložena hodnota `performance-metrics`.

```
mutate {
    add_field => { "[@metadata][target_index]" => "performance-metrics" }
}
```

Output

Nejprve je na základě podmínky, která kontroluje přítomnost dříve vytvořeného pole s názvem `[@metadata][target_index]`, rozhodnuto, jestli na výstup budou data poslána. Tímto je ošetřeno, že do ElasticSearch databáze nebudou poslána žádná nevalidní data, která by neprošla filtry. Pro výstup je použit opět rozšíření `elasticsearch`. Je třeba definovat, kde je komponenta ElasticSearch hostována a port, na kterém naslouchá. Dále je pak třeba definovat uživatele a jeho heslo, které jsme nastavili během zprovozňování nástroje ELK Stack. Na základě pole obsahujícího cílový index (`[@metadata][target_index]`) rozhodnuto, na který index bude výsledný dokument odeslán.

```
output {
    if ([@metadata][target_index]){
        elasticsearch {
            hosts => "elasticsearch:9200"
            user => "JMENO_UZIVATELE_PRO_ELASTIC_SEARCH"
            password => "HESLO_UZIVATELE"
            data_stream => "false"
            index => "Logstash-%{[@metadata][target_index]}"
        }
    }
}
```

6.5 Výkonnostní testování – vizualizace výsledků pomocí nástroje Kibana

6.5.1 Perf_Tests_Main_Dashboard

Tato obrazovka vizualizuje výsledky jednotlivých výkonnostních testů. Opět bylo využito nástroje Lens, ve kterém je vytvořena tabulka. Každý řádek v tabulce představuje poslední běh výkonnostního testu. Je zobrazován výsledek testu, doba trvání a taktéž je možno kontrolovat překročení definovaných prahových hodnot. Pomocí Drilldown uživatel může po kliknutí na jeden ze záznamů přejít na detail konkrétního testu, tedy na obrazovku `Performance_Test_Result`. Na této obrazovce je pak nastaven filtr nad polem `TestName` tak, aby byl zobrazen výsledek vybraného testu.

