Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Reporting v testování softwaru

Bc. David Pilař

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. David Pilař

Informatika

Název práce

Reporting v testování softwaru

Název anglicky

Reporting in software testing

Cíle práce

Cílem práce je navrhnout a implementovat systém, jenž bude zodpovědný za sběr dat z test management systému a jejich předávání do různých reportovacích či monitorovacích systémů. Cílem je konkrétní navržené řešení implementovat ve vybrané společnosti a předáváním dat do externích systémů umožnit analýzu výsledků testování širšímu spektru pracovníků.

Metodika

Metodika teoretické části bude založena na analýze a následné syntéze odborných informačních zdrojů zabývajících se zvolenou problematikou. Metodika praktické části bude založena na identifikaci potřebných dat, která je třeba získávat. Dále také na podrobné analýze odkud a jak tyto data získávat a do jakých reportovacích systémů je následně třeba data předávat. Na základě toho bude navržen a implementován vlastní systém, který takový sběr a reporting dat o testování aplikací umožní. Následně bude analyzován přínos systému ve zvolené společnosti pomocí vhodných metod společně s obecným návrhem využití systému v podobném typu společností. Na základě syntézy teoretických a praktických poznatků budou formulovány závěry práce.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

testování softwaru, zaručení kvality, reporting, zpracování dat, vývoj software

---

Doporučené zdroje informací

ENGLANDER, Irv. *The architecture of computer hardware and systems software : an information technology approach.* New York: Wiley, 2003. ISBN 0471073253.

Huang, J., Gotel, O., & Zisman, A. (Eds.). 2014. Software and Systems Traceability (2012th edition). Springer. ISBN 1447158199.

KEYES, Jessica. *Software engineering handbook.* Boca Raton: Auerbach, 2003. ISBN 0849314798.

LAURSEN, Gert H. N.; THORLUND, Jesper. *Business analytics for managers : taking business intelligence beyond reporting.* Hoboken, New Jersey: Wiley, 2017. ISBN 978-1-119-29858-8.

LEWIS, William E.; VEERAPILLAI, Gunasekaran. *Software testing and continuous quality improvement.* Boca Raton: Auerbach Publications, 2005. ISBN 0849325242.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Petr Benda, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 29. 6. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

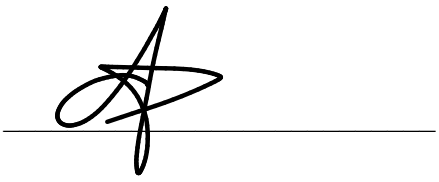
doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 20. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Reporting v testování softwaru" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. 3. 2024 

Poděkování

Rád bych touto cestou poděkoval Ing. Petru Bendovi, Ph.D. za vedení práce, trpělivost a cenné podněty. Kolegům z Alza.cz děkuji za možnost realizace praktické části této práce a odborné rady poskytnuté v průběhu zpracování. Své přítelkyni děkuji za velkou psychickou podporu a nekonečnou motivaci.

Reporting v testování softwaru

Abstrakt

Testování je klíčovou součástí vývoje každého softwaru a má velký dopad na výslednou podobu a kvalitu vyvinutého produktu. Reporting je jedním z pilířů business intelligence a jde o cenný nástroj pro podporu rozhodování ve firmách. Diplomová práce kombinuje obě tyto oblasti a zabývá se problematikou reportingu v testování softwaru, přičemž si klade za cíl navrhnout a vytvořit nástroj, který zastřeší sběr dat z test management systému a jejich následné předávání externím reportovacím a monitorovacím nástrojům.

Teoretická část v podobě literární rešerše poskytuje přehled o základní problematice vývoje softwaru, podrobněji se pak zaměřuje na technologii kontejnerizace, jsou popsány aspekty testování softwaru a rovněž nezbytné náležitosti ohledně business intelligence, reportingu i reportingu přímo v kontextu testování softwaru.

Vlastní práce začíná analýzou požadavků na výslednou aplikaci. Na základě specifikovaných požadavků probíhá analýza dat, datových zdrojů, je navržen způsob ukládání dat a způsob jejich předávání externím systémům. Poté je navržena architektura aplikace, detailně popsán princip jejího fungování a celý proces vývoje. Výsledná aplikace je nasazena do firemního Kubernetes clusteru a je vyhodnocen přínos celého řešení.

Klíčová slova: testování softwaru, business intelligence, reporting, zpracování dat, vývoj software, .NET, kontejnerizace, Kubernetes, Elasticsearch

Reporting in software testing

Abstract

Testing is a crucial part of software development and has a major impact on the final appearance and quality of the developed product. Reporting is one of the pillars of business intelligence and is a valuable tool for supporting decision-making in companies. The diploma thesis focuses on both areas and deals with reporting in software testing, while aiming to design and create a tool that covers the collection of data from the test management system and their subsequent transmission to external reporting and monitoring tools.

The theoretical part, in the form of a literature review, provides an overview of the basics of software development, then it focuses in more detail on containerization technology, aspects of software testing are described, as well as the necessary principles regarding business intelligence, reporting and reporting directly in the context of software testing.

The actual work begins with an analysis of the requirements for the final application. Based on the specified requirements, the analysis of data and data sources is carried out, the method of storing data and the method of transferring it to external systems is proposed. Then the architecture of the application is designed, the principle of its operation and the entire development process are described in detail. The resulting application is deployed in the company's Kubernetes cluster and the benefit of the entire solution is evaluated.

Keywords: software testing, business intelligence, reporting, data processing, software development, .NET, containerization, Kubernetes, Elasticsearch

Obsah

1 Úvod

Business Intelligence (BI) představuje klíčovou součást strategického rozhodování v podnikovém prostředí. Reporting, jako jeden z pilířů BI, umožňuje firmám přeměňovat surová data na cenné informace, které podporují efektivní rozhodování. Obrovské množství různorodých dat je generováno i v procesu testování softwaru, přičemž v kontextu vývoje softwaru je testování nezbytné pro zajištění kvality a spolehlivosti aplikací.

Důležitost reportingu v testování softwaru vychází z potřeby detailně porozumět výkonnosti, bezpečnosti a celkové kvalitě testovaného produktu. Efektivní reporting v tomto kontextu umožňuje týmům rychle identifikovat problémy, monitorovat pokrok ve vývoji a přizpůsobovat testovací strategie podle aktuálních potřeb. S narůstající složitostí softwarových projektů se zvyšuje i důležitost schopnosti efektivně zpracovávat a prezentovat data z testování, a to nejen členům testovacího týmu, ale především vedoucím pracovníkům nejen v rámci IT oddělení. I v dnešní době bohužel stále existují případy, kdy je hlavním nástrojem reportingu Microsoft Excel, což má velmi daleko od ideální situace.

Tato práce se proto ve své praktické části zaměřuje na vytvoření nástroje, který bude sloužit jako podpora pro reporting v rámci testování softwaru. Cílem je vyvinout řešení, které bude schopné sbírat data o testování přímo z test management systému a následně tato data zpřístupní pro analýzu pomocí externích reportovacích a monitorovacích nástrojů. Takový nástroj by měl po nasazení umožnit společnosti nejen efektivnější interpretaci výsledků testování, ale také zlepšit celkovou transparentnost a sledovatelnost procesů vývoje softwaru.

Vývoj tohoto nástroje představuje komplexní výzvu, která vyžaduje porozumění jak procesům testování softwaru, tak principům Business Intelligence, reportingu i samotného vývoje softwaru. Práce proto v teoretické části shrnuje zásadní poznatky a problematiku dotyčných oblastí a dále se v praktické části věnuje nejen technickému návrhu a implementaci takového nástroje, ale rovněž analýze požadavků uživatelů a možnostem integrace s existujícími systémy a procesy v organizaci, kde dojde k nasazení výsledného řešení a vyhodnocení jeho přínosu.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je navrhnout a implementovat systém, jenž bude zodpovědný za sběr dat z test management systému a jejich předávání do různých reportovacích či monitorovacích systémů. Cílem je konkrétní navržené řešení implementovat ve vybrané společnosti a předáváním dat do externích systémů umožnit analýzu výsledků testování širšímu spektru pracovníků.

2.2 Metodika

Metodika teoretické části bude založena na analýze a následné syntéze odborných informačních zdrojů zabývajících se zvolenou problematikou. Metodika praktické části bude založena na identifikaci potřebných dat, která je třeba získávat. Dále také na podrobné analýze odkud a jak tyto data získávat a do jakých reportovacích systémů je následně třeba data předávat. Na základě toho bude navržen a implementován vlastní systém, který takový sběr a reporting dat o testování aplikací umožní. Následně bude analyzován přínos systému ve zvolené společnosti pomocí vhodných metod společně s obecným návrhem využití systému v podobném typu společností. Na základě syntézy teoretických a praktických poznatků budou formulovány závěry práce.

3 Teoretická východiska

3.1 Vývoj softwaru

Vývoj softwaru je jedním ze základních pilířů moderního technologického pokroku a jde o odvětví, které formuje a ovlivňuje to, jakým způsobem interagujeme s technologiemi v každodenním životě. Jedná se o proces, jehož prostřednictvím jsou digitální aplikace, systémy a řešení koncipovány, navrhovány, provozovány a udržovány. Nejde ovšem čistě o technické úsilí, vývoj softwaru je mnohostrannou disciplínou, která propojuje řadu činností, mimo jiné analytické řešení problémů, kreativní design a strategickou implementaci, aby byly splněny potřeby uživatelů, specifikace zadavatelů a obchodní cíle společnosti. Celý proces se musí neustále přizpůsobovat novým přístupům a technologiím. (Langer, 2016)

3.1.1 Role ve vývoji softwaru

Jedním z předpokladů úspěchu každého softwarového produktu je především schopný tým lidí, kteří stojí za jeho vznikem. Metod pro vývoj softwaru existuje mnoho a napříč konkrétními projekty se může lišit i přesné složení týmu z hlediska rolí, ty základní jsou zmíněné níže. (Filipova, 2018)

Business owner

Jedná se o strategickou roli, jejímž cílem je primárně obchodní perspektiva produktu. Business owneři mají za úkol identifikaci nových příležitostí na trhu, možných zdrojů příjmů a navazování vhodných partnerství. Jejich cílem je zajistit, že vyvíjený softwarový produkt je v souladu s dlouhodobými obchodními cíli společnosti. Rozhodují se primárně na základě aktuálních trendů na trhu, potřeb zákazníků a vize společnosti, případně vlastní vize o tom, kam chtějí svůj svěřený produkt vývojově směřovat, vždy s ohledem na business. (Filipova, 2018)

Product manager

Slouží zpravidla jako propojovací článek mezi business ownerem a zbytkem vývojového týmu, mezi jeho typické úkoly patří priorizace úkolů, podpora netechnicky zaměřených oddělení zainteresovaných na vývoji či sběr zpětné vazby. Jsou zodpovědní

za pochopení cílového trhu, porozumění potřebám zákazníků a uvědomění si businessového hlediska projektu tak, aby mohli vývoj produktu správně řídit a směrovat. (Filipova, 2018)

Designer

Designéři jsou zodpovědní za to, jak bude výsledná aplikace vypadat po grafické stránce a jak dobře se uživatelům bude s aplikací pracovat. Při své práci propojují umění s technologiemi a vytvářejí rozhraní, která jsou nejen vizuálně přitažlivá, ale také intuitivní a uživatelsky přívětivá. Designéry lze velmi obecně rozdělit do dvou skupin na UI designéry a UX designéry. UI designéři mají na starosti návrh uživatelského rozhraní (User Interface), soustředí se primárně na to, jak budou jednotlivé komponenty z grafického hlediska vypadat. Oproti tomu UX designéři se soustředí celkově na uživatelský zážitek (User Experience), aby zajistili, že používání aplikace a cesta uživatele všemi kroky je logická, efektivní a příjemná. Jejich práce je základem při zpřístupňování softwarových produktů a jejich přitažlivosti pro uživatele všech cílových skupin, což významně ovlivňuje spokojenost uživatelů a míru přijetí produktu. (Filipova, 2018)

Velmi často roli UI i UX designéra ve firmách zastává jedna a ta samá osoba, může se to však lišit v závislosti na velikosti společnosti a dalších faktorech.

Backend vývojář

Backend vývojáři jsou prakticky architekty celé struktury softwaru a zaměřují se na technologie serverové, databázové a aplikační logiku, tedy vše, co funguje na pozadí celé aplikace. Zajišťují, že software plní své základní funkce efektivně, bezpečně a spolehlivě. Zpracováním dat, ukládáním a zabezpečením vytvářejí vývojáři backendu jádro softwaru, umožňující bezproblémovou interakci pro uživatele a poskytují frontendovým vývojářům podporu poskytováním nezbytných dat a funkcí. (Filipova, 2018)

Frontend vývojář

Frontend vývojáři přivádějí produkt k životu z pohledu uživatele. Vytvářejí vizuální a interaktivní prvky softwaru a převádějí koncepty popsané v návrhu do funkčních aplikací orientovaných na uživatele. Frontend vývojáři svou prací zpřístupňují a zpříjemňují používání softwaru uživatelům na různých platformách a zařízeních, čímž zajišťují konzistentní a responzivní prostředí, které splňuje potřeby a očekávání uživatelů. (Filipova, 2018)

Quality Assurance

QA oddělení neboli quality assurance zajišťuje kvalitu a spolehlivost softwaru pomocí mnoha metodik testování tak, aby identifikovali a opravili chyby dříve, než se produkt dostane ke koncovým uživatelům. Jejich role zahrnuje širokou škálu testovacích metodologií, aby bylo zajištěno, že software splňuje všechny funkční požadavky a výkonnostní standardy. Tím, že předcházejí problémům, které by mohly zhoršit uživatelskou zkušenost nebo narušit funkčnost aplikace, hraje quality assurance klíčovou roli při udržování integrity softwaru. (Filipova, 2018)

DevOps

Lidé pracující na DevOps pozicích působí na pomyslné křižovatce vývoje softwaru a IT operací s cílem zkrátit životní cyklus vývoje a poskytovat zejména technologie a pipeline pro kontinuální integraci a kontinuální nasazení. Implementují postupy a nástroje, které zajišťují lepší spolupráci vývojového týmu, vyšší efektivitu a škálovatelnost při zavádění softwaru a správě infrastruktury. Týmy DevOps umožňují společnostem rychle se přizpůsobit změnám trhu a potřebám zákazníků a zároveň zajistit spolehlivost a bezpečnost jejich softwarových produktů. (Filipova, 2018)

3.1.2 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru, zkráceně v angličtině SDLC (Software Development Life Cycle) označuje základní koncept softwarového inženýrství a řídí se dle něj proces vývoje od plánování až po údržbu. Existuje mnoho metodologií, které určují, jak mají organizace k vývoji softwaru přistupovat, jako agile, vodopád či spirála. Navzdory tomu však stále ve vývoji softwaru přetrvávají opakující se problémy, zejména nedodržení časového plánu, vysoká chybovost, a ve výsledku i nekvalitní aplikace.

Definované metodologie zkrátka mnohdy nejsou dostačující v prostředí moderního vývoje softwaru – ten je čím dál komplexnější, požadavky náročnější a průběh bývá velmi proměnlivý. Tradiční vodopádový model bývá kritizován pro svou nedostatečnou pružnost a lineární povahu, agilní metodiky oproti tomu sice nabízejí vysokou pružnost, mohou však trpět z hlediska nedostatečné dokumentace a jsou problematické v dlouhodobém plánování. (Conger, 2011)

Navzdory mnohdy problematické adaptaci lze popsat několik typických fází životního cyklu vývoje softwaru, ty jsou popsány v následujících odstavcích.

Plánování a studie proveditelnosti

V této počáteční fázi se hodnotí proveditelnost projektu z technického, finančního a provozního hlediska. Fáze obnáší zejména pochopení problému nebo příležitosti, definování rozsahu projektu a provedení předběžné analýzy k posouzení životaschopnosti projektu. (Conger, 2011)

Analýza požadavků

Tato fáze se zaměřuje na shromáždění podrobných požadavků od zúčastněných stran k definování očekávané funkcionality a případným omezením softwaru. Jde o zásadní fázi pro pochopení potřeb uživatelů a toho, jak by měl výsledný systém fungovat. Výstupem analýzy požadavků je zpravidla dokument s podrobnou specifikací těchto požadavků, na základě nich bývá následně postaveno zadání vývoje. (Conger, 2011)

Design

Fáze designu či návrhu slouží k transformaci sepsaných požadavků do grafického nebo funkcionálního návrhu pro vývoj softwaru. To zahrnuje architektonický návrh, definování celkové architektury systému a pokud to daný produkt vyžaduje, tak i wireframe či detailní grafický návrh, kde jsou navrženy konkrétní komponenty, rozhraní a datové modely. Výsledkem je designová specifikace, od které se odráží a řídí se dle ní další fáze vývoje. (Conger, 2011)

Implementace

Implementaci lze jinak označit jednoduše jako psaní kódu. Během implementační fáze se dle výstupů z předešlých fází, tedy analýzy požadavků a designu, píše samotný kód. Programátoři mají za úkol vytvořit veškeré softwarové komponenty a funkce tak, aby vše splňovalo dříve definované požadavky. Důraz je kladen na vývoj funkčního softwarového systému, který splňuje předem definovaná kritéria. (Conger, 2011)

Testování

Poté, co je software vyvinut, by měl procházet důkladným testováním, aby se identifikovaly a opravily případné závady a nedostatky. Cílem této fáze je zajistit, aby software splňoval požadavky, jak z hlediska funkčnosti, tak z hlediska designu a choval se dle očekávání za různých podmínek. Testování může samo o sobě zahrnovat několik fází, jako unit testování, integrační testování, systémové testování a akceptační testování. Tato fáze bývá mnohdy podceňována. (Conger, 2011)

Nasazení

Jakmile je software vyvinut, otestován a považován za hotový, je nasazen do produkčního prostředí, kde bude následně dostupný k použití. Nasazení může být buď postupné nebo jednorázové, v závislosti na povaze projektu a zejména platformě, kam se aplikace nasazuje. Tato fáze může zahrnovat rovněž zaškolení uživatelů a přípravu dokumentace pro nasazení. (Conger, 2011)

Údržba

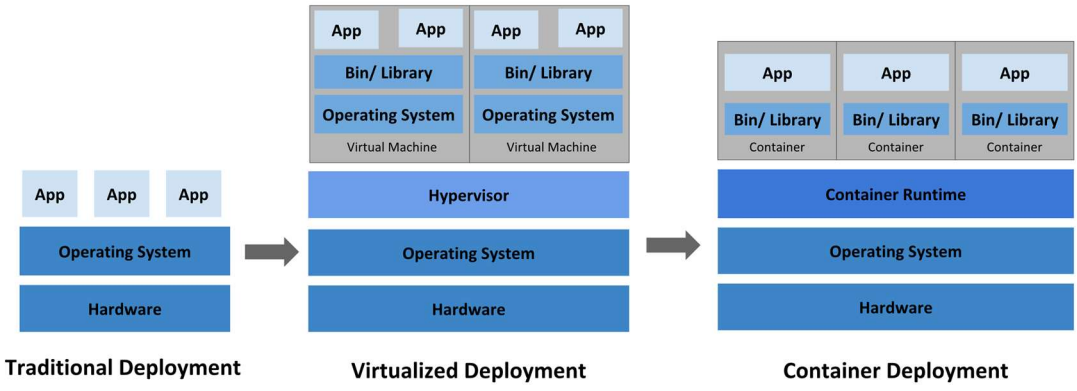
Po nasazení vstupuje software do fáze údržby, kde se dle potřeby aktualizuje, upgraduje a opravuje, aby se odstranily případné problémy, zlepšil výkon nebo přidaly nové funkce. Údržba zajišťuje, že software bude i nadále splňovat potřeby uživatelů v průběhu času. (Conger, 2011)

Jak ovšem vyplynulo z úvodu kapitoly a co rovněž Conger ve své publikaci zdůrazňuje je, že pevně definované metodologie jsou problematické a vždy je nutné přizpůsobit skladbu fází a jejich průběh konkrétnímu projektu. Jedním ze zásadních zmíněných nedostatků je i zanedbávání testování a obecně procesů pro zaručená kvality, s nimiž by se mělo počítat v průběhu celého vývojového procesu, a ne pouze ve specifické fázi.

