Univerzita Hradec Králové
Fakulta informatiky a managementu

Continuous Test Automation

Diplomová práce

Autor: Jan Chaloupka
Studijní obor: ai2-p

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Odborný konzultant: Ing. Miroslav Ježek, Unicorn Systems a. s.

Hradec Králové Duben 2023

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 4. 4. 2023 Jan Chaloupka

Poděkování:

Děkuji vedoucímu diplomové práce panu docentovi Ing. Filipovi Malému, Ph.D za cenné připomínky a za vedení této diplomové práce. Dále bych chtěl poděkovat panu Ing. Miroslavu Ježkovi ze společnosti Unicorn Systems a. s. za odborné vedení a konzultace během psaní této diplomové práce.

Anotace

Cílem této diplomové práce je shrnout tvorbu automatických testů a jejich využití v agilním vývoji. Práce se mimo jiné zabývá složitostmi automatizovaného testování v agilních prostředích a představuje různé nástroje, které usnadňují automatizaci testování. Nejprve popisuje výzvy spojené s testováním software, po nichž následuje vysvětlení agilních metodik a DevOps v jejich vztahu k testování software. Následně se práce podrobně věnuje automatizovanému testování a s ním spojeným nástrojům. Praktická část práce popisuje integraci CI/CD spolu se třemi nástroji pro integraci s automatickým testováním a obsahuje praktickou ukázku těchto nástrojů. Poté je představena společnost Unicorn a projekt Certigy, na kterém autor v rámci spolupráce napsal nové automatické testy, které napomáhají společnosti Unicorn k udržení vysoké spolehlivosti a kvality kódu na projektu Certigy.

Annotation

Title: Continuous test automation

The aim of this thesis is to summarize the creation of automated tests and their use in agile development. The thesis discusses the complexities of automated testing in agile environments and introduces various tools that facilitate test automation. It first discusses the challenges associated with software testing, followed by an explanation of agile methodologies and DevOps as they relate to software testing. The paper then discusses automated testing and its associated tools in detail. The practical part of the thesis describes CI/CD integration along with three tools for integration with automated testing and includes a practical demonstration of these tools. Then, Unicorn and the Certigy project are introduced, where the author has written new automated tests in a collaborative effort to help Unicorn maintain high reliability and code quality on the Certigy project.

Obsah

1 Úvod

Společnosti vyvíjející software čelí v současné době výzvě dodávat vysoce kvalitní produkty a zároveň si zachovat schopnost rychle se přizpůsobovat neustále se měnícím požadavkům trhu. Metodiky agilního vývoje se staly oblíbeným přístupem k řešení těchto výzev, protože upřednostňují iterativní vývoj, spolupráci a zpětnou vazbu od zákazníka. Úspěch agilního vývoje však do značné míry závisí na účinnosti a efektivitě procesů testování software. Cílem této práce je poskytnout ucelený přehled o automatizaci testování a zdůraznit význam a dopad automatického testování na proces vývoje software zejména pak v agilním prostředí.

Automatizace testování vychází z potřeby rychle ověřovat funkčnost, výkonnost a spolehlivost aplikací v průběhu jejich vývoje v každé iteraci agilního vývoje. Začleněním automatizovaného testování do každé fáze životního cyklu vývoje software mohou vývojové týmy včas odhalit a vyřešit problémy, čímž se sníží riziko vzniku chyb a zvýší se celková kvalita software. Automatizace procesu testování navíc umožňuje týmům získat okamžitou zpětnou vazbu o dopadu změn jejich kódu.

Diplomová práce shrnuje problematiku automatického testování software do přehledného celku, který lze využít v případě zavádění agilního přístupu a automatického testování software. Obsahuje popis tří nástrojů pro automatické testování software včetně praktických ukázek každého uvedeného nástroje. Dále v rámci práce vznikly nové automatické testy pro společnost Unicorn, které jsou využívány pro udržení kvality software psaného pro projekt Certigy.

Práce přináší ucelený přehled o automatizaci testování software včetně ukázky tvorby automatických testů. Zdůrazňuje význam a dopad automatického testování na proces vývoje software zejména v agilním prostředí.