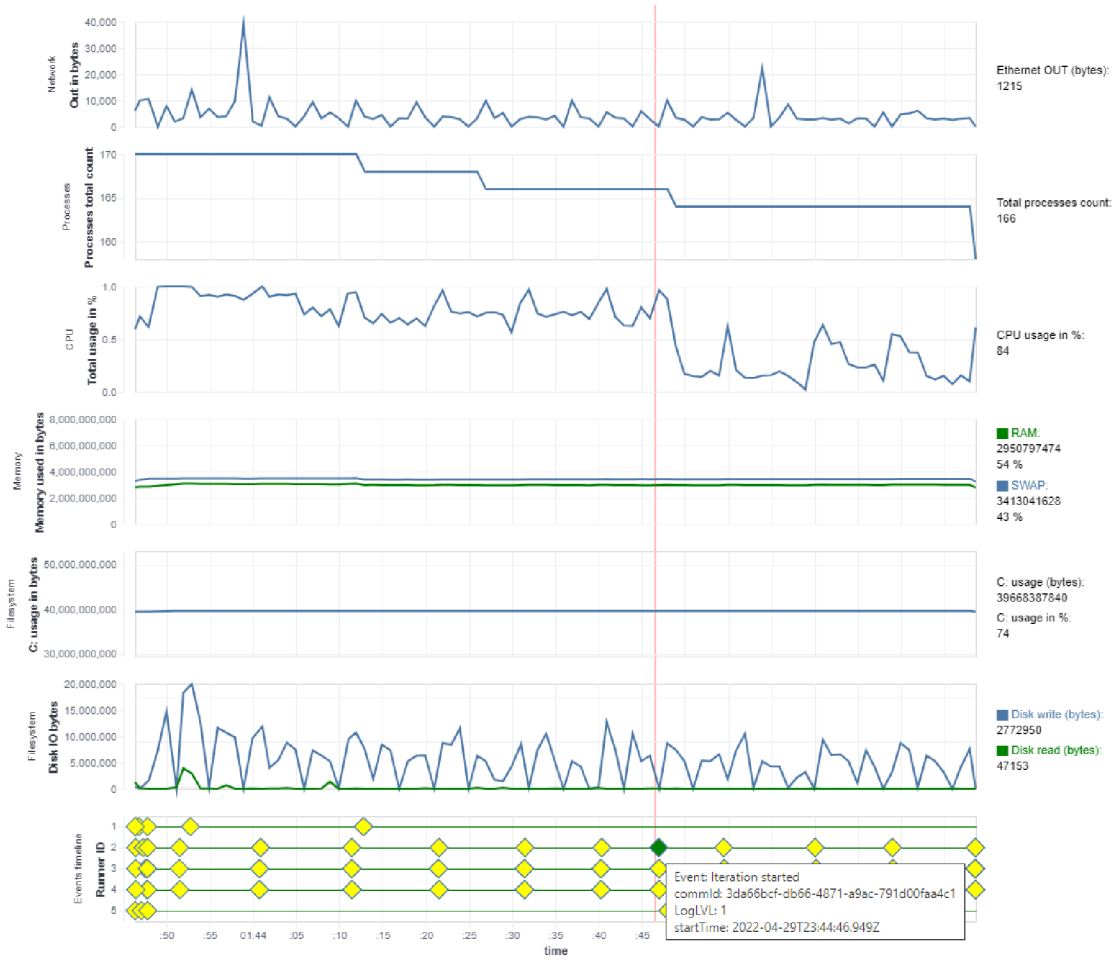
6.5.2 Performance_Test_Result

Obrazovka `Performance_Test_Result` zobrazuje výsledek výkonnostního testu. Opět byl implementován sloupcový graf v gramatice Vega. Každý sloupec představuje délku konkrétního běhu testu. Tento graf je interaktivní a po kliknutí na sloupec se změní filtr nad polem `runId` tak, aby byl vizualizován výsledek vybraného běhu testu.

Nejdůležitější komponentou je soubor několika grafů metrik a časová osa zobrazující události provedené v jednotlivých scénářích testu. Tato komponenta je opět napsána v gramatice Vega. Podstatou této komponenty je zobrazit naměřené hodnoty metrik v čase tak, aby uživatel mohl jednoduše zjistit, jaké události měly na měřené metriky vliv. Grafy jsou organizovány pod sebe a mají jednu společnou osu X, která představuje časový interval, během kterého byl test vykonáván. V této komponentě je velké množství ElasticSearch dotazů, pomocí kterých jsou získána data jednotlivých metrik. Před inicializací komponenty je nastaven časový filtr tak, aby začátek časového intervalu odpovídal startu testu a konec intervalu pak konci testu. V dotazech je pak opět používána proměnná `%timefilter%`, díky čemuž jsou vždy získány hodnoty metrik pouze pro vybraný test.

Časová osa zobrazující jednotlivé události vykresluje vždy několik horizontálních úseček, které představují jednotlivé scénáře. Události jsou pak zobrazeny jako symbol kosočtverců. Pro zobrazení detailu události byl implementován tooltip, který se zobrazí po najetí kurzorem nad tento symbol.

Jelikož je grafů zobrazujících hodnoty jednotlivých metrik poměrně velké množství, bylo pro lepší orientaci v grafu nutné implementovat vertikální ukazovátko, které protíná všechny tyto grafy, a uživatel tak může jednoduše pozorovat závislosti jednotlivých metrik. V pravé části tohoto elementu je pak zobrazována hodnota pro všechny metriky v bodě, kde je graf metriky protnut tímto vertikálním ukazovátkem. Tato hodnota je dopočítávána pomocí několika signálů.



Obrázek 6.6: Ukázka vizualizace výsledku testu.

6.6 Import vizualizačních obrazovek

Pro uvedení výše zmíněných vizualizačních obrazovek do provozu je nutné je importovat do dokerizovaného nástroje Kibana. Pro tento import byl napsán skript s názvem `prepareEnviromentScripts/importDashboardsAndVizualizations.sh`. Tento skript pak využívá nástroj Perfaide, který byl vytvořen v rámci projektu Prefekt [11]. Tento nástroj pomocí API, které poskytuje Kibana a ElasticSearch, ulehčuje práci uživatelů při importu a exportu vizualizací, vizualizačních obrazovek, indexů atd. Ve skriptu je pak využit následovně:

```
perfaide upload -f [NAZEV_IMPORTOVANEHO_SOUBORU]
```

Toto volání je pak provedeno pro každý soubor v podsložce `dashboardsAndVizualizations`, která obsahuje soubory s definicemi všech potřebných vizualizací, vizualizačních obrazovek, indexů a předpisů indexu (index pattern).