3.2 Kontejnerizace

Dalo by se říct, že technologie kontejnerizace způsobila revoluci ve způsobu, jakým jsou obecně aplikace vyvíjeny, nasazovány a spravovány. Jedná se rovněž o významný posun od tradičních virtualizačních přístupů. Princip kontejnerizace v jádru spočívá v tom, že se aplikace včetně veškerých svých závislostí zapouzdří do balíčku neboli kontejneru, který lze následně spustit na libovolném kompatibilním hostitelském systému. Toto zapouzdření zajišťuje konzistenci napříč různými prostředími, od vývoje až po nasazení, a efektivně se tím řeší různé problémy s kompatibilitou a případnými chybami specifickými pro odlišná prostředí. (Bentaleb, 2022)

Za počátky kontejnerizace lze svým způsobem považovat unixový mechanismus chroot, což je příkaz sloužící k izolaci souborových systémů a zaveden byl již v 70. letech minulého století. Globální nárůst popularity kontejnerizace nicméně nastal až s představením Dockeru v roce 2013. Docker totiž přišel že standardizovaným způsobem balení a distribuce aplikací, což vývojářům značně usnadnilo používání kontejnerů. (Turnbull, 2019)



Obrázek 1 - Architektura kontejnerizace v porovnání s tradiční virtualizací; zdroj: (Kubernetes Documentation | Kubernetes, 2023)

Na rozdíl od tradiční virtualizace, která se při vytváření virtuálních strojů (VM) spoléhá na hypervizory se samostatnými operačními systémy, kontejnery sdílejí stejné jádro operačního systému a izolují aplikační procesy od hostitele. Porovnání jednotlivých přístupů vizualizuje obrázek výše. Díky tomuto sdílenému použití operačního systému hostitele jsou

kontejnery efektivnější a mnohem méně náročné na výpočetní výkon než standardní virtuální počítače, což výrazně zkracuje dobu spouštění a zlepšuje využití zdrojů. (Bentaleb, 2022)

Zavedení kontejnerizace mělo obecně výrazný pozitivní dopad na vývoj a nasazování aplikací. S využitím kontejnerizace se pojí četné výhody, zejména vylepšená škálovatelnost, jelikož kontejnery lze rychle a jednoduše přidávat nebo odebírat dle potřeby a požadavků provozovaných aplikací. Z výhod lze dále zmínit například zvýšenou produktivitu vývojářů, díky konzistenci prostředí nebo provozní efektivitu, díky dobré optimalizaci zdrojů a sníženým režijním nákladům. (Bentaleb, 2022)

3.2.1 Docker a jeho základní komponenty

Ekosystém Dockeru se skládá z několika klíčových komponent, které společně zastřešují komplexní správu kontejnerů a všechny související procesy. (Turnbull, 2019)

O jednotlivých součástech Dockeru pojednávají následující odstavce.

Docker Engine

Je jádrem Dockeru, jde o runtime fungující jako klient-server aplikace, stará se o správu kontejnerů, obrazů, sítí a svazků. Jedná se o páteř celého Dockeru umožňující vytvářet, transportovat a provozovat kontejnery v různých prostředích.

Docker Images

Jsou to malé, samostatné, spustitelné softwarové balíčky, které obsahují vše potřebné ke spuštění určitého softwaru, včetně kódu, runtime, systémových nástrojů, knihoven a nastavení. Z obrazů se za běhu stávají kontejnery, které zapouzdřují celou aplikaci a její prostředí.

Kontejnery

Jedná se o runtime instance Docker obrazů. Jsou to izolovaná prostředí, kde běží aplikace. Tato izolace umožňuje efektivní využití zdrojů a zajišťuje konzistentní běh aplikací v různých výpočetních prostředích.

Dockerfile

Je textový dokument obsahující všechny příkazy, které by standardně uživatel musel volat postupně na příkazovém řádku k sestavení obrazu. Pomocí Dockerfile může Docker vytvářet obrazy automaticky načtením veškerých instrukcí ze souboru.

Docker Compose

Je nástroj pro definování a spouštění vícekontejnerových aplikací. Pomocí jediného příkazu následně mohou uživatelé vytvořit kompletně definované a vzájemně propojené aplikační prostředí s více kontejnery, což usnadňuje správu složitějších aplikací.

Tato architektura a komponenty představují celou podstatu Dockeru, přičemž vše je navrženo se zaměřením na jednoduchost, efektivitu a škálovatelnost. Tyto technologie umožňují vývojářům vytvářet, nasazovat a škálovat aplikace rychle a bezpečně v rámci kontejnerů a poskytují nezbytný základ pro pipeliny zastřešující kontinuální integraci a kontinuální doručování (CI/CD). (Turnbull, 2019)

3.2.2 Kubernetes

Kubernetes je open-source platforma navržená pro automatizaci nasazení, škálování a celkovou správu a orchestraci aplikačních kontejnerů napříč hostitelskými clustery. Tato technologie výrazným způsobem změnila prostředí cloud computingu a posunula možnosti správy kontejnerů. Architektura a ekosystém Kubernetes nabízí robustní řešení pro snadnou a efektivní správu, nasazení a škálování kontejnerizovaných aplikací. (Burns, 2019)

U zrodu Kubernetes stál Google, který tento orchestrační nástroj představil v roce 2014, od té doby technologie roste ve velké míře i díky komunitě, která se na jejím vývoji podílí. (Burns, 2019)

Koncept Kubernetes je vhodný pro mnoho různých řešení, využít lze jak pro stateless tak stateful aplikace, podporuje tradičnější monolitické architektury ale velmi vhodný je zejména pro aplikace postavené na takzvané mikroservisní architektuře. Nabízí podporu kontinuální integrace a kontinuálního nasazení (CI/CD), a například oproti jiným orchestrátorům umožňuj přímou adresovatelnost kontejnerů tím, že každý kontejner má vlastní přiřazenou IP adresu. (Poulton, 2021)

Podobně jako v případě Dockeru i chod Kubernetes clusteru a celkovou architekturu řešení zastřešuje několik komponent, o kterých pojednávají následující podkapitoly.

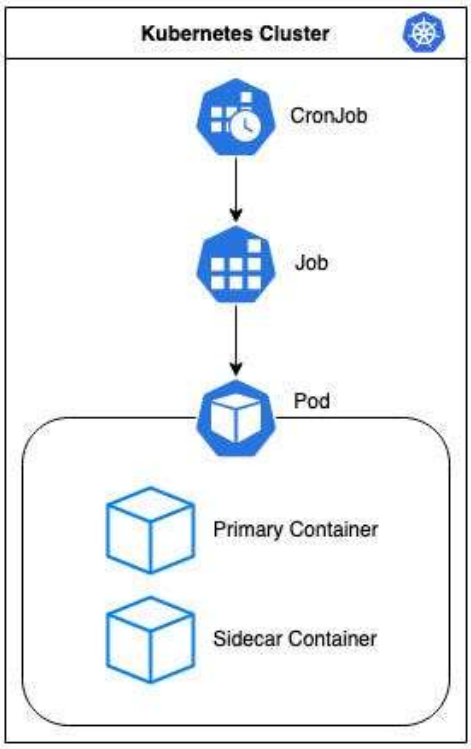
Pod

Je nejmenší jednotkou, která může být vytvářena, plánována a spravována. Jedná se o základní stavební kámen celých Kubernetes. Uvnitř podu může být začleněno vícero kontejnerů, navíc jsou jeho součástí sdílené zdroje jako síť či úložiště. Kontejnery uvnitř jednoho Podu jsou vždy společně umístěné, plánované a spouštěné ve sdíleném kontextu v rámci jednoho nodu, mají společný životní cyklus. (Poulton, 2021)

Job

V Kubernetes se využívá k vytvoření jednoho či vícero Podů za účelem vykonání nějaké jednorázové úlohy. Zajišťuje, že specifický počet Podů úspěšně splní svůj běh, poté je činnost Jobu považována za dokončenou. Zpravidla se využívají pro dávkové zpracovávání dat nebo časově omezené úlohy. Specifickým druhem je **CronJob**, což je v praxi jen rozšíření umožňující plánování běhu Jobů na nějaké pravidelné bázi, v určité časy či intervaly. Funguje na podobném principu jako cron úlohy v Unix/Linux systémech. (Maina, 2024)

Hierarchii mezi CronJobem, Jobem, Podem a kontejnerem v rámci Kubernetes clusteru vyobrazuje pro lepší představu následující obrázek.



Obrázek 2 - hierarchie CronJob-Job-Pod; zdroj: (Aldinger, n.d.*)*

Service

Jedná se o abstrakci definující logický soubor Podů ve formě síťové služby a slouží jako gateway pro přístup k Podům, včetně definice pravidel tohoto přístupu. Funguje také jako takzvaný load balancer mezi všemi zahrnutými pody. Services se dále dělí na čtyři specifické podtypy – ClusterIP (výchozí typ), NodePort, LoadBalancer a ExternalName. (Poulton, 2021)

ReplicaSet

Jde o kontroler, který má za úkol správu sady identických Podů a zajišťuje, že v daný čas běží specifikovaný počet těchto Podů. V případě potřeby přebytečné Pody ukončuje, pokud naopak není v provozu dostatečný počet Podů, spouští nové. Tento princip primárně zaručuje dostupnost a podporuje výkon. (Poulton, 2021)

Deployment

Stejně jako ReplicaSet má na starosti správu životního cyklu Podů, přičemž funguje na vyšší úrovni abstrakce. Poskytuje deklarativní aktualizace pro aplikace, umožňuje tím jejich snadné škálování, samoopravování a průběžné aktualizace. Využívá ReplicaSets pro udržování aplikací v požadovaném stavu a počtu vytvářením, aktualizací a odstraňováním Podů. Navíc poskytuje dodatečné funkcionality jako rollback či „rolling updates“ neboli průběžné aktualizace, čímž umožňuje takzvaný „zero-dowtime deployment“, česky jinými slovy nasazení bez ovlivnění dostupnosti celé služby. (Poulton, 2021)

Volume

Spravuje datová úložiště a zajišťuje, že jsou data zachovaná během celé životnosti Podu i po jeho expiraci. Umožňuje napojení lokálního, cloudového i síťového typu úložiště do Podu. Data ve Volumes jsou uchována i po restartu kontejneru nebo jeho smazání. Tato úložiště jsou sdílena mezi všemi kontejnery uvnitř Podu. (Poulton, 2021)

Namespace

Poskytuje mechanismus, jak v rámci jednoho Kubernetes clusteru izolovat a spravovat skupiny zdrojů mezi více týmy či projekty využívajících daný cluster. Namespaces zajišťují, že uvnitř clusteru může pracovat více uživatelů nezávisle na sobě tak, že se vzájemně

neovlivňují, mají každý svůj přidělený počet zdrojů i vlastní politiky přístupu. (Poulton, 2021)

3.2.3 Helm

Běžně bývá označován jako správce balíčků pro Kubernetes, jde o technologii ve velké míře řešící časté složitosti spjaté s nasazováním a správou aplikací v ekosystému Kubernetes. Je navržen tak, aby zjednodušil vývojářům celkovou administrativu okolo nasazování a správy aplikací s ohledem na to, aby zároveň dokázal využít plný potenciál Kubernetes. (Block, 2022)

Jádrem celého Helmu jsou takzvané „charts“, jde o strukturované balíčky předem konfigurovaných Kubernetes zdrojů, které zapouzdřují veškeré potřebné komponenty aplikace v rámci Kubernetes. Helm Charts jsou uloženy a sdíleny prostřednictvím Helm repozitářů a představují efektivní systém pro správu Kubernetes aplikací. Charts se skládají z několika souborů které společně definují co a jak se má do Kubernetes nainstalovat. (Butcher, 2021)

Nejdůležitější souborem celého řešení je Chart.yaml. Obsahuje samotnou definici a metadata celého chartu. Jsou v něm uvedeny informace o názvu, verzi, může obsahovat popis a klíčová slova či zdroje. Zásadní jsou však atributy name a version. Díky tomu je následně možné identifikovat balíček a dále s ním pracovat. V souboru values.yaml se poté definují výchozí hodnoty pro konfigurovanou službu, je důležité i v tomto souboru po celou dobu dodržovat název aplikace, který byl definován v souboru Chart.yaml. Pomocí vlastností definovaných ve values.yaml se následně řídí chování celé aplikace. (Butcher, 2021)

Helm rovněž umožňuje integraci s Azure DevOps pipelines, což usnadňuje celkově automatizaci nasazení a správy aplikací a podporuje CI/CD metodiky vývoje.

3.3 Testování softwaru

Testování je nezbytnou složkou procesu tvorby softwaru a mělo by probíhat kontinuálně během celého cyklu jeho vývoje – od počáteční specifikace požadavků až po finální dodání klientovi a následnou údržbu. Samotné testování softwaru je přitom jen podmnožinou oblasti zajišťování kvality softwaru, známé v anglickém jazyce jako "Quality Assurance", odkud také vychází běžně užívaná zkratka QA.

Cílem tohoto procesu je hodnotit softwarový produkt s cílem zjistit, zda vyhovuje stanoveným kritériím a požadavkům. Proces zahrnuje analýzu různých aspektů produktu, jako je jeho funkčnost, použitelnost, bezpečnost, spolehlivost a výkonnost, a je rozčleněn do různých kategorií v závislosti na technice testování, znalosti kódu a fázi vývojového procesu, v níž se testování uskutečňuje.

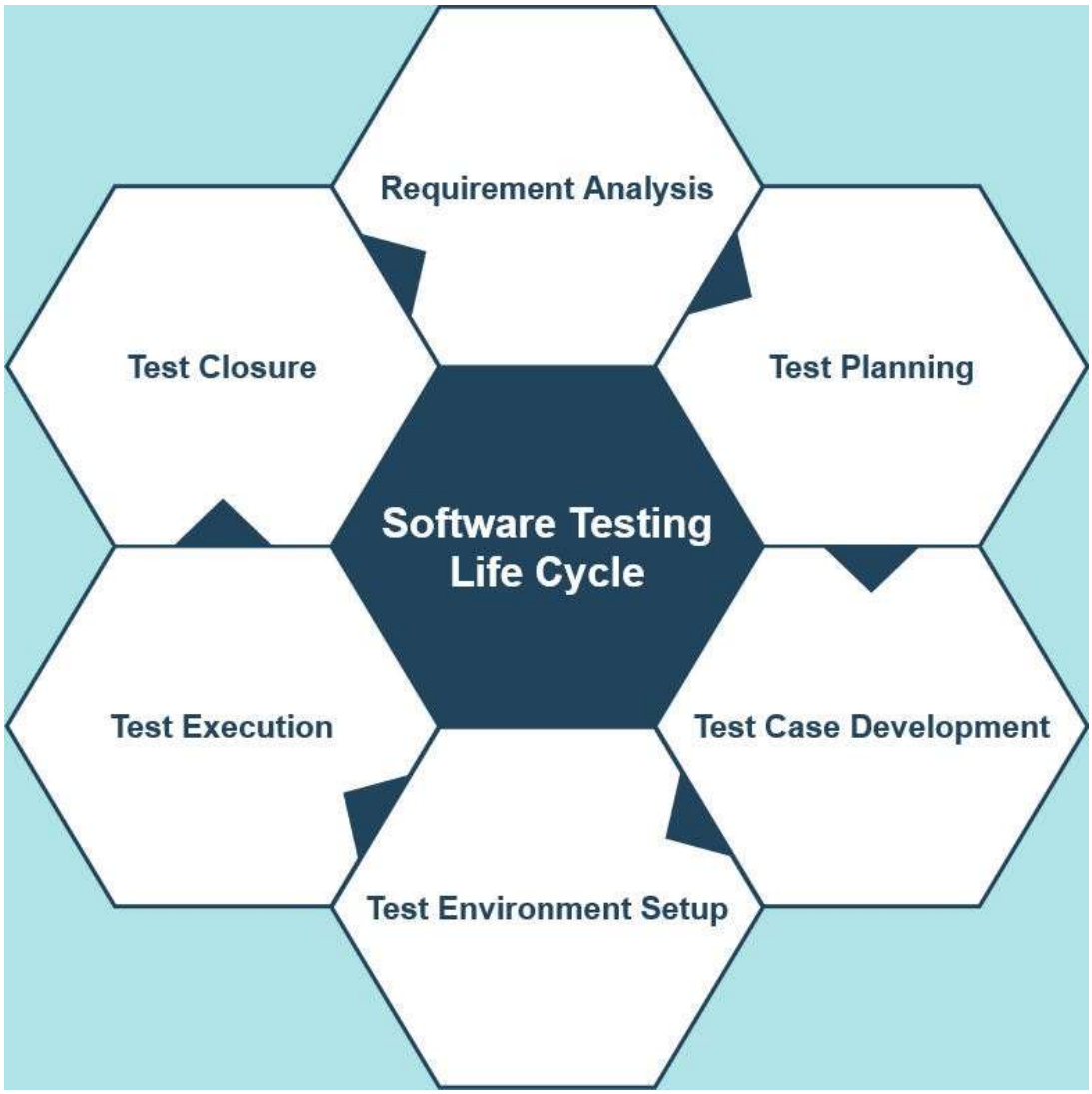
Záměrem testování je zejména ověřit, zda software splňuje všechny definované požadavky a identifikovat co největší počet nedostatků a chyb, které software obsahuje. Přestože je cílem odhalit co možná největší počet chyb, je třeba si uvědomit, že dosažení stavu, kdy je software zcela bez chyb, není možné.

