VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SBĚR VÝKONNOSTNÍCH PARAMETRŮ SYSTÉMU MES PHARIS

PERFORMANCE DATA COLLECTION OF MES PHARIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MARTIN OHÁŇKA**
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: **Oháňka Martin, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Softwarové inženýrství
Název: **Sběr výkonnostních parametrů systému MES PHARIS**
Performance Data Collection of MES PHARIS
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte informační systém MES PHARIS. Nastudujte problematiku monitorování výkonnostních parametrů.
2. Analyzujte požadavky firmy UNIS na sběr výkonnostních metrik. Navrhněte rozšíření stávajícího výrobního informačního systému i rozšíření procesu jeho vývoje umožňující sběr výkonnostních metrik. Zaměřte se na sběr dat při úlohách procesu vývoje systému (např. sestavení systému), úlohách nasazení systému (např. orchestrace, spuštění, import dat) a při automatizovaných úlohách zahrnující koncová zařízení systému.
3. Implementujte sběr výkonnostních metrik jako rozšíření stávajícího systému MES PHARIS.
4. Vytvořte automatické testy pro získání referenčních výkonnostních parametrů.

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.

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 sledováním automatizovaných úloh na integračních serverech a získání dat z těchto úloh. Další rovinou této práce je výkonnostní testování a z něj získat informace o vytížení hardwaru. Díky tomu je možné provádět výkonnostní analýzy implementovaného řešení z různých výkonnostních pohledů. Výsledkem této diplomové práce je programové řešení, které je schopné získat data o úlohách z integračních serverů DevOps a Jenkins. V oblasti výkonnostního testování je vytvořeno řešení pro paralelní provádění úloh. Výstupem této práce je formátovaný výstupu předávaný formátem JSON. Data jsou následně předávána do platformy Elastic, konkrétně Logstash, kde jsou následně vizualizována pomocí Kibana. Ke sběru dat z výkonnostního testování je využita platforma Beat. Řešení bylo aplikováno na výrobní informační systém MES PHARIS společnosti UNIS.

Abstract

This master's thesis deals with monitoring of automated tasks on integration servers and obtaining data from these tasks. Another area of this work is performance testing and to obtain information about hardware utilization from it. Thanks to this, it is possible to perform performance analysis of the implemented solution from different performance perspectives. The result of this master's thesis is a software solution that can obtain data about tasks from DevOps and Jenkins integration servers. In the area of performance testing, there is created a solution for parallel execution of tasks. The output of this work an output passed in JSON format. The data is then transferred to the Elastic platform, specifically Logstash, where it is subsequently visualized using Kibana. The Beat platform is used to collect data from performance testing. The solution was applied to the production information system MES PHARIS of the UNIS company.

Klíčová slova

monitorování, automatizované procesy, průběžná integrace, výkonnostní testování, DevOps, Jenkins, Elasticsearch, Logstash, .NET, C#, výrobní informační systém, MES PHARIS

Keywords

monitorig, automated processes, continuous integration, performance testing, DevOps, Jenkins, Elasticsearch, Logstash, .NET, C#, manufacturing execution systems, MES PHARIS

Citace

OHÁŇKA, Martin. *Sběr výkonnostních parametrů systému MES PHARIS*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Sběr výkonnostních parametrů systému MES PHARIS

Prohlášení

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

. . . .
Martin Oháňka
14. května 2022

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Aleši Smrčkovi, Ph.D. za jeho odborné rady a čas, který mi věnoval během tvorby této diplomové práce. Dále firmě UNIS a.s., která tuto práci podporovala a zaměstnancům firmy Ing. Josefu Konečnému, Mgr. Martinu Kosmákovi, Ing. Josefu Lolovi a Ing. Martinu Švédovi, se kterými byla práce diskutována.

Obsah

1 Úvod **5**

2 Analýza sledování výkonu informačního systému **7**
2.1 Výkonnostní testování . . . 7
2.1.1 Výkonnostní metriky . . . 8
2.1.2 Typy výkonnostního testování . . . 11
2.1.3 Možnosti sběru parametrů . . . 12
2.1.4 Problémy při sběru dat . . . 13
2.2 Analýza monitorovaného systému . . . 15
2.2.1 Informační systémy typu MES . . . 15
2.2.2 Systém MES PHARIS . . . 16
2.2.3 Architektura systému MES PHARIS . . . 18
2.2.4 Postupy vývoje systému . . . 22
2.2.5 Současný stav sledování výkonnosti systému . . . 24
2.3 Analýza požadavků firmy UNIS na sledované parametry . . . 25
2.3.1 Obecné požadavky . . . 25
2.3.2 Požadavky na systémové metriky . . . 26
2.3.3 Požadavky na nové výkonnostní testy . . . 26
2.3.4 Požadavky na sledování automatizovaných procesů . . . 27
2.3.5 Zpracování získaných metrik . . . 29

3 Návrh realizace sběru výkonnostních metrik **30**
3.1 Sledování automatizovaných procesů vývoje . . . 30
3.1.1 Server Jenkins . . . 31
3.1.2 Server Azure DevOps . . . 33
3.2 Automatizované výkonnostní testování . . . 34
3.2.1 Spouštěč scénáře . . . 37
3.2.2 Formát záznamu o události . . . 37
3.3 Sledované metriky . . . 38
3.3.1 Metriky ze serveru Jenkins . . . 39
3.3.2 Metriky ze serveru DevOps . . . 42
3.3.3 Metriky z výkonnostního testování . . . 44

4 Implementační detaily sběru výkonnostních metrik **45**
4.1 Měření automatizovaných úloh v rámci vývoje . . . 45
4.1.1 Společná část aplikací . . . 45
4.1.2 Aplikace pro sledování serveru Jenkins . . . 52
4.1.3 Aplikace pro sledování serveru DevOps . . . 55

4.2 Implementace výkonnostních testů 60
4.2.1 Rozhraní scénáře 60
4.2.2 Zapisování záznamů o události v průběhu scénáře 61
4.2.3 Spouštěč scénáře 61
4.2.4 Orchestrátor 65
4.2.5 Nastavení aplikací Beat 69
4.3 Výkonnostní testování systému MES PHARIS 70
4.3.1 Nástavba koncové aplikace systému MES PHARIS 72
4.3.2 Generování testovacích dat 72
4.3.3 Automatizované testy pro měření výkonnosti systému 73
4.4 Výkonnostní testování bez systému MES PHARIS 73
4.4.1 Řešení pro využívání zdrojů stroje 73
4.4.2 Vytvořené scénáře 76
4.4.3 Sestavení a spuštění 77
4.4.4 Ověření řešení 78

5 Závěr **79**
5.1 Odhalované problémy 79
5.2 Budoucí rozšíření 80

Literatura **81**

A Obsah přiloženého paměťového média **83**

B Ukázka implementace zatížení CPU **84**

C Ukázka výsledku ze serveru Jenkins **86**

D Ukázka výsledku ze serveru Devops **88**

Seznam obrázků

2.1 Umístění MES systému ve vertikální struktuře informačních systémů (převzato z [9]). . . . 15
2.2 Architektura systému MES PHARIS 18
2.3 Přehled technologických postupů ve webové aplikaci 19
2.4 Výrobní obrazovka terminálové aplikace 20
2.5 Ukázka přístroje dodávaného zákazníkům 21
2.6 Konfigurační stránka mobilní aplikace na systému Android 21
2.7 Proces vývoje včetně návaznosti automatizovaných procesů 23

3.1 Schéma fungování sledování automatizovaných procesů 30
3.2 Návrh orchestrace 35

4.1 Diagram aktivit běhu aplikací sledující servery 51
4.2 Diagram synchronizace procesů spouštěče scénáře 63
4.3 Diagram aktivit spouštěče scénáře 64
4.4 Diagram aktivit orchestrátoru 68
4.5 Jednotlivé části výkonnostního testování se systémem MES PHARIS 71
4.6 Jednotlivé části výkonnostního testování bez systému MES PHARIS 74

Seznam tabulek

2.1 Obecné požadavky 25
2.2 Požadavky na nové výkonnostní testy 27
2.3 Požadavky na sledování automatizovaných procesů 28

3.1 Struktura záznamu o události během výkonnostního testování 37
3.2 Položky výčtu ELogLevel 38
3.3 Položky výčtu ELogType 38
3.4 Struktura získaných parametrů ze serveru Jenkins 39
3.5 Struktura získaných parametrů v objektu typu Stage (Jenkins) 40
3.6 Struktura získaných parametrů v objektu typu TestsResult (Jenkins) 40
3.7 Struktura získaných parametrů v objektu typu TestRunResult (Jenkins) . . 41
3.8 Struktura získaných parametrů ze serveru DevOps 42
3.9 Struktura získaných parametrů v objektu typu PackageContent (DevOps) . 43
3.10 Struktura získaných parametrů v objektu typu Stage (DevOps) 43
3.11 Struktura získaných parametrů v případě běhu úlohy pro schválení žádosti o změnu (DevOps) 43
3.12 Struktura získaných parametrů v objektu typu Identity (DevOps) 44
3.13 Struktura získaných parametrů v objektu typu Log (DevOps) 44

4.1 Popis argumentů akceptovaných aplikací 49
4.2 Možnosti konfigurace aplikace pro server Jenkins v souboru `appsettings.json` 52
4.3 Přehled využívaných koncových bodů API (Jenkins) 54
4.4 Možnosti konfigurace aplikace pro server DevOps v souboru `appsettings.json` 55
4.5 Přehled využívaných koncových bodů API (DevOps) 59
4.6 Argumenty programu pro spuštění scénáře 65
4.7 Formát klíče Message v případě záznamu o události z orchestrátoru 66

Kapitola 1

Úvod

V dnešní době neustále roste počet informačních systémů, které různé společnosti nasazují pro zefektivnění svých vnitřních procesů. Pokud některý z těchto systémů má výpadek nebo špatnou dobu odezvy, klesá produktivita a spokojenost s daným systémem. To má za následek zvýšení nákladů pro společnost. Pokud má být takový systém úspěšný na trhu a zdatně konkurovat alternativním řešením, musí být spolehlivý. Pro zajištění tohoto požadavku musí být systém řádně testován, zda splňuje všechny funkcionality a také zda dokáže pracovat pod očekávanými hodnotami zatížení (počet uživatelů nebo požadavků v určitém čase). Na tuto oblast není někdy kladen dostatečný důraz i přesto, že má zásadní dopad na úspěšnost produktu, protože toto testování může být pro některé typy systémů velmi náročné. Proto je nutné toto testování co nejvíce zautomatizovat a výsledky přehledně reportovat.

Tato diplomová práce se zaměřuje na výkonnostní testování výrobního informačního systému MES PHARIS, který je vyvíjen brněnskou společností UNIS a.s. Práce bude mít za úkol se seznámit se systémem MES PHARIS, požadavky společnosti na sledování výkonnosti a vytvořit implementaci testování, která pomůže s ověřením funkcionality systému a tím zvýšit jeho hodnotu a konkurenceschopnost.

Souběžně s touto prací vznikala i diplomová práce Bc. Aleše Ondráčka [12], která se zaměřuje na analýzu a zpracování získaných dat o výkonnosti systému MES PHARIS. Obě práce spolu úzce souvisí a je na ni odkazováno v rámci dalšího textu. Právě z tohoto důvodu byly některé části konzultovány společně pro zajištění kompatibility obou řešení.

Spojením obou prací získá společnost UNIS možnost, jak sledovat výkonnost systému MES PHARIS a následně zjištěné informace budou zobrazeny na vizualizační obrazovce (anglicky dashboard). Zde budou vývojáři přehledně informováni o aktuálním stavu výkonnosti, včetně historického srovnání. To umožní vývojovému týmu pružněji reagovat na zjištěné problémy a včas zajistit nápravu.

Struktura textu

Kapitola 2 obsahuje úvod do problematiky výkonnostního testování, jeho přístupů a popis metrik, na které se testování zaměřuje. Dále jsou v kapitole popsány systémy typu MES a sledovaný systém MES PHARIS vyvíjený společností UNIS. Následuje popis požadavků firmy UNIS na výkonnostní testování, které má být náplní této práce. Kapitola 3 popisuje možné přístupy pro řešení požadavků firmy UNIS v různých oblastech vývoje systému MES PHARIS. V kapitole 4 jsou popsány implementační detaily řešení pro měření automatizovaných úloh v rámci vývoje, výkonnostního testování a provedené úpravy v rámci systému a jeho vývoje. Kapitola 5 popisuje možné budoucí rozšíření.

Kapitola 2

Analýza sledování výkonu informačního systému

V následující kapitole je popsána problematika výkonnostního testování, seznam nejčastěji sledovaných metrik, existujících řešení a problémy, které testování obnáší. Dále jsou obecně popsány informační systémy typu MES a sledovaný systém MES PHARIS. Poslední částí kapitoly je rozbor požadavků firmy UNIS, které mají být v této práci zpracovány.

2.1 Výkonnostní testování

Performance testing [10, 4, 16, 8, 5, 2] se zaměřuje na testování nefunkcionálních vlastností systému. Hlavním důvodem, proč se toto testování provádí, je porovnávání různých řešení implementace systému (např. nová verze, nová optimalizace, nové jádro (anglicky engine) atd.). Dalším důvodem je také ověření chování systému při reálném nasazení. V tomto případě je cílem vytvořit takové prostředí pro testování, aby co nejvíce simulovalo prostředí, ve kterém je systém používán.

Většina problémů s výkonem je v důsledku samotné rychlosti aplikace, ať už se jedná o dobu odezvy nebo dobu načítání. Rychlost je jednou ze základních vlastností aplikací, které jsou poměrně snadno viditelné koncovým uživatelem. Pokud aplikace běží příliš pomalu, vede to až ke ztrátě důvěry v dané řešení a přechod zákazníků ke konkurenčnímu řešení.

Výkonnostní testování umí identifikovat úzká místa (anglicky bottlenecks) v daném systému. Jedná se o překážky, které mají za následek celkové snížení výkonu systému. Tyto problémy nejčastěji vznikají chybou v kódu nebo architektuře. Tyto překážky není možné neustále řešit pouze změnou ve zdrojovém kódu. Jednou se dojde do stavu, kdy už další optimalizace nebude možná a jediným možným řešením bude změna nebo přidání hardwarových zdrojů. Nejčastějšími úzkými místy je využití procesoru, paměti, sítě, disku nebo samotného operačního systému.

Metodika, podle které se postupuje během výkonnostního testování, se může lišit. Obecně je ale cíl testování vždy stejný, a to prokázání zda software splňuje předem definovaná výkonnostní kritéria. Dále může být cílem porovnání výkonnosti dvou různých řešení pro stejnou problematiku (např. vytvoření objednávky ve dvou různých systémech, kolik souběžných relací aplikace zvládne atd.) nebo pomoc s identifikováním míst, které celý systém brzdí.

Výkonnostní testování má nejčastěji následující části:

Identifikace testovacího prostředí Aby byly výsledky zátěžového testování co nejbližší reálnému používání, je důležité provést analýzu, jak se daný produkt reálně využívá. Jde především o identifikaci hardwaru, na kterém daný software běží. Zjištění doprovodného softwaru, který běží souběžně s testovaným produktem a může mít vliv na chod systému a ovlivnění výsledků testování (např. antivirový program). Dalším klíčovým prvkem je znalost topologie sítě, pokud se jedná o webovou aplikaci a dochází k výměně dat mezi serverem a klientskou aplikací (např. volání API). Je také nutné určit databázový server, který se využívá pro hostování databáze.

Identifikace sledovaných výkonnostních metrik V této fázi se určují metriky, které se budou sledovat. Dále se stanoví kritéria úspěšnosti pro vybrané metriky, kdy je výsledek testu považován za úspěšný.

Plánování a návrh výkonnostních testů Pro plánování je nutné zjistit, jak koncoví uživatelé používají aplikaci. Díky tomu je možné stanovit klíčové scénáře pro testování. Je nutné brát v úvahu různé koncové uživatele, určit testovací data a metriky, které bude daný test sledovat.

Konfigurace testovacího prostředí Příprava jednotlivých prvků testovacího prostředí a nástrojů pro provedení testování. Tato fáze je velmi důležitá, jelikož pro reprodukovatelnost testování a schopnosti porovnat výsledky jednotlivých měření je nutné zajistit co nejvíce podobná prostředí, kde testy běží.

Implementace navržených testů Zde probíhá implementace testů podle definovaných testovacích scénářů. Dále testování samotné implementace testů do doby, než stav implementace odpovídá všem požadavkům a jejich běh přináší data dle očekávání.

Spuštění testů Provedení jednotlivých testovacích scénářů, monitorování výkonnostních metrik a vytvoření reportu.

Analýza výsledků, úpravy a opětovné spuštění testů Provádí se vyhodnocení naměřených dat. Na základně zjištěných údajů se určí úzké místo v kódu, které způsobuje problémy s výkonem. Následně se provede úprava a celý proces testování se zopakuje.

2.1.1 Výkonnostní metriky

Existuje řada parametrů neboli klíčových ukazatelů výkonu (KPI z anglického Key Performance Indicator), které mohou organizaci pomoci vyhodnotit aktuální výkon[1]. Mezi nejčastější parametry, které jsou sledovány při testování výkonu softwarových systémů, patří:

Využití procesoru

Definujeme jako množství času, kdy procesor vykonává daný proces. Jedná se o důležitou metriku, protože v případě, kdy je procesor využíván na 100 % již není schopen vykonávat další činnost a tím se zmenší celková propustnost daného systému. Tuto metriku je možné využít jako měřítko toho, jak jakákoliv změna, která byla provedena, ovlivňuje celkový výkon systému. Dále může určovat, jak je kód a celkový program efektivní.

[1] KPI se používá i pro hodnocení projektů, organizací nebo lidí.

Využití operační paměti

Ukazuje množství fyzické paměti, kterou daný proces využívá. Tato metrika může sloužit k identifikaci potencionálních úniků paměti. Pokud je využití paměti vyšší, než je ve fyzické paměti k dispozici, dochází k využití virtuální paměti, která je ukládána na disk. Čtení z disku je o několik řádů pomalejší než čtení z paměti. Pokud jsou požadována data z virtuální paměti, je nutné je načíst do fyzické paměti. Vzhledem k rychlosti disku to může způsobovat pomalou odezvu celého systému. V tomto případě je nutné navýšit kapacitu fyzické paměti, případně identifikovat úniky paměti.

Využití pevného disku

Využití můžeme brát ve dvou rovinách, jak z pohledu využité kapacity, tak vytížení disku IO operacemi.

Využití kapacity disku může být sledováno vzhledem k předchozí metrice a ukládání virtuální paměti, nebo v případě, kdy aplikace využívá dočasné soubory pro svůj běh, případně vytváří soubory se záznamy o událostech. Všechny zmíněné možnosti mohou zásadně zabírat místo na disku a to může mít za následek snížení jeho výkonnosti.

Úzká hrdla disku jsou v nejvíce případech spojena s časem. Nejčastěji se sleduje čas disku, což je doba, po kterou byl disk zaneprázdněn požadavky na čtení nebo zápis. Dále máme k dispozici další čítače, které nám pomáhají k určení úzkého místa. Jedná se například o čítač průměrné délky diskové fronty atd.

Garbage collection

Jde o proces, který spravuje paměť objektů na haldě. Pokud na objekt již neexistuje reference, je tento objekt uvolněn z paměti a paměť může být v budoucnu použita pro alokaci dalších objektů. Jinak objekt zůstává v paměti. Proces je převážně sledován z hlediska účinnosti.

Garbage collection [6] se skládá ze tří fází: označení, smazání a komprimování. V první fázi je procházena halda a dochází k označení všech objektů jako živých (existuje na ně reference) nebo nereferencovaných, zbytek je označen jako dostupná paměť. Následuje smazání nereferencovaných objektů a komprimace živých objektů.

Garbage collection pomáhá zlepšit výkon aplikace, ale neměl by být příliš častý nebo pozdní. Pokud je moc častý, může docházet ke snížení výkonu systému a způsobuje nárůst vytížení CPU. Garbage collection je výpočetně náročnou činností a může tedy neúměrně zabírat čas na CPU.

Dalším problémem je únik paměti (anglicky memory leake). V ideálním případě postupně dochází ke zvyšování využití paměti až do okamžiku, kdy se provede Garbage collection a využití paměti klesne. Využití haldy je většinou na stejné úrovni a uvolnění paměti by víceméně mělo mít stejný objem. Při úniku paměti každý další cyklus uvolní menší objem paměti. Využití haldy se bude zvyšovat, až bude halda zaplněná a dojde k vyvolání výjimky `OutOfMemory`. To značí stav, kdy v počítači došla volná operační paměť. Důvodem je, že proces uvolnění označí ke smazání pouze nereferencované objekty. Ty, na které existují refe-

rence a již se nevyužívají, nebudou odstraněny. V tomto případě je nutné upravit zdrojový kód.

V některých prostředích (např. Java [11]) existuje více typů Garbage collection. Při sériovém Garbage collection se využívá akce s názvem *Stop the World*. Tato akce má za následek pozastavení celé aplikace, než je proces ukončen. Tento typ je nevhodný u aplikací, kde je požadována nízká latence. Tento problém se ještě více prohlubuje, pokud je spojen s úniky paměti a tím častějším prováděním uvolnění paměti. Uživatel tuto dobu vnímá jako zamrznutí aplikace a zpožděnou odpověď.

Pro správné fungování Garbage collection je nutné vědět, jak tento proces funguje na daném prostředí, operačním sytému a využitém frameworku, dále je nutné optimalizovat současný kód, aby byl tento proces více efektivní.

Zámky v databázi

K zamykání [3] dochází z důvodu ochrany sdílených zdrojů. Nejčastěji se jedná o tabulky, datové řádky nebo datové bloky. Existuje více typů zámků, které jsou v databázích využívány (např. sdílené zámky, exkluzivní zámky nebo transakční zámky). V případě zamykání transakcí je hlavním cílem systému řízení báze dat (SŘBD) zajištění ACID vlastností. ACID je zkratka, která znamená atomicitu (atomicity), konzistenci (consistency), izolovanost (isolation) a trvalost (durability). Existuje řada problémů, které jsou způsobeny zamykáním. Obecně se jedná o čtyři kategorie (spor o zámek, dlouhodobé blokování, uváznutí databáze, uváznutí celého systému).

Spor o zámek nastává v případě, že mnoho databázových relací vyžaduje častý přístup ke stejnému zámku. Daný zámek je držen po krátkou dobu a poté uvolněn. To vede při větším počtu spojení k vytváření úzkého místa. Celkově má tento problém malý dopad, ale omezuje možnost škálovatelnosti dané aplikace. Soupeření o zámek může vést až k nadměrnému vytížení procesoru na databázovém serveru. K identifikaci tohoto problému můžeme využít informace o zámcích, které poskytuje samotný SŘBD.

Dlouhodobé blokování je velmi podobné se sporem o zámek. Opět je zde zámek, ke kterému přistupuje větší množství databázových relací. Rozdíl je v tom, že jedna relace neuvolní zámek okamžitě, ale je držen delší dobu a po tuto dobu jsou ostatní relace zablokovány. Dlouhodobé blokování bývá větší problém než spor o zámek, protože dokáže odstavit funkčnost celé oblasti nebo dokonce i celého systému. Důsledkem tohoto problému může být až vypršení času na zpracování (anglicky timeout) požadavku uživatele a nutnost celý proces opakovat, což vede k celkové nespokojenosti se systémem.

V databázi je *uváznutí* nechtěnou situací, ve které dvě nebo více transakcí nekonečně dlouho čekají, až se jedna pro druhou vzdá zámku. Uváznutí (anglicky deadlock) je považováno za jednu z nejobávanějších komplikací v SŘBD, protože přivádí celý systém k zastavení. Výhodou je, že SŘBD dokáže tento problém rozpoznat a dokonce i vyřešit. Nejčastěji SŘBD vybere jednu transakci, kterou je nejsnazší vrátit zpět. Pokud ovšem aplikace není navržena tak, aby dokázala na uváznutí reagovat a opakovat požadavek, má to negativní dopad na uživatele, protože jeho požadavek není dokončen a musí jej opakovat. Ve většině případů je tento problém způsoben špatnou aplikační logikou, kdy není k datům přistupováno v konzistentním pořadí. Uváznutí je také ovlivněno načasováním a daty, nad kterými je transakce vykonávána.

Doba odezvy

Dobou odezvy aplikace je myšlen čas od okamžiku, kdy uživatel zadal data do aplikace a zadal požadavek po okamžik, kdy aplikace na tento požadavek zašle odpověď. Opět by doba měla být co nejkratší, jinak ochota uživatele pracovat s aplikací klesá a celková výkonnost pracovníka také. V případě, kdy jsou požadovány specifické výpočty, simulace nebo operace, při kterých je očekávána delší doba zpracování a uživatel je na ní připraven, nejedná se o problém.

Doba načtení aplikace

Dobu načítání aplikace definujeme jako dobu, po kterou se aplikace spouští. Tento čas by měl být omezen na nezbytné minimum, aby uživatel dlouho nečekal, než začne samotnou aplikaci využívat. Existují ovšem i výjimky a u některých aplikací nebo způsobu použití je akceptovatelná i delší prodleva samotného spuštění aplikace.

Využití sítě

Tato metrika je velmi důležitá u aplikací typu klient-server, protože po síti dochází k výměně veškerých dat. Pokud je software provozován na lokální síti, můžeme díky testování odhalit problémy ve vnitřní infrastruktuře sítě. Řešením zjištěných nedostatků je poté modernizace prvků sítě nebo celková změna její architektury. Pokud je server umístěn mimo lokální síť, je nutné prověřit vstupní body lokální sítě, například fungování firewallu a dalších bezpečnostních prvků, zda neblokují legitimní provoz. V případě, kdy je na vině nedostatečná rychlost spojení, je nutné změnit technologii připojení, případně poskytovatele nebo změnit tarif připojení tak, aby odpovídal předpokládanému provozu.

2.1.2 Typy výkonnostního testování

Existuje několik typů testování, které se liší podle přístupu a cílů, kterým má být testovaný systém podroben.

Load testing

Tento typ testování pomáhá porozumět chování systému při určité očekávané hodnotě zatížení. V procesu testování se simuluje zátěž očekávaným počtem souběžných uživatelů a transakcí po určitou dobu. Získané informace slouží k ověření očekávané doby odezvy a lokalizaci úzkých míst ještě před samotným nasazením softwarové aplikace.

Stress testing

Toto testování se používá za účelem pochopení horních limitů aplikace z hlediska zátěže. Převážně se zaměřuje na kontrolu stability a výkonu softwaru, když je testován za bodem zlomu, tj. za definovanými limity maximálního zatížení. Zátěžové testování také kontroluje efektivní správu chyb a postup obnovy systému v případě selhání z důvodu nadměrného zatížení systému.

Capacity testing

Je podobné Stress testing v tom, že testuje zatížení na základě počtu uživatelů, ale liší se v jejich počtu. Toto testování se zaměřuje na zjištění hranice, při které systém přestává být spolehlivý.

Volume testing

Objemové testování, také známé jako Flood testing, se používá ke kontrole výkonu softwarové aplikace při různých objemech databáze. Testují se různé situace, kdy systém musí zpracovávat obrovské množství dat. Kontroluje se chování systému a identifikují se úzká místa v systému při zpracování objemných dat. Především se zaměřuje na dobu odezvy databáze, správné ukládání dat, ztrátu dat, integritu dat, využití paměti atd.

Soak testing

Vytrvalostní testování, také známé jako Soak testing, testuje schopnost systému fungovat při nepřetržité zátěži po delší dobu. Provádí se za účelem zjištění robustnosti systému. Během testování se monitorují metriky, jako je využití paměti, aby se zjistily jakékoli úniky atd.

Spike testing

Testování špiček ověřuje výkon systému, kdy dojde k extrémním změnám zatížení s náhlým zvýšením nebo snížením zátěže, také známým jako špičky. Testování špiček bere v úvahu počet uživatelů a složitost prováděného úkolu a testuje reakci softwaru na náhlé velké skoky v zátěži generované uživateli. Dále sleduje dobu zotavení mezi dvěma špičkami a zda je systém dokáže zvládnout. Toto testování se nejčastěji provádí před akcí, kdy je očekáváno, že systém bude zatěžován větším objemem požadavků (např. vánoční výprodeje, black friday atd.).