V případě, že jsou takto importované položky změněny v GUI nástroje Kibana, je nutné je opětovně exportovat a nahradit původními soubory v adresáři s vizualizacemi za nově exportované. Import je proveden pomocí následujícího příkazu.

```
perfaide ping
```

Kapitola 7

Testy výsledného řešení

Výsledné řešení monitoringu automatizovaných procesů a výkonnostních testů bylo a je testováno v provozu přímo vývojáři. Mimo jiné byl implementován skript, který spouští testy konfigurací zřetězených linek nástroje Logstash, a sada několika takových testů.

7.1 Automatická sada testů konfigurace nástroje Logstash

Implementace těchto testů byla inspirována řešením dostupným volně na GitHub repozitáři s názvem `logstash-test-runner` [26]. Pro spuštění těchto testů je nejprve nutné vytvořit obraz kontejneru. Pro vytvoření obrazu stačí ve složce `LogstashTests` spustit příkaz `docker-compose build`.

Pro spuštění testů je nutné mít nainstalovaný nástroj `jq` [7], který usnadňuje práci se soubory ve formátu JSON. Samotné testy se pak spouštějí provoláním hlavního skriptu `./runTests.sh`. Tento skript nemá žádné argumenty a je předpokládáno, že bude vždy spouštěn z adresáře, kde se nachází.

Testy jsou definovány v adresáři `TESTS` a jejich vyhodnocení probíhá následovně:

1. nejprve je ze souboru `testConfig.json` přečteno, která konfigurace zřetězené linky má být testována a který Ruby skript k této konfiguraci náleží.
2. konfigurace zřetězené linky je zkopírována do svazku, jenž je sdílen s obrazem nástroje Logstash. Totéž se stane i s Ruby skriptem.
3. konfigurace zřetězené linky je upravena následovně:
   (a) blok s definicí vstupu je odebrán a nahrazen definicí, která je obsažena v souboru `Logstash-input.conf`. Vstupem bude standardní vstup.
   (b) blok s definicí výstupu je odebrán a nahrazen definicí, která je obsažena v souboru `Logstash-output.conf`. Výstupem bude soubor s názvem `output.log`.

4. je spuštěn kontejner pomocí `docker-compose run -rm` (vykoná se pouze jeden příkaz, poté kontejner zanikne, což je zaručeno přepínačem `-rm`) a tomuto kontejneru je předán na standardní vstup soubor s předdefinovaným vstupem (`input.json`).

5. výstupní soubor je porovnán se souborem `expectedOutput.json`, pokud není nalezen rozdíl, je test úspěšný, jinak neúspěšný.

Porovnávání výstupu testu s očekávaným výstupem je provedeno pomocí příkazu:

```
diff <(jq -S -f walk.filter "$PWD/$test_dir/output.json") <(jq -S -f walk.filter "$PWD/$test_dir/testoutput.log")
```

Filtr `walk.filter` slouží k seřazení JSON souborů. Tento filtr byl převzat ze stránky Local Coder[1] a vypadá následovně:

```
# Apply f to composite entities recursively, and to atoms
def walk(f):
  . as $in
  | if type == "object" then
      reduce keys[] as $key
        ( {}; . + { ($key):  ($in[$key] | walk(f)) } ) | f
  elif type == "array" then map( walk(f) ) | f
  else f
  end;
walk(if type == "array" then sort else . end)
```

Níže je uvedena struktura adresáře s implementací těchto testů:

```
LogstashTests
├── TESTS ..................................... složka obsahující definice testů
│   ├── [TEST1] ..................................složka představující definici testu
│   │   ├── thresholds ..............složka obsahující konfiguraci prahových hodnot
│   │   ├── input.json ..................................... vstupní data testu
│   │   ├── expectedOutput.json ..........................očekáváná výstupní data
│   │   └── testConfig.json ..............................konfigurační soubor testu
│   ├── [TEST2] ..................................................... definice dalších testů
│   ├── .
│   ├── .
│   └── .
├── .env ..................................................obsahuje definici verze ELK
├── docker-compose.yml ..............slouží k vytvoření obrazu instance Logstash
├── Dockerfile ..................... Dockerfile k vytvoření obrazu instance Logstash
├── Logstash-input.conf ..................................... konfigurace vstupu
├── Logstash-output.conf ..................................konfigurace výstupu
├── runTests.sh ..............................hlavní skript, který spouští testy
└── walk.filter ..............................filtr k seřazení souborů formátu JSON
```