Často se stává, že opravením jedné chyby vzniknou chyby zcela nové. Docílit bezchybnosti v praxi zpravidla není možné a v každém produktu se vždy budou vyskytovat nějaké chyby. (Kitner, 2017)

3.3.1 Životní cyklus testování softwaru

Životní cyklus testování softwaru (z angličtiny zkráceně STLC) označuje proces, který se skládá z jednotlivých kroků, které je třeba v konkrétním pořadí provést, aby bylo zajištěno, že produkt splňuje zadané požadavky a další nároky na kvalitu. STLC se může lišit napříč organizacemi, základ je ale vždy stejný.

Každá fáze má určitá vstupní a výstupní kritéria. Za ideálních podmínek by se nemělo přecházet k další fázi do té doby, než jsou splněna výstupní kritéria z fáze předchozí. Toho však v praxi občas nelze dosáhnout. (Software Testing Life Cycle (STLC), 2019)



Obrázek 3 - Vizualizace fází životního cyklu testování softwaru; zdroj: (Hartanto, n.d.*)*

Analýza požadavků

V úvodní fázi je klíčové provést detailní analýzu zadání projektu. Tester obvykle vychází z business analýzy, která jasně definuje, co se od výsledného produktu očekává. Hlavním úkolem testera je zjistit, zda jsou všechny požadavky testovatelné. V této fázi lze diskutovat s ostatními zainteresovanými stranami o jakýchkoliv nejasnostech či požadavcích, které nelze testovat. Požadavky lze dělit na dvě skupiny – funkční, týkající se toho, jak má produkt pracovat, nebo nefunkční, související například s designem, výkonem či bezpečností. (Ganji, 2023)

Plánování testování

Druhá fáze spočívá ve vytvoření testovacího plánu, což je zásadní část celého procesu. V této fázi se stanovuje strategie testování a na základě testovacího plánu se řídí veškeré další postupy. Je nutné určit rozsah testování, aby byly zahrnuty všechny specifikované požadavky. Plán musí obsahovat i odhady potřebných zdrojů – jak lidských, tak časových. Přesný časový odhad je důležitý zejména pro projektové řízení. Dále se v plánu specifikují používané testovací nástroje, definuje se testovací prostředí a určí se potřebná testovací data. (Ganji, 2023)

Návrh testovacích scénářů

Po sestavení testovacího plánu přichází na řadu vytváření testovacích scénářů. Každý scénář podrobně popisuje postup, kterým se testuje určitá funkcionalita systému, s cílem dosáhnout očekávaného výsledku. Pro každý požadavek by měl existovat alespoň jeden testovací scénář. (Ganji, 2023)

Pokud je například předmětem testování nákupní aplikace pro e-shop a požadavkem je možnost vytvářet nákupní seznamy, bude existovat příslušný testovací scénář krok po kroku popisující, jak v katalogu najít specifické produkty a proklikat se postupně až do stavu, kdy je vytvořený nákupní seznam obsahující několik produktů.

Příprava testovacího prostředí

Nastavení správného testovacího prostředí je zásadní pro celý proces testování, protože určuje podmínky, v nichž se bude produkt testovat. Proces zahrnuje definici, proti jaké hardwarové a softwarové konfiguraci se budou testy provádět. Často při tomto procesu

pomáhají i členové vývojového týmu a databázoví specialisté, v závislosti na specifických požadavcích. (Ganji, 2023)

Je velmi důležité mít testovací prostředí dobře nastavené, zvláště když se testují nové funkce existujícího softwaru. Existuje totiž riziko, že by se mohlo nedopatřením testovat na produkčním prostředí, právě kvůli chybě v nastavení, což by mohlo následně nepříznivě ovlivnit systémy využívané skutečnými uživateli. To by mohlo vést k vážnějším problémům než testování v izolovaném prostředí, kde mají přístup pouze členové testovacího a vývojového týmu.

Provedení testů

Exekuce samotných testů přímo závisí na předchozí přípravě scénářů a nastavení testovacího prostředí. Testeři v této fázi provádějí plánované testy a sledují, zda jsou výsledky v souladu s očekáváním. Test je považován za úspěšný, pokud se podaří úspěšně dokončit všechny kroky testovacího scénáře a dosáhnout očekávaného výsledku. Pokud však tester narazí na odchylku od očekávaného výsledku v jakémkoliv kroku, je test považován za neúspěšný a je důležité toto zaznamenat a podat chybové hlášení. Chybové hlášení se následně přiřadí k příslušnému scénáři, aby bylo patrné, že spolu tyto dvě věci souvisí. Úkolem vývojáře je pak najít a opravit příčinu chyby. Poté tester opětovně prověří scénář, aby zjistil, zda byla chyba opravena. (Ganji, 2023)

Vyhodnocení výsledků

Po dokončení všech testů se vyhodnocují a jsou shrnuty výsledky celého procesu testování. Vyhodnocuje se úspěšnost provedených testů a porovnává se skutečná časová náročnost testování s původními odhady. U testů, které nedopadly úspěšně, se očekává oprava chyb, případně se dle potřeby upravují testovací scénáře, pokud problém nespočívá v chybném softwaru, ale v chybě v návrhu scénáře. (Ganji, 2023)

Pokud je to nutné, může proběhnout i schůzka, na které se diskutuje o úspěšnosti testování a případných nedostatcích testovacího procesu, s cílem identifikovat oblasti pro zlepšení v budoucnu.

3.3.2 Způsoby testování

Rozlišuje se v zásadě mezi dvěma hlavními způsoby, kterými lze realizovat testování softwaru. Jejich charakteristiku, výhody i nevýhody popisují následující podkapitoly. (Kitner, 2017)

Manuální testování

Manuální testování bylo v podstatě popsané v kapitole „Životní cyklus testování softwaru“. Jde o testy, které ručně provádí sám tester dle scénářů. Může být výhodné například při testování uživatelského rozhraní (UI), kde je lidský instinkt stále nenahraditelný. Zároveň se při manuálním testování dá lépe zjistit už z pohledu testera, kde přesně vzniká příčina určité chyby – člověk dokáže sledovat a vnímat souvislosti. Lidský faktor zde ale může být i negativem, jelikož může nějaké věci přehlédnout, špatně pochopit či na jejich otestování úplně zapomenout. Manuální testování tak nikdy nelze považovat za 100% spolehlivé.

Speciálním druhem manuálního testování je pak takzvané explorativní testování. Od klasického manuálního se liší v tom, že se tester neřídí předepsanými scénáři, ale prozkoumává přirozeným způsobem daný software, čímž se snaží zjistit, jak se daný software chová, jaké má vlastnosti a funkce. I při explorativním testování přitom může dojít k nalezení mnoha chyb. (Kitner, 2017)

Automatizované testování

Automatizace je považována za nejpokrokovější disciplínu testování softwaru, ačkoliv má již poměrně bohatou historii. Automatizované testování je prakticky kód, který kontroluje kód. Jde o krátké programy, psané v různých jazycích, které mají za úkol ověřit určitou funkcionalitu.

Jeho výhoda je především v rychlosti. To, co by tester procházel několik minut, má automatizovaný test mnohdy hotové jen za několik sekund. Kromě toho jsou testy jednoduše škálovatelné a zpravidla i cenově výhodnější.

Nevýhoda automatizace může být například ve vysokých počátečních nákladech na její zavedení. Samotná údržba testů a jejich konfigurace zabírá mnohdy spoustu času, zejména pokud se testovaný software často mění. S každou změnou podoby softwaru je třeba zrevidovat i související automatizované testy, aby skutečně kontrolovaly to, co mají.

Na rozdíl od manuálního testera nedokáže automatizovaný test posoudit faktory UX, takže v tomto ohledu stále vítězí lidský přístup. Automatizace může být také poměrně neefektivní pro menší software, u kterého by vývoj automatizovaných testů zabral neúměrně dlouhou dobu a nevyplatil by se oproti manuálnímu otestování. (Borovcová, 2008)

Automatizace nalezne dobré uplatnění zejména u regresních testů, které vyžadují opakované vykonávání těch samých testovacích scénářů na pravidelné bázi. Manuální testeři mohou takto opakovanou činností časem začít přehlížet i dobře patrné chyby, test již nemusí provádět s takovou pečlivostí a rychlostí, jako z počátku. Oproti tomu automatizované testy dodávají konzistentní výsledky, jejich exekuce trvá pokaždé přibližně stejnou dobu a nemůže se stát, že by takový test přehlédl nějakou chybu.

Při volbě testů ke zautomatizování je také dobré brát v potaz jejich prioritu z hlediska businessu a z technického hlediska. Pokud je testovací případ příliš složitý pro manuální testování a lze jej zautomatizovat, je určitě vhodné to zvážit. To samé v případě, že je testovaná oblast z pohledu businessu klíčová a její funkčnost je stěžejní pro daný software. (Bose, 2023)

Pro automatizaci testů slouží specializované nástroje, lišící se v několika oblastech. Rozdílný je například způsob zápisu testů. Některé nástroje dokážou zaznamenávat akce prováděné ručně uživatelem a ty následně automaticky reprodukovat, běžnější jsou však nástroje, kde se testy zapisují skriptovacím či plnohodnotným programovacím jazykem.

Jednotlivé automatizační softwary většinou dokážou zajistit automatizaci jen pro jednu platformu. Pokud tedy například firma vyvíjí program pro desktop, web i mobilní zařízení, je zpravidla nutné pro automatizaci na každé platformě volit specifický automatizační nástroj. (Testim, 2022)

3.4 Business intelligence

Společnosti po celém světě napříč všemi segmenty podnikání v dnešní době generují obrovské množství různorodých dat – schopnost tato data efektivně sbírat, analyzovat a interpretovat představuje stěžejní konkurenční výhodu. Business intelligence a reporting představují základní metodologie umožňující firmám převádět data na využitelné poznatky, čímž umožňují vykonávání důležitých rozhodnutí a přispívají k vyšší provozní efektivitě.

Oblast business intelligence zahrnuje široké spektrum nástrojů, aplikací a zavedených postupů, které dohromady zajišťují sběr dat, integraci, analýzu a prezentaci podnikových informací. Cílem toho všeho je podpora lepšího rozhodování.

S růstem společností a rozvojem jejich podnikání však rostou i jejich data, stávají se komplexnější a postupem času je čím dál složitější se v nich orientovat. Role BI tím pádem začíná přerůstat přes klasickou kompilaci dat a jejich interpretaci – firmy jsou nuceny inovovat v této oblasti a využívat různé techniky prediktivní analýzy či data mining, což vede k dynamičtějšímu a předvídavějšímu řízení podniku. (LAURSEN, 2017)

3.4.1 Reporting

Reporting lze jinými slovy označit i jako vizualizaci informací. Jeho účelem je přeměňovat určitá data ve znalosti, což se v závislosti na konkrétní situaci může lišit rozsahem a náročností jeho tvorby.

Je rovněž důležité vyjasnit si v úvodu rozdíl mezi pojmy „report“ a „dashboard“, je totiž velice jednoduché tyto pojmy vzájemně zaměňovat, jelikož hranice mezi nimi je velmi tenká. Report je klíčovým prvkem v rámci business inteligence a lze jej popsat jako dokument obsahující data a analýzy, které jsou určeny pro periodické čtení. Dashboard naproti tomu poskytuje interaktivní přehled klíčových ukazatelů v reálném čase a je určen pro kontinuální sledování. (Hroch, 2008)

3.4.1.1 Uživatelé reportingu

V základu je možné uživatele reportingu rozdělit na interní a externí. Za interní uživatele se považují zaměstnanci společnosti, kteří využívají reporting pro sledování výkonnosti, optimalizaci procesů, nebo pro strategické rozhodování. Externí uživatelé mohou zahrnovat investory, regulátory, nebo partnery, kteří reporty využívají pro hodnocení

výkonnosti společnosti, soulad s regulacemi, nebo pro posouzení potenciálu pro spolupráci. (Hroch, 2008)

3.4.1.2 Účel reportingu

Hlavním účelem reportingu je poskytnout uživatelům informace potřebné k podpoře rozhodování. To zahrnuje identifikaci trendů, měření výkonnosti oproti cílům, odhalování oblastí pro zlepšení, a sdílení poznatků s ostatními členy organizace nebo s externími partnery. Reporting také slouží k zajištění transparentnosti a dodržování regulatorních požadavků. (Hroch, 2008)

3.4.1.3 Druhy reportingu

Statický

Je vhodný v případě, kdy vizualizované informace mají jednotnou strukturu a fixní vstupní parametry. Zpravidla se jedná o statický dokument prezentující nějakou ucelenou sadu informací. V případě obchodu se může jednat například o přehled stovky nejprodávanějších produktů za poslední měsíc, v bankovním sektoru třeba přehled deseti nejvýkonnějších bankéřů z hlediska nově sjednaných produktů. (Hroch, 2008)

Dynamický

Dynamický reporting se v mnohém podobá statickému, liší se však tím, že uživatel může úpravou vstupních parametrů měnit konkrétní podobu a obsah reportu.

V případě výše uvedeného příkladu s přehledem o nejprodávanějších produktech by to znamenalo, že uživatel si v rámci reportu může zvolit, jak dlouhé časové období chce sledovat, že ho zajímají například jen specifické kategorie produktů, případně jen určitá skupina zákazníků. (Hroch, 2008)

Oproti statickému reportu tedy u dynamického není člověk omezen předem definovanou formou a mnohdy si může přizpůsobit i design reportu. Výhodou je zkrátka přizpůsobení potřebám konkrétního uživatele.

3.5 Reporting v kontextu testování softwaru

Reporting je v oblasti testování softwaru důležitým aspektem a hraje zásadní roli v rámci životního cyklu testování softwaru, jelikož test reporty poskytují cenný vhled do celého průběhu testování. Zpravidla informace v nich obsažené pomáhají ve výsledku činit důležitá rozhodnutí – například zda je testovaná aplikace připravena k vydání do produkce či nikoliv. Reporty rovněž zpřístupňují proces testování a výsledky všem zainteresovaným stranám, ať už přímo v IT či dalších odděleních. Mají však mnohem širší uplatnění. (Brik, 2020)

Periodicita poskytování test reportů může být různá dle toho, jaká je testovací strategie, případně dle typu a podmínek konkrétního vývojového projektu. Například pro regresní testování je typické velmi časté reportování o jeho aktuálním stavu a takové reporty mohou být poskytovány na denní či ještě častější bázi. Není výjimkou ani prakticky nepřetržitý monitoring průběhu regresních testů. (Li, 2023)

3.5.1 Hlavní funkce

Je možné říct, že test reporting spočívá primárně ve shromažďování, analyzování a prezentování důležitých dat o testování a výsledků stakeholderům. Pod stakeholdery si přitom nutně není třeba představovat jen úzké vedení společnosti, jedná se rovněž o vývojáře, samotné testery, product ownery, projektové manažery či kohokoliv dalšího, kdo je určitým způsobem zainteresovaný na vývoji daného produktu. (Li, 2023)

3.5.1.1 Sběr dat

V průběhu testovacího procesu dochází ke generování obrovského množství dat, jako jsou výsledky prováděných testů, hlášení o chybách, metriky ohledně pokrytí testů a podobně. Kromě výsledků jednotlivých testů zahrnuje sběr dat i získávání dalších údajů, jako informace o testovacím prostředí, specifické konfigurace hardwaru a softwaru a další parametry, které napomáhají pochopit kontext podmínek, za kterých testování probíhalo. Díky tomu je možné odhalit potenciální specifické faktory prostředí, které mohou ovlivnit výsledky testů. (Li, 2023)

3.5.1.2 Analýza dat

Sesbíraná data je nutno podrobit důkladné analýze za účelem odvození nějakých smysluplných poznatků. Cílem je identifikace trendů, vzorů a korelace v datech. Ve výsledku taková analýza poskytuje detailní vhled do testovaného softwaru a pomáhá porozumět jeho výkonu, díky čemuž je možné odhalit potenciální problémové oblasti. Z analýzy je například možné vypozorovat narůstající počet defektů okolo specifické funkční oblasti softwaru, v návaznosti na to pak lze podniknout nutné kroky k dalšímu šetření, nápravě a opatření ke zlepšení kvality konkrétní komponenty. (Li, 2023)

3.5.1.3 Prezentace dat

Finálním krokem je prezentace analyzovaných dat v nějakém srozumitelném a jasně čitelném formátu, aby bylo možné se v poskytnutých informacích snadno a rychle orientovat. K tomu v rámci reportingu napomáhá řada vizualizačních nástrojů, jako jsou tabulky, grafy či dashboardy. Toto všechno poskytuje souhrnný pohled na výsledky testování a snadná orientace v datech umožňuje urychlit další rozhodovací proces. Mnohdy mohou test reporty zahrnovat i textovou sumarizaci, a to především v případech, kdy prezentovaná data potřebují nějaké dodatečné vysvětlení pro lepší pochopení.

Vzhledem k tomu, že reporty mohou být dostupné širokému publiku, měly by být také uzpůsobené tomu, že do nich budou nahlížet lidé různých rolí, přičemž každého může zajímat trochu jiný údaj. (Brik, 2020)

3.5.2 Metriky a kvalita

Měření kvality je důležité hned z několika důvodů, vývoj softwaru stojí velké množství času i peněz a je spojen se značnými riziky, zpravidla také software generuje firmě nějakou formou zisk, ať už danou aplikaci používají její zaměstnanci či zákazníci. Je tedy třeba se pravidelně ujišťovat, že vývoj a výsledná aplikace přináší očekávané výsledky. (Roudenský, 2013)

Je několik důvodů, proč je kvalita softwaru měřena, z těch nejdůležitějších například:

1. Získání informací o aktuálním stavu.
2. Sledování postupu v porovnání s plánem.
3. Odhalení rizik.
4. Motivování týmu k lepším výkonům.

5. Měření může být podkladem pro návrh změny procesu.
6. Měřením bráníme chaosu, neměřené oblasti totiž mohou být nehlídané, neřízené nebo i v úpadku.