2.1.3 Možnosti sběru parametrů

Pro sběr parametrů můžeme využít několik různých přístupů. Ať už přes různé typy testů, kdy nám každý z přístupů odpovídá na jiné otázky nebo vložení sond do zdrojového kódu a následná analýza záznamů o událostech, napojení na aplikaci za běhu, využití existujících řešení a programů pro testování a další.

Většina existujících řešení [7] se zaměřuje na testování webových aplikací, mobilních aplikací nebo API. Tato řešení pomáhají s generováním zátěže, prováděním scénářů a poskytují přehledy o výkonu. Jednotlivá řešení jsou jak bezplatná, kdy mohou mít omezenou funkcionalitu nebo nenabízejí všechny funkce, tak i placená.

Mezi existující řešení patří například nástroj WebLOAD od RadView Software[2]. Nástroj je dostupný v cloudu jako SAAS nebo On-Prem. Podporuje všechny hlavní webové technologie. Dále dokáže simulovat stovky tisíc souběžných uživatelů, což umožňuje testovat velké

[2]RadView Software – https://www.radview.com/

zatížení a identifikovat úzká místa v aplikaci. Nástroj obsahuje IDE pro vytváření a editaci scénářů, které jsou založeny na technologii JavaScript. Pro záznam scénářů obsahuje nástroj záznamník, který je založen na proxy a zaznamenává aktivitu na HTTP. V něm je podporována analýza výsledků z libovolného prohlížeče nebo mobilního zařízení, která je zobrazena na vizualizační obrazovce.

Dalším řešením je nástroj Apache JMeter[3]. JMeter navíc podporuje integraci se Selenium, což umožňuje spouštět automatizované skripty spolu s testy výkonu nebo zátěže. JMeter obsahuje mnoho modulů s dalšími funkcemi. Další výhodou je podpora parametrických proměnných, souborů cookies pro jednotlivá vlákna nebo konfiguračních proměnných. Nástroj je psán v jazyce Java a má pro tento jazyk plnou podporu. Podporuje protokoly HTTP, HTTPS, SOAP, FTP a další.

LoadNinja[4] je dalším nástrojem, který můžeme použít pro zátěžové testování. Hlavní výhodou je, že umožňuje rychle vytvářet sofistikované zátěžové testy bez skriptování díky podpoře InstaPlay rekordéru. Ten zaznamenává interakci na straně klienta, kterou následně uloží jako testovací scénář. Dalším přínosem je, že je hostován v cloudu a není potřeba mít vyčleněný server a provádět nutnou údržbu instance. Podporuje protokoly HTTP, HTTPS, SAP GUI Web, Google Web Toolkit a další.

2.1.4 Problémy při sběru dat

Při výkonnostním testování je celá řada aspektů, na které musí být brán zřetel před zahájením samotného testování a sledování výkonnosti. Pokud nebudou reflektovány, může to vést až k častým chybám [14] v procesu výkonnostního testování. V následující sekci jsou popsány některé z nich.

Pokud měříme dobu odpovědi informačního systému, tak naměřená doba *zahrnuje více časových úseků*. Celková doba zahrnuje jak dobu generování požadavku, tak odeslání požadavku, jeho zpracování, komunikace s externími systémy, vyčíslení požadavku, zaslání odpovědi zpět a následné vykreslení uživateli. Většinou ale chceme měřit pouze výkonnost daného informačního systému a ne celé infrastruktury. V tomto případě je potřeba brát v úvahu výkon sítě, databázového serveru, webového serveru, případně i webového prohlížeče.

Další oblastí je *vedlejší zátěž*, která probíhá na daném systému nebo stroji, na kterém je systém hostován. Nejčastěji se jedná o dalšího přihlášeného uživatele, probíhající údržbu na daném systému (např. indexace, zálohování, probíhající aktualizace atd.). Pro snížení přepnutí kontextů nebo výpadků stránek paměti je třeba zajistit minimální konfiguraci stroje a tím minimalizovat kolize.

Testování by mělo být *reprodukovatelné*, zde se ale musí zajistit stejná konfigurace jak na úrovni HW (procesor, paměť a cache, disk atd.), tak SW (nainstalovaný operační systém včetně verze jádra, ovladače, knihovny a celkové nastavení stroje).

Vyrovnávací paměti mohou také velice ovlivňovat naměřené výsledky. Slouží k urychlení výpočtu a za tímto účelem využívají spekulace (kam v paměti bude následující operace

[3]Apache JMeter – https://jmeter.apache.org/
[4]LoadNinja – https://loadninja.com/

přistupovat, cache line, u CPU pipelining, která další instrukce se bude vykonávat, prediktor skoku atd.).

Sekvenční nebo souběžné spouštění procesů má také dopad na měření výkonnosti. Zde vstupuje režie spojená s přepínáním kontextu, cache miss nebo výpadek stránek. V důsledku přepnutí kontextu dochází v CPU k vyprázdnění cache jak datové, tak i instrukční. Například u procesoru s jedním jádrem, na kterém se budou vykonávat dva úkoly, dojde v případě paralelního zpracování k častému přepínaní kontextu. Dochází neustále k výměně stejných dat a to vede ke zhoršeným naměřeným výsledkům než v případě, kdy budou úkoly vykonány sekvenčně.

Dalším problém je *virtuální prostředí*. V podstatě není chybné využívat virtuální stroje pro měření výkonnosti, ale musí být zajištěné, že každé testování bude mít stejný efekt. Největším problémem u virtuálních prostředí je, že hostující systém spekulativně načítá data do chache a dále ovlivňuje virtuální prostředí svým zatížením. Řešením tohoto problému je, že do výsledku testování nebudou brány v úvahu výsledky prvních několika testů. To bude bráno jako příprava test fixture (prostředí, kde se provádí testování), kdy je virtuální stroj spuštěn a již na něm byly provedeny nějaké operace (testy, jejichž výsledky jsou zahozeny), které mají za cíl načíst potřebné údaje do paměti hostujícího stroje. Následuje samotné výkonnostní testování.

Narozdíl od běžné praxe je vhodné výkonnostní testy provádět nad různými daty stejné povahy. Výkonnostní testování by se mělo co nejvíce přiblížit reálnému použití, takže například použití vždy stejného HTTP požadavku by mohlo znamenat problém. Aplikace v dnešní době se snaží být co nejrychlejší, některé jsou schopny za tímto cílem spolu s databázemi rozpoznat takový požadavek a začít data ukládat do mezipaměti. Pokud se tak stane, tak aplikace začne pracovat rychleji a tím výsledky vracet rychleji. Toto chování bude mít také dopad na výsledky testování, které se budou jevit jako lepší. Takové výsledky jsou ale neplatné a celé výkonnostní testování ztrácí svůj přínos. Problém s opakujícími se daty se také projeví například při testování registrace nového uživatele. Většina systémů nedovolí opakovanou registraci třeba se stejnou emailovou adresou. To následně způsobí, že test projde pouze jednou. Pro vyřešení tohoto problému je vhodné v takových údajích použít generátory náhodných hodnot. Tento přístup zajistí, že každý požadavek bude unikátní, což bude pro testování přínosné.

Jako další problém bývá nevhodná *simulace virtuálního uživatele*. Často se v testech zapomíná na čas, který uživatel stráví nad přemýšlením na dané stránce, zpracování aktuální stránky, zadání údajů a odeslání požadavku. Takové testy, kdy chodí velké množství požadavků v krátkém čase, se hodí pouze ve specifických případech, například testování chování aplikace při útoku *Denial of Service*. Takové testy mohou přinášet výsledky s velmi pomalou dobou odezvy, jelikož je zatížen požadavky v krátkém čase. Následně při ručním provedení scénáře se výsledky jeví v pořádku. Řešením je přidat do testů vhodný způsob čekání mezi jednotlivé kroky virtuálního uživatele.

2.2 Analýza monitorovaného systému

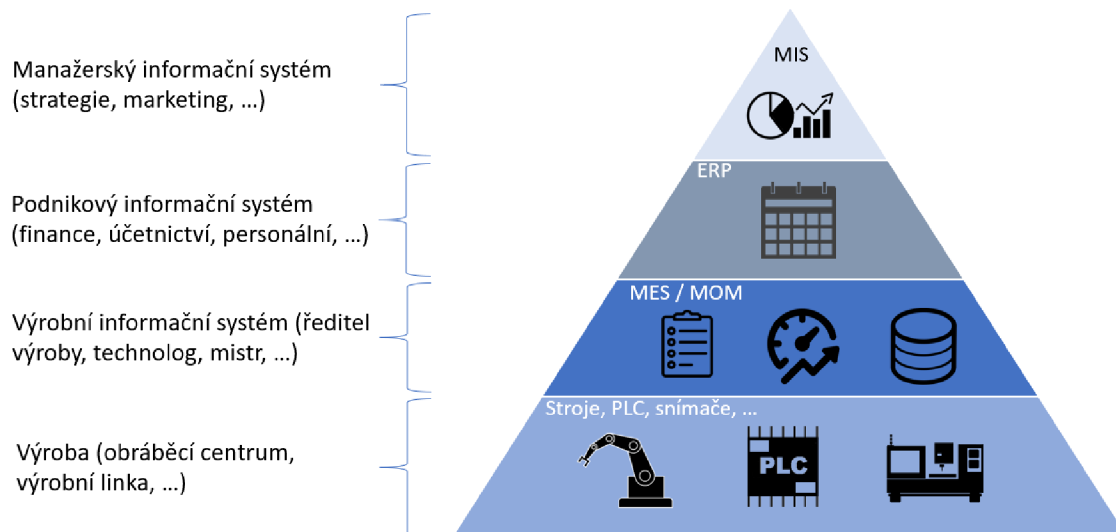
Sledovaným systémem je MES PHARIS brněnské společnosti UNIS.

Společnost UNIS [15] byla založena roku 1990 jako česká společnost bez účasti zahraničního kapitálu. Majitelem a členem představenstva je Ing. Jiří Kovář. Hlavním zaměřením firmy je dodavatelská činnost na klíč v oblasti zpracování ropy a zemního plynu, petrochemie a energetiky. Poslední akvizicí firmy se v roce 2017 stal dodavatel UNIS Power s.r.o., který se zaměřuje na zakázky v oblasti energetického průmyslu.

2.2.1 Informační systémy typu MES

Informační systémy typu MES (Manufacturing Execution Systems) [9] překládáme do češtiny jako Výrobní informační systémy. Hlavní využití těchto systémů je ve výrobních podnicích, kde řídí a monitorují výrobní procesy, informují o případných problémech ve výrobě a tím dodávají podklady řídícím pracovníkům pro včasné přijetí rozhodnutí. Hlavním znakem je, že tyto systémy pracují v reálném čase.

MES systémy nejčastěji spolupracují se systémy typu ERP (Enterprise Resource Planning) na jedné straně a systémy pro řízení výrobních procesů a sběru dat na straně druhé. Základní funkcionalita MES systémů byla vydefinována již v roce 1992 neziskovou organizací MESA International[5] v modelu MESA-11, který popisuje 11 základních funkcionalit.



Obrázek 2.1: Umístění MES systému ve vertikální struktuře informačních systémů (převzato z [9]).

[5]Manufacturing Enterprise Solutions Association (MESA) International – https://mesa.org

Systémy typu MES nejčastěji obstarávají tyto oblasti:

Správa výrobních zdrojů, která sleduje a přiřazuje zdroje a kapacity při výrobě. Jedná se především o materiál, osoby, zařízení, nástroje apod. Dále může kontrolovat specifické požadavky na daný zdroj (např. absolvování školení).

Správa výrobních postupů, která umožňuje evidenci, správu verzí a výměnu dat s okolními systémy. Nejčastěji se jedná o pravidla výroby finálního produktu, kusovník materiálu, výrobní zdroje atd. Díky všem těmto informacím je možné definovat postupy pro tvorbu finálního produktu.

Detailní plánování a rozvrhování výroby, které je zásadní pro dodržení termínů a plnění plánu během výroby. Existuje mnoho způsobů, jak plánování provádět, například plánování založené na jednoduchých algoritmech s využitím priorit zakázek, komplexní plánování založené na genetických algoritmech nebo dopředné a zpětné plánování. Výsledek plánování udává, v jakém pořadí se na výrobních zařízeních budou vykonávat jednotlivé operace. Během plánování je kladen důraz na minimalizaci zbytečného seřizování strojů, prostojů, spotřeby energie atd.

Řízení a sledování výroby, kdy jde o souhrn aktivit, které řídí výrobu přiřazováním práce jednotlivým zařízením a osobám, zajišťování potřebného množství surovin a energie, sledování aktuálního stavu výroby, operativní řešení výpadků atd. Také zajišťují činnosti, které řídí výrobu specifikovanou v plánu výroby. Pokud je řízení výroby zajištěno v řídicím systému, tak výrobní informační systém pouze kontroluje zdroje a informuje okolní systémy o aktuálním stavu výroby (odvody práce, zabezpečení kontrolních kroků výroby atd.). Řízení výroby v MES systémech je velmi důležité, jelikož jsou propojeny s ERP systémy.

Sběr dat, který slouží ke sběru a historizaci dat procesních, výrobních apod. Sběr dat se liší podle typu výroby. Může se jednat pouze o sběr základních informací (např. výrobní cyklus stroje) nebo se může jednat o sběr až tisíců údajů za minutu ve velmi automatizované výrobě.

Sledování výrobků a jejich rodokmen, kde se jedná o souhrn aktivit, které shromažďují a poskytují informace o zdrojích, které byly použity pro výrobu finálního produktu, spotřebu materiálu, výrobu meziproduktu apod. Tyto informace jsou důležité pro společnosti z hlediska legislativních požadavků, auditů nebo při řešení reklamací.

Výkonnostní analýzy, kde jde o klíčové výkonnostní ukazatele (KPI z anglického Key Performance Indicator), které podniky využívají pro stanovení úspěšnosti dané aktivity (výroby). Jednotlivé společnosti sledují rozlišné ukazatele podle zvolené strategie. Nejčastěji se jedná o celkovou efektivitu zařízení (OEE - Overall Equipment Effectiveness).

2.2.2 Systém MES PHARIS

Vývoj monitorovaného systému byl zahájen roku 2002. Původní záměr systému byl pro farmaceutický průmysl, kde byl určen pro řízení šaržovité výroby. Postupem času vznikl systém, který je schopen řešit komplexní řízení výroby. Nyní se systém nejčastěji nasazuje v diskrétních výrobách, jako jsou kovovýroba, lisování plastů nebo gumárenství.

MES PHARIS je modulární výrobní informační systém. Obstarává řízení výroby, její plánování a rozvrhování. Dále dokáže řešit výrobní logistiku a sběr dat ze strojů a technologií. Díky komunikaci s ERP je systém MES PHARIS základem pro výrobu v podniku podle standardů Industry 4.0.

Mezi přínosy systému řadíme: zefektivnění výrobního procesu, on-line přehled o rozpracované výrobě, efektivní správu údržby zařízení, dokladování výroby pro potřeby auditů, podporu bezpapírové výroby, přímé napojení na CNC stroje pro sběr dat, okamžitou signalizaci problému na lince, odhalování „úzkých hrdel“ montážních linek, podporu Just in Time (JIT), podporu Kanban, uživatelskou tvorbu vlastních reportů atd.

Systém MES PHARIS je dále schopen řídit montáže. Podporuje samostatné výrobní příkazy pro montážní i navazující operace. Systém plně podporuje automotive standard IATF 16949. Tento standard se uplatňuje v automobilovém průmyslu, kde zaručuje kvalitu výrobků a snižuje množství zbytečného odpadu.

Mezi zákazníky společnosti patří například firmy Kasko, BOSCH, LukovPLAST, ATEK, THK, KOVOKON, SELLIER & BELLOT nebo Nolato HU.

Systém MES PHARIS obsahuje několik modulů a ty celkově splňují model MESA-11. Níže následuje popis některých modulů systému.

Sběr dat z výroby Systém obsahuje podporu pro mnoho komunikačních protokolů. Ty umožňují získat data z výrobních zařízení, periferií a kontrolních stanic. Díky tomu jsou k dispozici podstatné informace o výrobě pro celý výrobní proces. Data jsou také historizována pro potřeby auditů a prokazování postupů při výrobě. Automatický sběr dat dále významně zpřesňuje vykazování, evidenci práce a hodnocení výkonnosti.

Jsou podporovány následující způsoby komunikace: Euromap 63, Euromap 77, OPC UA, OPC DA, sériová komunikace a přes řídící systémy strojů (Heidenhain, Sinumerik...) a další.

Sledování rozpracované výroby Díky evidenci provedené práce jednotlivých pracovníků dochází ke zprůhlednění rozpracovanosti výrobních zakázek. Dále systém obsahuje přehled a stav výroby s predikcí dokončení, znázornění případného skluzu nebo aktuální míru zmetkovitosti. Tyto údaje jsou důležité pro operativní řízení výroby.

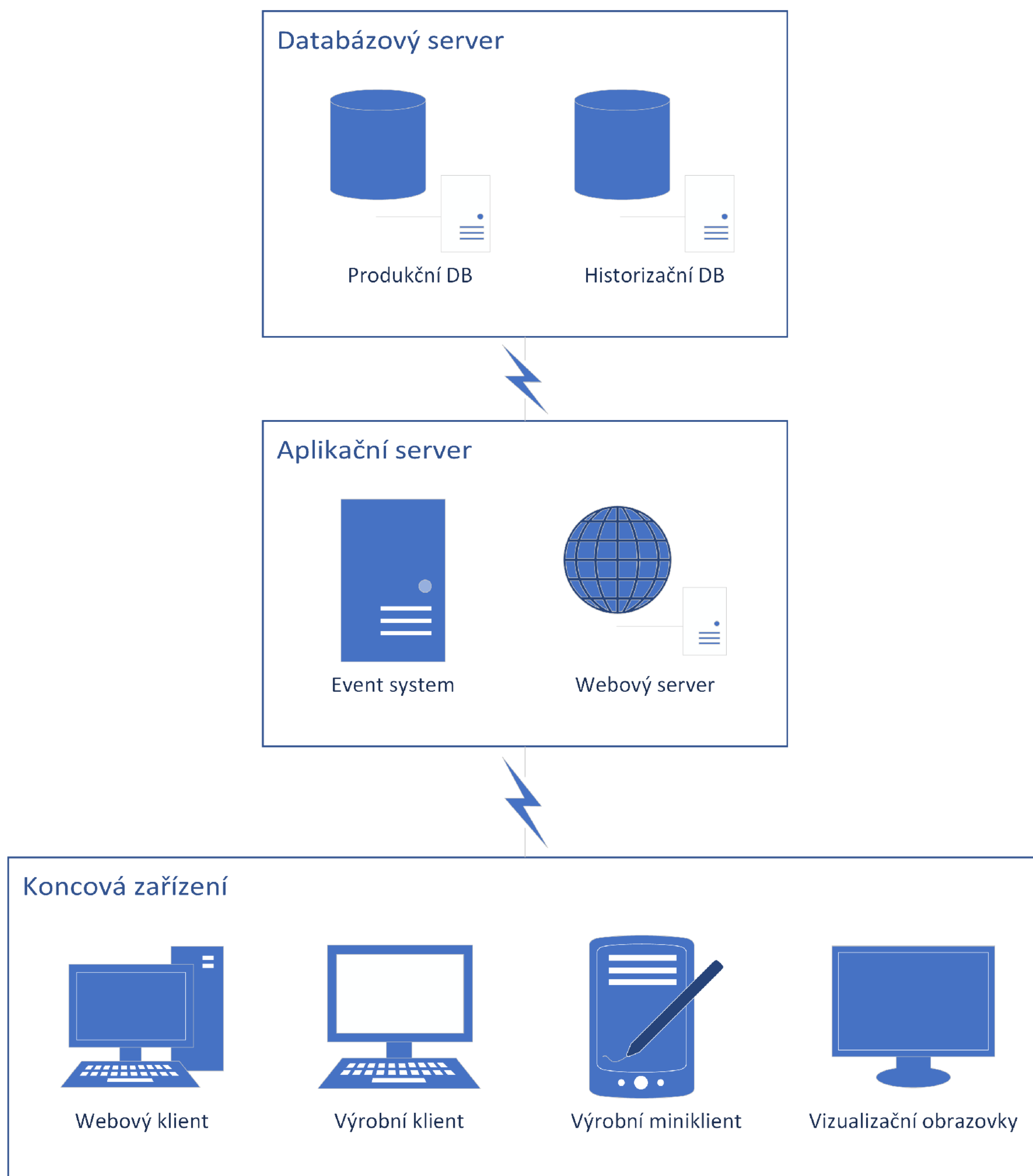
Včasné predikce dokončení výroby dále umožňují zavedení štíhlé výroby a programu SMED (Single Minute Exchange of Dies). Dochází k optimalizaci a snižování režijních časů při přeseřízení stroje na další výrobu, přípravě materiálu atd.

Kapacitní plánování a rozvrhování výroby Modul obsahuje nástroj pro optimalizaci výrobních rozvrhů. Tento rozvrhovač je vyvíjen ve spolupráci s Ing. Martinem Hrubým Ph.D. z Fakulty informačních technologií Vysokého učení technického v Brně. Cílem nástroje je výpočet optimální alokace výrobních operací na jednotlivá pracoviště a stroje s využitím pokročilých genetických algoritmů. Výsledek je zobrazován pomocí dynamického Ganttova diagramu. Díky tomu je zajištěno maximální a efektivní využití prostředků pro danou výrobu.

Dalšími moduly jsou: Řízení výroby a odvody práce, KPI - klíčové výrobní ukazatele, Vizualizace výroby, Řízení údržby, Správa a evidence nástrojů, Centrální správa řídicích programů, Alarmy a eskalační systém, Řízení kvality, Výrobní logistika (Kanban, Just in sequence).

2.2.3 Architektura systému MES PHARIS

MES PHARIS stojí na technologiích společnosti Microsoft a jedná se o systém využívající třívrstvou architekturu. Celý systém se skládá z několika částí, které budou rozepsány dále. Systém MES PHARIS je vyvíjen v jazyku C# na technologii .NET Framework. Architektura systému je znázorněna na obrázku 2.2.



Obrázek 2.2: Architektura systému MES PHARIS

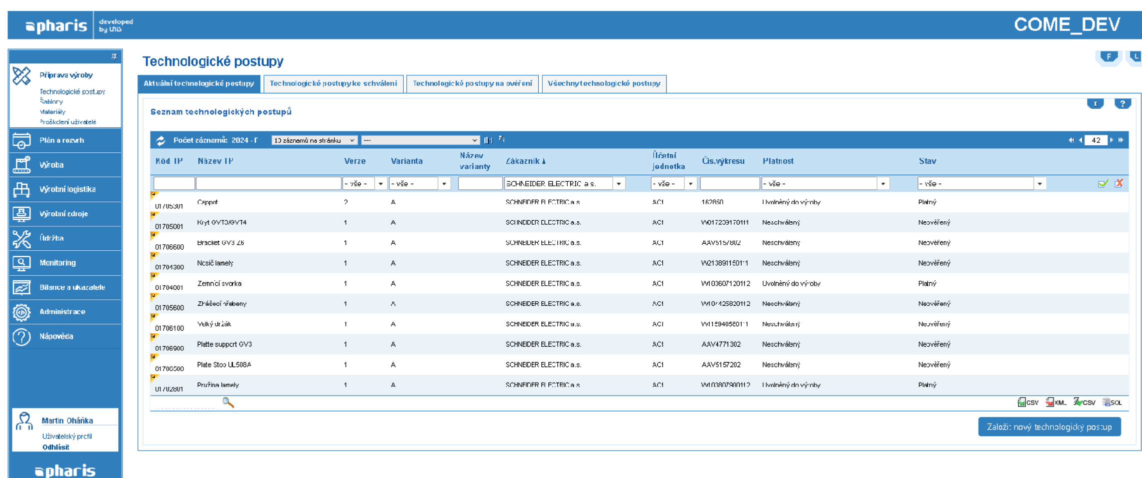
Databáze

Databáze systému je rozdělena na produkční a historizační část. Produkční databáze se využívá pro udržování aktuálního stavu výroby. Především jde o evidenci výroby, zařízení atd. Historizační databáze slouží k ukládání procesních dat z jednotlivých zařízení (CNC stroje, vstřikování atd.) a k jejich archivaci. Pro realizaci databází je využita technologie Microsoft SQL Server. Za účelem tvorby reportů se využívá MS SQL Reporting Services.

Webová aplikace

Webový server je hostován pomocí technologie IIS. Webová aplikace je postavena na technologii ASP.NET. Nejkomplexnějším klientem je internetový prohlížeč a proto je MES PHARIS k dispozici všem uživatelům společnosti. Výhodou tohoto řešení je snadná správa systému bez nutnosti instalace klientské stanice. Aktualizace systému se provádí pouze na serveru. Tuto výhodu ocení hlavně administrátoři, kteří se díky tomu nemusí starat o klientskou aplikaci systému MES PHARIS.

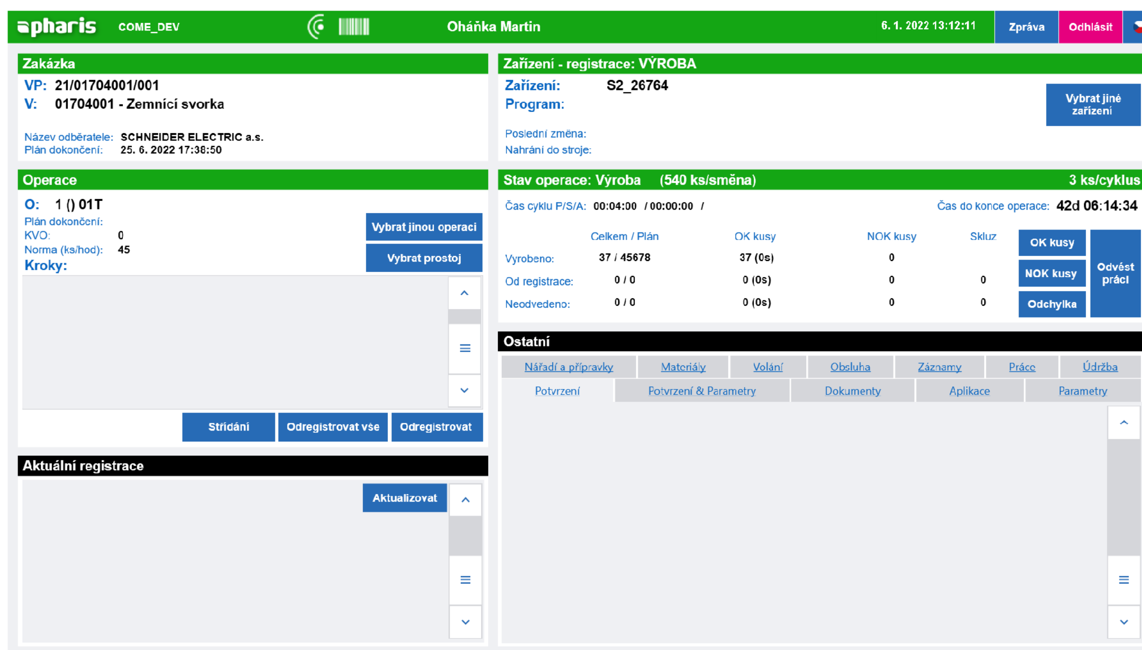
Webovou aplikaci využívají převážně řídící pracovníci, technologové nebo plánovači pro kontrolu aktuálního stavu jednotlivých výrobních zakázek a plnění plánu, generování reportů, zadávání nové výroby, úpravu uživatelských účtů, editaci základních číselníků, změny v technologickém postupu, změny v nastavení jednotlivých pracovišť a strojů atd.