[1] https://localcoder.org/transforming-the-name-of-key-deeper-in-the-json-structure-with-jq

Kapitola 8

Závěr

Nejdůležitějším cílem této diplomové práce bylo vytvořit řešení, které bude využíváno při vývoji MES PHARIS. Během implementace byl kladen důraz na zpětnou vazbu vývojářů (cílových uživatelů). Část zabývající se monitoringem automatizovaných procesů považuji aktuálně za nejpřínosnější část mé diplomové práce. Podařilo se realizovat sběr dat o úlohách spouštěných na automatizačních serverech Jenkins a DevOps. Takto sesbíraná data jsou centralizována a vizualizována pomocí nástroje ELK Stack. Docílilo se tak jednotných a přehledných vizualizací, které se nacházejí na jednom místě. Tyto vizualizace se již používají v praxi a osvědčily se při detekování několika problémů, jako je například růst balíčků vytvářených těmito úlohami. Proto považuji řešení první části diplomové práce za úspěšné a naplňující očekávání firmy UNIS. Struktura získávání dat a jejich vizualizace je poměrně snadno rozšiřitelná, a tak je možné, že při potřebě monitorovat i jiné úlohy bude toto řešení do budoucna rozšiřováno.

Druhá část zabývající se výkonnostním testováním systému MES PHARIS je základem pro další rozšiřování výkonnostního testování. V rámci diplomové práce Martina Oháňky byly vytvořeny základní scénáře testující kritickou část výroby. Vizualizace naměřených dat jsou přehledné a pro uživatele přívětivé. Zatím ale nemáme tak velkou zpětnou vazbu, jako tomu je u první části diplomové práce. Do budoucna je v plánu rozšiřovat sadu testů a v případě potřeby upravovat i vizualizace. Realizace výkonnostních testů a následná vizualizace výsledků je firmou taktéž hodnocena kladně. Úplný potenciál těchto testů však bude dosažen až po rozšíření o další scénáře a testování jiných částí systému.

Práci hodnotím kladně i z toho důvodu, že jsem se setkal s pro mě novými technologiemi jako je například Docker, Vega či Ruby. Největší osobní přínos je pak pro mě zvládnutí i pokročilých technik v rámci nástroje ELK Stack.

Literatura

[1] *Make dashboards interactive* [online]. [cit. 2022-3-4]. Dostupné z: https://www.elastic.co/guide/en/kibana/current/drilldowns.html#drilldowns.

[2] *Oficiální stránky Elastick Stack* [online]. [cit. 2021-15-12]. Dostupné z: https://www.elastic.co.

[3] *Oficiální stránky gramatiky Vega* [online]. [cit. 2022-28-2]. Dostupné z: https://vega.github.io/vega/.

[4] *Oficiální stránky Graylog* [online]. [cit. 2022-1-1]. Dostupné z: https://www.graylog.org/.

[5] *Oficiální stránky Loggly* [online]. [cit. 2021-28-12]. Dostupné z: https://www.loggly.com/.

[6] *Oficiální stránky nástroje Docker* [online]. [cit. 2021-27-12]. Dostupné z: https://www.docker.com/.

[7] *Oficiální stránky nástroje ./jq* [online]. [cit. 2021-28-12]. Dostupné z: https://stedolan.github.io/jq/.

[8] *Oficiální stránky skriptovacího jazyku Ruby* [online]. [cit. 2022-4-3]. Dostupné z: https://www.ruby-lang.org/en/.

[9] *Oficiální stránky Splunk* [online]. [cit. 2021-22-12]. Dostupné z: https://www.splunk.com/.

[10] *Oficiální stránky Sumo Logic* [online]. [cit. 2021-24-12]. Dostupné z: https://www.sumologic.com/.

[11] *Prefekt* [online]. [cit. 2022-20-3]. Dostupné z: https://pajda.fit.vutbr.cz/tacr-unis/prefekt.

[12] *Repozitář projektu Rison* [online]. [cit. 2022-16-4]. Dostupné z: https://github.com/Nanonid/rison.

[13] *Vega – Expressions* [online]. [cit. 2022-18-3]. Dostupné z: https://vega.github.io/vega/docs/expressions/.