Předmětem měření mohou být různé aspekty vývoje softwaru, většinou je třeba se při jejich výběru řídit tím, kdo je primárně cílovým konzumentem. Jiné údaje budou zajímat projektového manažera, jiné zase vedoucího vývoje.

Metriky se dělí na tvrdé a měkké podle toho, jak přesná jsou data, ze kterých vycházejí. Tvrdé metriky znamenají přesné hodnoty, například počet provedených testů, čas strávený vykonáváním jednotlivých testů a podobně. Měkké metriky jsou naopak nějaká subjektivní hodnocení, jako třeba spokojenost uživatelů s designem aplikace. Zatímco u měkkých metrik se výsledky jednotlivých měření mohou lišit v závislosti na tom, kdo zrovna danou problematiku posuzuje, jaké má preference či v jakém je rozpoložení, tvrdé metriky budou vždy jednoznačné, neovlivněné osobou provádějící měření, a především vycházejí z přesných údajů získaných v průběhu vývoje či testování. (Roudenský, 2013)

4 Vlastní práce

Následující kapitoly budou věnovány praktické realizaci zadání. Práce volně navazuje na autorovu bakalářskou práci, která se věnovala problematice automatizace v testování webových aplikací a kde v praktické části bylo několika aktivitami, primárně automatizací několika testovacích případů, přispěno k rozšíření automatizačního testovacího frameworku ve vybrané společnosti.

Nyní v rámci diplomové práce autor ve stejné společnosti (Alza.cz a.s.) zpracovává problematiku reportingu v testování softwaru. Primárním cílem je vyvinout nástroj, který dokáže zpracovat vybraná data o testování a následně tato data zpřístupnit pro analýzu pomocí reportovacích či monitorovacích nástrojů třetích stran. K naplnění vytyčeného cíle je nutné udělat několik činností a splnit různé dílčí povinnosti.

Nejprve musí být dobře specifikováno, co firma od výsledného řešení očekává, jaké má na aplikaci požadavky, podle čehož lze následně uvažovat nad architekturou celého řešení.

Je rovněž nezbytné zanalyzovat, jaká konkrétní data jsou pro firmu, respektive QA oddělení, pro potřeby reportingu a monitoringu důležitá. Pro účely práce je především důležité zjistit, s jakým datovým zdrojem či datovými zdroji se bude pracovat, a rovněž v jakém jsou tato data formátu. Rovněž je důležité se zamyslet nad způsobem získávání dat a případně periodicitou.

Dále je třeba navrhnout způsob ukládání získaných dat. Data by měla být uložena strukturovaně, bezpečně a mělo by být umožněno k nim snadno přistupovat, ať už napřímo či skrze požadované reportovací a monitorovací nástroje.

Rovněž je nezbytné zamyslet se nad tím, jak budou data do externích systémů předávána, jelikož každý systém může mít své unikátní funkce a odlišný způsob řešení datových zdrojů.

Na základě celkové analýzy všech požadavků může být následně přistoupeno k návrhu a vývoji aplikace, která toto vše zastřeší.

4.1 Aktuální stav a motivace

Za účelem bližšího porozumění kontextu této práce považuje autor za důležité objasnit, jak ve firmě funguje vývoj a základní procesy v IT, zejména z pohledu QA oddělení a testování.

Vývoj software je ve společnosti Alza.cz a.s. řízen interně, což znamená, že celý proces od konceptualizace, návrhu, vývoje až po uvedení produktu na trh je realizován týmy, které jsou přímo součástí společnosti. Firma si zajišťuje interně vývoj drtivé většiny všech svých softwarových produktů, včetně ERP, logistických systémů, webové aplikace i mobilních aplikací pro platformy Android a iOS. Tento přístup umožňuje firmě dobře a pohotově reagovat na potřeby trhu, rychle implementovat změny a inovace a udržovat kontrolu nad kvalitou a bezpečností svých systémů.

Důležitou součástí vývojového cyklu jsou i aktivity quality assurance, od návrhu testů, plánu jejich exekuce až po samotné testování všech aplikací adekvátními metodami v různých fázích vývoje, což zajišťuje, že vše funguje podle očekávání a že nově vyvinutý software je připraven na produkční nasazení bez závažných chyb, které by mohly negativně ovlivnit zákaznickou zkušenost nebo provozní efektivitu. Na testování je ve společnosti kladen velký důraz a zahrnuje širokou škálu testovacích metodik, od manuálních testů po pokročilé automatizované regresní testy.

Celý proces testování generuje obrovské množství výstupů a dat, primárním výstupem jsou samotné výsledky provedených testů. Sledují se ale i další data a ukazatele, jako informace o počtu nalezených produkčních bugů mezi jednotlivými releasy, počet chyb nalezených během testování v rámci vývoje nebo čas strávený vykonáváním různých druhů testů. Tato data jsou neocenitelným zdrojem pro vedení QA a IT oddělení, které vyžadují pravidelné a podrobné informace o stavu testování. Zvláštní pozornost je věnována i analýze úspěšnosti automatizovaných regresních testů, které hrají klíčovou roli v udržení agility a efektivity vývojových cyklů společnosti.

Jedna z výzev, se kterou se však společnost potýká, je absence jednotného systému pro sledování a vyhodnocování dat z testování. I přes pokročilé postupy v oblasti vývoje a testování softwaru se jako základ pro různé reporty stále používá Microsoft Excel, kde se do tabulek ručně vyplňují různé hodnoty získané z jiných systémů, což je jednak časově náročné, a rovněž zde vzniká prostor pro lidskou chybu při přepisu jednotlivých údajů.

4.2 Specifikace požadavků

Jako vůbec první aktivita spojená s praktickou realizací práce proběhla počáteční schůzka a diskuse s vedením QA oddělení a seniorními členy, kde byly definovány požadavky na očekávaný produkt, tedy aplikaci pro reporting. Bylo důležité dobře specifikovat, co by měla aplikace umožňovat a jakým způsobem by měla pracovat. Výsledkem této schůzky byla formulace následujících požadavků:

- Spojení se službou Azure DevOps pro přístup k potřebným datům.
- Automatický sběr dat o vybraných výsledcích testování.
- Návrh a implementace funkcí pro reportování výsledků do různých monitorovacích a reportovacích BI systémů (Kibana, Grafana, Power BI).
- Průběžné poskytování výsledků v daném časovém intervalu.

Ze zmíněných bodů tedy vyplývá zejména explicitní důraz na konkrétní službu, která bude, alespoň v prvotní fázi projektu, poskytovat veškerá data. Očekává se automatizovaný běh aplikace, s tím že data budou poskytována pravidelně a průběžně. Při následném návrhu tedy bude nezbytné zamyslet se nad způsobem fungování aplikace a celkově její architekturou tak, aby umožňovala automatické spouštění například v určitých intervalech.

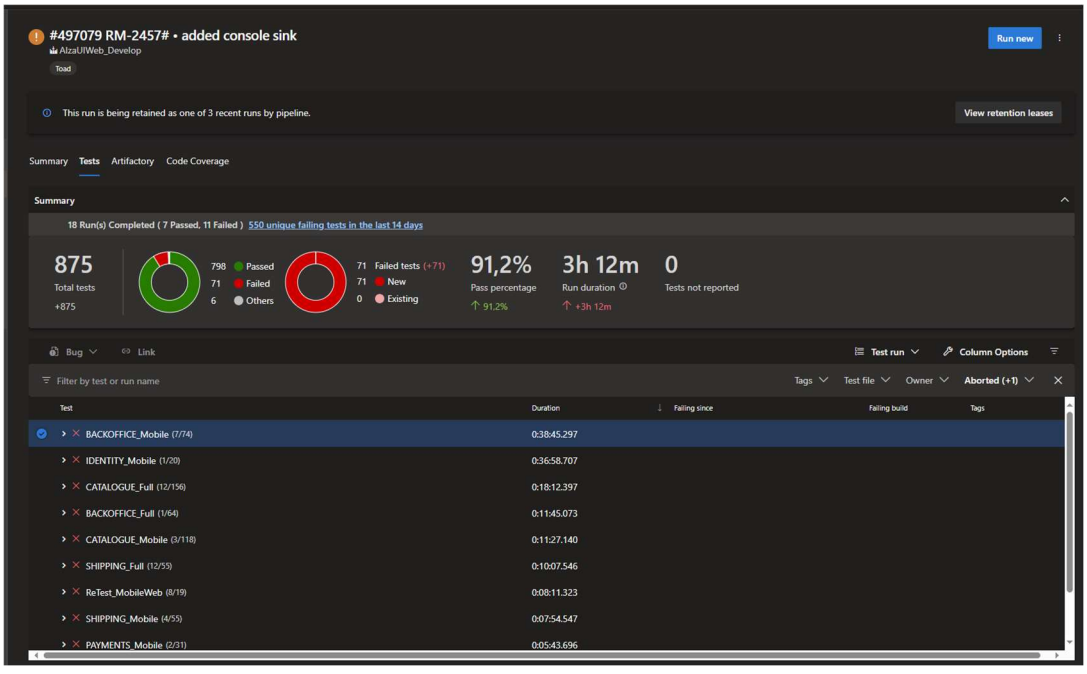
Očekává se, že bude možné nahlížet na data v reportovacích/monitorovacích systémech primárně v podobě interaktivních dashboardů a různých vizualizačních nástrojů, aby bylo možné rychle získat přehled o tom, jak se sledované testy vyvíjí v čase, kde jsou nejproblémovější oblasti a podobně.

4.3 Identifikace dat a datové zdroje

Primární datový zdroj vyplývá již ze specifikace požadavků a jedná se o službu Azure DevOps. Ta představuje sadu nástrojů od Microsoftu navrženou pro podporu vývoje softwarových projektů od plánování až po nasazení. Jedná se o cloudovou službu, kterou firma využívá pro správu většiny svých vývojových projektů. Umožňuje vytvářet, spravovat a nasazovat aplikace napříč platformami a prostředími.

Azure DevOps zahrnuje pět hlavních komponent – Azure Repos, Azure Pipelines, Azure Boards, Azure Artifacts a Azure Test Plans. Společně pokrývají celý životní cyklus vývoje softwaru, od správy verzí a projektového managementu po správu sestavení a nasazení, ukládání artefaktů a testování. Z pohledu této práce lze za nejdůležitější komponenty považovat Azure Test Plans a Azure Pipelines. Azure Test Plans nabízí komplexní sadu nástrojů a funkcionalit jak pro manuální, tak automatizované testování, včetně dokumentace testovacích případů, jejich vykonávání a sledování průběhu. Dobře se integruje se zmíněnými Azure Pipelines, což umožňuje právě automatizaci testovacích scénářů a celkově lepší navázání testování do celého procesu vývoje.

QA oddělení a weboví testeři využívají celou řadu pipeline, ať už pro spouštění různých sad automatizovaných testů, tak i pro generování takových sad, kontrolu testovacích dat a podobně. Jelikož výsledné reporty by měly sledovat primárně výsledky proběhlých testů, bude se dále práce zabývat ve velké míře touto oblastí.



Obrázek 4 - Přehled výsledků testů z jednoho běhu pipeline v Azure DevOps; zdroj: vlastní zpracování

Obrázek 4 na předchozí straně ukazuje, jak v prostředí Azure DevOps vypadá stránka s přehledem výsledků testů z jednoho běhu pipeline. Lze si všimnout i záložky se sumarizací výsledků a jednoduchými vizualizacemi. Azure sám o sobě poskytuje základní přehled o průchodnosti testů, postrádá však pokročilejší operace nad dostupnými daty. Uživatelé jsou omezeni na limitované možnosti filtrování a zdaleka ne vždy je možné z dat přímo v Azure vygenerovat požadovaný výstup. I toto byl jeden z motivačních faktorů pro vznik této práce, tedy dostupná data vhodným způsobem získat z Azure a umožnit jejich pokročilejší interpretaci pomocí jiných softwarových nástrojů.

Nejstěžejnější pro testování webu je regresní sada automatizovaných UI testů, což jsou testy vykonávané pomocí nástroje Selenium WebDriver, který dokáže interagovat s prohlížečem a procházet webovou stránku podobně jako uživatel dle v kódu nadefinovaných kroků. Běh testů lze skrze pipeline spouštět buď ručně, případně se dá nastavit i pravidelné automatické spouštění s předem pevně definovanou konfigurací.

Exekuce manuálních testů je neméně důležitá a rovněž je prováděna pomocí Azure, konkrétně přímo v nástroji Test Plans. I z těchto manuálních běhů jsou zaznamenávány výsledky a další detaily. Výsledky jak manuálních, tak automatizovaných testů, jsou v Azure uchovány ve stejném formátu, respektive Azure běhy obou typů testů uchovává stejným způsobem a lze k nim i jednotně přistupovat, přičemž jsou běhy vzájemně rozlišitelné skrze typ běhu specifikovaný jedním z atributů každého běhu.

Mimo prostředí Azure momentálně firma neprovádí exekuci testů a jedná se o centrální místo, skrz které probíhá evidence testovacích případů, exekuce všech typů testů, jejich výsledků a dalších náležitostí. Nicméně v době psaní této práce probíhají rané aktivity spojené s přechodem na zcela nový test management systém, zatím bez bližších podrobností. Tento fakt však motivuje k zamyšlení se nad tím, jakým způsobem aplikaci navrhnout, aby byla do budoucna připravena na zcela jiný zdroj dat, než je právě Azure.

4.4 Způsob ukládání dat a předávání externím systémům

V průběhu investigace celkové problematiky kolem reportovacích a monitorovacích nástrojů, které firma primárně využívá a v nichž by chtěla mít data přístupná, bylo zjištěno, že prakticky všechny fungují na principu, kdy se v samotném nástroji pouze definuje datový zdroj – odkud se mají data konzumovat. To celou situaci zkomplikovalo, respektive zjištění není v souladu s původním cílem a očekáváním, že vyvinutá aplikace bude schopna napřímo s těmito nástroji komunikovat a realizovat předávání dat. Z toho důvodu bylo nutné přehodnotit původní záměr a soustředit se dále v práci primárně na to, kam bude vytvářená reportovací aplikace získaná data ukládat.

Jako velmi vhodné řešení se nabízel ve firmě již zavedený nástroj Elasticsearch. Jedná se o open-source distribuovaný vyhledávací a analytický engine postavený na Apache Lucene, jenž je součástí a zároveň základem širšího balíčku produktů zvaného Elastic Stack. Primárně je navržen pro rychlé a škálovatelné vyhledávání v textech, ale díky své flexibilitě a schopnosti pracovat efektivně s JSON dokumenty a dalšími semistrukturovanými soubory se využívá i pro ukládání a analýzu velkého množství různorodých dat, lze jej tedy velmi dobře využít i pro účely této práce pro ukládání výsledků testování, které nejen Azure ale i jiné systémy zpravidla v tomto formátu generují.

Data v Elasticsearch jsou organizována do indexů, které si lze představit jako optimalizované databázové tabulky umožňující efektivní vyhledávání a agregaci dat v reálném čase. Ještě lépe terminologické rozdíly mezi relačními databázemi a Elasticem ilustruje následující obrázek.

Obrázek 5 - Relační databáze a Elasticsearch, porovnání názvosloví; zdroj: (Neves, 2019)

Aplikace Elasticsearch se často volí v situacích, kde je potřeba rychle vyhledávat v rozsáhlých datových souborech, jako jsou logovací systémy, e-commerce platformy pro vyhledávání produktů, nebo jako součást full-textového vyhledávání na webových stránkách. Díky dobrým schopnostem pro distribuované zpracovávání dat lze Elasticsearch snadno horizontálně škálovat, což z něj činí vhodné řešení například i pro práci s velkými daty. I kvůli těmto přednostem bylo zvoleno toto konkrétní řešení.

Dalším z důvodů je pak i fakt, že firma již Elastic Stack a jeho nástroje aktivně využívá, byť k jiným účelům (log management). Důležité je především to, že se nejedná o zcela novou technologii, kterou by bylo nutno implementovat a budovat od nuly, což by s sebou neslo kromě větší pracnosti zejména i vyšší náklady.

Alternativně by bylo možné řešit ukládání dat i pomocí klasické SQL databáze, to by však s sebou neslo spíše nevýhody než výhody, řešení by bylo pracnější, méně flexibilní a pravděpodobně i dražší jak na implementaci, tak na následnou údržbu. Následující seznam shrnuje hlavní přednosti zvoleného řešení oproti tradiční databázi a podtrhuje jedny z primárních důvodů, proč bylo zvoleno právě toto řešení.

1. **Rychlost vyhledávání**: Elasticsearch je optimalizován pro rychlé vyhledávání. To znamená, že pokud bude potřeba nějak pokročileji vyhledávat v uložených datech, například filtrovat podle specifických testů, jejich výsledků, dalších z mnoha evidovaných atributů a podobně, Elasticsearch bude velmi pravděpodobně poskytovat rychlejší výsledky než tradiční SQL databáze.
2. **Škálovatelnost**: Elasticsearch je navržen s důrazem na dobrou škálovatelnost a snadno se rozšiřuje na více nodů (uzlů), což umožňuje efektivní distribuci dat a zátěže v případě nějaké pokročilejší implementace a potřeby řešení v budoucnu rozšiřovat. Toto horizontální škálování je u tradičních SQL databází mnohem složitější a nákladnější na implementaci.
3. **Práce se semistrukturovanými daty**: Data nejen o výsledcích testování bývají většinou semistrukturovaná (např. formáty JSON, XML), což je přirozený formát pro Elasticsearch. Na rozdíl od tradičních SQL databází, které vyžadují pevně definované schéma tabulek, umožňuje Elasticsearch flexibilnější modelování dat, a především si dokáže vnitřní strukturu indexů sám nadefinovat podle struktury dokumentů, které mu jsou poskytnuty, tím odpadá jakákoliv nutnost navrhovat strukturu tabulek a datové typy.

4. **Analýza a agregace v reálném čase**: Elasticsearch poskytuje skrze Kibanu pokročilé možnosti pro real-time analýzu a agregaci dat, což je ideální pro generování různých přehledů, sledování trendů a identifikaci problémů v reálném čase.
5. **Full-textové vyhledávání**: V případě, že bude třeba provádět složité dotazy na textová data, například vyhledávat v textech chybových hlášení či dalších atributech jednotlivých výsledků, nabízí Elasticsearch pokročilé možnosti full-textového vyhledávání, včetně hledání s použitím synonym, stematizace a dalších textových analýz. To je něco, co SQL databáze sama o sobě poskytnout nedokáže.
6. **Snadná integrace**: Elasticsearch má širokou podporu pro integraci s různými nástroji a platformami, včetně mnoha cloudových služeb, což usnadňuje automatizaci procesů a workflow. Na data v Elasticu se dokáže dotazovat rovněž velké množství monitorovacích i business inteligence aplikací.