Obrázek 2.3: Přehled technologických postupů ve webové aplikaci

Terminálová aplikace

MES PHARIS Výrobní klient je používán zaměstnanci pracujícími v samotné výrobě. Výrobní klient umožňuje registrace na operace, zobrazovat informace o operaci, informovat příslušný personál, nahrávat a stahovat řídicí programy, vyměňovat nástroje a formy, vést záznamy o spotřebě materiálu a odvodech práce atd. Výrobní klient je navržený pro dotykové monitory a uživatelské rozhraní je optimalizováno tak, aby její použití bylo co nejjednodušší. Tento typ klienta je postaven na technologii Windows Presentation Foundation a distribuce nových verzí je zajištěna prostřednictvím technologie na principu Click Once. Tato technologie umožňuje automatické klientské aktualizace řízené z jednoho místa.



Obrázek 2.4: Výrobní obrazovka terminálové aplikace

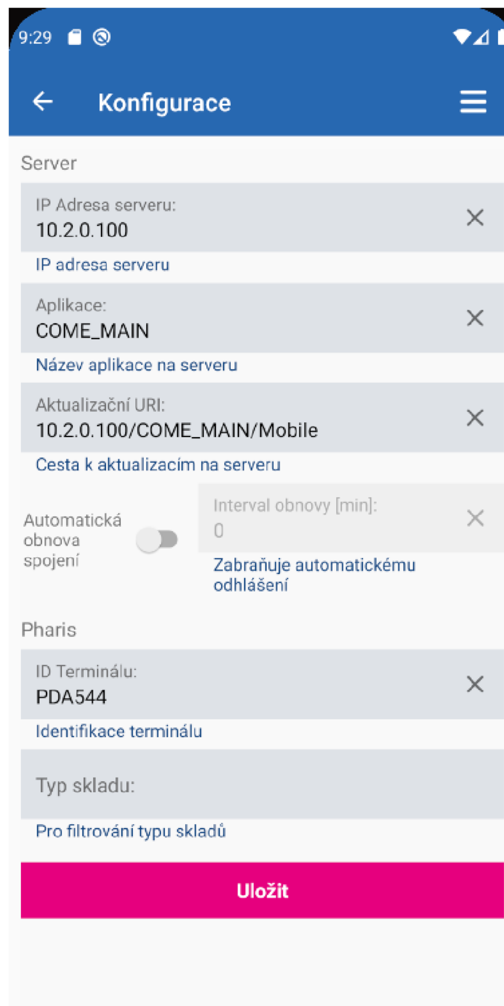
Event system

Event System (Systém událostí) je modulární .NET aplikace, která může spustit různé moduly aplikačního serveru. Je nainstalována jako služba Windows. MES PHARIS server se obvykle skládá z několika Event system s různými rolemi. To zajišťuje komunikaci s jinými systémy a systémy sběru dat, stejně tak s klienty MES PHARIS atd. Tato služba komunikuje prostřednictvím Microsoft Message Queue, WCF, WebSockets. Důležitou součástí služby je server pro řízení výroby a překladatelé, což jsou rozhraní pro komunikaci s externími systémy; např. OPC klient, protokol Euromap 63 nebo klient pro přístup do databáze technologie.

Mobilní aplikace

Výrobní Mini Klient MES PHARIS je přístupný pro mobilní zařízení s operačním systémem Windows Mobile. Toto řešení je založeno na principu Microsoft a .NET Compact Framework. Pokrývá specifické funkce dle přání zákazníka - obvykle aplikace pro správu nástrojů, odvody práce, načtení programů do strojů, řízení spolupráce a dodání výrobků atd. Tyto terminály jsou obvykle vybaveny RFID čipem nebo čtečkou čárového kódu. V současné době již tento klient není dále rozvíjen a je nahrazen novým. Ten je postaven na technologii Xamarin a cílí na zařízení s operačním systém Android.

Obrázek 2.5: Ukázka přístroje dodávaného zákazníkům



Obrázek 2.6: Konfigurační stránka mobilní aplikace na systému Android

Optimalizační server

Jde o server, který je hostitelem zařízení pro plánování výrobních příkazů. Nejpokročilejší verze vyžaduje stroj Linux a je poskytován pomocí kontejneru. Alternativním přístupem distribuce je plně nakonfigurovaný virtuální stroj, který lze spustit přes jakýkoliv virtualizační nástroj jako je HyperV, VM Ware nebo VirtualBox.

2.2.4 Postupy vývoje systému

Postup vývoje a návaznost kontrolních mechanismů je graficky znázorněn na obrázku 2.7.

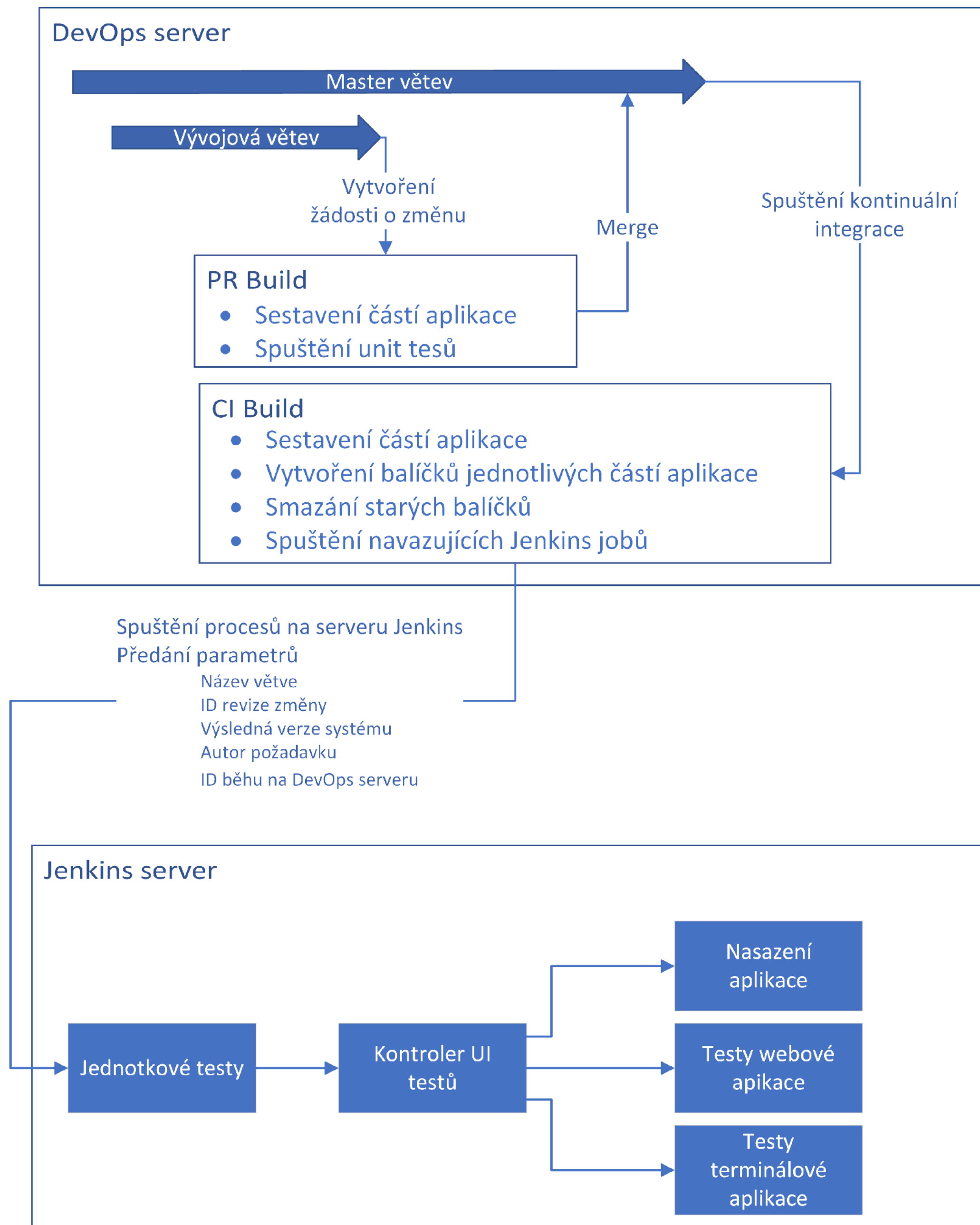
Zdrojový kód je verzován pomocí technologie GIT na lokální instalaci Azure DevOps Server[6]. Jsou udržovány dvě poslední verze, které jsou nasazeny u zákazníků a dále jedna verze, do které jsou implementovány nové požadavky na systém. Nad každou z těchto tří verzí jsou prováděny kontroly jak funkcionálního, tak nefunkcionálního charakteru.

Proces vývoje nové funkcionality je řízen agilní metodikou Scrum[7] po dvoutýdenních sprintech. Všechny pracovní položky (anglicky work item) jsou řízeny na serveru DevOps. Jednotlivé požadavky (anglicky requirment) se mohou skládat z více úkolů (anglicky task). Na začátku sprintu dostane vývojář přiřazené úkoly a chyby (anglicky bug) k opravě, které mají být ve sprintu vyřešeny. Ke každé položce, kterou vývojář řeší, je vytvořena nová větev v rámci repozitáře. To zajišťuje jasnou identifikaci změn, které byly nutné v průběhu řešení daného úkolu.

Pro nahrání nových změn je nutné využít žádost o změnu (anglicky pull request) s potvrzením minimálně jednoho dalšího vývojáře. Žádost o změnu má vydefinovaný postup sestavení, který je podmínkou pro přidání těchto změn. Jde o kontrolu sestavení všech součástí systému, běh jednotkových testů a kontrolu, zda je možné sestavit UI testy. Po splnění všech podmínek v žádosti o změnu následuje kontinuální integrace (CI build). Zde dochází k vytvoření balíčku aktuální verze, smazání starých verzí a spuštění následujících automatizovaných úkolů pomocí serveru Jenkins[8]. Systém Jenkins slouží k další automatizaci procesu vývoje systému. Tento server se stará o automatizované nasazení aplikací, běh jednotkových a UI testů a spouštění dalších kontrolních mechanismů a podpůrných procesů, například vytvoření instalačního balíčku.

Na serveru Jenkins je po úspěšné kontinuální integraci na serveru DevOps spuštěna úloha, která rozhoduje na základě parametrů, které dostala ze serveru DevOps (název větve a verze výsledného systému), jaký bude vybrán proces kontrol. Postup kontrol je totiž závislý na verzi systému. Následuje spuštění jednotkových testů. Pokud testování dopadlo úspěšně, je spuštěna úloha, která řídí průběh UI testů. Zde nejprve dochází k nasazení nové verze aplikace na speciálně určenou instanci pro účely automatizovaných testů. Během nasazení dochází k aktualizaci databázového schématu, nasazení prázdné databáze pouze se základními číselníky a aktualizaci jednotlivých částí aplikace (webová aplikace, terminálová aplikace a Event system). Pokud nasazení dopadlo úspěšně, pokračuje se spuštěním UI testů nejdříve webové aplikace a následně terminálové aplikace. Výsledky jednotlivých úloh jsou zobrazeny na monitoru pomocí přehledné vizualizační obrazovky, která pouze zobrazuje, zda byl poslední běh úspěšný nebo ne.

[6] Azure DevOps Server – https://azure.microsoft.com/en-us/services/devops/server/
[7] Co je Scrum – https://www.rascasone.com/cs/blog/co-je-scrum-jak-funguje
[8] Jenkins – https://www.jenkins.io/



Obrázek 2.7: Proces vývoje včetně návaznosti automatizovaných procesů

2.2.5 Současný stav sledování výkonnosti systému

V současné době není během vývoje informačního systému jeho výkonnost zásadně sledována. Jsou k dispozici informace o době sestavení aplikace, délce běhů jednotlivých jednotkových testů a UI testů, době nasazení jednotlivých částí aplikace a další. Údaje jsou poskytovány jak serverem Azure DevOps, kde dochází k sestavení aplikace, tak i Jenkins serverem, kde probíhají automatizované procesy. Tyto informace ale nejsou automaticky ani průběžně kontrolovány členy vývojového týmu. K jejich analýze se přistupuje pouze v případě nalezení problému a snahy o jeho vyřešení. Analýzu komplikuje to, že jsou jednotlivé procesy spouštěny na rozdílných serverech, které poskytují odlišnou reprezentaci dat a některé údaje nejsou lehce vyhledatelné a čitelné.

Ke sledování výkonnosti samotného systému z pohledu uživatele se přistupuje pouze v případě, kdy je hlášen problém, například dlouhá doba odezvy u zákazníka, nebo během testování nové funkcionality testovacím oddělením. Většina těchto případů je způsobena specifickými daty daného zákazníka, což značně komplikuje simulaci problému v prostředí vývojového týmu. Následuje analýza problému přímo u zákazníka a nalezení příčin problému. Případně je vytvořena záloha databáze a záznamů o událostech od zákazníka a následuje analýza na straně dodavatele se snahou o napodobení a vyřešení problému. Pro automatické zvyšování zátěže a provádění testů není připravena infrastruktura. Fyzické napodobení je také velice obtížné vzhledem k potřebě simulování činnosti vícero uživateli a také proto, že systém MES PHARIS v reálném provozu sbírá informace z jednotlivých výrobních strojů a zařízení.

Statická analýza zdrojového kódu

Mezi nástroje, které jsou nasazeny pro podporu vývoje, patří i SonarQube[9]. Jde o open-source produkt vyvinutý společností SonarSource. Produkt se zaměřuje na kontrolu kvality kódu, provádění statické analýzy kódu a vyhledávání chyb; podporuje přes 20 programovacích jazyků. Jedná se především o chyby, konstrukce s bezpečnostními riziky a charakteristiky kódu, které mohou znamenat hlubší problém v kódu (code smell). Díky tomu může produkt odhalit i problémy, které mohou mít souvislost s výkonem výsledného programu. Ve výsledném reportu je zobrazeno přesné místo v kódu a popis příslušného problému podle předem definovaných pravidel, které jsou v kódu kontrolovány.

[9] Domovská stránka SonarQube – https://www.sonarqube.org/

2.3 Analýza požadavků firmy UNIS na sledované parametry

Na základě několika schůzek s vývojovým týmem byly stanoveny požadavky, které mají být v rámci práce naplněny. Jedná se především o stanovení technologií, které mají být použity. Dále byly stanoveny požadavky na sběr metrik pro jednotlivé části systému MES PHARIS. Tyto požadavky se zaměřují na kontrolu problematických částí systému, kde již v minulosti byl zaznamenán problém. Jelikož jsou na systému závislé provozy několika podniků, je nutné tyto problémy včas detekovat a předcházet jim. Další požadavky se zaměřují na místa, která jsou z pohledu vývoje pro společnost zajímavá, nejsou zatím sledována a mohou přinést další zkvalitnění vývoje a odhalení případných chyb a tím zajistit větší spolehlivost a spokojenost uživatelů systému.

2.3.1 Obecné požadavky

Během vypracování diplomové práce bylo diskutováno mnoho požadavků na návrh a implementaci jednotlivých částí pro sběr metrik. Tyto požadavky byly podrobeny podrobné analýze. Byly specifikovány funkční (specifikace chování výsledného řešení) i nefunkční (specifikace upravující návrh a provedení řešení) požadavky.

Identifikátor	**Název**	**Kategorie**
req_code_style	Styl kódu	Kód
req_code_doc	Dokumentace kódu	Kód
req_testing	Testování	Kód
req_unit_tests	Jednotkové testy	Kód
req_code_struct	Struktura kódu	Kód
req_code_patterns	Návrhové vzory	Kód
req_configuration	Konfigurovatelnost	Funkcionalita
req_logging	Zápis záznamu o události	Funkcionalita
req_output	Získané údaje	Funkcionalita
req_run_report	Reportování postupu	Funkcionalita
req_instructions	Manuál	Dokumentace
req_extension	Rozšířitelnost	Návrh
req_technology	Využití technologií	Technologie

Tabulka 2.1: Obecné požadavky

Styl kódu bude konzistentní a bude dodržovat standardy a specifikace pro daný programovací jazyk (tabulka 2.1 – req_code_style). Kód bude strukturovaný dle logických bloků, aby byl přehledný a snadno srozumitelný (tabulka 2.1 – req_code_struct). Kód bude dokumentován obvyklým způsobem a dle standardů pro daný programovací jazyk (tabulka 2.1 – req_code_doc). Implementace bude využívat známé návrhové vzory na vhodných místech (tabulka 2.1 – req_code_patterns).

Testování funkcionality bude provedeno tam, kde to bude považováno za nutné ještě před samotným nasazením výsledného řešení (tabulka 2.1 – req_testing). Klíčové části kódu a základní funkcionalita budou pokryty jednotkovými testy (tabulka 2.1 – req_unit_tests).

Pro splnění požadavku (tabulka 2.1 – req_configuration) budou výsledné aplikace obsahovat konfigurační soubory pro snadnější úpravu klíčových parametrů a zamezení jejich pevnému zapsání ve zdrojovém kódu. Výsledná řešení pro sledování procesů na serverech a provádění testů budou navržena a vytvořena tak, aby bylo možné přidat další sledovanou úlohu na serverech nebo vytvoření dalších testů (tabulka 2.1 – req_extension). Během běhu budou výsledná řešení poskytovat údaje o svém průběhu, ať formou vypisování informací na standardní výstup nebo do souboru (tabulka 2.1 – req_run_report). Jednotlivá řešení mají výsledné informace poskytovat ve strojově čitelném formátu pro další zpracování (tabulka 2.1 – req_output). V případě problému během běhu bude chyba a další potřebné údaje pro následnou analýzu zapsány do záznamu o události. (tabulka 2.1 – req_logging).

Požadavek (tabulka 2.1 – req_technology) je ovlivněn stávajícím vývojem, jelikož je systém MES PHARIS vyvíjen na platformě Windows s použitím .NET frameworku a programovacího jazyka C#. Automatizované procesy a skripty jsou řešeny pomocí skriptů napsaných v PowerShell a spouštěných na serverech Azure DevOps a Jenkins. Tyto technologie mají být primárně využívány při řešení dané problematiky. Je to především z důvodu snadnější udržovatelnosti a také tím, že všichni členové vývojového týmu mají s těmito technologiemi bohaté zkušenosti.

2.3.2 Požadavky na systémové metriky

Jelikož má systém stanoveny hardwarové požadavky pro svůj běh, je nutné kontrolovat, zda některá součást sytému nezačíná brát více zdrojů než bylo stanoveno. To může mít negativní dopad na výkonnost celého systému, pokud nemá stroj, na kterém je systém provozován, žádné výkonnostní rezervy. Mezi parametry, které mohou být sledovány, patří využití operační paměti, využití procesoru a další podle požadavků společnosti.

2.3.3 Požadavky na nové výkonnostní testy

Současné testování funkcionality během vývoje nemá zahrnuty zátěžové testy. Firma UNIS by ráda zahrnula tento typ testů do procesu vývoje a kontinuální integrace. Tento typ testování by měl největší přínos pro samotný systém, protože zhoršení odezvy systému při větší zátěži má negativní dopad na výrobu u zákazníka. Navíc výsledky těchto testů mohou být prezentovány zákazníkům, jak současným tak i budoucím, za účelem garance určité úrovně doby odezvy a celkové funkcionality systému v reálné výrobě. Během těchto testů mohou být kontrolovány i systémové metriky (tabulka 2.2 – req_tests_metrics), jak tyto testy a různá zátěž ovlivňují jejich vytížení. Součástí práce je jak návrh postupu vývoje (tabulka 2.2 – req_tests_design) a fungování těchto testů, tak i implementace ukázkového testu pro prezentaci a získání měřených metrik (tabulka 2.2 – req_tests_implementation).

Systém obsahuje několik oblastí, ve kterých je zajištění správné odezvy na základě zátěže důležité. Jedná se především o terminálový klient (tabulka 2.2 – req_tests_terminal), který je umístěn na každém pracovišti a nejvíce ovlivňuje výrobu. Příkladem takového testu může být současné přihlášení k výrobě více uživatelskými stanicemi. V případě úspěšné implementace a časového prostoru se nabízí k testování i webová aplikace (tabulka 2.2 – req_tests_web). Ta je ale využívána podstatně méně než terminálová aplikace a nemá takový dopad na samotnou výrobu. Zde by se opět ověřovala doba odezvy na definované úkony při určitém počtu uživatelů.

Další oblastí, na kterou jsou kladeny vysoké nároky, je import dat do systému MES PHARIS (tabulka 2.2 – req_tests_import), nejčastěji z ERP systémů. Jedná se o data obsahující základní číselníky pro fungování systému, technologické postupy, výrobní příkazy a další. Existuje několik různých způsobů jak import provést. Zde by bylo nutné kontrolovat všechny tyto způsoby. Bylo by sledováno chování systému pod zátěží importu většího množství dat, které by se odvíjelo od reálného množství, které je v reálném provozu nahráváno do systému.

Systém také sbírá data z připojených strojů. Těchto údajů může být velké množství a systém je musí být schopen zpracovat. Další oblastí ke sledování jsou datové zdroje a tagy (tabulka 2.2 – req_tests_dataSource). Zde by bylo úkolem simulování strojů a jejich poskytovaná data (např. počet vykonaných cyklů, teploty, stav stroje atd.) a ta zasílat do systému. Cílem by bylo sledování doby odezvy pod zátěží více stroji a rozdílnou dobou zasílání jednotlivých údajů ze strojů.

Během testů má být zváženo zapojení profilátoru pro kód jazyku C# (tabulka 2.2 – req_tests_profiler). Jeho výsledky by poskytovaly informace o výkonnosti na úrovni tříd a jejich metod. Tyto informace by mohl vývojář snadno využít pro nalezení příčin zpomalení a při snaze zefektivnění dané metody a tím i celého kódu. Díky informacím o dřívějším běhu testů by bylo možné lehce identifikovat metody nebo třídy, které mají na zpomalení největší dopad oproti minulému běhu.

Identifikátor	**Název**
req_tests_terminal	Zátěžové testy terminálového klienta
req_tests_web	Zátěžové testy webové aplikace
req_tests_design	Vytvořit návrh pro vývoj testů
req_tests_implementation	Vytvořit ukázkové testy
req_tests_metrics	Měřit požadované metriky
req_tests_import	Zátěžové testy importu dat do systému
req_tests_dataSource	Zátěžové testy datových zdrojů a tagů ze strojů
req_tests_profiler	Využít profiler pro zisk některých metrik

Tabulka 2.2: Požadavky na nové výkonnostní testy

2.3.4 Požadavky na sledování automatizovaných procesů

Jak již bylo zmíněno v kapitole 2.2.5, jsou k dispozici informace o průběhu automatizovaných procesů. Z těchto procesů by firma UNIS chtěla získat detailnější data na jednotlivý běh důležitých automatizovaných procesů. Například na serveru Jenkins (tabulka 2.3 – req_jenkins) se běh jednotkových testů skládá ze samostatných částí (stažení repozitáře s aktuální změnou, stažení externích závislostí, sestavení testů, běh testů). Zde mají být získány jak informace o jednotlivých krocích procesu (tabulka 2.3 – req_jenkins_stage), tak časy jednotlivých testů (tabulka 2.3 – req_jenkins_tests). Pokud dojde v některém testu ke zpomalení, je to známka toho, že aktuální změna měla negativní vliv na výkonnost a je potřeba ji zkontrolovat. Velmi podobnými procesy jsou UI testy webové a terminálové aplikace. Zde budou také sledovány jednotlivé kroky procesu a výsledky testů, vzhledem k tomu, že se tyto testy zaměřují na jiný aspekt testování aplikace.

Dalším vytipovaným procesem je samotné nasazení sytému MES PHARIS na lokální testovací server, které se provádí automaticky každý večer nebo podle aktuální potřeby. Tento proces má být sledován z důvodu dřívějších problémů, kdy se postupem času začalo nasazení prodlužovat až do stavu, kdy byl proces nasazení automaticky ukončen z důvodu překročení časového limitu. Narůstání času bylo poté zpětně analyzováno a zjištěno, že probíhalo v průběhu dvou týdnů. Díky sběru detailních informací by mohl být tento problém odhalen dříve a jeho včasné řešení by předešlo nemožnosti nasazení nové verze. Toto sledování nemá za úkol sledovat výkonnost samotného systému, ale přispět k předcházení problémů během vývoje a nenarušování práce testovacího a vývojového oddělení. Po nasazení dochází k samotnému spuštění aplikace jak webového serveru, tak Event system. Zde by měla být zvážena možnost jak zjistit dobu, po kterou se jednotlivé části aplikace spouští (tabulka 2.3 – req_system_startup). Prodloužení času spouštění opět může znamenat problém ve výrobě, protože prodlužuje dobu, po kterou je výroba odstavena v případě, kdy je systém aktualizován nebo je potřeba jej z nějakého důvodu restartovat (např. výpadek lokální sítě, elektrické energie atd.).

Automatizované procesy jsou dále spouštěny na serveru DevOps (tabulka 2.3 – req_devops). Po nahrání nových změn a vytvoření žádosti o změnu na serveru DevOps je spuštěna úloha pro schválení žádosti o změnu. Zde je z pohledu firmy UNIS zajímavé kontrolovat dobu sestavení jednotlivých částí samotné aplikace (tabulka 2.3 – req_devops_stage). Po schválení žádosti o změnu je spuštěna úloha pro vytvoření balíčku jednotlivých částí aplikace. Zde má být kontrolována velikost jednotlivých částí aplikace (tabulka 2.3 – req_devops_packages). Dříve byl detekován problém až v produkci, kdy terminálová aplikace zabírala několikanásobně větší místo. Nová verze aplikace je distribuována pomocí sítě na jednotlivá pracoviště a to mělo za následek větší vytížení sítě a zmenšení dostupného místa na pracovních stanicích.

Samotná implementace nemá být závislá na současné podobě jednotlivých procesů, ta se může v budoucnosti libovolně měnit. Také výčet kontrolovaných procesů nemusí být konečný. Z těchto důvodů má být implementace co nejvíce univerzální (tabulka 2.1 – req_extension). Každé získané výsledky mají být spojitelné s konkrétní změnou v repozitáři pro snadnou identifikaci provedené změny (tabulka 2.3 – req_jobs_git).

Identifikátor	**Název**
req_jenkins	Sledovat vybrané úlohy na serveru Jenkins
req_jenkins_stage	Sledovat úlohy na úrovni jejich částí (kroků)
req_jenkins_tests	Sledovat časy existujících testů
req_devops	Sledovat vybrané úlohy na serveru DevOps
req_devops_stage	Sledovat úlohy na úrovni částí
req_devops_packages	Kontrolovat velikosti jednotlivých částí systému
req_jobs_git	Identifikovat změnu s aktuálním výsledkem
req_system_startup	Zjistit dobu spouštění systému

Tabulka 2.3: Požadavky na sledování automatizovaných procesů

2.3.5 Zpracování získaných metrik

Samotné navržení zátěžových testů a naměření hodnot nemá bez jejich následného zpracování tak velký přínos. Stále je nutné jejich ruční kontrolování a analyzování případných problémů. Tyto informace také nejsou k dispozici na jednom místě. Proto by firma UNIS dále chtěla zřídit centrální místo pro uložení získaných údajů a metrik. Dále by chtěla provádět automatizované spouštění procesů dle požadavků a následně provést analýzu získaných údajů třeba na základě historických měření, signalizovat případné odchylky a problémy pomocí přehledných reportů a vizualizačních obrazovek a následně zaslat notifikační email vybraným osobám.

Zpracování hodnot a jejich vizualizace bude mít velký přínos, protože již nebude nutné vybrané metriky analyzovat a sledovat ručně. Automatizovaný reporting bude vývojáře sám upozorňovat na vzniklý problém a ten dostane k dispozici informace o povaze problému, které pomohou ke snadnějšímu vyřešení problému.

Tato důležitá součást testování, tedy vyhodnocení výsledků, bude mít na výsledky této práce klíčový dopad. Vzhledem k předpokládané náročnosti této práce, sběr samotných hodnot a úprava systému MES PHARIS, je vyhodnocení, vizualizace a reportování součástí diplomové práce Aleše Ondráčka [12], která je obhajována současně s touto prací v roce 2022.