2 Testování software

Testování software je proces, při kterém se ověřuje, že software funguje tak, jak má, a že splňuje požadavky, které na něj byly kladené. Cílem testování je zjistit, zda software splňuje požadavky na jeho funkčnost, spolehlivost, bezpečnost a uživatelskou přívětivost. Testování může zahrnovat řadu různých aktivit, od ručního testování, automatizovaného testování přes integrační a regresní testy až po testování zátěže a životnosti software. Je důležité pro vývojáře a testery, aby si uvědomovali, že testování není jen o hledání chyb, ale také o ověření kvality a důvěryhodnosti software. Většina lidí má zkušenost s tím, že software nefunguje podle očekávání. Software, který nefunguje správně, může způsobit mnoho problémů, včetně ztráty peněz, času či reputace.

2.1 Pojmy v testování a vývoji software

V této podkapitole jsou krátce vysvětleny pojmy související s testováním a vývojem software, které jsou v této práci využívány.

- **Test-case** (testovací případ) je postup, jak otestovat určitou funkcionalitu software.
- **Test-scenario,** neboli testovací scénář, je sada testovacích případů.
- **SDLC** (software development life cycle) je životný cyklus vývoje software.
- **STANDUP** je rychlá každodenní schůzka, na které si členové vývojového týmu sdělují, čeho dosáhli od posledního standupu, na čem pracují a zda se potýkají s nějakými problémy během vývoje.
- **Akceptační kritéria** jsou specifikace, na základě kterých je rozhodnuto, zda je určitá funkcionalita přijatelná pro zákazníka.
- **Kompletní testování** (exhaustive testing) je testování všech možných kombinací vstupů a podmínek pro daný software.

2.2 Cíle testování software

V procesu testování software lze cíle popsat jako zamýšlené výstupy z testovaného projektu. Níže popsané cíle vychází z [1].

Verifikace a validace

Testování je kontrolní opatření kvality sloužící k ověření, že produkt funguje tak, jak bylo požadováno. Testování software poskytuje stavovou zprávu o skutečném produktu v porovnání s požadavky na produkt.

Proces testování musí ověřit, zda software splňuje podmínky stanovené pro jeho vydání a použití. Testování software by mělo odhalit co nejvíce chyb v testovaném software, zkontrolovat, zda splňuje požadavky na něj kladené, a celkově dopomoci k přijatelné úrovni kvality.

Prioritní pokrytí

Úplné a vyčerpávající testování je nemožné vyjma malých a jednoduchých systémů. Z tohoto důvodu bychom se měli snažit provádět testy efektivně a v rámci časových a rozpočtových mezí. Proto je potřeba testování důkladně plánovat a jednotlivé části testování pečlivě prioritizovat.

Případy užití, které s vysokou pravděpodobností budou využívány častěji, by měly mít větší pokrytí testy než scénáře, které se vyskytují zřídka nebo jsou nevýznamné.

Vyváženost

Testovací proces musí zahrnovat specifikace, technická omezení a požadavky uživatelů. Výsledky musí být reprodukovatelné a nezávislé na konkrétním testerovi, tedy musí být konzistentní a nestranné.

Dokumentace

Monitorování výsledků testů napomáhá zjednodušit proces testování. Co a jak bylo otestováno jsou otázky, na které je třeba mít uchovanou odpověď. Plány jednotlivých testů by měly být dostatečně jasné, aby se daly snadno pochopit. Organizace by měla přijmout jednotné metody dokumentace těchto testů, aby se vyhnula chaosu a využila tuto dokumentaci k poučení se z předešlých chyb.

Deterministické

Detekce chyb v testování by neměla být náhodná. Tester by měl vědět, co dělá, na co cílí a jaký bude možný výsledek. Kritéria pokrytí by měla odhalit všechny vady určité povahy. Objevující se chyby by měly být kategorizovány podle toho, v jaké části pokrytí nastaly, a tím mohou představovat určitou hodnotu při detekci těchto vad v budoucím testování. Jasný přehled o procesu testování umožňuje lépe odhadnout náklady a lépe řídit celkový vývoj.

2.3 Rozdělení testů

Testy software mohou být rozděleny do několika kategorií v závislosti na úrovni abstrakce, zaměření a perspektivy, ze které jsou prováděny. Toto rozdělení dle ISTQB[1] je popsáno v následujících podkapitolách a vychází z *„Foundation Level Syllabus“* vydaného výše uvedenou organizací [2].

Každý z těchto testů má své výhody a nevýhody a je používán v různých scénářích. V praxi se obvykle používá kombinace těchto přístupů pro větší pokrytí testů.