V rámci Elastic Stacku je mimo jiné k dispozici i webová aplikace Kibana, která slouží k průzkumu, analýze a vizualizaci dat uložených právě v Elasticsearch. Umožňuje vytvářet dynamické dashboardy zobrazující agregace dat, trendy či vzorce z dat uložených v Elasticsearch a sdílet je jednoduše dalším uživatelům. To rovněž ve velké míře podporuje rozhodnutí ukládat data právě do Elasticu, jelikož již v rámci tohoto řešení je k dispozici profesionální vizualizační nástroj pro tvorbu různých dashboardů a reportů, bez nutnosti řešit napojení dalších externích reportovacích systémů.

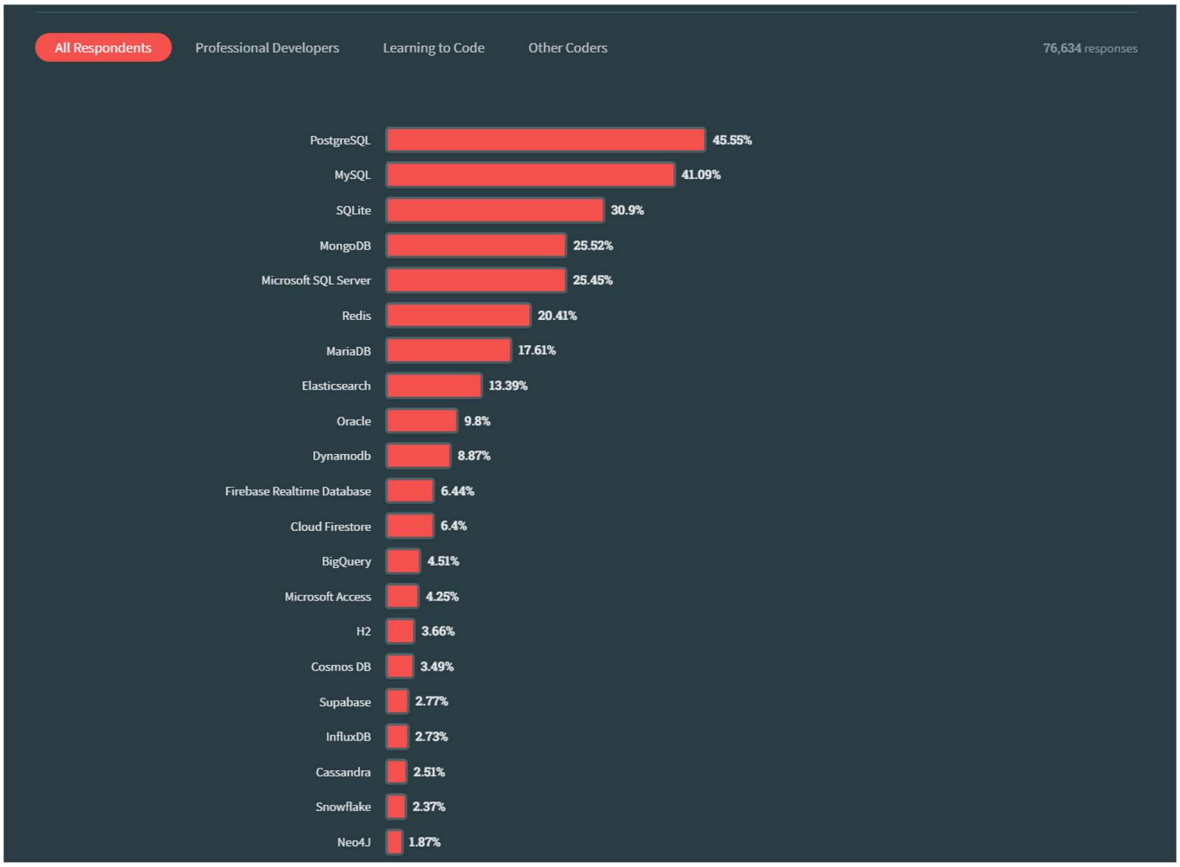
Jak již vyplynulo z popisu vlastností a výhod Elasticu, není samozřejmě Kibana jedinou možností, jak pracovat s uloženými daty. Velmi jednoduše se Elastic napojuje například na Grafanu, což je další firmou využívaný webový software pro analýzu a vizualizaci dat. Grafana již umožňuje třeba rozsáhlé možnosti konfigurace upozornění podle nastavených podmínek. Vybraní uživatelé tedy mohou obdržet upozornění například ve chvíli, kdy je zaznamenáno nějaké větší množství neprocházejících testů v jednom běhu a podobně.

Dále stojí za zmínku Power BI jako jeden z nejvýznamnějších zástupců reportovacích nástrojů a programů z oblasti business inteligence. I do Power BI se dá jako datový zdroj napojit Elasticsearch, nejedná se ale v tomto případě přímo o nativní podporu. Je nutné instalovat Elastic ODBC Driver, který umožňuje aplikacím podporujícím standard Open

Database Connectivity komunikovat s Elasticsearch. Power BI umožňuje využít obecně ODBC jako datový zdroj, zmíněný driver tedy umožní následně komunikaci těchto dvou nástrojů a následnou práci s daty z Elasticu v Power BI.

Kromě Power BI se na data v Elasticu dokáže dotazovat celá řada dalších business inteligence nástrojů, a to jak komerčně využívaných, tak i množství open source aplikací. Koncoví uživatelé tedy zkrátka nejsou odkázaní na jeden software. Právě to, aby byla umožněna flexibilita z hlediska konzumace dat o testování a uživatelé si dle svých preferencí a požadavků mohli vybrat, kde budou na data nahlížet a tvořit z nich vizualizace, byl jeden z primárních cílů při volbě datového úložiště.

Na základě průzkumu „Stack Overflow 2023 Developer Survey“, kterého se v kategorii „Most popular technologies – Databases“ zúčastnilo přes 70 000 respondentů z řad vývojářů, se Elasticsearch umístil mezi deseti nejpopulárnějšími technologiemi z oblasti databázových řešení, jak ukazuje následující obrázek.



Obrázek 6 - Most popular technologies – Databases; zdroj: (Stack Overflow Developer Survey 2023, 2023)

4.5 Architektura aplikace

S přihlédnutím k funkčním požadavkům a dalším výše zmíněným zjištěním byla jako finální řešení zvolena aplikace s mikroservisní architekturou založena na bázi .NET Worker Service, což je typ aplikace představený v .NET Core 3.0 umožňující vytváření dlouhodobě běžících služeb bez uživatelského rozhraní. Tento typ služeb je ideální pro různé úkoly prováděné na pozadí, jako je zpracování úloh ve frontě, monitorování aplikací či různé zpracování dat, což podporuje koncept aplikace pro tuto práci.

Je mimo jiné navržen pro snadné spuštění na různých platformách, včetně Windows a Linux, což umožňuje větší volnost v tom, kde finální program poběží.

Základní principy Worker Service jsou:

1. **Hostování**: .NET Worker Service využívá koncept hostování, který je zodpovědný za celou správu životního cyklu aplikace. Hostitel je odpovědný za inicializaci a spouštění služby, správu její konfigurace a všech závislostí, a také za řádné ukončení aplikace.
2. **Dependency Injection (DI)**: Worker Service podporuje využívání techniky DI, konkrétně pomocí interface „IServiceCollection“ pro konfiguraci a registraci služeb, včetně služeb na pozadí, což usnadňuje správu závislostí, podporuje lepší modulárnost kódu a velkou flexibilitu v konfiguraci jednotlivých služeb.
3. **Konfigurace**: Aplikace může snadno čerpat konfigurační nastavení z různých zdrojů, včetně souborů (např. appsettings.json), proměnných prostředí, argumentů příkazové řádky a dalších. Díky oddělení konfigurace od logické části je následně jednoduché měnit nastavení celé aplikace právě bez větších zásahů do kódu a logiky.
4. **Logování**: Integrovaná podpora pro logování umožňuje snadné sledování toho, co se děje v aplikaci, a pomáhá při diagnostice a odstraňování chyb. Jelikož Worker Service nemá uživatelské rozhraní, lze alespoň díky dobrému využití logování sledovat, co se s aplikací při běhu děje.

Mezi hlavní výhody zvoleného řešení patří již zmíněná multiplatformní podpora, snadné nasazení a správa aplikace díky integraci s moderními CI/CD nástroji a technologií pro kontejnerizaci, vysoká výkonnost a škálovatelnost, bezpečnost a případně i možnost integrace s cloudovými službami pro efektivnější využití zdrojů.

Ve firmě navíc existuje specifická implementace .NET Worker Service známá pod názvem Alza.Platform Scheduled Worker, kde je šablona projektu již modifikována tak, aby se nově vyvinutá aplikace následně dobře nasazovala a integrovala v podobě kontejneru do platformy běžící na technologii Kubernetes, což je také jeden ze záměrů.

V úvahu připadala i alternativní možnost realizace formou skriptu, který by byl přímo součástí Azure pipeline, pomocí níž se spouští automatizované testy. Výhodou tohoto řešení by byla zejména možnost zpracovávat výsledky ihned po uplynutém běhu testů, nebylo by tím pádem nutno myslet například na dodatečnou logiku získávání nových dat při každém běhu aplikace, jelikož by byl jasně definovaný kontext konkrétního běhu testů, ke kterému se běh skriptu vztahuje.

Samotná provázanost s Azure pipeline by však byla i největší nevýhodou, jelikož toto řešení počítá právě jen s tímto jedním datovým zdrojem, čímž se znemožňuje případné rozšíření o další zdroje dat o testování nejen z Azure, ale i odjinud. Service Worker je oproti tomu značně flexibilnější. Pokud bude v budoucnu nutné a chtěné čerpat data pro reporting i z jiných zdrojů, jednoduše se pro tento zdroj vytvoří a zaregistruje nová služba, která zastřeší sběr dalších dat z nového zdroje, přičemž logika jejich ukládání by mohla zůstat bez větších úprav.

4.5.1 MVP přístup

Pro účely ověření funkčnosti celého konceptu je k vývoji aplikace v rámci této práce přistupováno metodou MVP (Minimum Viable Product), prioritou je tedy vytvoření minimálního životaschopného produktu. Výsledná aplikace bude splňovat veškeré definované cíle a bude disponovat veškerou zásadní funkcionalitou nutnou pro uvedení do reálného provozu, v určitých ohledech však bude omezena, a to následovně:

- Sbíranými daty budou výsledky běhu webové regresní sady automatizovaných UI testů v rámci jedné specifikované pipeline, konkrétně pipeline, která běh této sady testů spouští jednou denně v noci. Testy jsou vykonávány oproti testovacímu beta prostředí webové aplikace.
- V prvotní fázi nebude řešena implementace vlastních instancí technologií Elastic Stacku. Aplikace bude data indexovat do existující infrastruktury na nody, které primárně využívá jiná aplikace. Vzhledem k tomu že však nabízí prozatím dostatečnou kapacitu pro data generovaná autorovou aplikací, bylo zvoleno toto

řešení. Z pohledu aplikace bude v budoucnu přechod na jiné servery Elasticu znamenat pouze to, že se v konfiguraci aplikace nahradí url adresy na jednotlivé nody za nové.

Díky tomu, že bude zpočátku sledována jen jedna pipeline a jedna sada testů běžící pravidelně nad beta prostředím, bude zajištěna dobrá porovnatelnost jednotlivých výsledků, bude možné data v dlouhodobém horizontu porovnávat, sledovat trendy v datech a další charakteristiky. Pokud se koncept a myšlenka v praxi osvědčí, může být přistoupeno k rozšíření v tomto ohledu, tedy konzumace výsledků z vícero pipeline či například obohacení o výsledky manuálního testování.

Dále eliminace stavění vlastního Elastic clusteru ušetří v počátku náklady, jelikož toto by obnášelo rozšíření infrastruktury o další výpočetní zdroje. Zároveň se nasazení aplikace celkově urychlí, jelikož ve firemním prostředí každé rozšíření infrastruktury, byť o jediný Elastic node, znamená netriviální proces schvalování, priorizace a realizace by nemusela nastat v termínu nutném pro dokončení této práce, jelikož priority společnosti jsou v době psaní tohoto textu upřené jiným směrem.

Pro ověření funkčnosti konceptu a sesbírání úvodní zpětné vazby od uživatelů postačí jakýkoliv existující cluster, který disponuje dostatečným úložištěm. Díky tomu bude možné vyvinutou aplikaci dříve nasadit do provozu, nechat uživatele seznámit se s jejím fungováním, konzultovat průběžně celé řešení a poté zapracovat případné poznatky a dále tak celou práci obohacovat a zdokonalovat.

4.6 Příprava vývojového prostředí

Aby mohla vůbec započít práce na tvorbě kódu a zhmotnění samotné aplikace, je nutné připravit si lokální vývojové prostředí. Je zapotřebí mít k dispozici vhodný editor pro psaní kódu a správu projektu, dále pak zprovozněné v lokálním prostředí nástroje Elastic Stacku.

4.6.1 Visual Studio Code

Pro psaní kódu byl zvolen editor Visual Studio Code od Microsoftu, ten autor dlouhodobě využívá jak k pracovním, tak soukromým účelům, primárně s ohledem na jeho jednoduchou použitelnost, podporu mnoha programovacích jazyků a rovněž dostupnost, jelikož je k dispozici zcela zdarma.

Jako první krok je nutné nainstalovat firemní šablony pro tvorbu platformních projektů, což lze udělat jednoduše zadáním příkazu „dotnet new install Alza.Platform.Templates“ do terminálu v aplikaci. Jakmile operace doběhne, lze již v editoru vytvářet projekty s novými šablonami.

Jak již bylo zmíněno, aplikace bude postavena na principu .NET Worker Service. Nový projekt se šablonou pro scheduled worker lze založit zadáním příkazu „dotnet new alza-scheduledworker -n NázevAplikace“ do terminálu. Pro svou aplikaci autor zvolil název Alza.TestReporting. Po doběhnutí operace je již k dispozici základní vývojové prostředí a jsou založeny stěžejní soubory celého projektu.

Dalším krokem je instalace knihoven (NuGet balíčků), které budou pro tvorbu a fungování aplikace nezbytné. Konkrétně je zapotřebí nainstalovat NuGet balíčky „Microsoft.TeamFoundationServer.Client“ a „NEST“. První jmenovaný zpřístupní v aplikaci funkce pro komunikaci se službou Azure DevOps, kam se bude aplikace alespoň v prvotní fázi dotazovat na veškerá zdrojová data. Druhý balíček slouží pro komunikaci se službou Elasticsearch, umožní tedy aplikaci získaná data do této služby indexovat. Instalace obou balíčků je realizována následujícími příkazy do terminálu.

```
➢ dotnet add package Microsoft.TeamFoundationServer.Client --version 16.205.1
➢ dotnet add package NEST --version 7.17.5
```

4.6.2 Elastic Stack

Pro účely vývoje je nezbytné zprovoznit také lokální instanci Elastic Stacku. Jednotlivé aplikace je možné spustit lokálně pomocí příslušných obrazů v rámci kontejneru, což usnadňuje významně vývoj i testování. K vývoji aplikace pro tuto práci je nutné stáhnout a spustit obrazy pro Elasticsearch a Kibanu. Díky tomu je následně možné v lokálním prostředí komunikovat s Elasticem, aniž by bylo nutné pokoušet se ihned posílat data do reálného firemního prostředí. Tento postup již předpokládá, že v počítači je zprovozněný kontejnerový virtualizační systém, jako například Docker Desktop nebo Rancher Desktop. Autor zvolil Rancher, jelikož je dostupný zdarma a nabízí plnohodnotnou podporu pro příkazy využívané i v Dockeru (ten je však pro firemní využití zpoplatněný), což usnadní následnou práci.

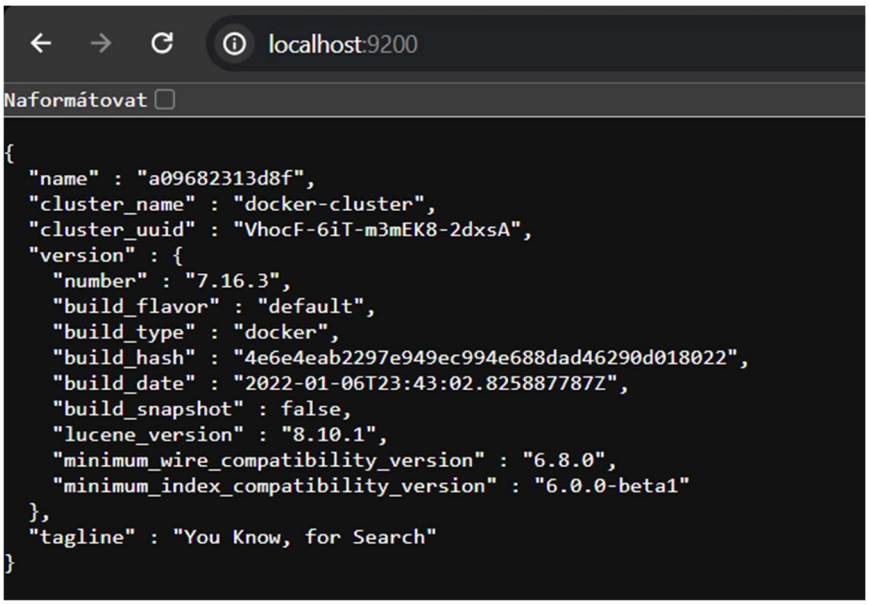
Je více způsobů, jak docílit lokálního spuštění kontejnerů. Jednou z možností je postupně jednotlivými příkazy pomocí příkazového řádku stáhnout všechny obrazy a poté je spustit s příslušnými parametry. Druhou možností je využít souboru docker-compose.yaml, což je efektivnější způsob, jak spustit najednou více kontejnerů, které spolu následně musí komunikovat. To je také varianta, která byla nakonec zvolena, jelikož jsou zapotřebí dvě navzájem úzce provázané služby – Elasticsearch a Kibana.

```yaml
version: "3.7"

services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.16.3
    container_name: elasticsearch
    restart: always
    environment:
      - discovery.type=single-node
    ulimits:
      memlock:
        soft: -1
        hard: -1
    ports:
      - 9200:9200
      - 9300:9300


  kibana:
    image: docker.elastic.co/kibana/kibana:7.16.3
    container_name: kibana
    restart: always
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - 5601:5601
    depends_on:
      - elasticsearch

volumes:
 certs:
   driver: local
 esdata01:
   driver: local
 kibanadata:
   driver: local
```

Obrázek 7 - Soubor docker-compose.yaml; zdroj: vlastní zpracování

Konkrétní příklad podoby docker-compose pro účely této práce lze vidět na obrázku výše. Uvnitř souboru se primárně definují jednotlivé služby, respektive kontejnery, které mají být spuštěny. Pro každou službu je nutné specifikovat obraz, název kontejneru, specifické porty, na kterých jednotlivé služby poběží, vzájemné závislosti a další podrobnosti dle požadavků. Kromě sekce services lze definovat i sekci volumes, díky čemuž se zajistí, že veškerá data vyprodukovaná při práci s kontejnery nebudou ztracena v případě, že se kontejnery zastaví nebo odstraní.