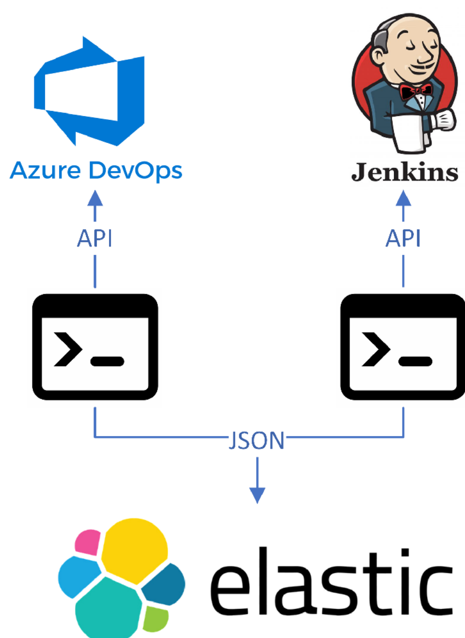
Kapitola 3

Návrh realizace sběru výkonnostních metrik

Tato kapitola obsahuje návrh řešení pro jednotlivé požadavky společnosti UNIS, viz 2.3. Dále je uveden seznam metrik, které budou z jednotlivých částí systému sbírány, včetně poskytovaného formátu.

3.1 Sledování automatizovaných procesů vývoje

V této části je popsána realizace programového řešení pro získávání informací o bězích automatizovaných úloh na jednotlivých integračních serverech. Obecná architektura řešení je na obrázku 3.1.



Obrázek 3.1: Schéma fungování sledování automatizovaných procesů

3.1.1 Server Jenkins

Existující řešení, které by splňovalo požadavky firmy UNIS, nebylo nalezeno i přes to, že samotný server Jenkins je open-source a poskytuje řadu doplňků. Nyní poskytuje informace a přehledy o jednotlivých úlohách a jejich bězích. Vzhledem k požadavku na jednotné místo, kde mají být údaje shromažďovány a k požadavku (tabulka 2.3 – req_jenkins_tests) na analýzu výsledků jednotlivých testů (jednotkové testy, UI testy), bude nutné vytvořit vlastní řešení.

Pro zajištění dat ze serveru Jenkins (tabulka 2.3 – req_jenkins) bude vytvořena konzolová aplikace v jazyku C# (tabulka 2.1 – req_technology). Ta bude konfigurovatelná pomocí konfiguračního souboru ve formátu JSON. V konfiguračním souboru budou vydefinované názvy úloh, které mají být sledovány, adresa a port, kde server běží. Dále bude existovat soubor s uloženými stavovými informacemi. Zde bude uvedeno ID běhu ke každé ze sledovaných úloh. Tento údaj bude značit poslední zpracovaný běh dané úlohy. Tato aplikace bude pravidelně spouštěna a ve svém běhu provede kontrolu, zda existuje novější běh.

Program bude využívat API, které poskytuje server Jenkins a jeho plugin [13]. API umožňuje získat informace jak o času jednotlivých částí, tak výsledek každé části (tabulka 2.3 – req_jenkins_stage). Využití dalších informací bude zváženo na základě komunikace s firmou UNIS. Pro požadavky typu GET API nevyžaduje na lokální instalaci přihlášení. Dále budou získány parametry, se kterými byl běh úlohy spuštěn (tabulka 2.3 – req_jobs_git). Ty jsou důležité pro identifikaci konkrétního běhu. Jedná se hlavně o verzi systému MES PHARIS, název větve, ze které aplikace vznikla, autora změny, u nasazení aplikace o název nasazované aplikace a další. Tyto informace jsou k dispozici v každé ze sledovaných úloh, jelikož jsou již předávány jako parametry ze serveru DevOps v rámci kontinuální integrace. Díky tomu není nutné provést zásahy do kterékoliv z implementací sledovaných úloh.

Výsledný program tedy zjistí seznam posledních běhů sledovaných úloh, pro každý provede získání potřebných informací, jejich zpracování a zapsání do výsledného formátu.

Může nastat problém při vyhodnocení získaných údajů a to v případě, kdy je některý z běhů spuštěn znovu, například ručně pro potřeby některého z oddělení během vývoje systému. Pro tento případ bude jako jednoznačná identifikace toho, že se jedná o další běh, který proběhl již dříve (běh měl stejné parametry v názvu větve a čísle verze výsledného systému MES PHARIS), sloužit název úlohy a ID běhu v rámci úlohy (jedná se o inkrementující se hodnotu, která vyjadřuje pořadí běhu v rámci jedné úlohy).

Zpracování testů

Jelikož mají být u úloh, které spouští i testy, sledovány časy jednotlivých testů (tabulka 2.3 – req_jenkins_tests), bude nutné upravit jejich stávající postup a ukládat do artefaktů výstupní soubor programu VSTest.Console.exe[1], který se využívá pro spouštění testů. Jde o soubor typu TRX. Tento soubor obsahuje výsledky jednotlivých testů ve formátu XML a je tedy vhodný pro další strojové zpracování. Uvnitř souboru jsou pro každý test uvedeny následující požadované vlastnosti a to jeho název, informace o délce běhu, času začátku a konce, výsledek nebo případný výstup.

[1] VSTest.Console.exe – https://docs.microsoft.com/en-us/visualstudio/test/vstest-console-options?view=vs-2022

Zde vyvstává problém, jelikož se některé testy mohu v rámci odlišných tříd jmenovat stejně. V rámci implementace testů ve společnosti UNIS se opravdu některé testy jmenují stejně. Je nutné jednoznačně identifikovat, který výsledek patří ke kterému testu. V souboru s výsledky jsou uvedeny další informace ke všem testům, ale mimo samotné výsledky. Pro jednoznačnou identifikaci bude stačit název třídy, protože v rámci jedné třídy bude vždy zaručeno, že se testy jmenují unikátně.

Spouštění aplikace

Pravidelné spouštění výsledné aplikace bude probíhat jako další automatizovaná úloha na serveru Jenkins. Časový interval mezi jednotlivými běhy bude určen na základě testování vytvořeného programu. Výhodou implementace pomocí úlohy na serveru Jenkins bude možnost snadného ručního spuštění běhu, pokud bude vyžadován sběr parametrů dříve, než uplyne doba mezi jednotlivými automatickými běhy.

Celkově má navržené řešení pouze menší nedostatek. Jde o to, že výsledky nejsou k dispozici ihned po dokončení běhu sledované úlohy. Výsledky budou k dispozici s menším zpožděním, ale vyváží to složitost údržby tohoto sledování. Navíc oproti současnému stavu sledování dojde k rapidnímu zlepšení. Po zvážení ostatních přístupů a jejich nedostatků bylo navržené řešení firmou UNIS schváleno k implementaci, protože mírné časové zpoždění je akceptovatelné vzhledem k celkovému přínosu.

Alternativní možnosti spouštění

Alternativním řešením by bylo upravit každou sledovanou úlohu tak, že by na konci svého běhu spustila další, která by zpracovala výsledky aktuálního běhu. Toto řešení by ale bylo náročnější na údržbu, jelikož by musela být upravena každá sledovaná úloha a také zajištěno provolání následující úlohy i v případě neočekávané chyby během běhu sledované úlohy. Poslední zmíněný bod by bylo možné obejít tak, že by se kontrolní úloha spouštěla jako první krok ve sledované úloze. Tato úloha by ale musela aktivně čekat a dotazovat se na stav sledované úlohy. Aktivní čekání by mělo za následek zvýšení zátěže na API a dále by zvýšilo počet aktivních běhů úloh dvojnásobně, což by vyčerpalo kapacitu jednotlivých agentů serveru Jenkins. Z těchto důvodů se přistoupí k implementaci samostatné aplikace popsané výše.

Dalším možnou úpravou navrženého řešení by bylo výslednou aplikaci spouštět jako službu přímo na serveru. To by mělo za následek zrušení všech požadavků na úpravy na server Jenkins. Také by to přineslo nevýhodu a to, že by nebyl snadno k dispozici záznam jednotlivých běhů včetně záznamů o událostech, které se budou jinak ukládat na serveru Jenkins.

3.1.2 Server Azure DevOps

Na tomto serveru probíhá sestavení aplikace a vytvoření balíčků jednotlivých částí výsledného informačního systému. Mají být sledovány dvě úlohy[2] (tabulka 2.3 – req_devops). Jedná se o úlohu při schvalování nové změny ve zdrojovém kódu a následující úlohu kontinuální integrace, ve které se vytváří balíčky jednotlivých částí systému. Na serveru probíhají i další úlohy, o které může být sledování rozšířeno, aktuálně ale nejsou pro společnost tak kritické.

Zde má být sledován celkový čas běhu, čas na jednotlivé kroky (tabulka 2.3 – req_devops_stage) a v případě vytvoření balíčků částí aplikace bude kontrolována jejich velikost a počet souborů (tabulka 2.3 – req_devops_package). Pro jednoznačnou identifikaci běhu bude využito ID přiřazené samotným serverem. Toto ID má zajištěnou unikátnost napříč všemi úlohami. Pro účely dalšího zpracování a lepší identifikaci jednotlivých běhů člověkem bude dále využit název větve, ve které daný běh běží a číslo verze výsledného systému.

Navrhovaný způsob řešení

Existující řešení, které by splňovalo požadavky firmy UNIS, nebylo nalezeno. Samotný server DevOps poskytuje informace a přehledy o jednotlivých bězích. Vzhledem k požadavku na jednotné místo, kde mají být údaje shromažďovány, bude nutné vytvořit vlastní řešení.

Řešením bude vytvoření další konzolové aplikace v jazyku C#. Chování bude velmi obdobné jako v případě aplikace pro získávání informací ze serveru Jenkins. Aplikace bude součástí stejné úlohy na serveru Jenkins a bude tedy pravidelně spouštěna. Ke konfiguraci bude opět využit soubor s formátem JSON. Bude obsahovat adresu a port, kde běží server DevOps, dále název organizace a projekt, kde je vyvíjen systém MES PHARIS a kde běží sledované úlohy. Dalším souborem bude soubor s informacemi o stavu, kde bude uvedeno ke každé sledované úloze ID posledního běhu. Po spuštění aplikace zjistí, který běh byl zpracován jako poslední a všechny novější postupně zpracuje. Všechny potřebné parametry pro identifikaci jednotlivých běhů jsou již obsaženy ve výsledcích o běhu sledovaných úloh a není nutné provést další dodatečné úpravy v jejich implementaci.

Pro získání těchto informací bude využito API [1], které poskytuje samotný DevOps server. Přístup na API vyžaduje přihlášení. Z tohoto důvodu budou využity přihlašovací údaje účtu, pod kterým jsou na serveru Jenkins úlohy spouštěny. Tento účet má přístup na server DevOps.

Navržené řešení bylo zvoleno na základě předchozího návrhu, aby byla zajištěna určitá podobnost jednotlivých návrhů, což přinese lepší udržitelnost řešení. Dalším přínosem je, že není nutné zasahovat do postupu sledovaných úloh. Vzhledem k tomu, že na serveru Jenkins vznikne nová úloha, která bude zajišťovat sbírání informací ze sledovaných úloh na tomto serveru, není nutné vytvářet další úlohu a celkově to přinese větší přehlednost výsledného řešení na sběr metrik z automatizovaných procesů. Podobnost řešení jak pro server DevOps a Jenkins byla oceněna a záměr návrhu firmou UNIS schválen k realizaci.

[2] V rámci serveru DevOps se úloha označuje jako pipeline.

Sledování velikosti částí systému

Postup kontinuální integrace bude nutné rozšířit o krok, který zjistí velikosti jednotlivých sestavených částí aplikace. Za tímto účelem bude vytvořen PowerShell skript, který potřebné informace zjistí a údaje zapíše do svého standardního výstupu. Tento výpis bude uložen v daném kroku v úloze ve formátu JSON. Při zpracování daného běhu budou tyto informace vyzvednuty z daného kroku, který bude vhodně pojmenován pro snadnou identifikaci k dalšímu zpracování. Tím bude zajištěno, že jsou všechny potřebné informace k dispozici v rámci výsledků jednotlivých běhů.

Alternativní možnosti spouštění

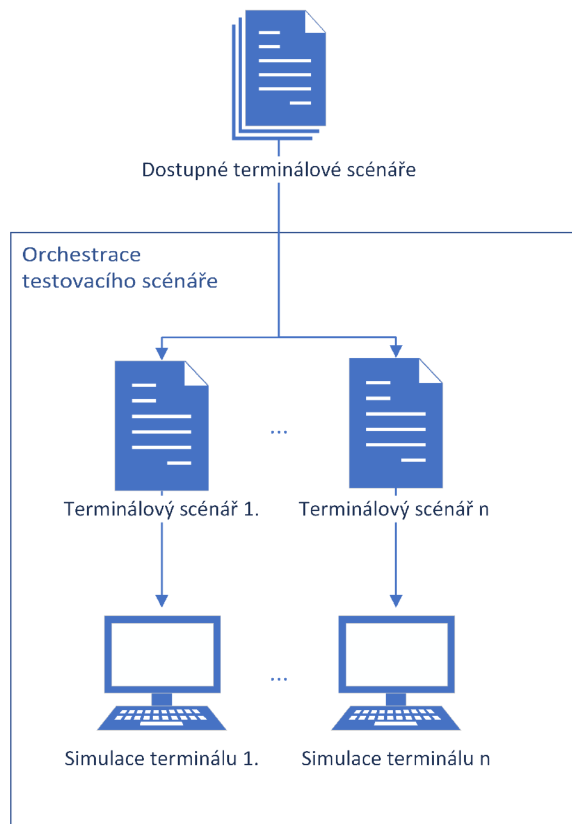
Alternativním řešením by bylo upravení postupů ve sledovaných úlohách na serveru DevOps, ale následkem by byla náročnější udržitelnost výsledného řešení a v případě změny zasahovat do každé ze sledovaných úloh. Na druhou stranu řešení by přineslo výsledky okamžitě v momentu, kdy jsou k dispozici. Další možností by bylo výslednou aplikaci spouštět jako službu a mít ji hostovanou přímo na operačním systému. Toto řešení by ale snížilo přehled o jednotlivých bězích a zhoršilo možnost snadného ručního spuštění.

3.2 Automatizované výkonnostní testování

Automatizované výkonnostní testování není v současné době firmou UNIS řešeno. Jelikož má být primárně testována terminálová aplikace, která je postavena na technologii WPF, nebylo nalezeno vhodné existující řešení. Další problém je, že pro simulaci činnosti uživatele je nutná interakce s UI aplikace. Aplikace navíc blokuje celou obrazovku a odchytává stisknuté klávesy a interakci myši. Tyto problémy se ještě umocní v případě, kdy má být prováděno zátěžové testování s více současnými uživateli.

Z důvodu minimalizace těchto problémů bude probíhat testování bez aktivního UI prostředí. Bude vytvořena mezivrstva, která bude simulovat činnost uživatele programově. Tato mezivrstva bude využívat logiku aplikace, která je volána z UI. Tím bude zajištěno stejné chování jako při samotné interakci s aplikací. Mezivrstva bude podporovat základní ovládání terminálu. Především jde o přihlášení uživatele, výběr zařízení, výrobního příkazu a operace k registraci, registrování uživatele k výrobě, zadání odvodu práce včetně možnosti zadání neshodných kusů, odregistrování se od výroby a odhlášení uživatele z aplikace. Další ovládání bude možno snadno rozšířit podle potřeb dalších testů. Zmíněné základní ovládání bude dostatečné pro simulování běžné činnosti uživatele a ověření základního výrobního scénáře.

Následně bude vytvořen předpis pro práci (scénář), kterou má daný terminál vykonat. Ten bude vytvořen pomocí již zmíněné mezivrstvy. Dále bude test definovat počet terminálů, které mají daný test vykonávat. Pro vykonání celého testu bude nutné vytvořit spouštěč, který se bude starat o orchestraci jednotlivých terminálů (scénářů). Obecná architektura běhu testů je na obrázku 3.2.



Obrázek 3.2: Návrh orchestrace

Infrastruktura pro běh testů

Pro samotné testování bude vytvořena infrastruktura mimo současně využívané servery. Tím bude zajištěno, že průběh testování nebude ovlivňován dalšími činnostmi. K tomu bude využit přiřazený fyzický server. Na tento server se bude nasazovat systém MES PHARIS pomocí standardních procesů na serveru Jenkins. Před každým spuštěním testů dojde k nasazení poslední verze systému a nahrání prázdné databáze. Pro testování se ale bude využívat interní síť firmy pro možnost využívat server Jenkins. To může ovlivňovat provádění testů a měření, protože síť může být zatížena jinými činnostmi v rámci společnosti. Databáze pro fungování systému se může nacházet buď na stejném serveru jako instance MES PHARIS nebo na samostatném serveru. V reálně nasazených systémech u zákazníků je nejčastěji volena možnost, kdy je databáze i samotná aplikace na stejném serveru. Z tohoto důvodu a také z důvodu, že by byl potřeba vyčlenit další fyzický server, bylo přistoupeno k řešení, že databáze i aplikace bude na stejném serveru.

Pro automatizované spouštění bude využit server Jenkins, na kterém bude za tímto účelem vytvořena nová úloha. V současné době bude testování zprovozněno pouze nad vývojovou větví, do které se implementují nové funkcionality. Zde bude samotná funkčnost testů sledována a v případě pozitivního přínosu implementace bude spouštění rozšířeno i na dvě další větve (viz kapitola 2.2.4) a samotné testování dále rozšiřováno podle potřeb společnosti. Frekvence spouštění bude stanovena podle doby běhu testovací sady a její náročnosti. V úvahu připadají dvě možnosti a to zařadit tyto testy do postupů kontinuální integrace

spouštěné po každém vytvoření nové verze systému, tedy nahrání nové změny (viz obrázek 2.7) nebo jednou za den, nejspíše v noci. V případě spouštění během kontinuální integrace bude přínosem podrobení testování každá změna zvlášť. To bude mít přínos ve snadnější identifikaci změn, které vedly k problému s výkonem. Na druhou stranu by toto testování bylo spouštěno během dne a mohlo by se na něm projevit jak již zmíněné vytížení sítě nebo jiné infrastruktury. Pokud by byl zvolen přístup s testováním v noci, mělo by to výhodu v tom, že bude testování probíhat v době, kdy je na interní infrastrukturu společnosti vyvíjeno minimální zatížení. Je ale nutné zvolit dobu mimo další automatizované procesy během noci, automatické zálohování atd. Jako nevýhodu tohoto řešení můžeme vidět to, že nebude testována každá změna zvlášť, což může zkomplikovat postup při hledání změny, která vedla ke zhoršení výkonnosti systému. Během dne totiž může dojít k nahrání více změn. Zde bude hrát roli zkušenost vývojáře a analýza pracovních položek (work item), které byly předchozí den vyřešeny.

Sledování metrik

Na serveru, kde bude provozována instance systému MES PHARIS, bude dále sledováno zatížení hardwarových zdrojů stroje během testování. To bude zajištěno pomocí Metricbeat[3] poskytované Elasticem. Tento sběr je zvolen z důvodu navazující práce Aleše Ondráčka [12], která pro vizualizaci má využívat řešení Kibana a pro práci s daty Logstash. Samotný sběr bude probíhat pouze během průběhu testování s vysokou frekvencí sběru. Tím bude minimalizován objem naměřených dat a také náročnost na diskovou kapacitu.

Pro měření doby odezvy bude každý terminál během běhu testů zapisovat informaci o délce čekání na odpověď ze serveru do záznamu o události. Například se bude jednat o dobu přihlášení uživatele, dobu registrace k výrobní operaci nebo dobu běhu celého scénáře. Dále se budou zapisovat další údaje a informace, které mohou lépe objasnit aktivitu systému.

Generování testovacích dat

Dalším problémem je samotná příprava testovacích dat, nad kterými bude test pracovat. Systém MES PHARIS umožňuje import dat pomocí importačního excelu nebo pomocí generátorů, které využívají bussiness vrstvu. Tyto generátory jsou využívány v současných UI testech. Pro jejich větší využití by bylo nutné provést jejich refactoring a rozšíření, aby podporovaly vytváření všech potřebných dat. Během vývoje testů budou vyzkoušeny obě varianty. Na základě zkušeností s používáním bude vybrána vhodnější varianta. Vytváření dat bude muset reflektovat počet terminálů, které v testu budou figurovat.

Povaha testovacích dat bude zvolena na základě poskytnutých údajů od zákazníků. Tyto informace pomohou testy co nejvíce přiblížit reálnému prostředí, což bude mít za následek větší kvalitu testů.

[3]Metricbeat – https://www.elastic.co/beats/metricbeat

3.2.1 Spouštěč scénáře

Terminálovou aplikaci je nutné z povahy její implementace spouštět v samostatných procesech. Z tohoto důvodu bude nutné samotné spuštění scénáře implementovat ve dvou hlavních částech. První bude třída, která se bude starat o spuštění jednotlivých scénářů pomocí samotného spouštěče. Tím bude zapouzdřeno jeho spouštění a předání nutných parametrů pro jeho běh. Tato třída bude také obstarávat případné ukončení jednotlivých procesů v případě překročení času na celý test nebo selhání některého z procesů. Druhou částí bude samotná konzolová aplikace, která provede načtení souboru s implementací scénáře a jeho vykonání.

Všechny scénáře budou vycházet ze stejného rozhraní (interface), tím bude sjednoceno chování a implementace všech scénářů. Rozhraní bude obsahovat dvě metody a to `Init`, která provede nutné úkony pro přípravu scénáře, které nejsou součástí samotného běhu. V našem případě se bude jednat o vygenerování dat, nad kterými bude terminál pracovat. Druhá metoda `RunScenario` provede samotný běh scénáře. To bude obnášet vykonání jednotlivých úkonů v aplikaci terminálu nad připravenými daty. Obě metody budou mít návratový typ logické hodnoty (bool), která bude signalizovat logický výsledek samotného scénáře.

3.2.2 Formát záznamu o události

Pro další zpracování bude během provádění testu každý scénář zapisovat důležité události ve formátu JSON do společného souboru. Ten bude pojmenován podle času začátku testu ve formátu *yyMMddHHmmss.txt*. Pro sběr záznamů bude využita aplikace Filebeat[4] poskytovaná Elasticem.

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 záznamu podle výčtu	ELogLevel viz 3.2
LogType	Typ záznamu podle výčtu	ELogType viz 3.3

Tabulka 3.1: Struktura záznamu o události během výkonnostního testování

[4]Filebeat – https://www.elastic.co/beats/filebeat

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

Tabulka 3.2: Položky výčtu ELogLevel

Záznam o události typu `Request` značí, že je očekáván další záznam tentokrát typu `Response` se stejnou hodnotou v klíči `CommunicationId`. Tím jsou tyto záznamy provázány a značí navazující odpověď. Takto označené záznamy nejsou ve smyslu, že by odesílaly požadavky na orchestrátor, ale že byl odeslán požadavek na server, například když se uživatel přihlašuje ke svému účtu je záznam typu `Request`. Jakmile server odpoví a uživatel je přihlášen, bude zapsán záznam s typem `Response`, kdy bude mít stejnou hodnotu klíče `CommunicationId`. Časový rozdíl mezi takto provázanými záznamy bude značit dobu zpracování požadavku na serveru.

Název	**Hodnota**	**Význam**
Request	0	Záznam o události typu požadavek.
Response	1	Záznam o události typu odpověď
Message	2	Záznam o události typu zpráva

Tabulka 3.3: Položky výčtu ELogType

3.3 Sledované metriky

Na základě požadavků firmy UNIS a provedených analýz byly vybrány metriky, které budou podstatné a zajímavé z hlediska dalšího zpracování. Dále byly zapracovány požadavky Aleše Ondráčka vzhledem k jeho navazující práci a to za účelem snazšího zpracování těchto dat. Jeho požadavky se týkaly zpracování časových údajů a jejich převedení do jednotného formátu a také vypočítání dob běhu. Tyto požadavky vyplývají z toho, že v rámci aplikace je zpracování poměrně snadné a rychlé, kdežto následné dopočítávání hodnot v proudovém zpracování na platformě ELK se jeví jako neefektivní. V následujících tabulkách jsou popsány jednotlivé informace, které budou získávány z jednotlivých částí. U každé metriky je uveden stručný popis a datový typ, kterým je reprezentována. Díky těmto údajům bude možné jednoznačně identifikovat jednotlivé běhy a konkrétní běh bude možné spárovat s provedenou změnou ve zdrojovém kódu. Další informace, jako například délka běhu jednotlivých částí úlohy nebo informace o jednotlivých testech, budou umožňovat provádění analýz. Díky historickým údajům bude možné odhalit potencionální problém. Získané metriky budou předávány ve formátu JSON, který podporuje snadné strojové zpracování.

3.3.1 Metriky ze serveru 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 3.5
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 mili-sekundách	long
testsResult	Výsledek testů, pokud je běh obsahoval	TestsResult viz 3.6

Tabulka 3.4: Struktura získaných parametrů ze serveru Jenkins

Údaj *parameters* u sledovaných úloh s testy bude vždy obsahovat klíče *Branch, Commit, MESPharisVersion*. Jde o identifikaci větve, nad kterou byl daný běh spuštěn, jednoznačnou identifikaci revize kódu (commit) pomocí jeho hash a údaj o výsledné verzi systému MES PHARIS. Tyto údaje budou sloužit k identifikaci změn, které mohou být příčinou odhaleného problému.

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 mili-sekundách	long

Tabulka 3.5: Struktura získaných parametrů v objektu typu Stage (Jenkins)

Název	**Popis**	**Datový typ**
testsRunResult	Kolekce s výsledky jednotlivých testů	List<TestRunResult> viz 3.7
startTime	Čas začátku části	DateTime
finishTime	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 3.6: 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 3.7: Struktura získaných parametrů v objektu typu TestRunResult (Jenkins)

3.3.2 Metriky ze serveru 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 3.10
packages-ContentInfo	Obsahuje informace o výsledných balíčcích systému. Jako klíč je použit název balíčku	Dictionary <string, Package-Content> viz 3.9
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 viz 3.12
requestedBy	Informace o účtu, který běh spustil	Identity viz 3.12

Tabulka 3.8: Struktura získaných parametrů ze serveru DevOps

Pokud daný běh nevytvořil balíčky, bude parametr *PackagesContent* prázdný. Parametr *parameters* nemá jednotný formát z důvodu univerzálního použití nad více úlohami na severu DevOps. V některých případech může být parametr prázdný. Získané informace mohou být použity ve vizualizaci jako doplňkové informace. V případě běhu úlohy pro schválení žádosti o změnu jsou zde uvedené informace podle tabulky 3.11.

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 3.9: 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 viz 3.13

Tabulka 3.10: 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 3.11: Struktura získaných parametrů v případě běhu úlohy pro schválení žádosti o změnu (DevOps)

Název	Popis	Datový typ
displayName	Neunikátní jméno uživatelského účtu, které je primárně zobrazováno	string
uniqueName	Unikátní jméno uživatelského účtu ve formátu (doména/uživatelské_jméno)	string

Tabulka 3.12: Struktura získaných parametrů v objektu typu Identity (DevOps)

Název	Popis	Datový typ
url	Url adresa	string

Tabulka 3.13: Struktura získaných parametrů v objektu typu Log (DevOps)

3.3.3 Metriky z výkonnostního testování

Během výkonnostního testování budou sbírány informace o využití hardware zdrojů na serveru s instancí MES PHARIS. Především se bude jednat o celkové vytížení CPU, využití paměti RAM nebo zatížení disku, využití sítě a další. Půjde o modul Metricbeat s názvem System[5]. Z tohoto modulu budou sledovány následující oblasti `cpu, memory, network, process, process_summary, socket_summary, diskio`. Dále je upřesněno sledování procesů a to pouze na procesy spouštěče scénářů.

Dalším údajem bude doba odezvy serveru, kdy se bude měřit doba mezi odesláním a přijetím odpovědi. To bude zajištěno pomocí záznamů o událostech ze samotného systému MES PHARIS. Dále se bude sledovat doba běhu celého scénáře. Podle potřeby bude možné sbírat i více detailnějších informací o využití jednotlivých systémových zdrojů.