2.3.1 Unit testy

Unit testy (jednotkové testy) se zaměřují na testování samotných komponent. Cílem unit testů je ověření, že jednotky fungují správně a že splňují požadavky na ně kladené. V agilním vývoji software, kdy dochází k častým změnám kódu, dobře napsané unit testy hrají klíčovou roli v budování důvěry v testované komponenty a snižují riziko postupu chyby do vyšších úrovní testování. To zahrnuje testování vstupů a výstupů jednotek a jejich následné chování včetně výjimek. Unit testy píše většinou vývojář, který napsal danou komponentu, a jsou navrženy tak, aby ověřily samostatnou funkčnost dané komponenty v izolaci od ostatních částí systému. Jednotkou mohou být různě samostatně testovatelné komponenty software jako například metody, třídy, objekty či moduly. Výhodou unit testů je, že umožnují odhalit chyby a problémy v rané fázi vývoje, což má za následek rychlejší a tím pádem také levnější opravu. Chyby odhalené unit testy se většinou opravují ihned bez formálního řízení a unit testy jsou většinou spouštěny před

[1] International Software Testing Qualifications Board je mezinárodní nezisková organizace a v oblasti testování software je nejznámější na světě. Organizace mimo jiné vydává řadu mezinárodně uznávaných certifikátů v oboru testování software.

integračními testy. Přesto však, pokud vývojář nahlásí chybu odhalenou unit testem, může tím poskytnout důležité informace pro analýzu příčiny chyby a pro zlepšování procesů.

2.3.2 Integrační testy

Integrační testy se zaměřují na testování komunikace mezi jednotlivými částmi aplikace. Typicky se těmito testy pokrývá komunikace s API, databází, subsystémy a jinými mikroservisami. Cílem těchto testů je zajistit, že jednotlivé části aplikace spolupracují správně a produkují očekáváné výsledky. Stejně jako v případě unit testů poskytují integrační testy jistotu, že nové změny neporušily stávající rozhraní komponenty či systému. Integrační testy se obvykle provádějí po vytvoření jednotlivých modulů aplikace a před jejím spuštěním v produkčním prostředí. Těmito testy lze zajistit, že aplikace bude fungovat správně i po nasazení do produkčního prostředí. Integrační testy se dělí na dvě úrovně:

- **Integrační testování komponent** se zaměřuje na testování interakce a rozhraní mezi integrovanými komponentami.
- **Testování systémové integrace** testuje, jak spolu různé systémy, balíčky či mikroservisy komunikují. Může se provádět po testování systému nebo současně s probíhajícími činnostmi testování systému. Integrační testování může být náročné zejména, pokud externí rozhraní nejsou řízena vyvíjející organizací, protože může vyžadovat odstranění chyb v externím kódu, které blokují testování.

Samotné integrační testování by se mělo primárně zaměřovat na testování integrace. Například při integraci dvou komponent X a Y by se správně napsané integrační testy měly soustředit výhradně na komunikaci mezi komponentami X a Y a neměly by testovat funkčnost samotných komponent. Ta by měla být pokryta během testování komponent. Stejně tak komunikace mezi dvěma systémy X a Y by se měla zaměřovat výhradně na komunikaci mezi těmito systémy, a ne na funkčnost systémů jako takových.

2.3.3 Systémové testy

Systémové testy se zaměřují na ověření funkčnosti a spolehlivosti celého systému jakožto celku s ohledem na end-to-end úkoly včetně nefunkčního chování při zpracovávání jednotlivých úkolů. Cílem systémového testování je zjistit, zda systém splňuje požadavky na funkčnost a zda je schopen pracovat v reálných podmínkách dle požadavků.

Funkční testování je proces ověřování, že software splňuje definované funkční požadavky a specifikace. Tyto požadavky jsou definovány například v podobě business požadavků nebo případů užití a zahrnují popis toho, „co“ by měl systém dělat.

Funkční testování by mělo být provedeno ve všech fázích testování, i když se zaměření těchto testů může lišit v závislosti na konkrétní fázi testování. Funkční testování bere v úvahu chování software, a proto se využívá „black-box“ testing, kdy tester nemá přístup ke zdrojovému kódu, ale ověřuje vstupy a výstupy. Návrh funkčních testů vyžaduje znalost konkrétního problému, který software řeší, např. geolokační modelovací software.

Nefunkční testování se zaměřuje na vlastnosti systému jako takového a je to testování toho, „jak dobře“ se systém chová. Příkladem nefunkčního testování může být testování:

- výkonnosti
- bezpečnosti
- kompatibility
- použitelnosti.