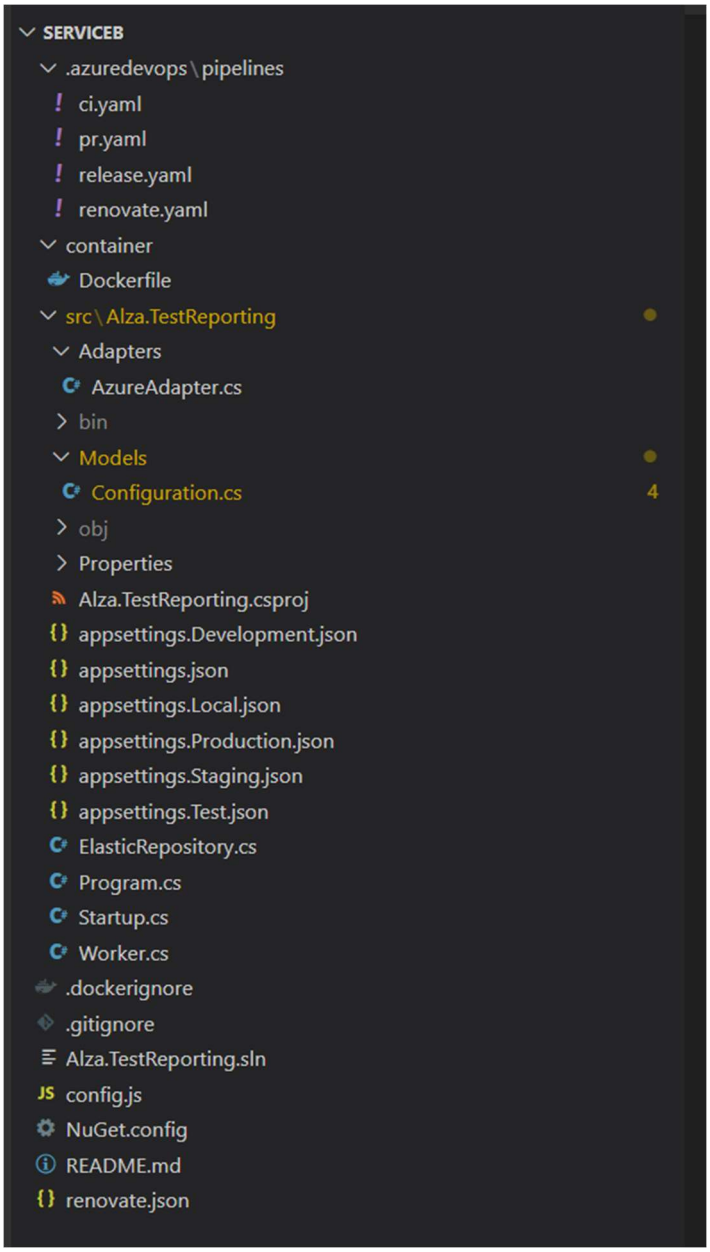
Spuštění následně probíhá jednoduše zadáním příkazu „docker compose up“ nad složkou, ve které se soubor nachází. Tím dojde ke stažení a instalaci specifikovaných obrazů a následně spuštění všech služeb dle zadané konfigurace. Úspěšnost operace a funkcionalitu služeb si následně lze ověřit v prohlížeči zadáním adresy a portu na příslušnou službu. V tomto případě pro kontrolu funkční instalace Elasticsearch lze zadat „localhost:9200“, přičemž očekávaným výsledkem je validní JSON response z Elasticu jako na obrázku níže, kde jsou v jednotlivých polích informace o konkrétní instalaci.



Obrázek 8 - Odpověď serveru obsahující údaje o instalaci Elasticu; zdroj: vlastní zpracování

4.7 Struktura a logika aplikace

Jádrem celého řešení na bázi .NET Worker Service je několik souborů, které společně zastřešují základní fungování aplikace. Kompletní souborová struktura výsledné aplikace je k vidění i na níže přiloženém obrázku.



Obrázek 9 - souborová struktura projektu; zdroj: vlastní zpracování

Vstupním bodem aplikace je soubor Program.cs, ten zodpovídá za spuštění hostitelského prostředí aplikace, včetně nastavení logování, konfigurace a závislostí. Tento soubor zajišťuje celkově nastavení a spuštění programu.

Dále pro konfiguraci služeb slouží soubor Startup.cs, v něm se (konkrétně v metodě ConfigureServices) registrují veškeré závislosti, které aplikace potřebuje. Díky dependency injection je pak umožněno využívání a zpřístupnění těchto služeb napříč celou aplikací. Konkrétně se ve startupu registrují tři hlavní služby – konfigurace, AzureAdapter a ElasticRepository. Poslední dvě zmíněné služby jsou popsány detailně později v této kapitole.

```
10  namespace Alza.TestReporting
11  {
        2 references | You, 7 days ago | 1 author (You)
12      internal class Startup : IScheduledWorkerStartup
13      {
            4 references
14          private readonly IConfiguration config;
15
            0 references
16          public Startup(IConfiguration config) {
17              this.config = config;
18          }
19
            0 references
20          public void ConfigureServices(IServiceCollection services)
21          {
22              services.AddSingleton<ElasticRepository>();
23              services.AddSingleton<AzureAdapter>();
24              services.Configure<Configuration>(config.GetSection("Configuration"));
25              AddElastic(services, config);
26              AddAzure(services, config);
27          }
28
```

Obrázek 10 - Soubor Startup.cs, detail metody ConfigureServices; zdroj: vlastní zpracování

Jak konkrétně vypadá metoda ConfigureServices lze vidět na obrázku výše počínaje řádkem 20. Kromě metody ConfigureServices se zde nachází ještě dvě další autorem vytvořené metody:

- **AddElastic** – Zde se definuje a inicializuje připojení ke službě Elasticsearch. Metoda přijímá dva argumenty, services a configuration. V metodě se nejprve načte do proměnné elasticConf aktuální konfigurace běhu a následně se definuje connectionPool, kde se určuje, na které adrese běží instance Elasticu, ke které se aplikace připojuje. Dále se definuje nastavení připojení, kde se nastavuje formátování JSON odpovědí z Elasticu a zejména pak výchozí mapování pro zvolený index. Tím se Elasticu dává informace o tom, jaká je struktura dat,

která mu bude aplikace posílat, konkrétně se zde tedy určuje, že index s názvem „testresults“ bude mapován podle třídy „TestCaseResult“. Následně se inicializuje nový ElasticClient s definovaným nastavením. Po inicializaci se již v Elasticu zakládá nový index, pojmenovaný „testresults“ a opět se zde určuje mapování dle třídy „TestCaseResult“, navíc s metodou AutoMap, čímž se zajistí, že Elastic přebere a automaticky použije datové typy všech atributů. Nakonec se již jen nová služba zaregistruje.

```
1 reference
private IServiceCollection AddElastic(IServiceCollection services, IConfiguration configuration)
{
    var elasticConf = configuration.GetSection("Configuration").Get<Configuration>()
        ?? throw new InvalidCastException($"Could not load settings for {nameof(Configuration)}");

    var connectionPool = new StaticConnectionPool(elasticConf.ElasticNodes);

    var settings = new ConnectionSettings(connectionPool)
        .PrettyJson()
        .DefaultMappingFor<TestCaseResult>(m => m.IndexName("testresults"));

    var client = new ElasticClient(settings);

    client.Indices.Create("testresults", i => i.Map<TestCaseResult>(x => x.AutoMap()));

    services.AddSingleton<IElasticClient>(client);

    return services;
}
```

Obrázek 11 - Soubor Startup.cs, detail metody AddElastic; zdroj: vlastní zpracování

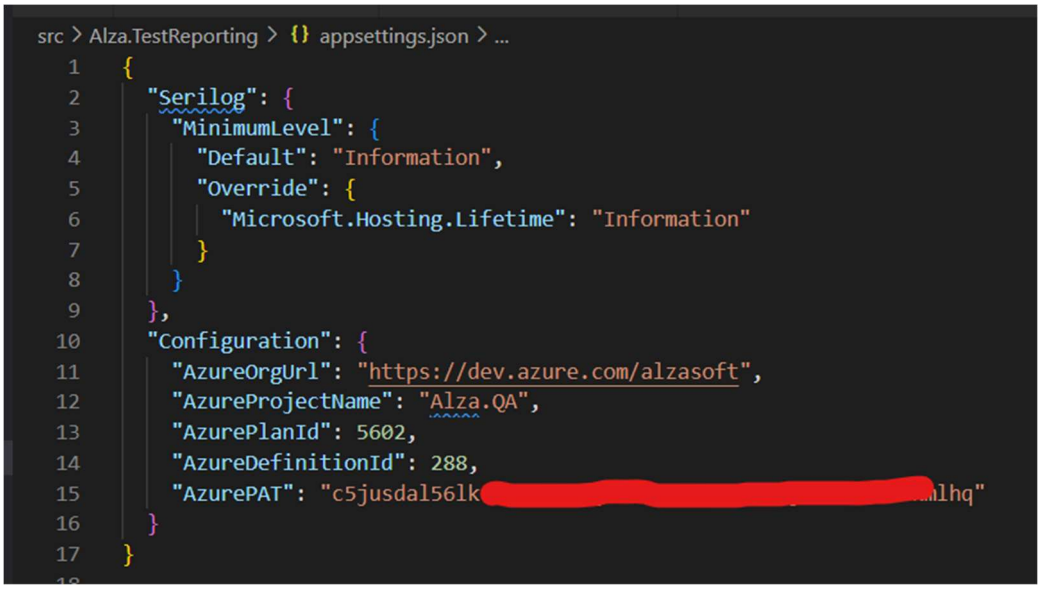
- **AddAzure** – Obdobně jako výše i připojení k Azure DevOps je potřeba ve startupu inicializovat. I zde se, tentokrát do proměnné azureConf, načítá aktuální konfigurace běhu. Následně se definuje samotné připojení pomocí adresy organizace a autentizace pomocí access tokenu, obojí získané z konfiguračního souboru. Poté se definuje připojení ke dvěma samostatným klientům, buildClient slouží pro komunikaci se službou BuildHttpClient a používá se následně k získávání informací o jednotlivých buildech (česky sestaveních), testClient slouží pro komunikaci se službou TestManagementHttpClient a ten zajišťuje, že je aplikace schopná získávat veškerá data ohledně testování. Opět se obě služby na konci metody zaregistrují.

```
48
        1 reference
49      private IServiceCollection AddAzure(IServiceCollection services, IConfiguration configuration) {
50
51          var azureConf = configuration.GetSection("Configuration").Get<Configuration>()
52              ?? throw new InvalidCastException($"Could not load settings for {nameof(Configuration)}");
53
54          VssConnection connection = new VssConnection(new Uri(azureConf.AzureOrgUrl), new VssBasicCredential(string.Empty, azureConf.AzurePAT));
55
56          var buildClient = connection.GetClient<BuildHttpClient>();
57          var testClient = connection.GetClient<TestManagementHttpClient>();
58
59          services.AddSingleton(buildClient);
60          services.AddSingleton(testClient);
61
62          return services;
63      }
64  }
65  }
66
```

Obrázek 12 - Soubor Startup.cs, detail metody AddAzure; zdroj: vlastní zpracování

Důležitou součástí celého řešení je i soubor appsettings.json plus jeho odvozené variace dle konkrétního prostředí (local, development, production…). V základní verzi souboru jsou definované konfigurační proměnné, jejichž hodnoty jsou stejné nezávisle na prostředí. V případě této aplikace tedy například adresa organizace v Azure DevOps, či ID označující specifickou pipeline, kompletní podoba souboru je k vidění na obrázku 13 níže. Soubor je možné vnitřně dělit ještě na jednotlivé sekce, samotná konfigurace se nachází v sekci „Configuration“ začínající na řádku 10, sekce „Serilog“ výše obsahuje základní informace pro logování.

V ostatních mutacích souboru appsettings se nad rámec tohoto základního definují jen ty proměnné, jejichž hodnoty se liší v závislosti na daném prostředí, zde se jedná zejména o adresy na jednotlivé nody Elasticu, které jsou odlišné pro lokální a zbylá prostředí.



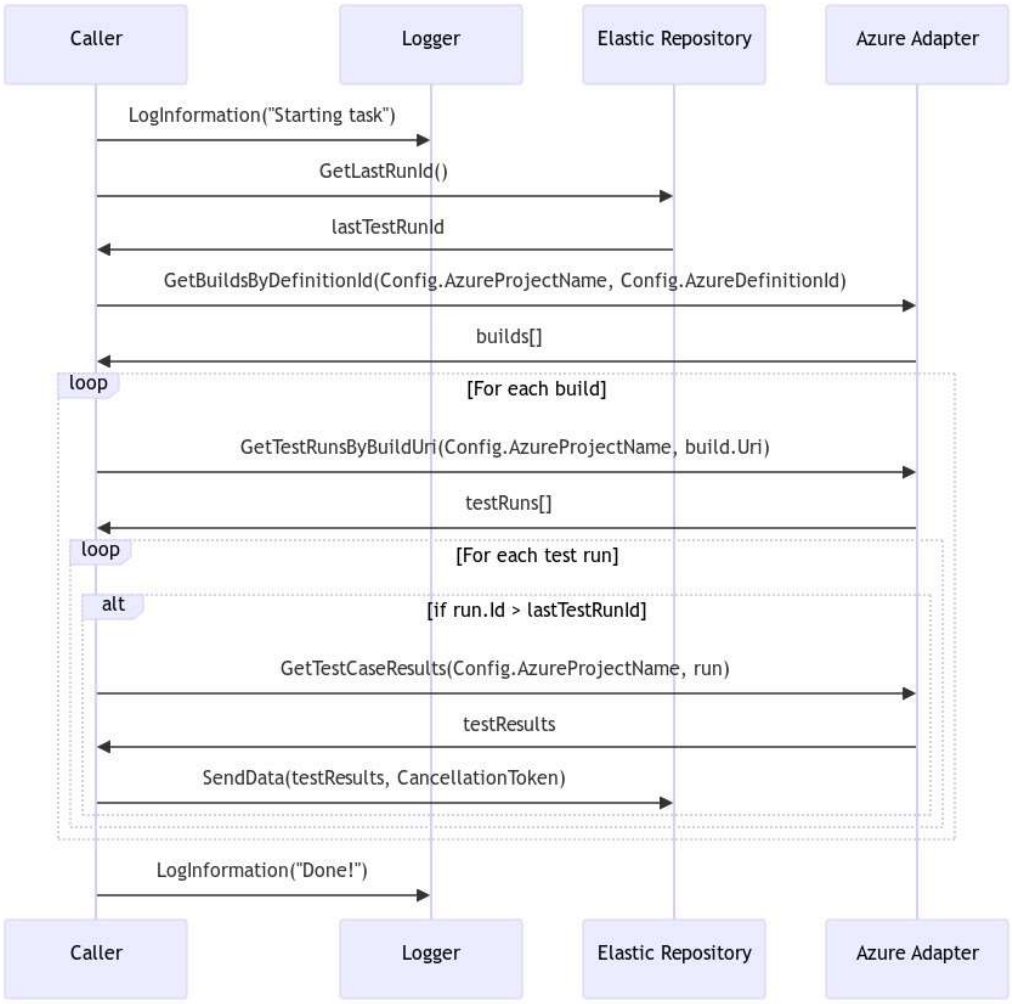
```
src > Alza.TestReporting > {} appsettings.json > ...
1   {
2     "Serilog": {
3       "MinimumLevel": {
4         "Default": "Information",
5         "Override": {
6           "Microsoft.Hosting.Lifetime": "Information"
7         }
8       }
9     },
10    "Configuration": {
11      "AzureOrgUrl": "https://dev.azure.com/alzasoft",
12      "AzureProjectName": "Alza.QA",
13      "AzurePlanId": 5602,
14      "AzureDefinitionId": 288,
15      "AzurePAT": "c5jusdal56lk                                    lhq"
16    }
17  }
```

Obrázek 13 - Soubor appsettings.json; zdroj: vlastní zpracování

Projekt obsahuje ještě Dockerfile, aby mohla být celá aplikace kontejnerizovaná. V souboru se definují instrukce pro sestavení dockerového obrazu aplikace, veškeré kroky pro kopírování souborů projektu, obnovení závislostí a nastavení příkazu pro spuštění aplikace (podrobněji bude vysvětleno v kapitole popisující princip nasazení aplikace).

Nejdůležitějším ze základních souborů tohoto řešení je Worker.cs – právě v něm je definovaná celá logika programu, která má být vykonávaná. Třída dědí z rozhraní IScheduledWorker, čímž se implementuje logika umožňující spouštění programu v pravidelných intervalech či ve specifických časech.

V případě této aplikace tedy Worker.cs obsahuje kompletní logiku pro sběr, zpracování a indexování dat. Celé jádro algoritmu je pro přehlednost znázorněno na následujícím sekvenčním diagramu (obrázek 14). Vyobrazená logika zachycuje primárně funkcionalitu kódu, který se nachází v metodě RunAsync právě uvnitř souboru Worker.cs.



Obrázek 14 - Sekvenční diagram vyobrazující logiku programu; zdroj: vlastní zpracování

Jelikož aplikace nedisponuje uživatelským rozhraním, slouží pro informování o běhu a všech událostech primárně logování, ze kterého je možné vyčíst proběhlou aktivitu programu. Nejprve se na počátku zaloguje obecná informace o tom, že program začíná se zpracováním. Následně proběhne zavolání metody GetLastRunId, jež se dotazuje na Elastic Repository a slouží k získání ID běhu z posledního uloženého záznamu. Metoda se nachází v souboru ElasticRepository.cs a její konkrétní podobu zachycuje obrázek 15 níže.