[5]Modul System v Metricbeat – https://www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-module-system.html

Kapitola 4

Implementační detaily sběru výkonnostních metrik

Tato kapitola popisuje finální implementaci navrženého řešení pro jednotlivé požadavky společnosti UNIS, viz 2.3.

4.1 Měření automatizovaných úloh v rámci vývoje

Jak bylo popsáno v návrhu, pro sledování automatických procesů vznikly dvě samostatné aplikace, které využívají společné jádro.

Obě řešení jsou umístěna ve zvláštním repozitáři na serveru DevOps. Pro spouštění je využit server Jenkins, kde vznikla nová úloha, která provede stažení zdrojových kódů, jejich sestavení a následné spuštění. Toto řešení bylo vybráno proto, že dojde k automatickému projevení nových změn nahraných do repozitáře a odpadá nutnost provádět sestavení a distribuování zkompilovaných souborů.

Pro sledování vybraných úloh, byl stanoven interval mezi jednotlivým spuštěním automatizované úlohy, která spouští jednotlivé aplikace pro sběr dat pro každý ze serverů. V pracovní dny je úloha spouštěna ve dvaceti minutovém intervalu v čase od šesti ráno do pěti odpoledne. Jelikož některé sledované úlohy jsou spouštěny i o víkendu, je provedeno i jedno spuštění v nepracovní dny v ranních hodinách.

4.1.1 Společná část aplikací

Pro implementaci byla využita platforma .NET 6. Jde aktuálně o nejnovější dostupný framework pro jazyk C#.

Společná část implementuje funkcionalitu pro zpracování argumentů příkazové řádky při spuštění aplikace. Načtení a práci s konfigurací aplikace, jelikož obě aplikace využívají konfigurační soubor ve formátu JSON. Třídu zajišťující zasílání zpráv pomocí protokolu TCP. Základní funkcionalitu třídy, která provádí dotazování pomocí API na jednotlivé servery, nebo třídu pro zápis záznamu o události včetně rozhraní (interface) pro případnou změnu zápisu bez nutnosti zásahu do použití.

Zápis záznamu o události

Pro snadnější změnu implementace zápisu záznamu o události v průběhu aplikace bylo vytvořeno rozhraní (interface) `ILogger`. Toto rozhraní obsahuje pouze jednu veřejnou metodu `Log` s parametrem typu `string`. Tím je umožněna snadná implementace další třídy pro zápis záznamů podle dalších požadavků.

Pro základní zápis byla implementována třída `ConsoleLogger`. Metoda `Log` provede zapsání záznamu do konzole ve formátu *aktuální_čas | zpráva*.

Jelikož je instance pro zapisování potřebná ve více částech (blocích) programu, byl pro správu instance využit návrhový vzor jedináček (anglicky singleton). Tím je zajištěno, že v programu existuje pouze jedna instance a ta je dostupná kdekoliv v programu. Implementace byla vytvořena ve třídě `LoggerManager`. Instance je vytvořena až když je potřeba. V případě, že je potřeba změnit třídu, která provádí zápis záznamů, bude změna provedena v této třídě. Implementace je ukázána ve výpisu 4.1. Díky použití zámku je implementace odolná i při použití ve vícevláknovém řešení.

```
1  public class LoggerManager
2  {
3      private static readonly object locker = new();
4      private static ILogger logger;
5      public static ILogger Logger
6      {
7          get
8          {
9              if (logger == null)
10             {
11                 lock (locker)
12                 {
13                     if (logger == null)
14                     {
15                         logger = new ConsoleLogger();
16                     }
17                 }
18             }
19             return logger;
20         }
21     }
22 }
```

Výpis 4.1: Implementace jedináčka pro práci s instancí pro zápis záznamu o události

Konfigurace aplikace

Konfigurovatelnost aplikací je zajištěna pomocí souboru *appsettings.json*, který je ve formátu JSON a je očekáván v adresáři spolu s výsledným programem. Pro načtení souboru je využita třída předepisující vlastnosti konfiguračního souboru (anglicky Data Transfer Objects - DTO), ze které je vytvořena instance, do které je obsah souboru nahrán. Zpraco-

vání souboru a vytvoření instance provádí knihovna `System.Text.Json`[1], která je součástí .NET 6.

Jelikož má každá aplikace rozdílný konfigurační soubor, bylo řešení pro práci s konfigurací implementováno genericky. Tento přístup je zvolen z důvodu maximalizace snahy o opětovné použití kódu a snadnější udržitelnosti výsledného řešení. Dále je tím odsunuta nutnost specifikace typu (třída DTO), se kterým bude výsledná třída pracovat až do momentu, kdy je třída deklarována a vytvořena kódem.

Implementace je ukázána ve výpisu 4.2. Jedná se o stejnou konstrukci jako v případě implementace jedináčka pro účely zápisu záznamu o události ve výpisu 4.1. Opět je řešení bezpečné pro vícevláknové řešení a instance je vytvořena, až když je vyžadována. Výsledná třída pro použití v aplikaci má následující podobu `public class LocalConfig : ConfigManager<Config>{ }`, kde `Config` je DTO třída s předpisem možností konfigurace. Přístup k instanci s konfigurací se provádí voláním `config = LocalConfig.Config;`. K využití mezitřídy bylo přistoupeno z důvodu, aby nebylo nutné ve všech voláních definovat typ DTO třídy pro konfiguraci. Následně toto řešení umožňuje doimplementovat specifické řešení pro danou konfiguraci.

Třída `ConfigManager` dědí od třídy `Location`, kde jsou vydefinovány cesty ke konfiguračním souborům. V případě nutné změny stačí změnit cestu v této třídě a projeví se v celém programu.

Popis možností konfigurace je znázorněn pro každou aplikaci zvlášť v tabulkách 4.2 a 4.4.

```
1  public abstract class ConfigManager<T> : Location
2  {
3      private static readonly object locker = new();
4      private static T config;
5      public static T Config
6      {
7          get
8          {
9              if (config == null)
10             {
11                 lock (locker)
12                 {
13                     if (config == null)
14                     {
15                         config = ConfigLoader.Load<T>(ConfigPath, false);
16                     }
17                 }
18             }
19             return config;
20         }
21     }
22 }
```

Výpis 4.2: Implementace jedináčka pro práci s instancí obsahující konfiguraci

[1]Knihovna System.Text.Json – https://docs.microsoft.com/en-us/dotnet/api/system.text.json

Stav aplikace

V rámci konfigurace aplikace je řešen i stav aplikace. Jedná se o soubor ve formátu JSON, ve kterém jsou uloženy informace o posledním zpracovaném běhu dané úlohy, aby bylo možné při dalším spuštění aplikace navázat pouze na nové běhy a znovu neprovádět zpracování starých. Pro načtení souboru je opět připravena DTO třída podle informací, které je nutné ukládat, instance této třídy je opět vytvořena za pomocí knihovny `System.Text.Json`. K načtení stavu dochází v momentě načtení konfiguračního souboru a je dostupný v rámci instance konfigurace. Stav umožňuje své uložení jak ze své instance, tak z instance konfigurace.

Pro práci se stavem je implementována třída `StateBase`, která rovněž vychází z třídy `Location`, tím je zaručen jednotný přístup k cestám, kde jsou soubory umístěny. Načtení probíhá pomocí volání `StateBase.Load<State>()`, kde `State` je DTO třída s předpisem možností stavu aplikace.

Pokud není soubor se stavem nalezen, je to aplikací považováno za prvotní spuštění a dojde ke zpracování všech dostupných běhů sledovaných úloh. Soubor bude poté vytvořen a při dalším běhu aplikace využíván a aktualizován.

Soubor se stavem má v obou aplikacích stejné schéma. Jedná se o objekt, který obsahuje pouze jeden klíč `LastBuildId`. Ten je rovněž typu objekt a obsahuje dvojice klíč hodnota, kde jako klíč je identifikátor sledované úlohy a hodnota je identifikátor poslední úspěšně zpracované úlohy. Díky objektu může být obsah dynamický a přizpůsobí se počtu sledovaných úloh. Částečná implementace třídy pro práci se stavem je znázorněna ve výpisu 4.3. Ukázka obsahuje pouze vlastnost starající se o načtení objektu s dvojicemi úloha:běh. Pro přístup k hodnotě je využito následující volání `Config.State.LastBuildId[pipelineId]`.

```
1  public class State : StateBase
2  {
3      /// <summary>
4      /// Dvojice identifikator ulohy a identifikator
5      /// posledniho zpracovaneho behu ulohy
6      /// </summary>
7      public Dictionary<string, long> LastBuildId { get; set; } = new();
8
9      ...
10
11 }
```

Výpis 4.3: Implementace DTO třídy pro práci se stavem

Odesílání výsledku do navazující aplikace

Po zpracování každého běhu je možné výsledek odeslat pomocí TCP na určenou adresu a port. Řešení je zapouzdřeno třídou `LogstashSender`, jelikož je výsledek nejčastěji posílán do platformy ELK konkrétně do Logstash. Bezpečnost odesílaných dat není v práci řešena, jelikož odesílání probíhá v rámci interní sítě.

Třída odesílá výslednou DTO třídu, která obsahuje všechny informace podle stanovených formátů (viz tabulky 3.4 a 3.8). Tato třída je před odesláním serializovaná do formátu JSON,

který je výstupem těchto aplikací a platforma ELK v navazující práci Aleše Ondráčka [12] jej očekává.

Během odesílání se aplikace pokusí odeslat výsledek na zadanou adresu, pokud se to z nějakého důvodu nepodaří, vyčká aplikace pět sekund a odeslání se opakuje. Případná chyba je vypsána do záznamu o události aplikace. V případě, že se ani opětovné odeslání nezdaří, je chyba rovněž zanesena do záznamu o události a celá aplikace ukončena. K ukončení dochází z důvodu, že se nepodařilo odeslat aktuální výsledek a pokud by byl odeslán další, došlo by k narušení sekvenčního zpracování běhů úloh. Takto přeskočený běh úlohy by již nebyl zpracován v dalším běhu aplikace. Dále by to způsobovalo problém s kontinuálním zobrazením a ve vizualizaci by mohl být chybějící běh snadno přehlédnut.

Výsledky je rovněž možné uložit do souboru ve formátu JSON. Konfigurační soubor dovoluje současné nastavení jak pro odeslání pomocí TCP, tak uložení do souboru. U formátu souboru je možné definovat, zda má být jeho obsah formátovaný nebo zda se má jednat o jeden řádek. Soubor je uložen s názvem ve formátu `pipelineId_buildId.json` do adresáře, odkud byla aplikace spuštěna.

Argumenty příkazové řádky

Výsledné aplikace podporují zadání argumentů při spuštění. Jsou k dispozici dva přepínače. Pokud mají být argumenty využity, je nutné zadat oba dva. Pro zadání přepínačů je k dispozici krátký i dlouhý formát, popis argumentů je v tabulce 4.1.

Popis	Krátký formát	Dlouhý formát
Identifikátor úlohy, která bude zpracovávána	`-p`	`--pipeline`
Identifikátor běhu úlohy, který bude zpracován	`-b`	`--build`

Tabulka 4.1: Popis argumentů akceptovaných aplikací

V případě zadání argumentů dojde ke zpracování pouze zadaného běhu. Aplikace během svého běhu nezmění soubor se stavem a dále z konfigurace ignoruje nastavené úlohy, které mají být sledovány. Ostatní nastavení zůstávají v platnosti. Dále není kontrolováno, zda běh byl již zpracován. Více o konfiguracích jednotlivých aplikací v kapitolách 4.1.2 a 4.1.3.

Běh aplikací

Hlavní jádro běhu aplikací je u obou implementací stejné. Obecný diagram aktivit je na obrázku 4.1. Po spuštění aplikace dojde nejprve k načtení konfiguračního souboru se stavem. Následuje zpracování argumentů příkazové řádky. Pokud jsou argumenty obsaženy a validní, dojde ke zpracování příslušného běhu definovaného parametry. V případě, že jsou argumenty zadány chybně, dojde k ukončení aplikace a vypsání nápovědy. Pokud během spuštění aplikace nejsou zadány argumenty, budou zpracovány všechny nové běhy sledovaných úloh podle konfigurace. Další informace o běhu aplikace s parametry v kapitole 4.1.1.

Po úspěšném zpracování každého běhu dojde k aktualizaci souboru se stavem. To je z důvodu, aby v případě vyskytnutí chyby ve zpracování dalšího běhu byl uložen stav s posledním úspěšně zpracovaným během.

Zpracování úloh probíhá v pořadí vydefinovaném v konfiguračním souboru. V rámci zpracovávání úlohy je ze serveru získán seznam se soupisem dostupných běhů dané úlohy. Z něj jsou odfiltrovány běhy, které mají identifikátor menší nebo roven než posledně zpracovaný běh. Oba servery (Jenkins a DevOps) používají k identifikaci běhů číselný identifikátor, který je postupně inkrementován. Následně je kolekce seřazena od nejstaršího k nejnovějšímu běhu (ten s nejvyšším identifikátorem). Poté je každý běh z kolekce zpracován. Seznam může obsahovat i běhy, které jsou aktuálně na serveru prováděny. Pokud je takový běh nalezen, je zpracování běhu a celé příslušné úlohy ukončeno a pokračuje se na další úlohu. Tento přístup je zvolen z důvodu, aby byla zajištěna kontinuálnost zpracování úlohy a jejich běhů a také z důvodu, že z probíhajícího běhu nejsou k dispozici všechny potřebné údaje. V případě, že jsou běhy v rámci úlohy spouštěny paralelně a další běh je již dokončen, tak nebude zpracován a k jeho zpracování dojde až po dokončení předchozího běhu na serveru a jeho zpracování aplikací. Příklad běhu aplikace s jejím výpisem je znázorněn ve výpisu 4.4.

```
1  4/26/2022 12:34:10 PM | Processing pipeline: 86
2  4/26/2022 12:34:10 PM | Last buildId: 47229
3  4/26/2022 12:34:10 PM | Found 1 new builds
4  4/26/2022 12:34:10 PM | Processing build: 47230
5  4/26/2022 12:34:10 PM | Pipeline Done
6
7  4/26/2022 12:34:10 PM | Processing pipeline: 85
8  4/26/2022 12:34:10 PM | Last buildId: 47228
9  4/26/2022 12:34:10 PM | Found 0 new builds
10 4/26/2022 12:34:10 PM | Pipeline Done
11
12 4/26/2022 12:34:10 PM | Total build done: 1
```

Výpis 4.4: Ukázka výpisu aplikace

Ověření správnosti řešení

Každá aplikace má svou sadu testů, více budou popsány v kapitolách 4.1.2 a 4.1.3.

Mimo samotné testy bylo řešení nasazeno ve společnosti UNIS, kde byl zahájen provoz řešení během měsíce leden roku 2022. Z počátku byl provoz spíše testovací a docházelo k dokončování implementace a řešení případných problémů. Během této doby bylo řešení prověřeno standardním provozem ve společnosti a sledovalo automatizované procesy během vývoje systému MES PHARIS. Výsledky byly úspěšně předávány do platformy ELK a vizualizovány řešením, které vznikalo souběžně v rámci diplomové práce Aleše Ondráčka [12].

Sestavení a spuštění

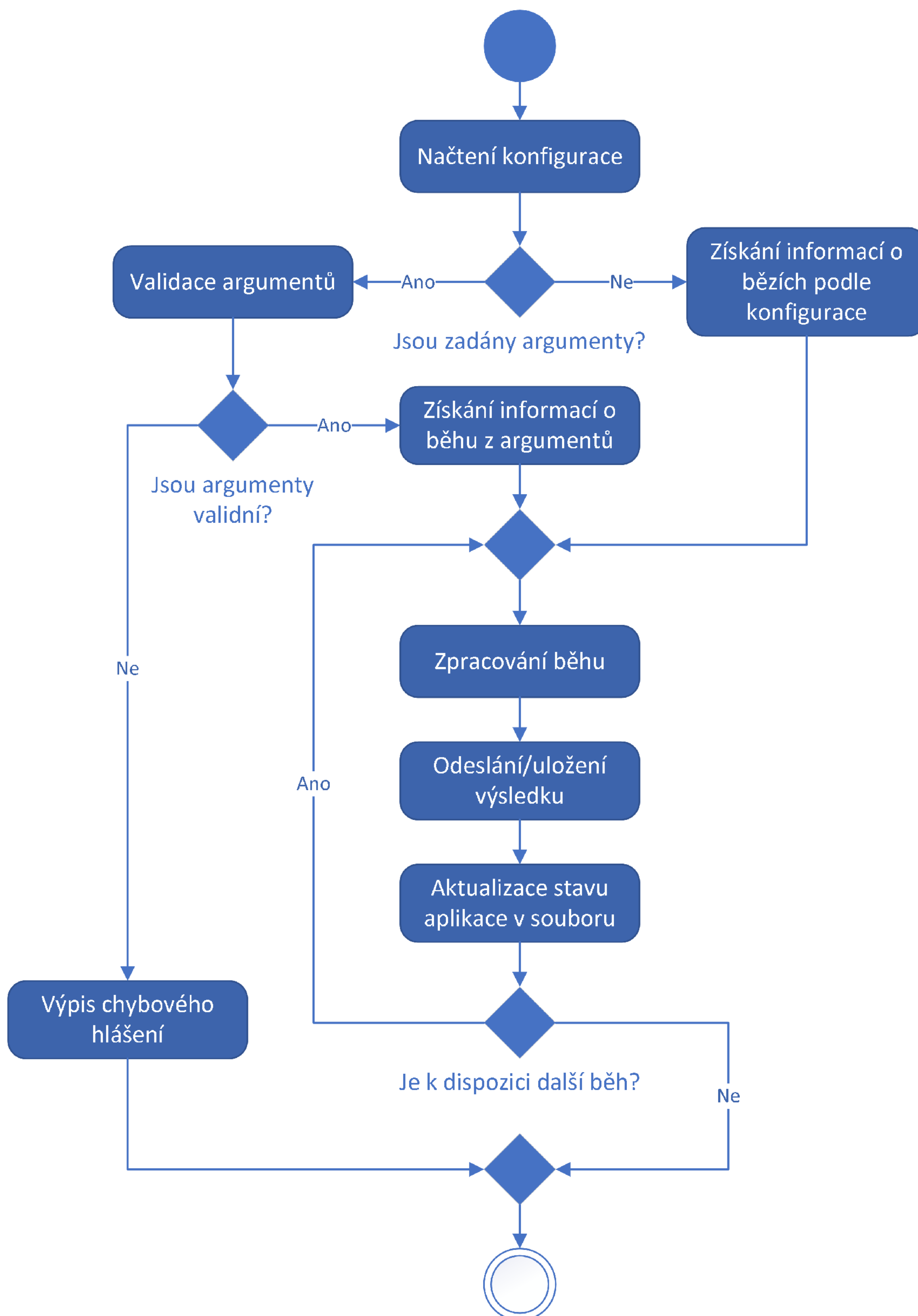
Pro sestavení je nutné mít nainstalován .NET SDK[2] verze 6.0.

Pro spuštění je nutné mít nainstalován balíček .NET SDK nebo .NET Runtime[3] verze 6.0. Pro sestavení je možné využít .NET CLI[4]. Příkazy pro jednotlivé aplikace jsou popsány v kapitolách 4.1.2 a 4.1.3.

[2] .NET SDK – https://docs.microsoft.com/en-us/dotnet/core/sdk

[3] .NET Runtime – https://docs.microsoft.com/en-us/dotnet/core/introduction#sdk-and-runtimes

[4] .NET CLI – https://docs.microsoft.com/en-us/dotnet/core/tools/



Obrázek 4.1: Diagram aktivit běhu aplikací sledující servery

4.1.2 Aplikace pro sledování serveru Jenkins

Aplikace je implementována v rámci projektu `JenkinsJobReader.csproj`. U projektového souboru je i soubor `README.md`, který obsahuje základní popis aplikace, postup sestavení a spuštění, možnosti konfigurace a popis výsledného formátu.

Konfigurace

Tabulka 4.2 obsahuje popis možného nastavení aplikace pomocí souboru `appsetting.json`. Popis obsahuje název klíče, jeho význam a datový typ. V ukázce 4.5 je uveden příklad konfiguračního souboru, který je používán ve společnosti UNIS.

Název	Popis	Datový typ
UrlSchema	Schéma pro komunikaci se serverem Jenkins (http, https)	string
Host	Adresa serveru Jenkins	string
Port	Port serveru Jenkins	int
Pipelines	Kolekce identifikátorů úloh (pipeline), které budou sledovány	List<string>
LogstashHost	Adresa serveru s instancí Logstash	string
LogstashPort	Port, na kterém Logstash přijímá komunikaci TCP	int
SendOutputToLogstash	Příznak, zda má být výsledek uložen do souboru	bool
SaveOutputToFile	Příznak, zda má být výsledek zaslán do Logstash	bool
WriteIndentedOutputToFile	Příznak, zda má být výsledek v souboru formátovaný	bool
ServerIdentification	Identifikace serveru použita ve výsledku v klíči `server`. Viz tabulka 3.4.	string

Tabulka 4.2: Možnosti konfigurace aplikace pro server Jenkins v souboru `appsettings.json`

Konfigurace vyžaduje jak zadání schématu komunikace (http nebo https), tak i port, na kterém tato služba běží. Zadání je nutné i v případě, kdy služby běží na svých výchozích portech (http - 80, https - 443). Tato možnost konfigurace byla zvolena z důvodu, že se často porty mapují na jiný a není možné se řídit pouze číslem portu nebo schématem komunikace.

```
 1 {
 2   "UrlSchema": "http",
 3   "Host": "jenkinsserver",
 4   "Port": 80,
 5   "Pipelines": [
 6     "UNIT_TESTS_COME_DEV",
 7     "UNIT_TESTS_IMPLEMENTATION_DEV",
 8     "TEST-Website_GUI-COME_DEV",
 9     "TEST-Terminal_GUI-COME_DEV",
10     "UNIT_TESTS_COME_MAIN",
11     "UNIT_TESTS_IMPLEMENTATION_MAIN",
12     "TEST-Website_GUI-COME_MAIN",
13     "TEST-Terminal_GUI-COME_MAIN",
14     "UNIT_TESTS_COME_RELEASE",
15     "UNIT_TESTS_IMPLEMENTATION_RELEASE",
16     "TEST-Website_GUI-COME_RELEASE",
17     "TEST-Terminal_GUI-COME_RELEASE"
18   ],
19   "LogstashHost": "12.5.100.183",
20   "LogstashPort": 5000,
21   "SendOutputToLogstash": true,
22   "SaveOutputToFile": false,
23   "WriteIndentedOutputToFile": false,
24   "ServerIdentification": "jenkins"
25 }
```

Výpis 4.5: Příklad konfiguračního souboru v prostředí firmy UNIS (Jenkins)

Zpracování běhu s testy

Pokud mají být u běhu zpracovány testy, které byly provedeny pomocí `vstest.console`, je nutné při spuštění testů použít přepínač `/Logger:trx`. Tím dojde k vytvoření souboru, ve kterém jsou informace o vykonaných testech. V rámci předpisu úlohy na serveru Jenkins je dále nutné udělat krok `archiveArtifacts allowEmptyArchive: true, artifacts: 'Results/*.trx', followSymlinks: false`. Tím bude zajištěno, že v artefaktech běhu bude tento soubor dostupný pro další zpracování. Během běhu aplikace je zkontrolováno, zda se v artefaktech zpracovávaného běhu nalézá souboru typu TRX. Pokud takový soubor existuje, je získán jeho obsah a dále zpracován. V opačném případě je klíč `testsResult` prázdný viz tabulka 3.4.

Pro zpracování souboru je připravena třída `TestResultParser`. Jelikož je vhodné namapovat obsah souboru na instance DTO tříd pro jeho další snazší zpracování, byly tyto třídy vygenerovány automaticky na základě reálného souboru a tím bylo zachyceno jeho schéma. Jejich implementace se nalézá v souboru `TestResultDTO.cs`.

Po provedení deserializace z formátu XML jsou z jednotlivých instancí DTO tříd získány všechny potřebné údaje viz popis formátu 3.6.

Běh aplikace

Obecné chování je popsáno v kapitole 4.1.1. Aplikace během svého běhu využívá několik koncových bodů API serveru Jenkins. Na serveru je nutné mít rozšíření *Pipeline Stage View Plugin* [13]. Jednotlivé koncové body jsou popsány v tabulce 4.3. Z jednotlivých odpovědí jsou získány všechny potřebné informace a ty jsou následně zpracovány do stanoveného formátu viz 3.4.

Adresa koncového bodu	Popis získaných informací
`job/{pipelineId}/api/json`	Vrátí základní informace o dostupných bězích dané úlohy
`job/{pipelineId}/{buildId}/wfapi`	Vrátí informace o běhu pomocí API rozšíření
`job/{pipelineId}/{buildId}/api/json`	Vrátí informace o běhu pomocí standardního API
`job/{pipelineId}/{buildId}/wfapi/artifacts`	Získá kolekci dostupných artefaktů daného běhu
`job/{pipelineId}/{buildId}/artifact/Results/{fileName}`	Získá obsah konkrétního souboru z artefaktů

Tabulka 4.3: Přehled využívaných koncových bodů API (Jenkins)

Spuštění aplikace

Postup sestavení je psán za využití .NET CLI. Příkazy je nutné provolávat ve složce se souborem `JenkinsJobReader.csproj`.

Sestavení aplikace: `dotnet build`

Spuštění aplikace: `dotnet run`

Spuštění aplikace pro konkrétní běh: `dotnet run - -p <identifikátor úlohy> -b <identifikátor běhu>`

Testy aplikace

Pro účely testování byla vytvořena třída `ApiCallerMock`, která má během testu za úkol nahradit volání na API serveru Jenkins. Tato třída implementuje rozhraní `IApiCaller`, které definuje metody, které jsou používány pro komunikaci se serverem. Díky tomu je možné velice snadno vyměnit třídu pro komunikace se serverem. Třída pro účely testování má před spuštěním testu naprogramované návratové hodnoty jednotlivých metod. Tím je snadné simulovat odpovědi ze serveru Jenkins a ověřit tak celé řešení.

Byla implementována sada několika testů, které prověřují funkčnost řešení při různých datech, které by mohl server Jenkins vrátit.

Pro spuštění lze využít .NET CLI a příkaz `dotnet test`. Provedení příkazu je nutné provést ve složce `JenkinsReader.Tests`.

4.1.3 Aplikace pro sledování serveru DevOps

Aplikace je implementována v rámci projektu `DevOpsJobReader.csproj`. U projektového souboru je i soubor `README.md`, který obsahuje základní popis aplikace, postup sestavení a spuštění, možnosti konfigurace a popis výsledného formátu.

Konfigurace

Tabulka 4.4 obsahuje popis možného nastavení aplikace pomocí souboru `appsetting.json`. Popis obsahuje název klíče, jeho význam a datový typ. V ukázce 4.6 je uveden příklad konfiguračního souboru, který je používán ve společnosti UNIS.