Cílem nefunkčního testování je odhalit problémy, které by mohly ovlivnit funkčnost software či snížily užitnou hodnotu pro uživatele. Nefunkční testování by mělo být prováděno na všech úrovních a to co nejdříve. Pozdní objevení nefunkčních vad může být závažné a potenciálně může ohrozit projekt jako celek.

2.3.4 Akceptační testy

Akceptační testy se podobně jako systémové testy zaměřují na testování systému jako celku, aby bylo potvrzeno, že software je připraven pro nasazení a používání zákazníkem. Testování probíhá ke konci vývoje software, nebo po úpravách, které byly provedeny v průběhu testování. Mezi typy akceptačního testování patří uživatelské akceptační testy, operační akceptační testy, smluvní akceptační testy a alfa a beta akceptační testy.

Uživatelské akceptační testy (User acceptance testing – UAT) mají za cíl ukázat vhodnost testovaného software pro jeho zamýšlené uživatele pomocí reálných scénářů.

Tyto testy musí prokázat, že testovaný systém je vhodný pro ostrý provoz a efektivně vykonává uživatelské požadavky a potřeby.

Provozní akceptační testy (Operational acceptance testing – OAT) se zaměřují na provoz v simulovaném produkčním prostředí. Tyto testy se soustředí na provozní aspekty testovaného software a mohou zahrnovat testování:

- Obnovení po pádu software
- Správu údržby
- Instalace, odinstalace, aktualizace
- Zálohování a obnovení

Provozní akceptační testy by měly dokázat, že provozovatel je schopen udržet funkční software v produkčním prostředí.

Alfa & beta testování jsou metody testování k vyhodnocení a získání zpětné vazby k produktu před jeho uvolněním pro širokou veřejnost. Alfa testování obvykle realizují interní týmy vývojářů přímo na místě a slouží k identifikaci a řešení chyb a dalších problémů. Beta testování provádí širší skupina externích testerů, obvykle zákazníků, nebo koncových uživatelů a slouží ke shromáždění zpětné vazby o použitelnosti produktu, jeho celkovém výkonu a dalších problémech, které nemusely být odhaleny při alfa testování.

Cílem alfa i beta testování je identifikovat a odstranit případné problémy s produktem před jeho uvedením na trh, což pomůže zajistit, že software bude kvalitní a bude splňovat potřeby uživatelů. Dalším cílem je detekce chyb související s využitím v prostředí, ve kterém bude systém používán, zejména pokud je takové prostředí těžko replikovatelné vývojovým týmem.

Smluvní a regulační testování (Contractual and regulatory acceptance testing)
Smluvní akceptační testování je proces posouzení software, který byl speciálně vyvinut podle dohodnutých kritérií uvedených ve smlouvě oběma stranami. Provádí se za účelem ověření, zda software splňuje požadavky, které byly ve smlouvě ujednány.

Naproti tomu regulační akceptační testování je proces kontroly, při kterém se ověřuje, jestli software splňuje vládní či právní předpisy. Tento druh testování se provádí s cílem zajistit, aby software vyhovoval platným předpisům a normám.

Konečným cílem smluvního i regulačního akceptačního testování je zajistit, aby software splňoval požadavky uvedené ve smlouvě nebo předpisech.

2.4 Automatické vs manuální testování

Další možností, jakou pohlížet na rozdělení testování, je rozlišení toho, zda testování probíhá automaticky nebo manuálně. Automatické a manuální testování software se používá k ověřování funkčnosti aplikace a oba druhy testování mají své výhody a nevýhody.

- **Automatické testování** je proces, při kterém se testování provádí bez lidského zásahu pomocí jiného software. Tento proces je rychlý a efektivní a může být použit k otestování velkého množství různých vstupů a scénářů. Díky tomu, že lidský faktor je vyřazen z procesu testování, snižuje se tak míra chybovosti způsobená lidským faktorem. Nevýhodou automatického testování je počáteční investice do infrastruktury vhodné pro automatické testování a horší testování uživatelského rozhraní či uživatelské přívětivosti. Automatické testování je podrobněji popsáno v kapitole č. 4.
- **Manuální testování** je proces, při kterém tester ručně testuje funkčnost aplikace. Nevýhody manuálního testování zahrnují nižší rychlost oproti automatickému testování a vyšší náchylnost k chybám způsobeným lidským faktorem. Na druhé straně manuální testování poskytuje testerům výhodu v podobě plné kontroly nad procesem testování a umožňuje jim tak odhalit chyby, které by automatické testování nemuselo zaznamenat. Manuální testování je také cenově výhodnější oproti automatickému testování, zejména u menších projektů.