[14] *Vega – Transforms* [online]. [cit. 2022-2-2]. Dostupné z: https://vega.github.io/vega/docs/transforms/.

[15] *What is Performance Testing?* [online]. Create IT [cit. 2021-12-15]. Dostupné z: https://www.microfocus.com/en-us/what-is/performance-testing?utm_source=everyonesocial&utm_medium=social&utm_campaign=00164964.

[16] Astrakhan, Y. *Rison online decoder* [online]. [cit. 2022-16-4]. Dostupné z: https://observablehq.com/@nyurik/rison-online-decoder-encoder.

[17] Cotten, A. a Lapenna, A. *GIT repozitář projektudocekr-elk* [online]. [cit. 2022-19-3]. Dostupné z: https://github.com/deviantony/docker-elk#how-to-disable-paid-features.

[18] Farrell Vinay, P. *Manage Software Testing.* 1. vyd. Auerbach Publications, 2008. ISBN 9780849393839.

[19] Gheorghiu, G. *Performance vs. load vs. stress testing* [online]. Systemonline, únor 2005 [cit. 2021-12-18]. Dostupné z: http://agiletesting.blogspot.com/2005/02/performance-vs-load-vs-stress-testing.html.

[20] Hons, L., Patočka, M., Adam, R., Schneebauer, K., Reshetnikov, I. et al. *Co je to MES systém* [online]. MES Center, srpen 2012 [cit. 2021-12-03]. Dostupné z: http://www.mescenter.org/cz/clanky/5-co-je-to-mes-system.

[21] Hons, L., Patočka, M., Adam, R., Schneebauer, K., Reshetnikov, I. et al. *ERP - Výrobní informační systém* [online]. Mes Center, srpen 2012 [cit. 2021-12-03]. Dostupné z: http://www.mescenter.org/cz/slovnik/16-erp-vyrobni-informacni-system-enterprise-resource-planning.

[22] Jiří, S. a Tomáš, S. *Seriál: Výkonnostní testy podnikových aplikací* [online]. Systemonline, březen 2009 [cit. 2021-12-15]. Dostupné z: https://www.systemonline.cz/sprava-it/vykonnostni-testy-podnikovych-aplikaci-2-dil.htm?mobilelayout=false.

[23] Kiesel, M. *Manažerský informační systém* [online]. WikiKnihovna, únor 2012 [cit. 2021-12-03]. Dostupné z: https://wiki.knihovna.cz/index.php?title=Mana%C5%BEersk%C3%BD_informa%C4%8Dn%C3%AD_syst%C3%A9m.

[24] Oháňka, M. *Sběr výkonnostních parametrů systému MES PHARIS.* Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[25] Ondruška, M. *Slovníček pro začínající testery* [online]. Create IT, červen 2019 [cit. 2021-12-15]. Dostupné z: https://www.create-it.cz/Blog/Stranky/typy_testu.aspx.

[26] Saada, M. *Logstash-test-runner* [online]. [cit. 2022-8-4]. Dostupné z: https://github.com/agolo/logstash-test-runner.

[27] Vodrážka, M. *Diplomová práce na téma: Přístupy definování požadavků pro výkonnostní testování softwaru.* Vysoká škola ekonomická v Praze.

Příloha A

Odevzdané soubory

V této příloze je uvedena struktura odevzdaného řešení na přiloženém DVD.

- `/`
  - `DP_xondra51` ..... text diplomové práce
  - `tex_src` ..... složka se zdrojovým kódem TEXu diplomové práce
  - `README.txt` ..... instrukce ke zprovoznění řešení
  - `src` ..... zdrojové kódy výsledného řešení
    - `elasticsearch/` ..... potřebnné soubory pro ElasticSearch
    - `extensions/` ..... rozšíření třetí strany pro nástroj ELK
    - `jenkinsScripts/` ..... pomocné skripty používané na serveru Jenkins
    - `kibana/` ..... potřebnné soubory pro Kibana
    - `logstash/` ..... potřebnné soubory pro Logstash
    - `logstashTests/` ..... obsahuje testy Logstash konfigurací
    - `prepareEnviromentScripts/` ..... obsahuje importační skripty
    - `dockerignore`
    - `.env` ..... zde je definována verze ELK
    - `docker-compose.yml`
    - `docker-stack.yml`
    - `LICENSE` ..... soubor s MIT licencí

Příloha B

Struktura získaných dat z automatizačních serverů

V této příloze jsou tabulky vytvořené Martinem Oháňkou. Tabulky znázorňují jaký formát mají data, jež jsou získávána z automatizačních serverů Jenkins a DevOps.