Název	**Význam**	**Datový typ**
UrlSchema	Schéma pro komunikaci se serverem DevOps (http, https)	string
Host	Adresa serveru Devops	string
Port	Port serveru DevOps	int
Organization	Organizace v rámci serveru DevOps, která bude sledována	string
Project	Projekt v rámcí organizace na serveru DevOps, který bude sledován	string
ApiVersion	Verze api[5]	string
Pipelines	Kolekce identifikátorů úloh (pipeline), které budou sledovány	List<string>
UseDefaultCredentials	Příznak, zda pro přihlášení k serveru DevOps využít údaje uživatele, který aplikaci spustil	bool
UserName	Uživatelské jméno pro přihlášení k serveru	string
PersonalToken	Osobní token (PAT) pro komunikaci se serverem s oprávněním Build-Read	string
LogStashHost	Adresa serveru s instancí LogStash	string
LogStashPort	Port, na kterém LogStash příjímá komunikaci TCP	int
SendOutputToLogStash	Příznak, zda má být výsledek uložen do souboru	bool
SaveOutputToFile	Příznak, zda má být výsledek zaslán do Logstash	bool
WriteIndentedOutput-ToFile	Příznak, zda má být výsledek v souboru formátovaný	bool
ServerIdentification	Identifikace serveru použita ve výsledku v klíči `server`. Viz 3.8.	string
FilterOnlyTask	Příznak, zda mají být v běhu filtrovány pouze části typu `Task`	bool
RemoveBranchName-InCheckoutTask	Příznak, zda při klonování zdrojových kódů ponechat v názvu název zdrojové větve	bool

Tabulka 4.4: Možnosti konfigurace aplikace pro server DevOps v souboru `appsettings.json`

Obdobně jako aplikace pro server Jenkins konfigurace vyžaduje jak zadání schématu komunikace (http nebo https), tak i port, na kterém tato služba běží. Zadání je nutné i v případě, kdy služby běží na svých výchozích portech (http - 80, https - 443). Tato možnost konfigurace byla zvolena z důvodu, že se často porty mapují na jiný a není možné se řídit pouze číslem portu nebo schématem komunikace.

```
1  {
2    "UrlSchema": "https",
3    "Host": "devopsserver",
4    "Port": 8080,
5    "Organization": "Pharis",
6    "Project": "Come",
7    "ApiVersion": "5.1-preview",
8    "Pipelines": [ "86", "85" ],
9    "UseDefaultCredentials": true,
10   "UserName": "",
11   "PersonalToken": "",
12   "LogstashHost": "12.5.100.183",
13   "LogstashPort": 5000,
14   "SendOutputToLogstash": true,
15   "SaveOutputToFile": false,
16   "WriteIndentedOutputToFile": false,
17   "ServerIdentification": "devops",
18   "FilterOnlyTask": true,
19   "RemoveBranchNameInCheckoutTask": true
20 }
```

Výpis 4.6: Příklad konfiguračního souboru v prostředí firmy UNIS (DevOps)

Osobní přístupový token (PAT)

Pro přístup na server DevOps lze využít osobní přístupový token (z anglického Personal Access Token - PAT[6]). Díky použití PAT není nutné do konfiguračního souboru zadávat osobní heslo, ale pouze uživatelské jméno a PAT. PAT jsou na ochranu stejně kritické jako hesla, takže s nimi musí být tak nakládáno. Při vytváření PAT je možné nastavit rozsah oprávnění pro daný PAT. Pro účel této aplikace je dostačující udělení oprávnění pouze ze sekce *Build* s rozsahem pouze pro čtení *Read*. V případě narušení bezpečnosti a úniku konfiguračního souboru toto nastavení zajistí minimální nebezpečí pro uživatele, jelikož pomocí PAT bude přístup pouze k informacím o sestavení a pouze ve formě čtení, ostatní oblasti jsou mimo oprávnění PAT. Dále je možné PAT ihned deaktivovat a tím zamezit neoprávněnému přístupu pomocí kompromitovaného PAT.

Testování probíhalo na verzi DevOps 2020 hostované firmou Microsoft pomocí řešení Azure, kde byla ověřen funkčnost řešení s použitím PAT a nastavenými přístupy popsanými výše.

[5] Verze API serveru DevOps – https://docs.microsoft.com/en-us/azure/devops/integrate/concepts/rest-api-versioning?view=azure-devops

[6] Osobní přistupový token – https://docs.microsoft.com/en-us/azure/devops/organizations/accounts/use-personal-access-tokens-to-authenticate

Měření obsahu jednotlivých částí systému MES PHARIS

V rámci běhu úlohy kontinuální integrace dochází k sestavení aplikace a vytvoření nuget balíčků jednotlivých částí systému (web, ES, terminálová aplikace). Jelikož by po zabalení do nuget balíčku bylo nutné jej znovu rozbalit za účelem získání počtu souborů a z důvodu, že zabalení s sebou přináší jistou úroveň komprese a tím ovlivňuje výsledné měření velikosti, je nutné tyto informace zjistit přímo v běhu dané úlohy. Pro zjištění potřebných informací byla úloha rozšířena o další krok, který provede skript v prostředí Powershell. Krok byl pojmenován `Measure of packages content`.

V rámci daného kroku se provede daný skript, který výsledek ve formátu JSON zapíše do svého výstupu. Při běhu aplikace jsou kontrolovány názvy jednotlivých kroků a pokud je nalezen krok s odpovídajícím názvem, je získán výstup jeho průběhu, který je zpracován a načten do interní reprezentace k dalšímu zpracování. Pokud daný krok není nalezen, je klíč `packagesContentInfo` ve výsledku prázdný, viz 3.8.

V ukázce 4.8 je implementace skriptu, který se stará o zjištění velikostí jednotlivých balíčků. Výstup ze skriptu je poté v ukázce 4.7. Implementace zjistí potřebné informace o balíčku a to celkovou velikost adresáře, počet souborů, počet složek a počet všech prvků v balíčku.

Implementace může být i jiná, ale je nutné dodržet pojmenování kroku v rámci úlohy a formát, který je vypsán do výstupu kroku. Formát výsledku má podobu objektu, kde jako klíče jsou použity názvy balíčků a hodnotou je objekt, který nese informace o balíčku viz ukázka 4.7.

```
1  {
2      "Phoenix.Phar.EventSystem.Package":
3      {
4           "dirs": 22,
5           "totalItems": 597,
6           "files": 575,
7           "size": 208017775
8       },
9      "Phoenix.Phar.Terminal.Package":
10     {
11           "dirs": 9,
12           "totalItems": 81,
13           "files": 72,
14           "size": 35977785
15       },
16     "Phoenix.Phar.Web.Package":
17     {
18          "dirs": 372,
19          "totalItems": 5412,
20          "files": 5040,
21          "size": 269079712
22      }
23 }
```

Výpis 4.7: Příklad informací o jednotlivých balíčcích v prostředí firmy UNIS

Díky použití dynamického objektu mohou být v rámci běhu úlohy zjištěny informace i o více balíčcích. Interní reprezentace ve výsledném kódu je pomocí slovníku `Dictionary<string, PackageContentDTO>`, kde jako klíč slouží název balíčku a hodnotou je třída nesoucí podrobné informace. Popis třídy viz 3.9.

```
1  $packages = @{
2      "$(PharEventSystemPackageName)" = "$(BuildBinariesEventSystemPackage)"
3      "$(PharTerminalPackageName)" = "$(BuildBinariesTerminalPackage)"
4      "$(PharWebPackageName)" = "$(BuildBinariesPublishedWeb)"
5  }
6
7  function GetFolderInfo{
8      param($directory)
9
10     $dirCount = (Get-ChildItem $directory
11         -Force -Recurse -Directory | Measure-Object).Count
12     $fileCount = (Get-ChildItem $directory
13         -Force -Recurse -File | Measure-Object).Count
14     $totalCount = $dirCount + $fileCount
15
16     $totalSize = (Get-ChildItem $directory
17         -Force -Recurse | Measure-Object -Property Length -sum).Sum
18
19     $result = @{
20         "dirs" = $dirCount
21         "files" = $fileCount
22         "totalItems" = $totalCount
23         "size" = $totalSize
24     }
25
26     return $result
27 }
28
29 function GetPackagesInfo{
30     $result = @{}
31
32     foreach($package in $packages.GetEnumerator()){
33         $result[$package.name] = GetFolderInfo $package.value
34     }
35
36     return $result
37 }
38
39 GetPackagesInfo | ConvertTo-Json -Depth 10 | Write-Output
```

Výpis 4.8: Kód pro zjištění obsahu balíčků

Běh aplikace

Obecné chování je popsáno v kapitole 4.1.1. Aplikace během svého běhu využívá několik koncových bodů API serveru DevOps. Ty jsou popsány v tabulce 4.5. Z jednotlivých odpovědí jsou získány všechny potřebné informace a ty jsou následně zpracovány do stanoveného formátu viz 3.8.

Adresa koncového bodu	Popis získaných informací
`_apis/pipelines/{pipelineId}/runs`	Vrátí základní informace o posledních 10000 běhů úlohy
`_apis/pipelines/{pipelineId}/runs/{buildId}`	Vrátí základní informace o konkrétním běhu
`_apis/build/builds/{buildId}/timeline`	Vrátí detailní informace o jednotlivých částech úlohy
`_apis/build/builds/{buildId}`	Vrátí detailní informace o konkrétním běhu
`_apis/build/builds/{buildId}/logs/{logId}`	Vrátí obsah výstupu konkrétního kroku běhu

Tabulka 4.5: Přehled využívaných koncových bodů API (DevOps)

Spuštění aplikace

Postup sestavení je psán za využití .NET CLI. Příkazy je nutné provolávat ve složce se souborem `DevOpsJobReader.csproj`.

Sestavení aplikace: `dotnet build`

Spuštění aplikace: `dotnet run`

Spuštění aplikace pro konkrétní běh: `dotnet run - -p <identifikátor úlohy> -b <identifikátor běhu>`

Testy aplikace

Pro účely testování byla vytvořena třída `ApiCallerMock`, která má během testu za úkol nahradit volání na API serveru DevOps. Tato třída implementuje rozhraní `IApiCaller`, které definuje metody, které jsou používány pro komunikaci se serverem. Díky tomu je možné velice snadno vyměnit třídu pro komunikace se serverem. Třída pro účely testování má před spuštěním testu naprogramované návratové hodnoty jednotlivých metod. Tím je možné velice snadno simulovat odpovědi ze serveru DevOps a ověřit tak celé řešení.

Byla implementována sada několika testů, které prověřují funkčnost řešení při různých datech, které by mohl server DevOps vrátit.

Pro spuštění lze využít .NET CLI a příkaz `dotnet test`. Provedení příkazu je nutné provést ve složce `DevOpsReader.Tests`.

4.2 Implementace výkonnostních testů

Podle návrhu popsaného v kapitole 3.2 bylo vytvořeno řešení, které řeší požadavky firmy UNIS na provádění výkonnostního testování. Díky celkovému návrhu je řešení možné využít i mimo prostředí firmy UNIS a jejího produktu MES PHARIS, který je testování podroben.

Celkové řešení zahrnuje obecný předpis scénáře, jedná se o jednotku, která bude vykonávána paralelně. Dále aplikace, která se stará o spuštění scénáře. Tím je zajištěno, že je každý scénář spuštěn v samostatném procesu a jednotlivé scénáře tak mají vlastní programový prostor. Aby byl uživatel odstíněn od přímého spouštění aplikace, vznikla třída, která zapouzdřuje spouštění aplikace pro jednotlivé scénáře, v textu je tato třída dále označována jako orchestrátor. Sběr výkonnostních metrik zajišťuje aplikace Metricbeat a sběr záznamů o události z jednotlivých scénářů zajišťuje aplikace Filebeat.

Řešení je implementováno pomocí platformy .NET framework 4.8. Tato verze byla zvolena z důvodu, že řešení firmy UNIS je implementováno pomocí této platformy a také z důvodu minimalizace případných problémů v referencování. Výsledná implementace a použité knihovny umožňují konverzi na nejnovější platformu .NET 6. Pro ověření kompatibility byl využit nástroj `.NET Portability Analyzer`[7].

4.2.1 Rozhraní scénáře

Pro zajištění jednotného přístupu ke scénářům bez znalosti jejich výsledné implementace bylo vytvořeno rozhraní `IScenario`, které je implementováno v knihovně `ScenarioBase`. Z tohoto rozhraní musí výsledné implementace scénářů vycházet. Rozhraní vyžaduje implementaci metody `Init`, která je provedena ve všech scénářích a čeká se na její dokončení před dalším pokračováním. Metoda akceptuje jeden textový argument, který slouží pro předání nutných parametrů až za samotného běhu scénáře a tím parametricky ovlivnit jeho chování. Způsob předání bude dále popsán u popisu orchestrátoru scénářů. Metoda má návratovou hodnotu pravdivostního typu, která označuje, zda byla metoda úspěšná. Uvnitř této metody je očekáváno provedení všech nutných příprav a inicializací pro samotný běh scénáře. Pokud tato fáze není scénářem vyžadována, metoda musí být implementována a stačí v ní pouze provolat následující příkaz `return true;`. Pokud by bylo řešení převedeno na platformu .NET 6, bylo by možné využít funkcionalitu z jazyka C# verze 8. Nová funkcionalita umožňuje implementovat funkcionalitu metody již v definici rozhraní a tím by nebylo nutné provádět implementaci, pokud není metoda `Init` vyžadovaná.

Další metodou je metoda `RunScenario`, ve které bude proveden vlastní běh scénáře. Metoda má opět návratovou hodnotu pravdivostního typu, která značí, zda byl scénář úspěšný po logické stránce.

Rozhraní má dále několik vlastností. Jedná se o vlastnost `Id`, která je číselného typu a má za úkol jednoznačnou identifikaci scénáře. Vlastnost se uplatňuje při zápisu události o průběhu, aby bylo možné spárovat jednotlivé záznamy s konkrétním scénářem. Další vlastností je `TestName` textového typu, která nese název testu včetně třídy, ve které je implementován. Tím je možné od sebe odlišit jednotlivé záznamy o události podle jednotlivých testů a určit, v rámci kterého testu daný scénář běžel. Následuje vlastnost `RunnerStartTime`, která

[7].NET Portability Analyzer – `https://docs.microsoft.com/en-us/dotnet/standard/analyzers/portability-analyzer`

v textové podobě uchovává start celého testovacího scénáře, slouží k odlišení jednotlivých běhů stejného testu. Formát hodnoty je následující: `yyMMddHHmmss`. Poslední vlastností je `Logger` typu `ILogger`. Jde o instanci, která umožňuje zapisovat informace v průběhu běhu scénáře. Další informace v následující sekci.

4.2.2 Zapisování záznamů o události v průběhu scénáře

Pro účely zápisu informací během běhu scénáře je připravené rozhraní `ILogger`. Z tohoto rozhraní musí vycházet všechny implementace pro zápis záznamů. Rozhraní obsahuje vlastnost `LogLevel` typu výčet `ELogLevel` viz 3.2. Určuje, od které úrovně včetně mají být zprávy zapisovány. Tím mohou být například pomocné záznamy snadno vypnuty nebo zapnuty v případě ladění scénáře. Dále rozhraní vyžaduje implementaci metody `Log`, kde jako parametr má být předána instance třídy `Log` viz 3.1. Tím je zaručeno zachování formátu záznamu i v případě změny implementace pro zápis záznamů.

S využitím rozhraní byla implementována dvě řešení pro zápis záznamů o události. Prvním je třída `ConsoleLogger`, která provádí výpis do konzole a podle úrovně události je záznam jinak obarven. Červená - úroveň `Error`, žlutá - úroveň `Warning`, šedá - úroveň `Debug`, výchozí barva - úroveň `Info`.

Z důvodu potřeby zápisu do centrálního souboru byla vytvořena třída `OneFileLogger`. Ta zajišťuje zápis ze všech scénářů do společného souboru. Jelikož se jednotlivé scénáře spouštějí v samostatných procesech, je nutné provést synchronizaci v přístupu ke sdílenému souboru pro zápis. To je zajištěno pomocí synchronizačního primitiva typu mutex, který zaštiťuje třída `Mutex`. Během vytváření instance třídy pro zápis záznamů je vytvořena instance mutexu, který je pojmenován. Tím je možné získat v každé instanci třídy stejný mutex a provádět tak zápis do souboru v rámci kritické sekce. Pro pojmenování mutexu byl použit následující text: `OneFileLogger_86392801-e04f-4c70-ad59-80447328e6ae`. Jde o název třídy a náhodně vygenerovaný `GUID`. Tím je z názvu mutexu patrné k čemu slouží a část `GUID` slouží k minimalizaci konfliktu v názvu mutexů.

Instance třídy `Log` je pro zápis serializována do formátu JSON. Metoda pro převod instance je implementována přímo ve třídě `Log`.

4.2.3 Spouštěč scénáře

Pro spuštění scénáře byla vytvořena samotná konzolová aplikace. Diagram popisující činnost spouštěče scénářů je na obrázku 4.3. Implementace se nalézá v `ScenarioRunner.csproj`. Ve výchozím stavu aplikace očekává, že se vše zapisuje do společného souboru. K pojmenování souboru se využije hodnota z argumentu `startDate` a soubor bude umístěn do podsložky `logs`. Aplikace po svém spuštění zpracuje zadané argumenty a zvaliduje jejich správnost. Podporované parametry s jejich popisem a přepínači se nacházejí v tabulce 4.6. Pokud je při zpracování argumentů nalezena chyba, není možné se spolehnout na obsah argumentu `startDate`. Z tohoto důvodu je nejprve prohledán adresář `logs` a nalezen nejnovější soubor, který není starší deseti sekund. Pokud je takový soubor nalezen, je považován za soubor, do kterého se aktuálně zapisuje prováděný testovací scénář. Existenci souboru zajišťuje orchestrátor, který před spuštěním jednotlivých scénářů provede jeho vytvoření a zápis prvního záznamu o události. Popis orchestrátoru se nalézá v další kapitole. Pokud není soubor nalezen, je vytvořen nový s formátem: `yyMMddHHmmss_runner_pid.txt`. Do souboru

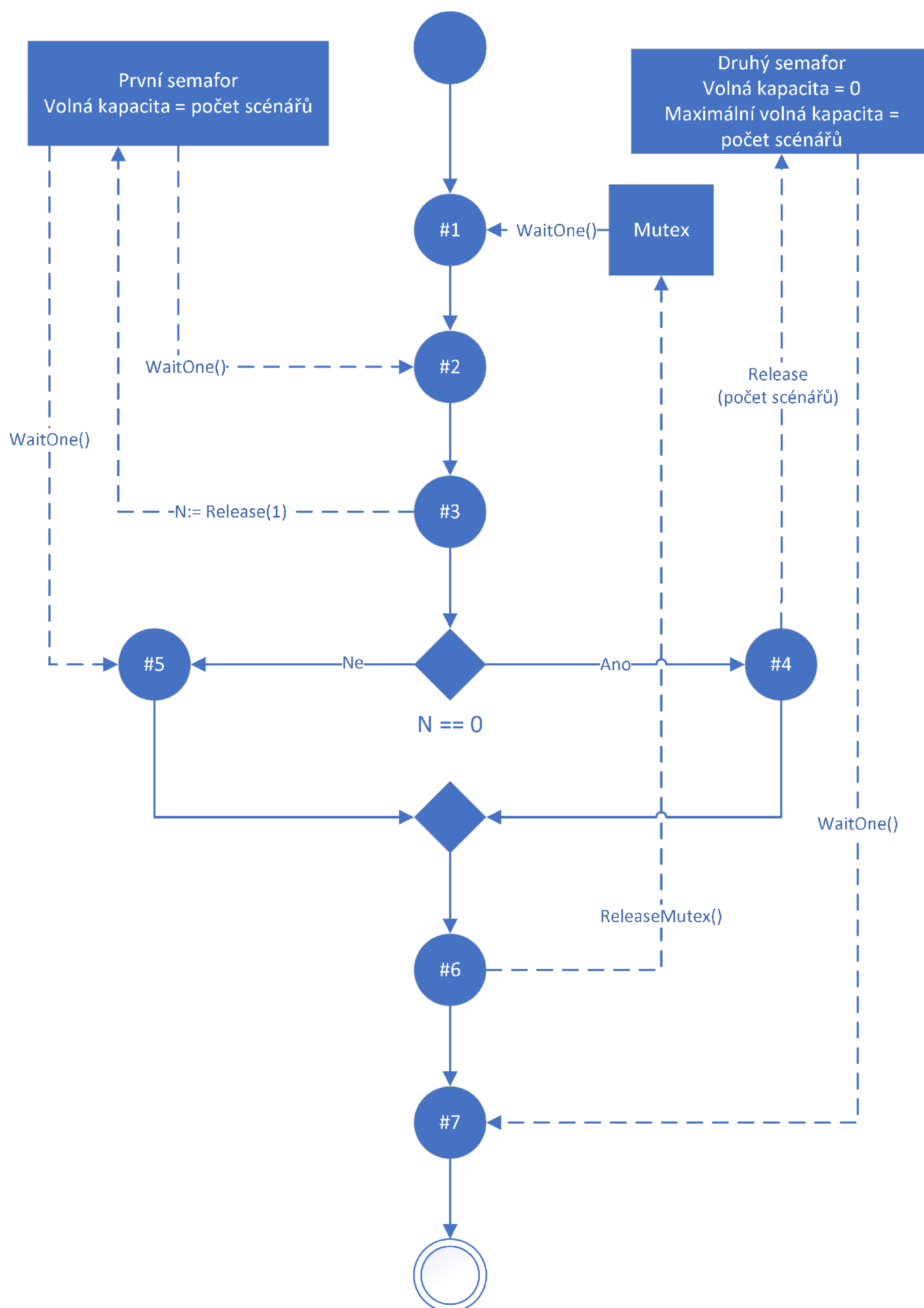
je poté zapsána chybová hláška ve standardizovaném záznamu. Aplikace po svém spuštění přesměruje standardní výstup a standardní chybový výstup. Pokud obsahují nějaký text, je jejich obsah zapsán ve formátu třídy `Log`. Chybový výstup má úroveň `Error` a standardní výstup má úroveň `Info`.

Pokud jsou argumenty validní, dojde k načtení DLL knihovny z cesty předané v argumentu `library`. V této knihovně je následně hledána třída odpovídající názvu z parametru `scenario`. Tato třída dále musí implementovat rozhraní `IScenario`. Pokud není třída nalezena, aplikace ukončí svůj běh chybou. V opačném případě je vytvořena instance nalezené třídy. Instanci jsou dále nastaveny vlastnosti podle předaných argumentů. Dále je vytvořena instance pro zapisování záznamů do společného souboru a tato instance je rovněž předána do instance scénáře.

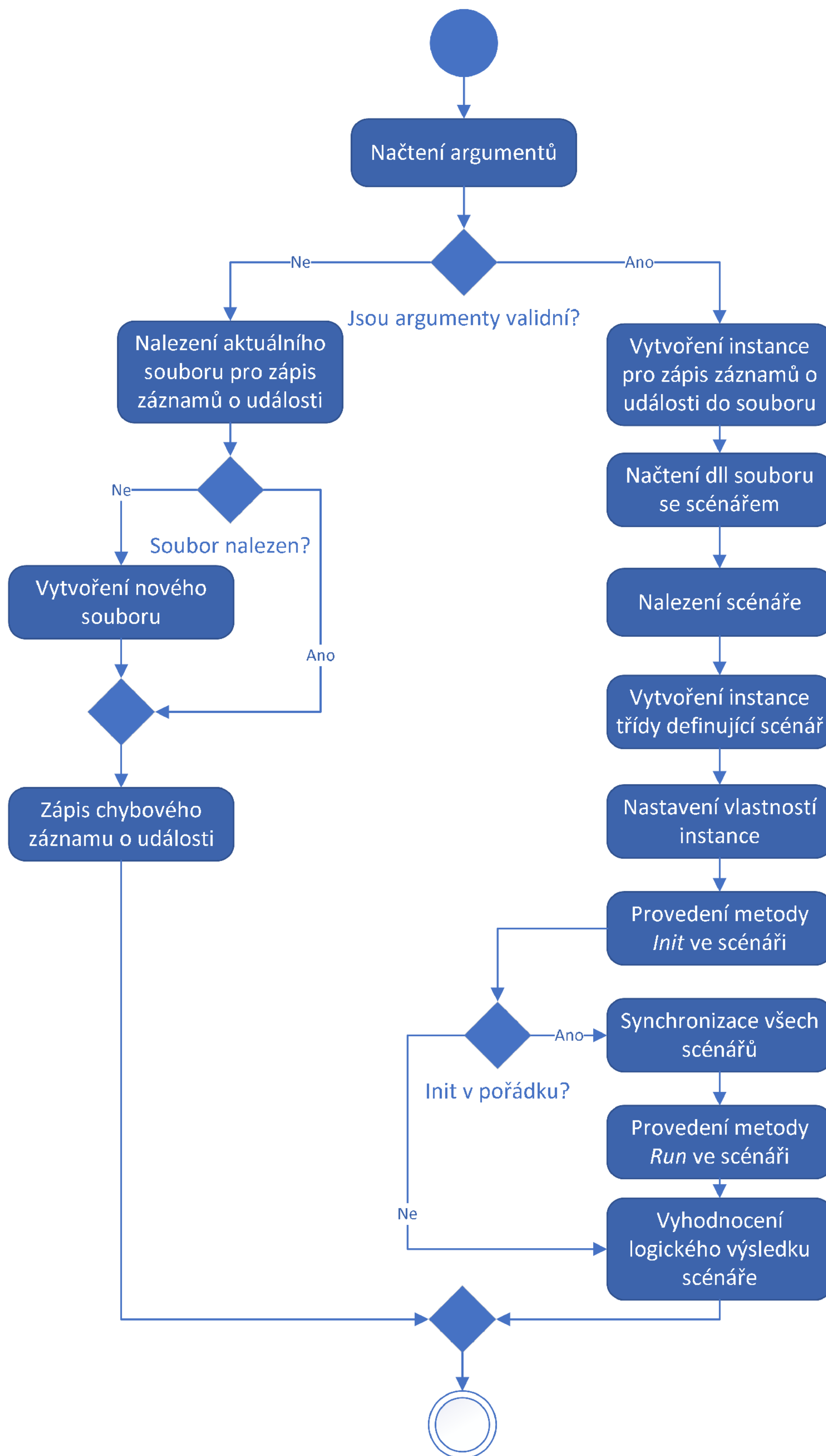
Následuje spuštění metody `Init` a jako parametr této metody je předán obsah argumentu `initArgs`. Dále je provedeno vyhodnocení výsledku metody a pokud metoda selhala, je celá aplikace ukončena chybovým stavem. V opačném případě je zahájena synchronizační fáze, kdy všechny scénáře čekají na dokončení metody `Init` u ostatních scénářů.

Synchronizace je řešena pomocí dvou semaforů a jednoho mutexu. Diagram synchronizace se nalézá na obrázku 4.2. Všechny synchronizační primitiva jsou pojmenovaná a proto je lze kdykoliv získat. Pro pojmenování je použit prefix podle účelu a suffix je obsah parametru `startDate`. Tím je zaručen vznik unikátních instancí pro každý test. Odpovědnost za vytvoření těchto primitiv má orchestrátor. V případě, že primitiva nejsou nalezena, pokračuje aplikace dále bez této fáze. Toto řešení bylo aplikováno z důvodu ladění a přímého spuštění aplikace. Jelikož orchestrátor ví celkový počet scénářů, vytvoří jeden semafor s kapacitou rovnající se celkovému počtu souběžných scénářů. Dále vytvoří druhý semafor, který má volnou kapacitu nula a maximální kapacitu rovnající se počtu všech scénářů. Aplikace v synchronizační fázi zamče mutex, aby s primitivy pracovala jen jedna instance aplikace, a dále zamče první semafor. Jelikož u pojmenovaného semaforu není možné přímo zjistit jeho volnou kapacitu, provede aplikace uvolnění u prvního semaforu. Při této operaci je vrácena volná kapacita před uvolněním. Pokud byla volná kapacita rovna nule, znamená to, že jde o poslední scénář, který musí uvolnit ostatní (posloupnost 1, 2, 3, 4 v diagramu 4.2). V takovém případě nastaví volnou kapacitu u druhého semaforu na celkový počet scénářů (posloupnost 1, 2, 3, 5 v diagramu 4.2). V opačném případě znovu uzamče první semafor, aby došlo k nastavení správné hodnoty volné kapacity. Následuje uvolnění mutexu a čekání na druhém semaforu, než jej poslední scénář uvolní. Tímto řešením je zajištěno, že všechny scénáře nejprve vykonají metodu `Init` a až poté pokračují do metody `Run`.

Následuje samotný běh scénáře. Po skončení metody je ověřen logický výsledek podle návratové hodnoty a určen celkový logický výsledek scénáře. Podle tohoto výsledku je aplikace ukončena s příslušným návratovým kódem.