Obecně se oba přístupy kombinují pro dosažení nejlepšího možného výsledku testování.

2.5 Dynamické vs statické testování

Dalším pohledem, jak lze rozdělit testy, je podle toho, zda je potřeba testovaný software spustit či nikoliv. Tato podkapitola vychází z [3]. Dynamické testy ke svému provedení vyžadují software, který lze spustit. Z tohoto důvodu jsou používány v pozdějších fázích vývoje a slouží k ověření chodu aplikace. Statické testy se používají v raných fázích vývoje ještě před tím, než je k dispozici funkční prototyp aplikace, a napomáhají kontrole požadavků a analýze zdrojového kódu.

2.6 Techniky testování software

V současnosti existuje velké množství publikovaných technik testování software, lépe řečeno je jich více než 200. Některé z těchto technik se mohou překrývat, doplňovat či mohou být použity ve spojení s jinými. Využití kombinace těchto technik však může vést k ucelenějšímu testování. [4]

2.6.1 Black-box

Black-box je technika testování software. Tato technika se zaměřuje na funkčnost aplikace a její očekávané vstupy a výstupy. Vychází z požadavků a specifikací software a nevyžaduje znalost vnitřního fungování a kódu. Tester hodnotí výhradně vnější vstupy a výstupy systému, aniž by rozuměl vnitřnímu fungování kódu. [5],[6]

Obrázek 1: Black-box
(zdroj: vlastní zpracování)

Mezi benefity black-box techniky patří : [5],[6]

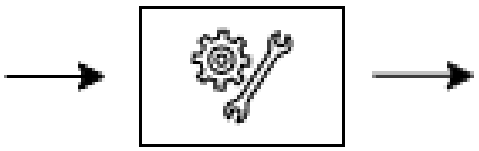
- Testování se provádí z pohledu požadavků zákazníka.
- Testeři nepotřebují znalost programování nebo implementace.
- Testování je efektivní i pro rozsáhlejší systémy.
- Pohled uživatelů je oddělen od pohledu vývojářů, tzn. programátor a tester jsou na sobě nezávislí (vývojář je ovlivněn implementací).

Mezi negativa black-box techniky lze uvést následující: [5], [6] ,[7]

- Návrh testů může být složitý, pokud nejsou požadavky jasně definovány.
- Některé častí backendu nemusí být otestovány vůbec.
- Provádí se jen omezený počet testovacích scénářů, což má za následek omezené pokrytí testy.
- Výsledky těchto testů bývají často nadhodnoceny.

2.6.2 White-box

White-box je technika, která se zaměřuje na interní logiku a strukturu testované aplikace. Pro její použití jsou potřeba komplexní informace o testované aplikaci a znalost jazyka, ve kterém je aplikace napsaná. White-box se snaží o otestování všech možných cest a podmínek v kódu. Tato technika vyžaduje zkušené testery se znalostí vnitřní implementace. [5]



Obrázek 2: White-box
(zdroj: vlastní zpracování)

Mezi benefity white-box techniky patří: [5],[6]

- Takto testovaná aplikace má větší pokrytí testy.
- Testování může pomoci s optimalizací kódu.
- Testovací scénáře jsou psány tak, aby bylo dosaženo maximálního pokrytí.
- Testování začíná v ranější fázi vývoje, jelikož není závislé na GUI.

Mezi negativa white-box techniky lze uvést následující: [5], [6]

- Výdaje na tento typ testování mohou být vysoké, protože vyžadují kvalifikované testery.
- Neustálé změny v implementaci vyžadují nepřetržitou údržbu testů.

2.6.3 Grey-box

Grey-box testování (testování šedé skříňky) je technika, která kombinuje prvky testování černé a bílé skříňky. Při testování černé skříňky tester nezná vnitřní strukturu testovaného produktu, zatímco při testování bílé skříňky je vnitřní struktura známa. Při testování šedé skříňky je vnitřní struktura známa pouze z části. To znamená, že tester má přístup k vnitřním datovým strukturám a algoritmům pro návrh testovacích případů, ale testování probíhá v rámci GUI, nebo na úrovni testování černé skříňky. [7]

Obrázek 3: Grey-box
Zdroj: vlastní zpracování

Testování šedé skříňky kombinuje výhody testování černé a bílé skříňky. Tester se zaměřuje spíše na definici rozhraní a funkční specifikaci než na zdrojový kód. Díky kombinaci přístupu černé a bílé skříňky může tester navrhnout velice efektivní testovací případy. [6]