Název	**Popis**	**Datový typ**
pipelineId	ID úlohy (pipeline)	long
buildId	ID běhu úlohy	long
server	Identifikace serveru	string
stages	Informace o jednotlivých částech běhu	List<Stage> viz B.2
parameters	Parametry běhu	Dictionary <string,object>
isResultValid	Příznak, zda během zpracování došlo k chybě. Další informace o chybě jsou uvedeny v záznamu o události během běhu aplikace.	bool
status	Stav běhu podle výčtu	string
queueTime	Čas zaplánování běhu	DateTime
startTime	Čas začátku běhu	DateTime
finishTime	Čas konce běhu	DateTime
durationMillis	Doba běhu samotné úlohy v milisekundách. Rozdíl mezi finishTime a startTime	long
startTimeMillis	Čas začátku běhu ve formátu Unix	long
finishTimeMillis	Čas konce běhu ve formátu Unix	long
queueDuration-Millis	Doba čekání na zahájení běhu v milisekundách. Rozdíl mezi startTime a queueTime	long
pauseDuration-Millis	Doba pozastavení běhu po jeho zahájení v milisekundách	long
testsResult	Výsledek testů, pokud je běh obasahoval	TestsResult viz B.3

Tabulka B.1: Struktura získaných parametrů ze serveru Jenkins

Název	Popis	Datový typ
name	Název části úlohy	string
startTime	Čas začátku části	DateTime
endTime	Čas konce části	DateTime
durationMillis	Doba běhu samotné části v milisekundách. Rozdíl mezi endTime a startTime	long
status	Stav části podle výčtu	string
startTimeMillis	Čas začátku běhu ve formátu Unix	long
pauseDuration-Millis	Doba pozastavení části po jejím zahájení v milisekundách	long

Tabulka B.2: Struktura získaných parametrů v objektu typu Stage (Jenkins)

Název	Popis	Datový typ
testsRunResults	Kolekce s výsledky jednotlivých testů	List<TestRunResult> viz B.4
startTime	Čas začátku části	DateTime
endTime	Čas konce části	DateTime
durationMillis	Doba běhu samotné části v milisekundách. Rozdíl mezi endTime a startTime	long
total	Celkový počet všech nalezených testů	long
executed	Počet spuštěných testů	long
passed	Počet úspěšných testů	long
failed	Počet neprošlých testů	long
error	Počet testů s chybou	long
timeout	Počet testů ukončených časovým limitem	long
aborted	Počet zrušených testů	long
inconclusive	Počet neprůkazných testů	long
passedButRun-Aborted	Počet testů, které prošly, ale byly přerušeny	long
notRunnable	Počet nespustitelných testů	long
notExecuted	Počet neprovedených testů	long
inProgress	Počet probíhajících testů	long
pending	Počet čekajících testů na spuštění	long

Tabulka B.3: Struktura získaných parametrů v objektu typu TestsResult (Jenkins)

Název	Popis	Datový typ
testName	Název testu	string
testClass	Název třídy, ve které je test implementován	string
result	Výsledek testu podle výčtu	string
durationMillis	Doba trvání testu v milisekundách	double
startTime	Začátek testu	DateTime
stopTime	Konec testu	DateTime

Tabulka B.4: Struktura získaných parametrů v objektu typu TestRunResult (Jenkins)