Obrázek 4.2: Diagram synchronizace procesů spouštěče scénáře



Obrázek 4.3: Diagram aktivit spouštěče scénáře

Popis	Krátký formát	Dlouhý formát	Datový typ
Úplná cesta k souboru s implementací scénáře (dll knihovna)	-l	--library	string
Název třídy scénáře, který bude vykonán	-s	--scenario	string
Celkový počet scénářů v rámci testu	-c	--count	int
Identifikace aktuálního scénáře	-i	--id	string
Text, který bude využit jako parametr ve volání metody Init	-a	--initArgs	string
Název testu	-t	--testName	string
Text s reprezentací začátku testovacího scénáře	-d	--startDate	string

Tabulka 4.6: Argumenty programu pro spuštění scénáře

Argument `initArgs` není povinný, ostatní ano.

4.2.4 Orchestrátor

Z důvodu zapouzdření spouštění jednotlivých scénářů a sledování jednotlivých procesů, byla vytvořena třída `Runner` v knihovně `ScenarioBase`.

Konstruktor přijímá dva parametry, prvním je název knihovny, kde jsou scénáře implementovány, název musí obsahovat i typ souboru `.dll`. Cesta ke knihovně je ve výchozím stavu hledána ve stejném adresáři, kde se nalézá knihovna s implementací orchestrátoru. Cestu je možné změnit podle potřeby prostřednictvím vlastnosti `PathToScenariosDll`. Výslednou cestu lze vidět prostřednictvím vlastnosti `ScenariosDllPath`. Tato vlastnost je pouze pro čtení a nelze ji měnit. Druhým parametrem je název testovacího scénáře.

Orchestrátor dále předpokládá, že program pro spuštění scénáře je ve stejném adresáři s výchozím názvem `ScenarioRunner.exe`. Obě vlastnosti lze změnit prostřednictvím vlastností `RunnerName` a `PathToRunner`. Výslednou cestu lze vidět prostřednictvím vlastnosti `RunnerPath`. Tato vlastnost je pouze pro čtení a nelze ji měnit.

Pro zaplánování scénáře je připravena metoda `AddScenario`. Tato metoda je přetížena a má celkem tři variace. Možnosti volání jsou následující:

```
AddScenario(string scenarioName, int scenarioId)
AddScenario(string scenarioName, string initArgs)
AddScenario(string scenarioName, int scenarioId, string initArgs)
```

Všechna volání vyžadují jako první parametr název třídy, která obsahuje implementaci scénáře, který má být proveden. Pro získání názvu třídy je doporučeno následující volání `nameof(<třída>)`[8]. Pomocí tohoto volání je získán název scénáře a také je vytvořena reference v kódu. Pokud dojde k přejmenování třídy nebo k jejímu smazání, bude změna odhalena při sestavení kódu a ne až v případě běhu testovacího scénáře. Dále je možné do parametru zadat přímo textovou reprezentaci názvu scénáře. V případě, že je název zadán chybně, bude tato chyba odhalena až při spuštění aplikace pro spuštění scénáře a chybnému pokusu o nalezení scénáře podle zadaného jména.

[8] Volání `nameof` – https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof

Dalším parametrem je identifikátor konkrétního scénáře. Jedná se o celočíselnou hodnotu. Očekává se zadání kladného čísla. Pokud je využito volání bez tohoto parametru, dojde k přidělení automatického identifikátoru, který se automaticky inkrementuje a začíná od čísla jedna. Pokud je identifikátor zadán, případně je využita kombinace ručního zadání s automatickým, není případná kolize identifikátorů sledována nebo signalizována. V případě identifikace orchestrátoru je k identifikaci využito číslo mínus jedna a jako název `ScenarioName` je `RunnerOrchestrator`.

Posledním parametrem je textový řetězec s argumenty pro metodu `Init` v rámci scénáře. Pokud není využit, dojde k předání hodnoty `String.Empty`. Text může obsahovat jakékoliv znaky podporované konzolovou aplikací. Pro předání více informací v rámci jednoho argumentu může být využito vhodného oddělovače, například mezera nebo středník. Zpracování a rozdělení na jednotlivé argumenty je už na vlastní implementaci v rámci metody `Init`.

Orchestrátor během svého běhu provede zápis dvou záznamů o události. Formát záznamu odpovídá formátu, který je popsán v tabulce 3.1. První záznam obsahuje v klíči `Message` pouze položky `Text, TotalRunner a StartTime` z tabulky 4.7, ukázka je ve výpisu 4.9. Tento záznam je zapsán na začátku testovacího scénáře. Tím dojde k vytvoření souboru pro zápis záznamů, který je případně hledán programem pro spouštění scénářů. Druhý záznam obsahuje v klíči `Message` všechny položky podle tabulky 4.7 a ukázka je ve výpisu 4.10.

Název	**Popis**	**Datový typ**
Text	Text zprávy	string
TestResult	Výsledek testu. Podle následujícího seznamu: SUCCEEDED, TIMEOUTED, FAILED	string
ExitCodes	Mapování identifikátoru scénáře na ukončovací kód spouštěče scénářů	Dictionary <int,int>
Killed	Mapování identifikátoru scénáře na příznak, zda byl program spouštějící scénáře ukončen voláním Kill	Dictionary <int,bool>
PidMap	Mapování identifikátoru scénáře na identifikátor procesu spouštějící daný scénář	Dictionary <int,int>
TotalRunner	Celkový počet scénářů v rámci testovacího scénáře	int
StartTime	Čas začátku testovacího scénáře	DateTime
EndTime	Čas konce testovacího scénáře	DateTime
Duration	Čas běhu testovacího scénáře v milisekundách. Rozdíl mezi EndTime a StartTime	double

Tabulka 4.7: Formát klíče Message v případě záznamu o události z orchestrátoru

```
1 {
2     "Text":"Test started",
3     "TotalRunner":2,
4     "StartTime":"2022-04-30T01:52:16.4635484+02:00"
5 }
```

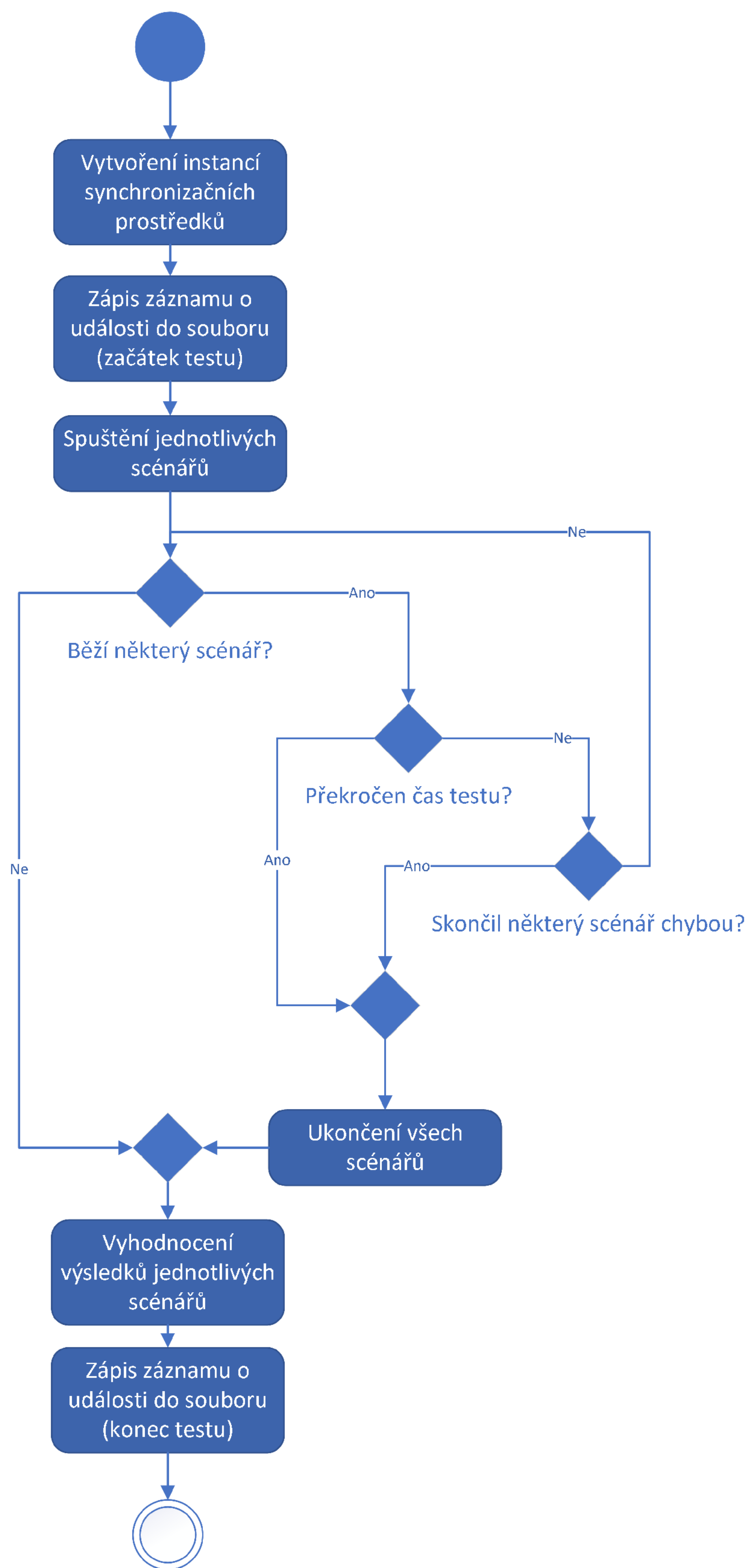
Výpis 4.9: Příklad klíče Message v úvodní zprávě z orchestrátoru

```
1  {
2      "Text":"Test completed",
3      "TestResult":"SUCCEEDED",
4      "ExitCodes":{"1":0, "2":0},
5      "Killed":{"1":false, "2":false},
6      "PidMap":{"1":6812, "2":3480},
7      "TotalRunner":2,
8      "StartTime":"2022-04-30T01:52:16.4635484+02:00",
9      "EndTime":"2022-04-30T01:52:56.5270488+02:00",
10     "Duration":40063.500400000004
11 },
```

Výpis 4.10: Příklad klíče Message v koncové zprávě z orchestrátoru

Po přidání všech scénářů, které mají být v rámci jednoho testovacího scénáře spuštěny, následuje provolání metody `Run`. Té může být předána jako parametr celočíselná hodnota, reprezentující maximální přípustnou dobu, po kterou scénář poběží. Tuto dobu lze také nastavit pomocí vlastnosti `TimeoutInMinute`. Výchozí hodnotou je deset minut. Metoda má návratovou hodnotu typu `bool`, která reprezentuje logický výsledek testu. Ten je stanoven na základě toho, že všechny aplikace spouštějící scénář byly ukončeny s návratovým kódem nula a nebyl překročen čas na testovací scénář.

Diagram aktivity třídy po zahájení testovacího scénáře je popsán na obrázku 4.4. V rámci metody `Run` dochází k inicializaci synchronizačních primitiv pro potřeby běhu aplikace spouštějící scénáře. Popis synchronizace je v rámci kapitoly 4.2.3. Následně jsou vytvořeny a spuštěny jednotlivé procesy pro spouštěč scénáře pro každý požadovaný scénář. Ty jsou sledovány v pravidelném intervalu a to deset sekund. V rámci kontroly je nejprve zjištěno, zda některá aplikace neskončila s chybou. Pokud se tak stane, jsou ostatní procesy ukončeny. Dále je ověřeno, zda celková doba běhu nepřekročila maximální dobu běhu testovacího scénáře. Pokud je to pravda, jsou všechny procesy ukončeny. Tato kontrola probíhá do ukončení všech procesů. Po dokončení běhu všech procesů je vyhodnocen výsledek a proveden zápis příslušného záznamu.



Obrázek 4.4: Diagram aktivit orchestrátoru

4.2.5 Nastavení aplikací Beat

Pro správný běh vizualizace a potřeby dalšího zpracování je zapotřebí nakonfigurovat aplikace Metricbeat[9] a Filebeat[10]. Samotné aplikace byly staženy a na stroji zaregistrovány jako služby.

Metricbeat

Tato aplikace slouží pro sběr výkonnostních metrik sledovaného stroje. Pro sledování všech potřebných informací stačí aktivovaný modul `system.yml` s nastavením podle ukázky 4.11. Toto nastavení sleduje vybrané parametry a samotné aplikace spouštějící jednotlivé scénáře v intervalu jedné sekundy a informace přeposílá k dalšímu zpracování do Logstash. K tomu je nutné v rámci samotné konfigurace aplikace nastavit klíč `output.logstash.hosts`, kde je definována adresa a port, kde instance Logstash běží. Detailnější dokumentace poskytovaných hodnot a samotného modulu `system` je dostupná přímo na stránkách aplikace[11].

```
1  - module: system
2    period: 1s
3    metricsets:
4      - cpu
5      - memory
6      - network
7      - process
8      - process_summary
9      - socket_summary
10     - diskio
11   processes: ['.*Scenario.*']
12
13 - module: system
14   period: 1s
15   metricsets:
16     - filesystem
17   processors:
18   - drop_event.when.regexp:
19       system.filesystem.mount_point:
20         '^/(sys|cgroup|proc|dev|etc|host|lib|snap)($|/)'
```

Výpis 4.11: Příklad doplněných položek v rámci klíče processors v konfiguraci

Filebeat

Aplikace slouží ke sledování souborů na stroji, kde běží testovací scénáře a kde dochází k zápisu záznamů o události z jejich průběhu. Následně dochází k přeposílání jejich obsahu. V rámci konfigurace nebyl aktivován žádný další modul. Jelikož je další zpracování

[9] Metricbeat – https://www.elastic.co/beats/metricbeat
[10] Filebeat – https://www.elastic.co/beats/filebeat
[11] Dokumentace modulu system aplikace Metricbeat – https://www.elastic.co/guide/en/beats/metricbeat/master/metricbeat-module-system.html

prováděno pomocí `Logstash`, musí být výstup přesměrován na příslušnou adresu a port. K tomu slouží v konfiguraci klíč `output.logstash.hosts`. Klíč `processors` je nutné doplnit o další položky, které jsou popsány v ukázce 4.12. Těmito úpravami dojde ke zpracování klíče `message` ze záznamu (viz tabulka 3.1) a jeho obsah je přímo dostupný v rámci zprávy. Tím byl také eliminován problém se zplošťováním vnořených objektů, na který narazil Aleš Ondráček při zpracovávání dat [12]. Dále je zprávě nastaven čas vzniku na skutečnou hodnotu zápisu záznamu. Jinak je zprávě nastaven čas, kdy byla zpracována aplikací Filebeat. Jelikož aplikace pracuje periodicky, byl by tento čas ovlivněn a neodpovídal by správnému času, navíc by mohlo docházet i k předbíhání jednotlivých záznamů.

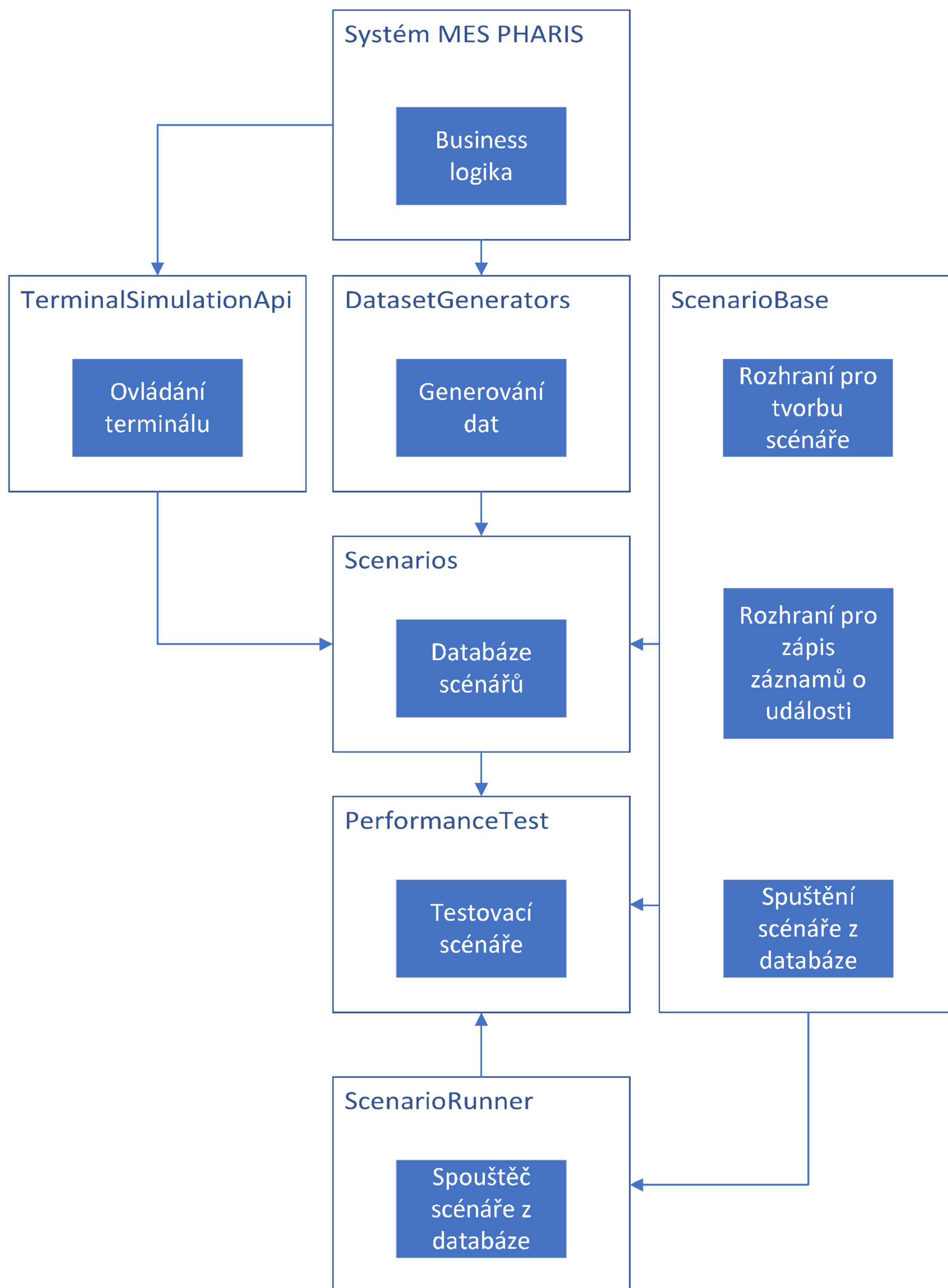
```
1  {
2  - decode_json_fields:
3      fields: ["message"]
4      process_array: false
5      max_depth: 5
6      target: ""
7      overwrite_keys: false
8      add_error_key: true
9  - timestamp:
10     field: DateTime
11     layouts:
12       - '2022-03-16T08:46:33.8697387+01:00'
13       - '2006-01-02T15:04:05.999-07:00'
14     test:
15       - '2020-08-03T07:10:20.123456+02:00'
16       - '2022-03-16T08:46:33.8697387+01:00'
17 - drop_fields:
18     fields: [DateTime, message]
19 },
```

Výpis 4.12: Příklad doplněných položek v rámci klíče processors v konfiguraci

4.3 Výkonnostní testování systému MES PHARIS

Vzhledem ke komplexnosti systému MES PHARIS a všem požadavkům, je výsledné řešení převážně zaměřeno na testování terminálové aplikace systému. K výsledné implementaci je využita implementace infrastruktury popsaná v kapitole 4.2. Diagram celého zapojení výsledné implementace se systémem MES PHARIS je znázorněn na obrázku 4.5.

Popsané řešení v této kapitole zůstává majetkem firmy UNIS a pouze ukazuje možné řešení a reálné využití celkového návrhu výkonnostního testování. Jelikož je řešení závislé na logice samotného systému, která je majetkem společnosti, nebylo by možné řešení nezávisle spustit a ověřit jeho funkčnost. Z tohoto důvodu byla dále vytvořena implementace, která nevyžaduje systém MES PHARIS. Řešení je popsáno v rámci kapitoly 4.4.



Obrázek 4.5: Jednotlivé části výkonnostního testování se systémem MES PHARIS

4.3.1 Nástavba koncové aplikace systému MES PHARIS

Pro možnost testování s terminálovou aplikací systému bez jejího grafického rozhraní byla vytvořena implementace, která využívá pouze business logiku samotné aplikace. Aplikace je spuštěna na pozadí bez grafického rozhraní, aby měla uchovaný stav a reagovala jako běžná aplikace. Výsledná simulace momentálně umožňuje provést následující akce: připojení aplikace k aplikačnímu serveru, přihlášení a odhlášení uživatele, vybrání stroje v rámci výroby, vybrání operace, která bude operátorem prováděna, provedení registrace k výrobě, vytvoření odvodu práce s úpravou počtu shodných a neshodných kusů, provedení odregistrace, získání dostupných tiskáren, vytisknutí štítku (například pro obalovou jednotku) a odpojení terminálu od serveru. Celkem tedy byla implementována logika, která umožňuje provádět základní scénáře a hlavní činnosti, které jsou prostřednictvím aplikace prováděny. Řešení ovládání terminálové aplikace splňuje požadavky firmy, aby bylo možné spouštět a ovládat více terminálů, které budou provádět základní činnosti a ověřovat tak případné problémy.

Řešení bude možné snadno rozšiřovat o novou funkcionalitu a umožní tak provádění méně častých operací na terminálové aplikaci. Zde je v budoucnu zvažována možnost provádět další typy registrací, jako například prostoj nebo seřizování.

4.3.2 Generování testovacích dat

Pro generování testovacích dat byla přepracována a rozšířena současná implementace generátorů, které vytvářejí data pomocí logiky samotné aplikace. Toto řešení je velice rychlé a není nutné data připravovat prostřednictvím SQL skriptů nebo uchováváním obrazů databáze pro jednotlivé scénáře. Postupem času by navíc nemuselo již odpovídat schéma databáze a obrazy se skripty by nebyly funkční. Také udržitelnost těchto řešení by byla velice náročná. Jelikož je výsledné řešení programové a využívá logiku systému, je jeho udržitelnost a projevení změn zaručeno již změnami během vývoje. Pokud dojde k nějaké zásadní změně, je nefunkčnost řešení odhalena již při sestavení implementace generátorů. Dále toto řešení vytvoří všechny potřebné vazby v databázi.

Řešení lze snadno rozšiřovat dopisováním dalších generátorů, které budou umožňovat vytvoření dalších bussines objektů a upravovat jejich další vlastnosti. Pro možnost vytvoření základních testovacích scénářů byla vytvořena sada generátorů. Tato sada umožňuje vytvořit například následující záznamy: uživatelský účet, vytvoření role s oprávněními, vytvoření technologického procesu s operacemi, z technologického postupu vytvoření výrobního příkazu, dále vytvoření materiálu, vytvoření zařízení a jeho umístění ve výrobním modelu a další.

Tato sada generátorů tedy dokáže vytvořit základní objekty, které jsou nutné pro vytvoření scénáře, který kontroluje základní funkcionalitu výroby.

Současná podoba generování dat může být využita i testovacím oddělením, jelikož vytvoření některých dat pro potřeby ručního testování je neúměrně zdlouhavé.

4.3.3 Automatizované testy pro měření výkonnosti systému

Výsledná implementace je součástí repozitáře ve společnosti UNIS. Pro běh testů je vyčleněn samostatný stroj, kde běží všechny scénáře současně. Na dalším stroji běží instance systému MES PHARIS. Toto řešení sice nereflektuje skutečné fungování systému v reálném světě, kdy je každá terminálová aplikace spouštěna na vlastním mini PC, ale usnadňuje ovládání terminálů. Vzhledem k většímu počtu terminálů, které běží současně, by bylo potřeba několik samostatných strojů, což by bylo finančně i co do schopnosti orchestrace náročné.

Výsledná implementace obsahuje scénář, který si díky generátorům v metodě `Init` připraví potřebná data. Samotný scénář poté provede připojení terminálu k serveru, přihlášení uživatele, výběr stroje a operace k registraci, samotnou registraci, následně provede odvod práce s úpravou shodných a neshodných kusů a odregistraci uživatele. Tento scénář je následně spuštěn současně na patnácti instancích terminálu. Scénář nereflektuje celkovou dobu směny výroby, jelikož se zaměřuje na nejvíce kritické fáze výroby a to například zahájení směny a současné přihlašování všemi pracovníky k výrobě a následné odhlášení směny nebo předání rozpracované výroby.

4.4 Výkonnostní testování bez systému MES PHARIS

Z důvodu, že výsledné řešení je úzce spjato se systémem MES PHARIS, byla pro účely odevzdání a možnosti nezávislého spuštění výsledného řešení a jeho prezentace vytvořena implementace, která simuluje využití hardwaru daného stroje. Toto řešení umožňuje libovolné využití paměti RAM, procesoru na úrovni vláken nebo diskové kapacity stroje.

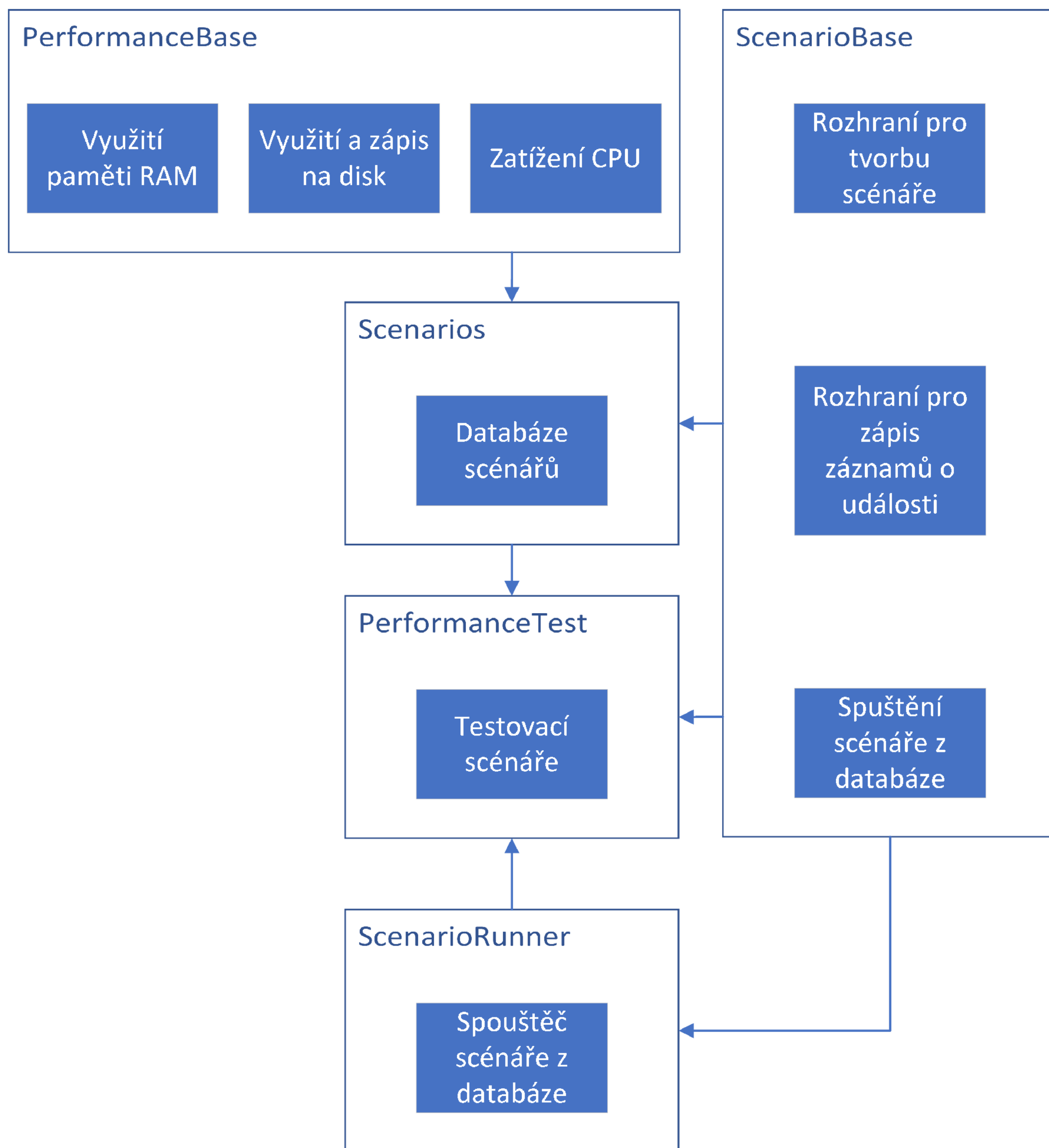
Díky celkovému návrhu řešení a přístupu k implementaci je také prezentována vlastnost nezávislosti výsledného řešení, jeho další možný rozvoj a použití i mimo společnost UNIS a systém MES PHARIS.