Nevýhodou testování pomocí šedé skříňky může být opakování předchozích testů, které byly napsány v rámci bílé či černé skříňky. [7]

2.6.4 Regresní testování (regression testing)

Regresní testování software je opakované testování, které slouží ke zjištění, zda nové změny v aplikaci negativně neovlivnily její funkčnost. Obvykle se toto testování provádí po provedení změn v kódu aplikace, aby bylo zajištěno, že tyto změny nezpůsobily žádné nové chyby ani nezpůsobily selhání jakýchkoli stávajících funkcí. Potřeba regresního testování roste spolu s popularitou agilního vývoje a je třeba zajistit, že nové změny nerozbijí stávající funkcionalitu. Průzkumy ukazují, že 80 % nákladů na testování software tvoří regresní testování. Regresní testování může být prováděno v průběhu různých fází projektu, v různých úrovních a s odlišnou frekvencí. Regresní testy lze provádět buď manuálně nebo automatizovaně, v praxi se většinou oba přístupy kombinují.[8]

2.6.5 Stress testování (stress testing)

Stress testování znamená použít testovaný software v provozu a sledovat, jak software pracuje v extrémních podmínkách. Cílem stress testování je zjistit, zda software zvládne nadměrnou zátěž, aniž by došlo k jeho selhání. Stress testování zahrnuje generování velkého množství uživatelů, které současně daný software používá, nebo záměrně vystavit software extrémním vstupům či podmínkám s cílem sledovat jeho limity. V některých situacích lze považovat zhoršený výkon systému při extrémní zátěži za zcela přijatelný, interpretace výsledku daného stress testu je spíše subjektivní. Stress test, který zatěžuje část testovaného software, může vést k nežádoucím vedlejším účinkům v jiné oblasti software. Z tohoto důvodu je třeba pro správnou analýzu výsledů vyhodnotit chování celého systému jako celku. [9]

2.6.6 Testování na základě rizik (risk-based testing)

Testování na základě rizik je přístup k testování software, při kterém se určí priority testů na základě rizika jejich selhání, obvykle na začátku vývojového cyklu. Smyslem je organizovat testovací úsilí tak, aby se snížila míra rizika v době dodání testovaného software. Testování na základě rizik by se dalo popsat jako prioritizace testování funkcí, u kterých je největší pravděpodobnost, že selžou. Čím větší je pravděpodobnost selhání nějaké funkce software, tím důležitější je její včasné a důkladné otestování. Při vývoji projektu s omezenými finančními či časovými zdroji je nutné, aby testování pokrylo primárně kritické části scénářů užití s vysokou prioritou, aby byla zajištěna co nejvyšší kvalita při co nejnižších nákladech.
Proces testování na základě rizik lze popsat následovně:

- Popsání všech požadavků z hlediska rizik, která jsou s nimi spojena. Analýza místa a důvodů výskytu rizik
- Stanovení priorit požadavků na základě posouzeni rizik kritických, složitých a potenciálně náchylných k chybám
- Podle prioritizace požadavků definovat a naplánovat testy
- Provedení testů podle priorit a akceptačních kritérií

Mezi výhody testování na základě analýzy rizik patří:

- Provádění testů v pořadí podle rizika přináší nejvyšší pravděpodobnost odhalení chyb podle závažnosti.
- V případě omezených časových, finančních a kvalifikovaných personálních zdrojů se klade důraz na testování nejdůležitějších testovacích scénářů.
- Jednotlivá rizika lze průběžně sledovat a zjistit tak stav projektu a jeho kvalitu.

Cílem testování založeného na analýze rizik je co největší pokrytí a efektivní využití omezených zdrojů. [10]

2.6.7 Uživatelské testování

Uživatelské testování software zahrnuje koncové uživatele, kteří daný software testují, aby se ověřila jeho použitelnost a uživatelská přívětivost (user experience). Cílem uživatelského testování je zajistit, aby byl software snadno použitelný pro co nejširší okruh uživatelů. [11]
Uživatelské testování zahrnuje následující kroky: [11]