Princip spočívá v obecném dotazu na Elasticsearch klienta prostřednictvím metody SearchAsync. Definuje se index, ve kterém se má vyhledávat a následně řazení výsledků. Řadí se podle pole CompletedDate v sestupném pořadí. Nakonec se sada výsledků omezí příkazem Size(1) na jeden záznam, jelikož index v čase výrazně roste a pokud by se musela pokaždé vracet celá sada výsledků, znamenalo by to postupem času pomalejší odpovědi z Elasticu a celkové zpomalení běhu programu.

```
1 reference
public async Task<string> GetLastRunId(CancellationToken ct) {
    var searchResponse = await client.SearchAsync<TestCaseResult>(s => s
        .Index("testresults")
        .Sort(sort => sort
            .Field(f => f
                .Field(p => p.CompletedDate)
                .Order(SortOrder.Descending)
            )
        )
        .Size(1), ct
    );

    if (searchResponse.IsValid)
    {
        var latestDocument = searchResponse.Documents.FirstOrDefault();

        if (latestDocument != null)
        {
            logger.LogInformation("Latest runId indexed in Elastic is {id}", latestDocument.TestRun.Id);
            return latestDocument.TestRun.Id;
        }
        else
        {
            logger.LogInformation("There are no valid documents in Elastic!");
            return "0";
        }
    }
    else
    {
        logger.LogInformation("Elastic search response is not valid! {error}", searchResponse.ServerError);
        return "0";
    }
}
```

Obrázek 15 - Soubor ElasticRepository.cs, detail metody GetLastRunId; zdroj: vlastní zpracování

Poté následuje větvení podle toho, zda je odpověď v searchResponse validní. Pokud ano, znamená to, že vyhledávání neskončilo chybou a vrátilo nějaký validní výsledek. Do proměnné latestDocument se aplikace pokusí načíst obsah prvního a zároveň jediného

dokumentu, který by měl být získán. To že je odpověď z Elasticu validní totiž ještě nutně nemusí znamenat, že byly vráceny nějaké dokumenty. Pokud obsah proměnné latestDocument není prázdný, získá se obsah pole TestRun.Id ze získaného dokumentu, zaloguje se informace o posledním zaindexovaném ID běhu a metoda nakonec vrátí tuto hodnotu. Všechny ostatní případy, tedy nevalidní obsah searchResponse či prázdná proměnná latestDocument, vyústí v zalogování patřičné chybové hlášky a vrací se výchozí hodnota „0“. V tuto chvíli je tedy aplikace ve stavu, kdy je získané buď ID posledního běhu, případně výchozí hodnota „0“

Následuje volání metody GetBuildsByDefinitionId skrze Azure klienta. Nachází se v souboru AzureAdapter.cs a přijímá dva argumenty – projectName a definitionId. Metoda podle těchto specifikací následně vrací zpět seznam buildů, které vyhovují argumentům, z nichž nejdůležitější je definitionId a specifikuje ID pipeline, jejíž buildy jsou požadované.

Následně se v cyklu procházejí všechny buildy ze získaného seznamu. Z každého buildu se získá seznam běhů testů do proměnné testRuns voláním metody GetTestRunsByBuildUri, která přijímá opět dva argumenty – projectName a buildUri. Specifikací URI buildu v rámci každého cyklu je docíleno právě toho, že se získá seznam běhů testů pro každý build ze seznamu.

Poté následuje druhý vnořený cyklus, kde se prochází všechny běhy testů v daném buildu. Uvnitř cyklu je podmínka kontrolující, zda je ID běhu vyšší než ID posledního běhu zaindexovaného v Elasticu. Pokud je podmínka splněna, znamená to, že aktuální nalezený běh je nový a jeho výsledky ještě nejsou uložené, získají se tedy do proměnné testResults detailní výsledky testů v daném běhu skrze metodu GetTestCaseResults. Opět se jedná o metodu v AzureAdapteru a přijímá argumenty projectName a run, kde se předává celá instance běhu pro získání výsledků. Metoda vrací opět výsledky formou seznamu, ten se následně předává metodě SendData v ElasticRepository a zde již dochází k samotnému indexování nových výsledků testů do specifikovaného indexu.

Metoda vnitřně funguje na principu volání metody BulkAsync, čímž se se posílá POST request na BULK API klienta. BULK API slouží právě k zaindexování většího množství dokumentů pomocí jednoho requestu. Uvnitř BulkAsync metody se ještě specifikuje konkrétní jméno indexu a v metodě IndexMany se předává celá proměnná obsahující seznam výsledků testů, konkrétně tedy seznam instancí třídy TestCaseResult.

Následně se ještě kontroluje odpověď serveru na odeslaný request. Pokud je odpověď validní, zaloguje se informace o úspěšném zaindexování nových dokumentů. V opačném případě se loguje podrobná chybová hláška, aby následně bylo možné z logů zjistit, kde nastal problém a proč konkrétní indexování selhalo.

Po proběhnutí všech cyklů a operací popsaných výše se zaloguje ještě finální informace „Done!“ značící celkový konec algoritmu, běh aplikace tímto končí.

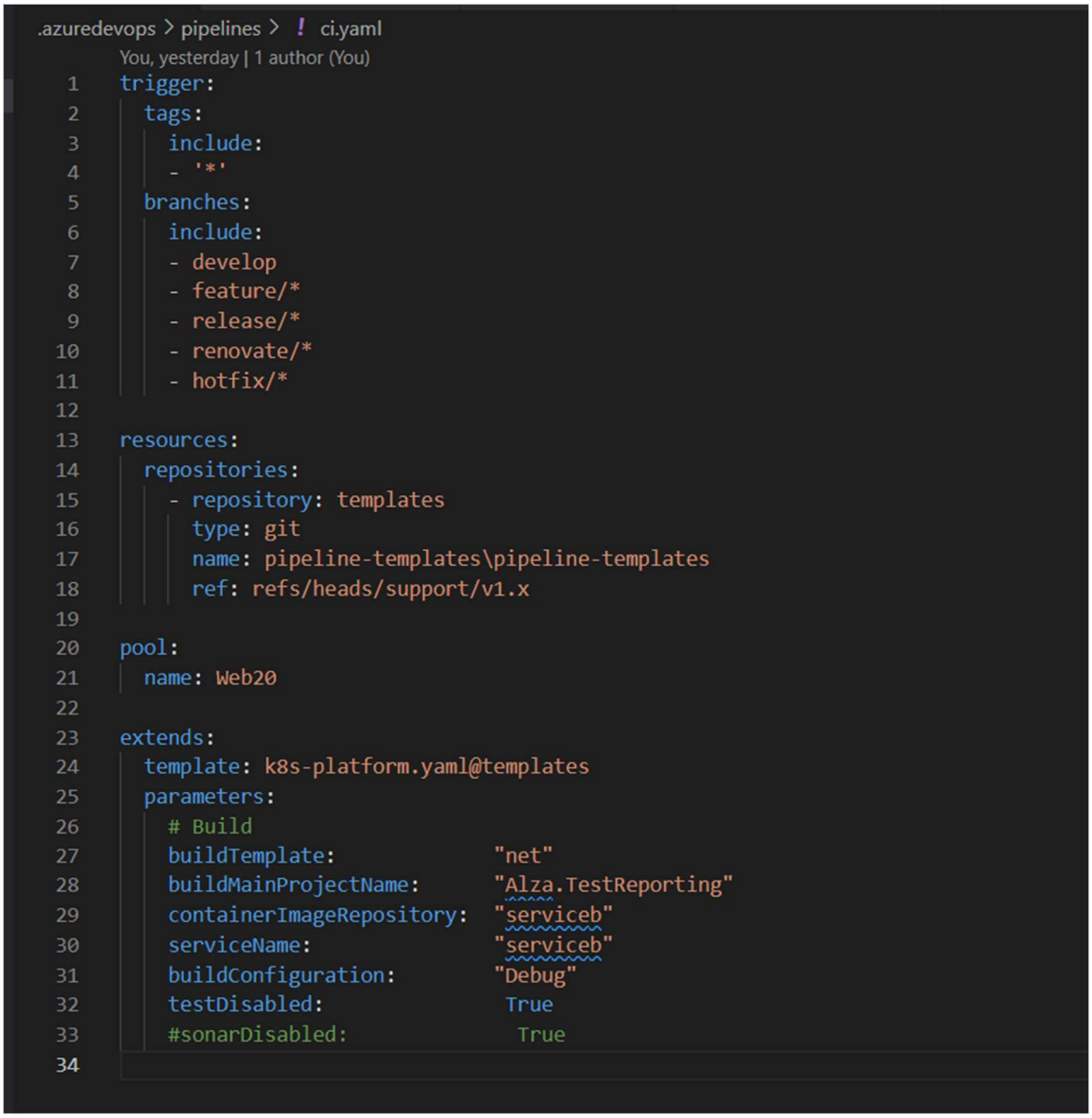
4.8 Nasazení aplikace

Jakmile je aplikace v popsané podobě dokončena, lze přistoupit k přípravě jejího nasazení do Kubernetes. Základním předpokladem je v projektu správně definovaný soubor Dockerfile, který zabezpečuje vytváření Docker obrazu jako předlohy pro výsledný kontejner, v němž aplikace poběží. Podoba souboru Dockerfile je vyobrazena na obrázku níže, přičemž první řádek odkazuje na základní obraz, ze kterého tento nový vychází. Zde se konkrétně jedná o Red Hat Universal Base Image 8. Další řádky slouží k definicím několika proměnných a instrukcí pro kopírování souborů projektu, na řádku 11 se nakonec definuje příkaz, který se provede při spuštění obrazu z kontejneru – jde o příkaz pro spuštění běhu programu.

Obrázek 16 - Soubor Dockerfile; zdroj: vlastní zpracování

Dále je třeba nakonfigurovat pipeline pro kontinuální integraci, ta má na starost automatické sestavení při každé úpravě zdrojového kódu aplikace, respektive spouští se s každým nahráním změn na server. Jedná se o soubor ve formátu YAML, vychází z obecné šablony projektu k8s-platform.yaml, stěžejní je uvnitř souboru definovat správně parametry jako název aplikace, repozitář pro kontejner, název služby či konfiguraci sestavení. Podobu souboru ukazuje obrázek 17 na další straně.

Takto nakonfigurovaná aplikace je uložena v Azure DevOps repozitáři s prostupem do Kubernetes. Aby mohlo celé sestavení kontejneru a jeho nasazení pomocí pipeline proběhnout úspěšně, je nutné v konfiguračním repozitáři ještě zaregistrovat nově přidanou službu, definovat její typ a veškeré závislosti. Využívá se při tom balíčkovacího nástroje Helm.

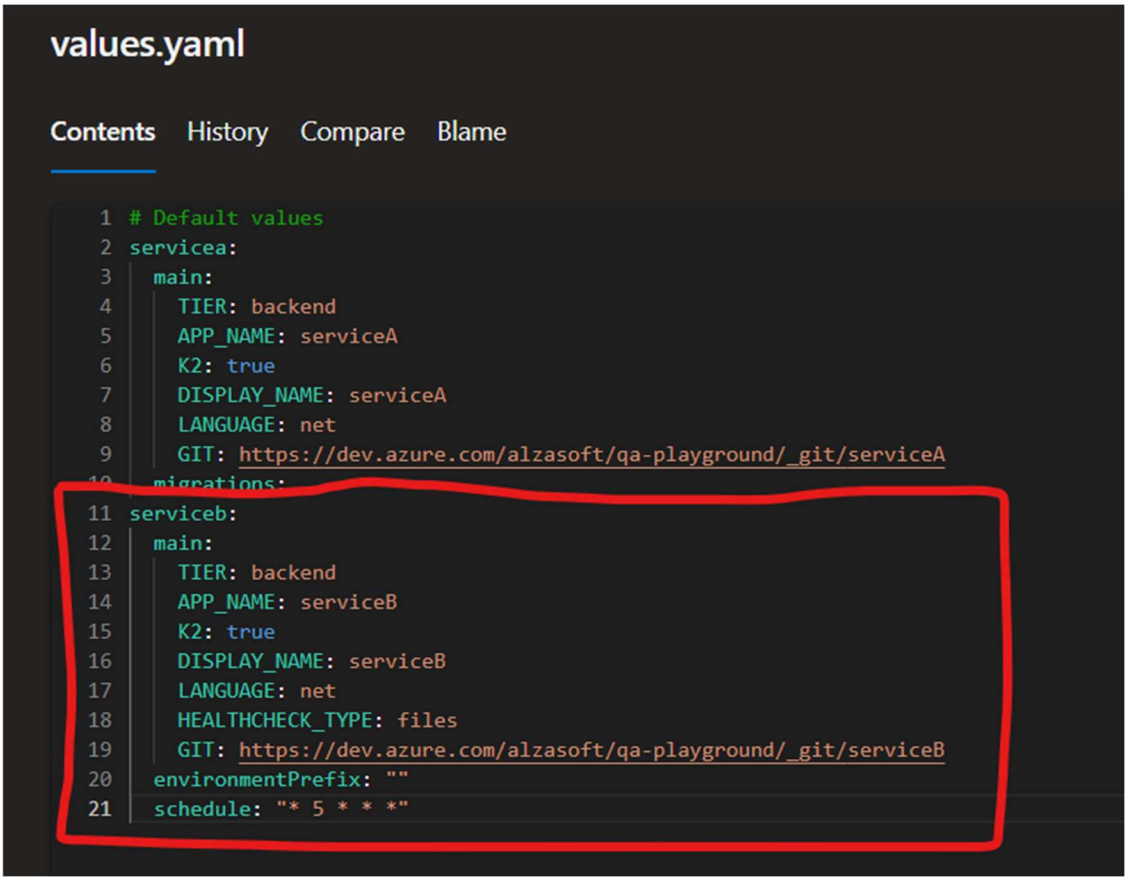


Obrázek 17 - Konfigurace pipeline pro kontinuální integraci; zdroj: vlastní zpracování

Repozitář aplikace se jmenuje „serviceB“, pod tímto pojmenováním tedy bude výsledná služba a kontejner dále figurovat. V konfiguračním souboru Chart.yaml se definuje nová závislost pro generic-cronjob, tedy aplikaci opakovaně spouštěnou dle definovaného plánu. V souboru environment.yaml je nutné pro službu definovat podrobnosti ohledně prostředí spuštění, v případě reportovací služby serviceb se konkrétně definuje výchozí větev develop, prostředí Development, rovněž je zde specifikována adresa Docker obrazu aplikace.

V souboru values.yaml se pak pro aplikaci specifikují výchozí hodnoty, jako název, jazyk aplikace, odkazuje se na zdrojový GIT adresář a také se zde nastavuje plán spouštění celého cronjobu, jak lze vidět na řádku 21 na obrázku na následující straně.

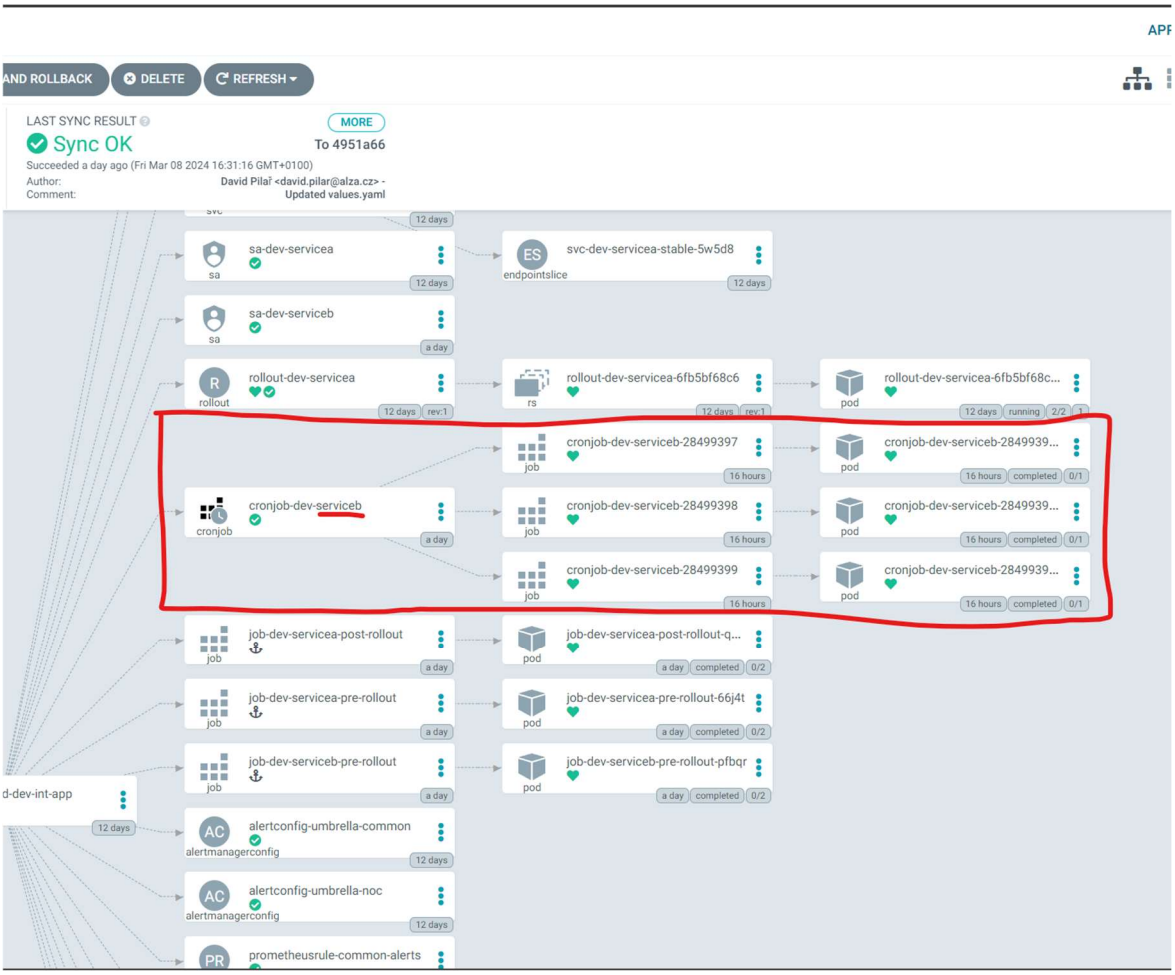
Pole schedule se zadává ve formátu „* * * * *“, přičemž klíč čtení tohoto zápisu je „minuta hodina den(v měsíci) měsíc den(v týdnu)“. Hvězdička udává libovolnou hodnotu. Konkrétně ve vyobrazeném zápisu tedy „* 5 * * *“ znamená, že se cronjob bude spouštět každý den v 5:00 ráno, bylo by však možné nastavit delší interval, například spouštění jen v úterý, případně interval zkrátit a nechat aplikaci spouštět každých pět minut.