Výsledná implementace je psána pomocí platformy .NET framework, konkrétně verze 4.8. Je to z důvodu, že celková implementace pro orchestraci a práci se scénáři je touto platformou psána také a usnadňuje to referencování na ostatní projekty a minimalizuje meziplatformní problémy. Ovšem vzhledem k obsahu implementace a použitých knihoven není problém řešení konvertovat na platformu .NET 6. Pro ověření kompatibility byl využit nástroj `.NET Portability Analyzer`[12].

4.4.1 Řešení pro využívání zdrojů stroje

Řešení se nachází v knihovně `PerformaneBase` a její zapojení do celkové implementace je znázorněno na obrázku 4.6. Toto řešení tedy pouze nahradilo části systému MES PHARIS. Pro porovnání rozdílů slouží obrázek 4.5.

[12].NET Portability Analyzer – https://docs.microsoft.com/en-us/dotnet/standard/analyzers/portability-analyzer



Obrázek 4.6: Jednotlivé části výkonnostního testování bez systému MES PHARIS

Zatížení CPU

Pro simulování zátěže CPU v různých úrovních byla implementována třída `CpuAllocation`. Ukázka implementace v příloze B. Ta obsahuje veřejnou metodu `UseCpu`, která provede konstantní zatížení CPU podle parametrů. Parametry je definována úroveň zatížení vlákna procesoru v rozmezí 0-100 %, doba v sekundách, po kterou mají být vlákna zatížena a počet vláken, která budou zatížena. Pro zatížení procesoru na 100 % je nutné nastavit hodnotu zatížení vlákna na 100 % a počet vláken na hodnotu dostupných logických procesorů. Tu můžeme získat pomocí volání v jazyku C# `Environment.ProcessorCount`.

Zde musíme brát v úvahu, že využití procesoru je vždy průměr za nějaký čas. Procesor v daném momentě buď pracuje nebo ne. Není možné, aby procesor pracoval na poloviční výkon. Můžeme ale simulovat určitou úroveň zatížení během sekundy tím, že procesor bude pracovat 0,5 sekundy a 0,5 sekundy bude uspán. Tím získáme průměrné využití procesoru za jednu sekundu na úrovni 50 %. Tato implementace tedy provádí střídání nekonečného cyklu a pozastavení vlákna. Výsledek se projevuje jako konstantní zátěž procesoru.

Veřejná metoda provádí v cyklu s počtem iterací rovnající se počtu požadovaných vláken vytvoření instancí vláken, které provádějí soukromou metodu. Vytvořené vlákno je okamžitě spuštěno a jeho instance uložena do kolekce. Díky vytvoření nových vláken, která provádějí samotnou činnost, může hlavní vlákno čekat na uplynutí požadovaného času. Po jeho uplynutí jsou jednotlivá vlákna ukončena.

Soukromá metoda provádí nepřetržitě cyklus `while`. V tomto cyklu dochází k měření doby běhu a pokud je doba běhu v milisekundách větší než požadované vytížení v procentech, je vlákno uspáno na dobu podle vzorce $dobaSpanku = 100 - vytizeni[\%]$. Doba spánku je v milisekundách. K uspání vlákna je využito volání `Thread.Sleap`.

Využití paměti RAM

K využití paměti RAM byla vytvořena třída `MemoryAlloaction`. Ta obsahuje kolekci struktury `IntPtr`[13], jedná se o ukazatel do paměti. Tato kolekce je veřejná a k ukazatelům může být přistoupeno pro další využití, kolekce má název `IntPtrs`.

Pro alokaci paměti je využita třída `Marshal`[14]. Tato třída se stará o alokaci nespravované paměti, její kopírování, uvolnění a další metody pro práci s tímto typem paměti.

Implementovaná třída obsahuje veřejnou metodu `Allocate`, která má jako parametr velikost paměti k alokaci v bajtech. Metoda provede alokaci pomocí volání `Marshal.AllocHGlobal`. Po získání ukazatele je paměť naplněna náhodným obsahem pomocí třídy `Random` a metodou `WriteByte` ve třídě `Marshal`. Ukazatel je následně uložen do kolekce `IntPtrs`. Opakované volání metody `Allocate` provede vytvoření nového ukazatele o požadované velikosti, naplnění náhodným obsahem a umístěním ukazatele do kolekce.

Takto alokovaná paměť musí být uvolněna. K tomu slouží veřejné metody `Free`. Bez parametrů provede metoda uvolnění všech ukazatelů do paměti z kolekce a tuto kolekci vyprázdní. Je možné zadat této metodě parametr. Jde o index do kolekce a ukazatel na tomto indexu bude uvolněn. Dojde k odebrání ukazatele z kolekce, tím dojde k posunu všech

[13] Struktura IntPtr – https://docs.microsoft.com/en-us/dotnet/api/system.intptr
[14] Třída Marshal – https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal

dalších ukazatelů o mínus jedna. Tyto metody použijí nad ukazatelem do paměti volání `Marshal.FreeHGlobal` a tím dojde k uvolnění paměti.

Využití diskové kapacity

Další oblast, pro kterou byla připravena podpora, je zápis na pevný disk a tím využití jeho kapacity. Byla implementována třída `DiskAllocation`. Třída obsahuje veřejnou vlastnost typu `string` s názvem `FilePath`. Zde je uložena absolutní cesta k souboru, do kterého je zapisováno, a tím dochází k využití kapacity disku. Cesta k souboru a jeho název je získán pomocí volání `Path.GetRandomFileName`. Tím je zajištěno získání unikátního názvu a nemůže tedy dojít ke kolizi s existujícím souborem.

Třída implementuje metodu `WriteBytes`, která zapíše náhodný počet bajtů podle parametru do souboru. Pokud soubor neexistuje, vytvoří nový. Další metodou je `CreateFile`, která vytvoří prázdný soubor. Je možné předat parametr s počtem bajtů, který bude do souboru hned zapsán.

Pro smazání souboru je dostupná veřejná metoda `DeleteFile`. Tato metoda smaže soubor, do kterého je zapisováno, a nastaví cestu k souboru na hodnotu `null`.

4.4.2 Vytvořené scénáře

Pro ověření funkcionality bylo vytvořeno několik scénářů a následně testy, které využívají tyto scénáře. Testy dále slouží jako ukázkové řešení pro vytvoření jednotlivých scénářů a z nich celkového testovacího scénáře.

Scénář pro zatížení CPU

V rámci fáze `Init` jsou v textovém parametru očekávány údaje oddělené mezerou, které nesou informace o délce zatížení vlákna v sekundách, zatížení vlákna v úrovni 0-100 % a počet vláken, která mají být zatížena. Pořadí v textu je nutné zachovat.

V rámci běhu samotného scénáře dojde k vytvoření instance třídy, která zajišťuje zatížení procesoru. Zatížení se odvíjí od předaných parametrů pro samotný scénář.

Scénář pro využití paměti RAM

V rámci fáze `Init` jsou v textovém parametru očekávány údaje oddělené mezerou, které nesou informace o množství paměti, která má být alokována v bajtech a doba, po kterou má být paměť držena. Pořadí v textu je nutné zachovat.

V rámci běhu samotného scénáře dojde k vytvoření instance třídy, která zajišťuje alokaci paměti. Množství paměti se odvíjí od předaných parametrů pro samotný scénář. Po uplynutí doby, po kterou má být paměť alokována, je uvolněna.

Scénář pro využití disku

V rámci fáze `Init` jsou v textovém parametru očekávány údaje oddělené mezerou, které nesou informace o velikosti souboru, který má být vytvořen v bajtech a doba, po kterou má soubor existovat. Pořadí v textu je nutné zachovat.

V rámci běhu samotného scénáře dojde k vytvoření instance třídy, která zajišťuje vytvoření souboru na disku. Velikost souboru se odvíjí od předaných parametrů pro samotný scénář. Po uplynutí doby, po kterou má soubor existovat, je soubor smazán.

Komplexnější scénář

V rámci fáze `Init` jsou v textovém parametru očekávány údaje oddělené mezerou, které nesou informace o množství paměti RAM, která má být ve scénáři využita. Pořadí v textu je nutné zachovat.

V rámci běhu samotného scénáře dojde v cyklu o počtu deseti iterací k postupnému alokování paměti RAM, zápisu 20 MB do souboru a vytížení procesoru. Zatížení procesoru se odvíjí od náhodně získaných hodnot. Konkrétně úroveň zatížení vlákna v úrovni 10-30 %, doba zatížení vlákna 5-10 sekund a počet zatížených vláken je vždy jedna. Mezi jednotlivými iteracemi je vlákno pozastaveno na dobu v rozmezí 5-20 sekund. Po dokončení cyklu jsou veškeré zdroje uvolněny.

4.4.3 Sestavení a spuštění

Celé řešení je dostupné v rámci souboru `Performance.sln`. Pro sestavení je nutné mít nainstalován .NET Framework 4.8 Developer Pack[15]. Další z možností je mít nainstalováno Visual Studio[16]. Tím je zajištěna dostupnost všech nutných součástí pro sestavení, spuštění nebo případnou úpravu řešení.

Pokud je nainstalováno Visual Studio 2022, jsou umístění nutných prostředků následující:

Program pro sestavení **msbuild.exe**: `C:\Program Files\Microsoft Visual Studio\2022\Enterprise\MSBuild\Current\Bin\MSBuild.exe`

Program pro spuštění testů **VSTest.Console.exe**:
`C:\Program Files\Microsoft Visual Studio\2022\Enterprise\Common7\IDE\Extensions\TestPlatform\vstest.console.exe`

Sestavení a spuštění řešení je následně možné v prostředí Visual Studia, případně následujícími příkazy:

Sestavení aplikace: `dotnet build`
Který je proveden ve složce se souborem `Performance.sln`.

Alternativní sestavení aplikace: `<msbuild> Performance.sln -t:restore -p:RestorePackagesConfig=true`
Provede stažení nutných závislostí pro sestavení, je nutné provolat pouze poprvé.
Následně `<msbuild> Performance.sln`

[15] .NET Framework 4.8 – https://dotnet.microsoft.com/en-us/download/dotnet-framework/net48
[16] Visual Studio – https://visualstudio.microsoft.com/

Spuštění připravených testů: `<vstest> PerformanceTest.dll`
Soubor se bude nalézat ve složce:
`PerformanceTest\bin\{Konfigurace (Debug|Release)}\PerformanceTest.dll`

4.4.4 Ověření řešení

Implementace ukázkového řešení se nalézá ve zvláštním repozitáři na serveru DevOps v rámci infrastruktury firmy UNIS. Následně byla vytvořena nová úloha na serveru Jenkins, která má za účel spouštět tyto testy. Úloha má za úkol naklonování aktuálního repozitáře, sestavení zdrojových kódů, spuštění služby Metricbeat a následné spuštění testů. Po jejich dokončení je služba ukončena, aby nedocházelo k posílání dat pokud aktuálně neběží testy.

Pro běh ukázkových testů byl využit nezávislý počítač, který není zatěžován jinými procesy a úlohami. Díky tomu není výkonnostní měření ovlivňováno jinou aktivitou. Na tomto počítači jsou nakonfigurované aplikace Metricbeat a Filebeat, které zajišťují přenos veškerých potřebných informací.

Výsledky byly úspěšně vizualizovány řešením Aleše Ondráčka [12]. Zobrazené informace korespondovaly s očekáváním a implementací testovacích scénářů, jelikož je v nich zadefinována náročnost na jednotlivé zdroje stroje. Tím byla ověřena funkcionalita simulování zátěže stroje a reprezentace naměřených výsledků.

Kapitola 5

Závěr

Výsledné řešení naplnilo hlavní požadavky firmy UNIS. Vzhledem k samotné náročnosti vytvoření řešení pro výkonnostní testování terminálové aplikace a přípravu testovacích dat, nebylo možné vytvořit implementace pro další části systému MES PHARIS.

Aplikace pro sledování automatizovaných úloh jsou schopné sledovat a poskytovat všechna relevantní data a navazující práce Aleše Ondráčka tato data přehledně zobrazuje. V rámci vývojové kanceláře byla realizována vizualizační obrazovka, na které je neustále zobrazen přehled a stav kritických úloh. Z hodnocení společnosti se výsledné řešení jeví jako přínosné, jelikož díky finální reprezentaci dat je přehlednější a také sdružuje informace z více serverů na jednom místě a data z obou serverů jsou prezentována identickým způsobem.

Řešení výkonnostního testování pro terminálovou aplikaci bylo pravidelně konzultováno se společností UNIS a tak je výsledné řešení také hodnoceno kladně. Vytvořený scénář pokrývá kritickou část výroby a je tedy základním kamenem pro budoucí rozšiřování výkonnostního testování celého systému. K výslednému hodnocení dále přispívá kvalitní reprezentace výsledků testů, která byla součástí práce Aleše Ondráčka.

Samotná implementace výkonnostního testování a orchestrace jednotlivých scénářů není závislá na řešení společnosti UNIS a je možné ji dále rozvíjet. Aplikace pro sběr informací o automatizovaných procesech během vývoje je rovněž možné využít na jiných instancích serveru DevOps a Jenkins. Případné rozšíření aplikací by z pohledu jejich návrhu a implementace mělo být možné. Veškerý kód je řádně zdokumentován a během vývoje byl kladen důraz na čitelnost a jasnost celé implementace.

5.1 Odhalované problémy

Výkonnostní testování si klade za cíl sledovat kritické procesy, které jsou prováděny samotnými zákazníky systému MES PHARIS. Tím bude předcházeno a včas odhaleno nežádoucí chování systému po stránce výkonnosti. Vývojové oddělení tak bude mít dost času a podkladů pro provedení nápravných opatření. Díky celkové implementaci bude možné do vytvoření výsledných testovacích scénářů zapojit i testovací oddělení a tím tak zajistit pokrytí důležitých scénářů a zautomatizování procesu kontroly samotného systému MES PHARIS.

Sledování automatizovaných procesů si klade za cíl odhalovat infrastrukturní problémy, například nedostatečný výkon některého ze strojů nebo neefektivní úpravu předpisu úlohy. Díky sledování jednotlivých testů je možné odhalit výkonnostní problém řešení již na úrovni bloku kódu.

5.2 Budoucí rozšíření

V rámci společnosti UNIS je nyní plánováno sledování dalších vytipovaných automatizovaných procesů na serveru DevOps a Jenkins. Mezi další úlohy k budoucímu sledování patří úlohy sestavení dalších částí systému, například nový PDA klient nebo implementační balíček, kde jsou uchovány zákaznické úpravy. Dále následují úlohy, které provádějí testy nad těmito částmi systému.

Ze zjištěných údajů byl odhalen výkonnostní problém jednoho z agentů serveru Jenkins. Proto se společnost rozhodla k investici do nového serveru. Ten je momentálně připravován k zapojení do procesu. Předběžné testování ukazuje zrychlení v některých úlohách až o cca 50 %.

Co se týče výkonnostního testování, má společnost v plánu jeho rozšiřování jak v oblasti vytvořeného řešení pro terminálovou aplikaci, tak v dalších oblastech, které byly popsány v kapitole 2.3. Převážně půjde o rozšiřování funkcionality řešení pro generování dat a ovládání terminálu. Tím bude umožněno vytvoření složitějších scénářů pro výkonnostní testy s terminálovou aplikací. Následně bude výkonnostnímu testování podroben i webový klient.

Literatura

[1] *Azure DevOps Services REST API Reference* [online]. 2022 [cit. 2022-01-11]. Dostupné z: https://docs.microsoft.com/en-us/rest/api/azure/devops/.

[2] Bakshi, A. *Performance Testing Types & Metrics* [online]. Březen 2021 [cit. 2021-11-22]. Dostupné z: https://www.webomates.com/blog/software-testing/performance-testing/.

[3] Callison, J. *Database Locking: What it is, Why it Matters and What to do About it* [online]. 2009 [cit. 2021-11-24]. Dostupné z: https://www.methodsandtools.com/archive/archive.php?id=83.

[4] Farrell Vinay, P. *Manage Software Testing*. USA: Auerbach Publications, 2007. ISBN 978-0-8493-9383-9.

[5] Gillis, A. S. *Performance testing* [online]. Prosinec 2020 [cit. 2021-12-14]. Dostupné z: https://searchsoftwarequality.techtarget.com/definition/performance-testing.

[6] Groves, B. *Improve Application Performance with These Advanced GC Techniques* [online]. Září 2021 [cit. 2021-11-23]. Dostupné z: https://www.overops.com/blog/improve-your-application-performance-with-garbage-collection-optimization/.

[7] Hamilton, T. *10 BEST Performance Testing Tools | Load Testing Tools (2022)* [online]. Prosinec 2021 [cit. 2022-01-04]. Dostupné z: https://www.guru99.com/performance-testing-tools.html.

[8] Hamilton, T. *Performance Testing Tutorial: What is, Types, Metrics & Example* [online]. Listopad 2021 [cit. 2021-11-22]. Dostupné z: https://www.guru99.com/performance-testing.html.

[9] *MES centrum* [online]. 2012 [cit. 2021-11-15]. Dostupné z: http://mescenter.org.

[10] Molyneaux, I. *The Art of Application Performance Testing: From Strategy to Tools*. O'Reilly Media, 2014. Theory in practice. ISBN 9781491900505. Dostupné z: https://books.google.cz/books?id=jc7UBQAAQBAJ.

[11] Nayak, S. *Garbage Collection in Java – What is GC and How it Works in the JVM* [online]. Leden 2021 [cit. 2021-11-23]. Dostupné z: https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/.

[12] Ondráček, A. *Monitorování výkonnosti systému MES PHARIS*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[13] *Pipeline Stage View Plugin* [online]. 2016. 2022 [cit. 2022-01-10]. Dostupné z: https://github.com/jenkinsci/pipeline-stage-view-plugin.

[14] Rizio, J. *6 Common Performance Testing Mistakes* [online]. Leden 2020 [cit. 2021-12-16]. Dostupné z: https://www.flood.io/blog/6-common-performance-testing-mistakes.

[15] *UNIS* [online]. 2020 [cit. 2021-11-15]. Dostupné z: https://www.unis.cz/.

[16] Weyuker, E. a Vokolos, F. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*. 2000, sv. 26, č. 12, s. 1147–1156. DOI: 10.1109/32.888628.

Příloha A

Obsah přiloženého paměťového média

Adresářová struktura je následující:

- `doc` - obsahuje dokumentaci
- `src` - obsahuje zdrojové kódy
  - `jobReader` - obsahuje zdrojové kódy aplikací pro sledování serverů
    - `Common` - obsahuje implementaci sdílené funkcionality
    - `DevOpsJobReader` - obsahuje implementaci aplikace pro server DevOps
    - `DevOpsJobReader.Tests` - obsahuje implementaci testů nad aplikací pro server DevOps
    - `JenkinsJobReader` - obsahuje implementaci aplikace pro server Jenkins
    - `JenkinsJobReader.Tests` - obsahuje implementaci testů nad aplikací pro server Jenkins
  - `perfTest` - obsahuje zdrojové kódy s implementací výkonnostního testování
    - `PerformanceBase` - obsahuje implementaci pro vytvoření zátěže nad hardwarem stroje
    - `PerformanceTest` - obsahuje implementaci výsledných testovacích scénářů
    - `ScenarioBase` - obsahuje implementaci pro scénář a podpůrné třídy
    - `ScenarionRunner` - obsahuje implementaci aplikace pro spuštění scénáře
    - `Scenarios` - obsahuje implementaci jednotlivých scénářů
- `example` - obsahuje ukázky výstupu aplikací pro sledování úloh
- `text` - obsahuje vlastní text práce

Příloha B

Ukázka implementace zatížení CPU

```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Threading;
5
6  namespace PerformanceBase
7  {
8      public class CpuAllocation
9      {
10         private void UseCpu(object cpuUsageObj)
11         {
12             int cpuUsage = (int)cpuUsageObj;
13             if (cpuUsage < 0 || cpuUsage > 100)
14                 throw new ArgumentException(
15                     $"Parameter {nameof(cpuUsage)} out of range (0-100)"
16                 );
17
18
19             Stopwatch watch = new Stopwatch();
20             watch.Start();
21             while (true)
22             {
23                 if (watch.ElapsedMilliseconds > cpuUsage)
24                 {
25                     Thread.Sleep(100 - cpuUsage);
26                     watch.Reset();
27                     watch.Start();
28                 }
29             }
30         }
31
```

```
32          /// <summary>
33          /// Provede konstantni vytizeni CPU po zadanou dobu
34          /// </summary>
35          /// <param name="cpuUsage">
36          /// Hodnota zatizeni jednoho vlakna (0-100)
37          /// </param>
38          /// <param name="timeoutSeconds">
39          /// Doba, po kterou bude CPU zatizen
40          /// </param>
41          /// <param name="processorCount">
42          /// Pocet vlaken, ktera budou zatizena
43          /// </param>
44          public void UseCpu
45              (int cpuUsage, int timeoutSeconds, int processorCount)
46          {
47              List<Thread> threads = new List<Thread>();
48              for (int i = 0; i < processorCount; i++)
49              {
50                  Thread t = new Thread(new ParameterizedThreadStart(UseCpu));
51                  t.Start(cpuUsage);
52                  threads.Add(t);
53              }
54
55              Thread.Sleep(timeoutSeconds * 1000);
56              foreach (Thread t in threads)
57              {
58                  t.Abort();
59              }
60          }
61      }
62  }
```

Výpis B.1: Implementace třídy zajišťující využití CPU

Příloha C

Ukázka výsledku ze serveru Jenkins


```
1  {
2      "server": "jenkins",
3      "pipelineId": "UNIT_TESTS_COME_DEV",
4      "buildId": 2944,
5      "status": "SUCCESS",
6      "startTimeMillis": 1651758705464,
7      "startTime": "2022-05-05T13:51:45.464Z",
8      "finishTime": "2022-05-05T13:54:24.122Z",
9      "queueTime": "2022-05-05T13:51:45.458Z",
10     "finishTimeMillis": 1651758864122,
11     "durationMillis": 158658,
12     "queueDurationMillis": 6,
13     "pauseDurationMillis": 0,
14     "parameters": {
15         "Branch": "refs/heads/master",
16         "Commit": "9798513b935dee1b44789cacb19b8d95eec42b16",
17         "BuildResult": "Succeeded",
18         "DevOpsBuildName": "4.18.0.250",
19         "Requestor": "Ohanka Martin",
20         "MESPharisVersion": "4.18.0.250",
21         "BuildId": "47442"
22     },
23     "stages": [
24         {
25             "name": "Unit Testing ...",
26             "status": "SUCCESS",
27             "startTimeMillis": 1651758758520,
28             "durationMillis": 101884,
29             "pauseDurationMillis": 0,
30             "startTime": "2022-05-05T13:51:45.464Z",
31             "endTime": "2022-05-05T13:54:24.122Z"
```

```
32          }
33          ...
34      ],
35      "testsResult": {
36          "total": 1250,
37          "executed": 1243,
38          "passed": 1243,
39          ...
40          "testsRunResult": [
41              {
42                  "testName": "GetDateTimeByInnerProviderTest",
43                  "testClass": "Phoenix.Phar.Core.Stone.Tests.Data.
44                      CProgrammableDataProviderTest",
45                  "result": "Passed",
46                  "durationMillis": 2.939,
47                  "startTime": "2022-05-05T15:54:00.9965501+02:00",
48                  "stopTime": "2022-05-05T15:54:00.9995503+02:00"
49              }
50              ...
51          ],
52          "startTime": "2022-05-05T15:53:10.1441899+02:00",
53          "finishTime": "2022-05-05T15:54:19.9960723+02:00",
54          "durationMillis": 69851
55      },
56      "isResultValid": true
57  }
```

Výpis C.1: Ukázka výstupu z aplikace sledující server Jenkins

Ukázka obsahuje reálný výstup z aplikace. Běh obsahoval vykonání 1250 testů, které byly zpracovány. Dále se úloha skládá z osmi kroků. Výsledek byl v klíčích `stages`, `testsResult` a `testsRunResult` zkrácen z důvodu přehlednosti a rozsahu ukázky. Průměrná doba zpracování běhu této úlohy se pohybuje kolem 500 milisekund. Aplikace běžela na stroji s procesorem Intel Core - i7-4770 a 16GB RAM. Pro síťovou komunikaci byla použita interní síť.

Příloha D

Ukázka výsledku ze serveru Devops

```
1  {
2      "buildId": 47442,
3      "server": "devops",
4      "stages": [
5          {
6              "id": "3338ca83-c763-54a4-3064-f850429ce430",
7              "parentId": "fd490c07-0b22-5182-fac9-6d67fe1e939b",
8              "name": "Build project PharisWebClient.csproj",
9              "startTime": "2022-05-05T13:40:28.5866667Z",
10             "finishTime": "2022-05-05T13:48:08.82Z",
11             "durationMillis": 460233,
12             "state": "completed",
13             "result": "succeeded",
14             "errorCount": 0,
15             "warningCount": 0,
16             "log": {
17                 "url": url
18             },
19             "type": "Task",
20             "order": 9
21         }
22         ...
23     ],
24     "packagesContentInfo": {
25         "Phoenix.Phar.EventSystem.Package": {
26             "dirs": 22,
27             "files": 575,
28             "totalItems": 597,
29             "size": 208482273
30         },
31         "Phoenix.Phar.Terminal.Package": {
```

```
32             "dirs": 9,
33             "files": 72,
34             "totalItems": 81,
35             "size": 35995193
36         },
37         "Phoenix.Phar.Web.Package": {
38             "dirs": 373,
39             "files": 5042,
40             "totalItems": 5415,
41             "size": 269497507
42         }
43     },
44     "pipelineId": 86,
45     "pipelineName": "Pharis-CI-4.15\u002B",
46     "pipelineRevision": 86,
47     "durationMillis": 788885,
48     "queueDurationMillis": 3312,
49     "parameters": {},
50     "isResultValid": true,
51     "buildNumber": "4.18.0.250",
52     "status": "completed",
53     "result": "succeeded",
54     "queueTime": "2022-05-05T13:38:26.5678093Z",
55     "startTime": "2022-05-05T13:38:29.8802922Z",
56     "finishTime": "2022-05-05T13:51:38.7655701Z",
57     "sourceBranch": "refs/heads/master",
58     "sourceVersion": "9798513b935dee1b44789cacb19b8d95eec42b16",
59     "reason": "manual",
60     "requestedFor": {
61         "displayName": "Ohanka Martin",
62         "uniqueName": uniqueName
63     },
64     "requestedBy": {
65         "displayName": "Ohanka Martin",
66         "uniqueName": uniqueName
67     }
68 }
```


Výpis D.1: Ukázka výstupu z aplikace sledující server DevOps

Ukázka obsahuje reálný výstup z aplikace. Úloha se skládá ze 35 kroků. Výsledek byl v klíčích `stages` zkrácen z důvodu přehlednosti a rozsahu ukázky. Klíče `uniqueName` a `url` byly anonymizovány. Průměrná doba zpracování běhu této úlohy se pohybuje kolem 200 milisekund. Aplikace běžela na stroji s procesorem Intel Core - i7-4770 a 16GB RAM. Pro síťovou komunikaci byla použita interní síť.