- Před začátkem testování je nutné vytvořit skupinu uživatelů, kteří budou software testovat. Tato skupina by měla co nejvíce připomínat skutečné uživatele, kteří budou software používat. Do této skupiny se mohou řadit i budoucí uživatelé daného software (zákazníci).
- Je třeba připravit prostředí, ve kterém se budou provádět uživatelské testy. Do tohoto kroku patří zajištění vybavení, které budou uživatelé používat, a zajištění všech potřebných aplikací a dat, která budou uživatelé v průběhu testu používat.
- Zajištění scénářů a úkolů, které budou uživatelé během testu plnit. Tyto scénáře a úkoly by měly odpovídat způsobu, jakým budou uživatelé software reálně používat. Tyto scénáře by měly být jasně formulované a srozumitelné pro uživatele a měly by obsahovat různé typy úkolů, které by mohly software vystavit různým výzvám a potencionálním problémům.
- Po přípravě uživatelů, prostředí a scénářů je zahájeno samotné uživatelské testování. Uživatelé v tomto kroku plní zadané scénáře a úkoly a dávají najevo své myšlenky a pocity při používání software. Evaluátor sleduje uživatele, naslouchá jejich myšlenkám a pozoruje, jak uživatelé se softwarem pracují.

- Posledním krokem je analýza a vyhodnocení výsledků. Cílem je identifikovat problémy s použitelností, které byly během testu zjištěny. Výsledkem uživatelského testování jsou poznatky o přívětivosti a použitelnosti testovaného software.

Uživatelské testování umožňuje vývojářům a designerům zjistit, zda je software snadno použitelný a splňuje požadavky na uživatelskou přívětivost. [11]

2.6.8 Volné testování (exploratory testing)

Volné testování je druh testování software, při kterém tester používá testovaný software bez předem připravených scénářů a snaží se zjistit, zda software funguje správně a splňuje požadavky uživatelů. Tester používá software tak, jak by se používal v reálném prostředí, a snaží se objevit potencionální nedostatky. [12]

Mezi vlastnosti volného testování patří: [12]

- Nejsou definované test-scenarios. Tester provádí testování bez konkrétních pokynů.
- Tester se řídí výsledky předchozích testů. Používá veškeré dostupné informace o cíli testování, jako jsou požadavky na software nebo dokumentace. Tester využívá tyto požadavky při plánování a provádění testů, aby mohl co nejlépe ověřit kvalitu a funkčnost testovaného software.
- Při testování je kladen důraz na odhalování chyb pouhým průzkumem, a ne pomocí sady testovacích scénářů.
- Efektivita volného testování závisí na schopnostech, zkušenostech a znalostech testera, který dané testování provádí.

Ve většině případů se volné testování provádí jako doplněk k ostatním typům testování a je považováno za užitečnou techniku při testování software. [12]

2.7 Test management

Organizace ISTQB definuje test management jako *„The planning, scheduling, estimating, monitoring, reporting, control and completion of test activities“*. [13] Jedná se tedy o proces koordinace, plánování a kontroly testovacích činností projektu. Test

management zahrnuje plánování a organizaci testování, jakož i sledování a podávání zpráv o průběhu a výsledcích testování. Test management může mimo jiné zahrnovat i identifikaci a obstarání zdrojů pro testování (např. lidí, hardware, software) a také odhad úsilí potřebného pro testování software. Test managment může být podpořen speciálními nástroji.
Jednotlivé části výše uvedené definice a test management nástroje jsou blíže popsány v následujících podkapitolách a vychází z pokročilého sylabu pro test manažery „Advanced Level Syllabus Test Manager“ dostupného na [14] od ISTQB organizace.

2.7.1 Plánování testů

Plánování testů je proces organizace činností a zdrojů k provedení testování software pro nějaký projekt. Plánování zahrnuje určení strategií a metod, které budou použity k testování v rámci projektu, měření jeho průběhu testování a hodnocení toho, jestli byly splněny cíle, které byly ohledně testování nastaveny. Plánování testů zahrnuje také určení metrik, které budou použity ke sledování pokroku projektu, jako je počet nalezených chyb a dosažené pokrytí kódu.

Konkrétní úkoly, které jsou zahrnuty do plánování testů, budou záviset na zvolené strategii testování pro daný projekt. Ve fázi plánování manažer plánování definuje rozsah testování včetně konkrétních funkcí software, které budou testovány, a těch, které testovány nebudou. Manažer testování dále spolupracuje s architekty projektu na definování počáteční specifikace testovacího prostředí a zajistí, aby byly k dispozici potřebné zdroje pro testování projektu.