Název	Popis	Datový typ
pipelineId	ID úlohy (pipeline)	long
buildId	ID běhu úlohy	long
server	Identifikace serveru	string
stages	Informace o jednotlivých částech běhu	List<Stage> viz B.7
packages-ContentInfo	Obsahuje informace o výsledných balíčcích systému. Jako klíč je použit název balíčku	Dictionary <string, Package-Content> viz B.6
pipelineName	Název úlohy (pipeline)	string
pipelineRevision	ID verze postupu úlohy	long
queueDuration-Millis	Doba čekání na zahájení běhu v milisekundách. Rozdíl mezi startTime a queueTime	long
durationMillis	Doba běhu samotné úlohy v milisekundách. Rozdíl mezi finishTime a startTime	long
parameters	Parametry běhu	Dictionary <string, string>
isResultValid	Příznak, zda během zpracování došlo k chybě. Další informace o chybě jsou uvedeny v záznamu o události během běhu aplikace.	bool
buildNumber	Název běhu	string
status	Stav běhu podle výčtu	string
result	Výsledek běhu podle výčtu	string
queueTime	Čas zaplánování běhu	DateTime
startTime	Čas začátku běhu	DateTime
finishTime	Čas konce běhu	DateTime
sourceBranch	Název větve, nad kterou běh běží	string
sourceVersion	ID revize kódu, nad kterým běh běží	string
reason	Informace o důvodu spuštění běhu podle výčtu	string
requestedFor	Informace o účtu pro který je běh spuštěn	Identity
requestedBy	Informace o účtu, který běh spustil	Identity

Tabulka B.5: Struktura získaných parametrů ze serveru DevOps

Název	Popis	Datový typ
dirs	Počet složek v balíčku, počítáno rekurzivně	long
files	Počet souborů v balíčku, počítáno rekurzivně	long
totalItems	Celkový počet prvků v balíčku, počítáno rekurzivně	long
size	Celková velikost balíčku v jednotce bytes	long

Tabulka B.6: Struktura získaných parametrů v objektu typu PackageContent (DevOps)

Název	Popis	Datový typ
id	Identifikátor části úlohy	Guid
parentId	Identifikátor nadřazené části	Guid
name	Název části úlohy	string
startTime	Čas začátku části	DateTime
finishTime	Čas konce části	DateTime
durationMillis	Doba běhu samotné části v milisekundách. Rozdíl mezi finishTime a startTime	long
state	Stav části podle výčtu	string
result	Výsledek části podle výčtu	string
errorCount	Počet chyb v části	long
warningCount	Počet varování v části	long
order	Pořadí části	long
type	Typ části podle výčtu	string
log	Uvnitř objektu je klíč url, kde se nachází záznam běhu příslušné části	Log

Tabulka B.7: Struktura získaných parametrů v objektu typu Stage (DevOps)

Název	Popis	Datový typ
system.pullRequest .pullRequestId	ID žádosti o změnu, pro kterou proběhl běh	string
system.pullRequest .sourceBranch	Název větve, pro kterou je žádost o změnu	string
system.pullRequest .targetBranch	Název větve, do které se provede žádost o změnu	string
system.pullRequest .sourceCommitId	ID revize kódu, nad kterým běh běží	string
system.pullRequest .sourceRepositoryUri	Adresa repozitáře	string
system.pullRequest .pullRequestIteration	Počet změn nahraných do vytvořené žádosti o změnu	string

Tabulka B.8: Struktura získaných parametrů v případě běhu úlohy pro schválení žádosti o změnu (DevOps)

Příloha C

Struktura zaznamenávaných událostí během výkonnostních testů

V následujících tabulkách, jejichž autorem je Martin Oháňka, je uvedena struktura zaznamenávaných událostí během výkonnostních testů.

Název	**Popis**	**Datový typ**
ScenarioName	Název prováděného scénáře. V případě orchestrátoru je hodnota 'RunnerOrchestrator'	string
TestName	Název testu, který je vykonáván	string
RunnerId	Jednoznačná identifikace scénářů v rámci jednoho testu. Pokud je zpráva od orchestrátoru, má hodnotu -1.	long
Communication-Id	Identifikace zprávy nebo komunikace. V rámci závislých záznamů je hodnota stejná.	Guid
DateTime	Datum vytvoření zprávy	DateTime
Message	Samotný obsah zprávy. Přizpůsobitelný obsah podle potřeby.	Dictionary <string,object>
LogLevel	Úroveň logu podle výčtu	ELogLevel viz. C.2
LogType	Typ logu podle výčtu	ELogType viz. C.3

Tabulka C.1: Struktura logu během výkonnostního testování

Název	**Hodnota**	**Význam**
Debug	0	Log úrovně debug
Info	1	Log úrovně info
Warning	2	Log úrovně warning
Error	3	Log úrovně error

Tabulka C.2: Položky výčtu ELogLevel

Název	**Hodnota**	**Význam**
Request	0	Log typu požadavek.
Response	1	Log typu odpověď
Message	2	Log typu zpráva

Tabulka C.3: Položky výčtu ELogType