Obrázek 18 - Soubor values.yaml, konfigurace výchozích hodnot aplikace; zdroj: vlastní zpracování

Důležité je zmínit, že veškeré změny prováděné v těchto konfiguračních souborech se ihned po uložení automaticky propagují do Kubernetes, není tedy nutné řešit kromě samotné definice všech hodnot žádné další operace.

Úspěšně nakonfigurovanou, nasazenou a běžící aplikaci lze ověřit skrze webové rozhraní nástroje Argo CD, který slouží ke správě Kubernetes clusterů a jejich monitoring. Jednotlivé objekty reprezentující zde vytvářenou službu jsou vyznačeny v obrázku 19 na následující straně. Je patrný vztah základního CronJobu, z něj rozvětvené jednotlivé Joby a pro každý Job jeden Pod.

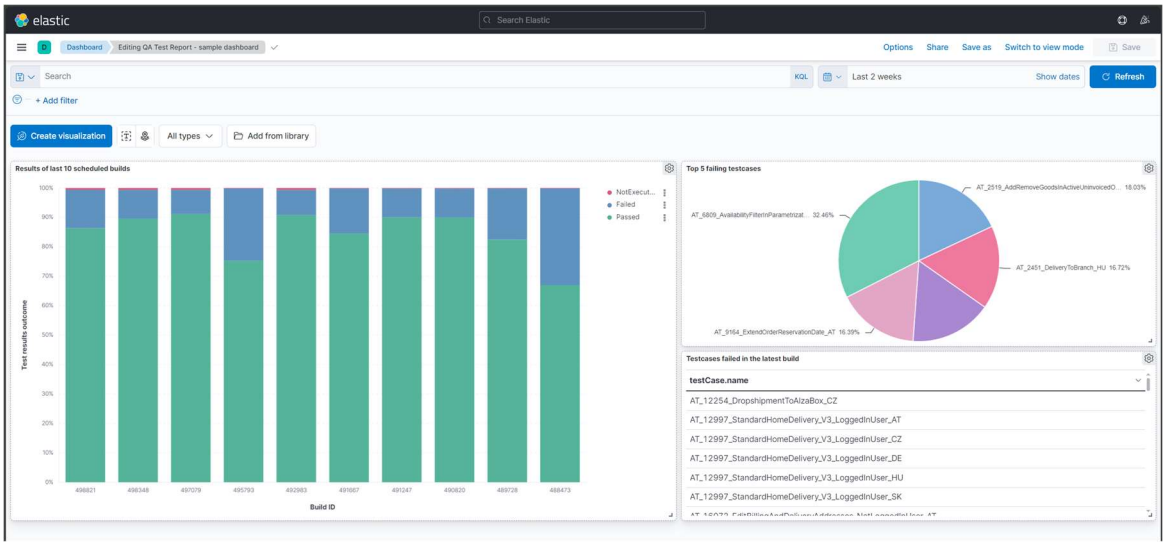


Obrázek 19 - Rozhraní aplikace Argo CD pro sledování a správu clusteru; zdroj: vlastní zpracování

5 Výsledky a diskuse

Jak bylo již uvedeno v kapitole popisující architekturu aplikace, k vývoji bylo přistupováno tak, aby výsledkem byl určitý koncept, respektive MVP, jenž poslouží pro demonstraci schopností celého návrhu, který bude dále dle zpětné vazby upravován a rozšiřován. Výsledkem praktické části práce je reálná aplikace, která v momentě psaní tohoto textu již běží dle pravidelného plánu jednou denně v 5 hodin ráno, získává z Azure DevOps výsledky automatizovaných regresních UI testů (z nočního běhu) běžících oproti webové aplikaci, a ty poté indexuje do Elasticsearch, čímž umožňuje jejich pravidelný reporting. Z Elasticu následně tato data mohou zaměstnanci volně přebírat a dále interpretovat pomocí libovolných reportovacích a monitorovacích nástrojů umožňujících napojení Elasticu jakožto datového zdroje. To byl i hlavní cíl celé práce – zpřístupnit data o testování pro analýzu skrze nástroje třetích stran. Momentálně se takto sice zpracovává jen jedna specifická oblast testování, i přes to však má již nyní aplikace pozitivní dopad a přínos.

Ve vizualizační aplikaci Kibana byl vytvořen nad sbíranými daty demonstrativní dashboard, který zobrazuje několik metrik a funguje mimo jiné jako příklad toho, jaké informace je možné například na základě sbíraných dat prezentovat. Zmiňovaný dashboard lze vidět na obrázku níže a zahrnuje tři vizualizace – ve sloupcovém grafu procentuálně vyjádřený přehled výsledků z posledních deseti plánovaných buildů sledované pipeliny, dále ve výsečovém grafu pět nejčastěji selhávajících testovacích případů za zvolené období, a nakonec tabulku se seznamem testovacích případů, které selhaly v posledním běhu pipeliny.

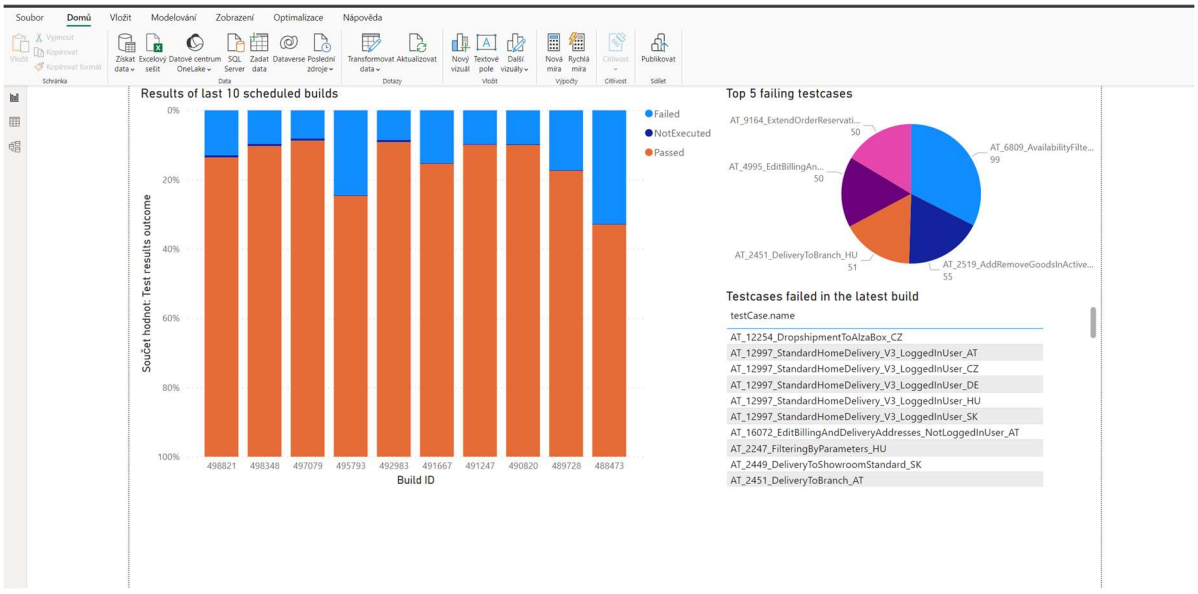


Obrázek 20 - Dashboard v aplikaci Kibana; zdroj: vlastní zpracování

Tento dashboard může na jednu stranu vypadat jednoduše, nabízí však cenný vhled do sledované oblasti. Azure totiž neumožňuje na jednom místě takto přehledně sledovat zvolené ukazatele, díky promítnutí těchto informací na jednom místě a jejich pravidelnému sledování se mimo jiné zlepšil přehled o trendech ve výsledcích testování. Ve sloupcovém grafu lze například dobře vidět, pokud najednou dojde k výraznému poklesu v poměru úspěšně provedených testů oproti předchozím dnům.

Díky vizualizaci nejproblémovějších testovacích případů a zároveň seznamu čerstvě selhaných testovacích případů je možné velmi brzy odhalit, pokud dochází k selhávání některých testů, se kterými dříve nebyl problém, což následně pomáhá s dřívějším odhalováním chyb. Jelikož dashboard sleduje data z testů vykonávanými nad beta prostředím, znamená zde odhalení případných bugů především možnost reportovat tato odhalení dříve, než se chyby dostanou do RC (release candidate), tím pádem je jejich oprava rychlejší a je zde výrazný potenciál celý release proces díky tomuto zrychlit, jelikož je příležitost odhalit některé chyby mnohem dříve.

Jak již bylo zmíněno i v kapitole popisující praktickou realizaci, řešení postavené na Elastic Stacku bylo zvoleno i kvůli dobré integraci s nástroji třetích stran. Dashboard podobný tomu na předchozím obrázku lze tedy bez problému vytvořit i například pomocí Power BI Desktop, jak ukazuje pro změnu následující obrázek. Je zde nyní větší flexibilita z hlediska toho, v jakém softwaru lze na dostupná data nahlížet a pracovat s jejich vizualizací.



Obrázek 21 - Dashboard v aplikaci Power BI Desktop; zdroj: vlastní zpracování

Je však důležité zmínit, že je zde stále velký prostor pro vylepšení. Aby mohla vyvinutá aplikace podporovat komplexnější reporting, bude nutné zahrnout do sběru dat mnohem více datových zdrojů a celkově rozšířit sledovanou oblast. Autor především doufá, že aktuální zkušební provoz na vybrané oblasti testování webové aplikace umožní ověření funkčnosti tohoto konceptu a na základě zpětné vazby poskytnuté od zaměstnanců bude i nadále docházet k rozvoji již vypracovaného řešení. Už aktuální pozitivní zkušenost však naznačuje, že vyvinuté řešení s vysokou pravděpodobností nalezne v budoucnu širší uplatnění v praxi, aplikace tedy nebude sloužit pouze jako podklad této kvalifikační práce, ale má šanci stát se součástí procesů v reálném firemním prostředí.

6 Závěr

Tématem práce byl reporting v testování softwaru, přičemž hlavním cílem práce byl návrh systému, který umožní sběr dat z test management systému ve vybrané společnosti a který bude tato data předávat reportovacím a monitorovacím nástrojům třetích stran tak, aby umožnil analýzu výsledků testování širšímu publiku. Cílem rovněž bylo ve vybrané společnosti toto navržené řešení implementovat.

V teoretické části práce došlo k vypracování literární rešerše ohledně tematiky spjaté s tématem práce, navíc se kromě problematiky testování, reportingu a business intelligence zaměřovala i na vývoj softwaru a technologii kontejnerizace.

Praktická část se dělila na několik dílčích kapitol, ve kterých byl postupně realizován vývoj samotné aplikace pro podporu reportingu, počínaje specifikací a analýzou požadavků, ze kterých vyplynuly podstatné detaily nezbytné pro návrh aplikace. Dále bylo nutné analyzovat data a zdroje těchto dat, aby bylo zřejmé, co a odkud má výsledná aplikace získávat. Nejen ze specifikace požadavků vyplynulo, že primárním a prozatím jediným zdrojem dat o testování má být Azure DevOps, přičemž pro potřeby reportingu jsou klíčové zejména výsledky proběhlých testů. Neméně důležité pak bylo analyzovat, kam mají tato získaná data dále putovat, přičemž bylo zjištěno, že aplikace pro reporting a monitoring fungují na principu definice datových zdrojů. Oproti původnímu očekávání tedy nebylo možné realizovat přímé předávání získaných dat externím nástrojům – v návaznosti na toto zjištění proběhla analýza dostupných systémů pro ukládání dat a bylo vyhodnoceno, že se získaná data budou ukládat, respektive indexovat, do nástroje Elasticsearch, o čemž pojednává podrobněji kapitola 4.4.

Jakmile bylo zřejmé, jaká data získávat, odkud je získávat a kam je dále předávat, bylo přistoupeno k návrhu architektury aplikace, přičemž byla zvolena aplikace na bázi .NET Worker Service. V kapitole 4.7 došlo k popisu celkové realizace vývoje ve smyslu psaní kódu, přičemž byl detailně vysvětlen a znázorněn algoritmus a princip fungování, včetně popisu dílčích funkcí, metod a úloh jednotlivých souborů v programu. Po dokončení vývoje došlo k nasazení hotové aplikace do provozu ve firmě, přičemž zde byla využita technologie kontejnerizace. Výsledná aplikace je tedy provozována ve firemním Kubernetes clusteru, kde je v podobě CronJobu na pravidelné bázi dle plánu spouštěna a na základě definované logiky zastřešuje sběr a ukládání dat, která jsou poté skrze Elasticsearch

přístupná k další analýze a interpretaci pomocí externích reportovacích a monitorovacích nástrojů.

Na základě dosavadního testovacího provozu a úvodní zpětné vazby bylo možné vyhodnotit, že již v aktuální podobě představuje autorovo řešení přínos pro řešení reportingu výsledků testování v momentálně sledované oblasti, přičemž je zde zjevný potenciál pro další vývoj a zdokonalení celého řešení tak, aby přinášelo dlouhodobě i nadále přidanou hodnotu.

7 Seznam použitých zdrojů

ALDINGER, Steven. Properly Running Kubernetes Jobs with Sidecars. In: *Medium.com* [online]. [cit. 2024-03-13]. Dostupné z: https://medium.com/teamsnap-engineering/properly-running-kubernetes-jobs-with-sidecars-ddc04685d0dc

BENTALEB, Ouafa, Adam S. Z. BELLOUM, Abderrazak SEBAA a Aouaouche EL-MAOUHAB, 2022. Containerization technologies: taxonomies, applications and challenges. *The Journal of Supercomputing* [online]. **78**(1), 1144-1181 [cit. 2024-03-10]. ISSN 0920-8542. Dostupné z: doi:10.1007/s11227-021-03914-1

BLOCK, Andrew a Austin DEWEY, 2022. *Managing Kubernetes Resources Using Helm*. Second Edition. Packt Pub. ISBN 9781803242897.

BOROVCOVÁ, Anna, 2008. *Testování webových aplikací*. Praha. Diplomová práce. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra teoretické informatiky a matematické logiky.

BOSE, Shreya, 2023. What is Automation Testing: Benefits, Strategy, Tools. *BrowserStack* [online]. Mumbai [cit. 2024-03-09]. Dostupné z: https://www.browserstack.com/guide/automation-testing-tutorial

BRIK, Evgeniy, 2020. Five Simple Steps to Write an Engaging QA Report. *Software Development Company – Andersen* [online]. [cit. 2024-03-12]. Dostupné z: https://andersenlab.com/blueprint/engaging-qa-report

BURNS, Brendan, Joe BEDA a Kelsey HIGHTOWER, 2019. *Kubernetes: Up and Running*. 2nd Edition. O'Reilly Media. ISBN 9781492046530.

BUTCHER, Matt, Matt FARINA a Josh DOLITSKY, 2021. *Learning Helm: managing apps on Kubernetes*. Beijing: O'Reilly. ISBN 978-1-492-08365-8.

CONGER, Sue, 2011. Software Development Life Cycles and Methodologies: Fixing the Old and Adopting the New. *International Journal of Information Technologies and Systems Approach*.

FILIPOVA, Olga a Rui VILÃO, 2018. *Software Development From A to Z*. Apress. ISBN 978-1-4842-3945-2.

GANJI, Suma, 2023. Understanding the Software Testing Life Cycle. *ACCELQ: #1 AI-Powered Codeless Test Automation QA Tool* [online]. [cit. 2024-03-16]. Dostupné z: https://www.accelq.com/blog/software-testing-life-cycle/

HARTANTO, Krisnawan. 6 Major Steps of Software Testing Life Cycle (STLC). In: *Medium.com* [online]. [cit. 2024-03-16]. Dostupné z: https://k-hartanto.medium.com/software-testing-life-cycle-stlc-b26581ae3051

HROCH, Michal, 2008. Proč potřebujete corporate reporting. *SystemOnLine* [online]. [cit. 2024-03-12]. Dostupné z: https://m.systemonline.cz/business-intelligence/proc-potrebujete-corporate-reporting-1.htm

KITNER, Radek, 2017. O čem je testování software? (pro znalé). *Radek Kitner - konzultant, lektor testování softwaru* [online]. Modřice [cit. 2021-02-25]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-software/

KITNER, Radek, 2017. Typy testování software (třídění testů). *Radek Kitner - konzultant, lektor testování softwaru* [online]. Modřice [cit. 2021-02-25]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/

Kubernetes Documentation | Kubernetes [online], 2023. [cit. 2024-03-13]. Dostupné z: https://kubernetes.io/docs/concepts/overview/

LANGER, Arthur, 2016. *Guide to Software Development*. Springer London. ISBN 978-1-4471-6799-0.

LAURSEN, Gert H. N. a Jesper THORLUND, 2017. *Business analytics for managers: taking business intelligence beyond reporting*. Hoboken, New Jersey: Wiley. ISBN 978-1-119-29858-8.

LI, Turbo, 2023. The Role of Test Reporting in Software Testing: A Comprehensive Overview. *HeadSpin* [online]. [cit. 2024-03-12]. Dostupné z: https://www.headspin.io/blog/a-step-by-step-guide-to-optimize-test-reporting-in-continuous-testing

MAINA, Mucheru, 2024. *Pod vs job vs cronjob in Kubernetes* [online]. [cit. 2024-03-12]. Dostupné z: https://mucheru.notion.site/pod-vs-job-vs-cronjob-in-Kubernetes-0d2cc97705104083b52d40ae8790c205

NEVES, Gabriel, 2019. Elasticsearch. *Avenue Code Snippets* [online]. [cit. 2024-03-14]. Dostupné z: https://blog.avenuecode.com/elasticsearch

POULTON, Nigel, 2021. *The Kubernetes Book*. 2021 edition. ISBN 9798703756065.

ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ, 2013. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press. ISBN 978-80-251-3816-8.

Software Testing Life Cycle (STLC), 2019. *GeeksforGeeks* [online]. Noida [cit. 2021-02-25]. Dostupné z: https://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc/

Stack Overflow Developer Survey 2023 [online], 2023. [cit. 2024-03-14]. Dostupné z: https://survey.stackoverflow.co/2023/

TESTIM, 2022. Test Automation Tool: Definition and 5 Best Ones. *Testim.io Blog - Test Automation, Software Quality and AI Articles* [online]. Tel Aviv [cit. 2024-03-09]. Dostupné z: https://www.testim.io/blog/what-is-a-test-automation-tool/

TURNBULL, James, 2019. *The Docker Book: Containerization Is the New Virtualization*. V18.09.2. Turnbull Press. ISBN 9780988820203.

8 Seznam obrázků