2.7.2 Monitorování testů

Pro efektivní monitorování testování nad projektem by měl manažer testování vytvořit rámec, který zahrnuje plán a systém monitorování. Tento rámec by měl umožnit sledování úkolů, výstupů a zdrojů spojených s testováním a porovnání s celkovým plánem testování projektu. Aby bylo zajištěno, že testování je v souladu s plánem a se strategickými cíli, měl by rámec obsahovat konkrétní opatření a cíle, které lze použít k vyhodnocení průběhu a efektivity testování. U menších a méně složitých projektů může být sladění testování s plánem jednoduší, ale obecně je k tomu nutné definovat podrobnější cíle. Těmito cíli se rozumí konkrétní cíle nebo úkoly, které jsou pro testování stanoveny. Je důležité, aby

bylo možné stranám zainteresovaným do testování jasně sdělit stav testování způsobem, který je relevantní a smysluplný pro daný projekt. K tomu může dopomoci definování dříve uvedených cílů. Testování by tedy mělo být průběžně monitorováno a řízeno podle potřeby, aby bylo zajištěno, že stav projektu je v souladu se stanovenými cíli projektu.

2.7.3 Odhadování testů

Odhad testování je proces vytváření odhadů nákladů, úsilí a doby trvání testovacích činností spjatých s testovaným projektem. Odhad testování je důležitým aspektem při řízení projektu a používá se k plánování a přidělování zdrojů pro testovací činnosti. Odhad testů by měl zohlednit všechny faktory, které mohou ovlivnit náklady, úsilí a dobu trvání testů. Mezi tyto faktory mimo jiné patří:

- Požadovaná úroveň kvality systému
- Velikost testovaného systému
- Dovednosti a zkušenosti testovacího týmu
- Procesní faktory včetně strategie testování a SDLC
- Materiální faktory včetně automatizace testování a nástrojů, testovací prostředí, testovací data a projektová dokumentace
- Osvojení nebo vývoj nových nástrojů, procesů a technik pro testování software

Odhadování testů lze provést buď top-down nebo bottom-up přístupy[2]. Existuje několik technik, které lze použít jednotlivě či v kombinaci k odhadu nákladů, úsilí a doby trvání testovacích činností v projektu. Mezi tyto techniky patří využití intuice a předchozí zkušenosti nebo využití firemních standardů a norem. Je důležité mít na paměti, že každý odhad vychází z informací z doby, kdy byl vytvořen. Na počátku projektu mohou být informace značně omezené, anebo se mohou v průběhu času změnit. Aby zůstala zachována přesnost těchto odhadů, je důležité odhady aktualizovat tak, aby odrážely nové a změněné informace.

[2] V tomto kontextu se jedná o postupy, které určují pořadí, v jakém by měly být testy prováděny. Top-down jako první testuje celkovou funkčnost software a až poté následuje testování konkrétních modulů. Oproti tomu bottom-up přístup jako první testuje jednotlivé komponenty a poté následuje jejich integrace a testování jako celku.

3 DevOps a agilní metodiky v testování software

DevOps a Agilní přístup jsou techniky, které se zaměřují na zlepšení efektivity a produktivity týmů a urychlení doručování nových funkcí k zákazníkovi. Přestože jsou oba pojmy velmi podobné, existují mezi nimi určité rozdíly, které jsou vysvětleny níže.

Zatímco DevOps se zaměřuje na celý životní cyklus aplikace od vývoje až po provoz, agilní vývoj se zaměřuje pouze na vývoj software. DevOps vyžaduje silnou spolupráci mezi vývojáři a operačními týmy a zahrnuje procesy pro automatizaci testování a nasazování aplikací. Agilní vývoj klade důraz na týmovou spolupráci a schopnost rychle reagovat na změny. Oba přístupy se nevylučují a jsou užitečné při zlepšování efektivity a produktivity týmů a při urychlení vydávání nových funkcí aplikace. Mohou být použity v kombinaci k dosažení maximalizace výše zmíněných výhod. [15]

Ještě před samotným popisem testováním v rámci DevOps a Agilních metodik jsou tyto přístupy blíže vysvětleny v následujících dvou podkapitolách.

3.1 Agilní vývoj

Agilní vývoj software je metoda vývoje software, která se zaměřuje na flexibilitu a schopnost rychle reagovat na změny. Tato metoda vyžaduje úzkou spolupráci mezi vývojáři a zákazníky a podporuje sebeorganizaci vývojových týmů. K dosažení výše zmíněných cílů se využívají agilní postupy, které zlepšují komunikaci během vývoje software. Nejdůležitější charakteristikou agilního vývoje je vývoj v iterativních cyklech, kdy se práce rozdělí do kratších časových úseků nazývaných sprinty. Mezi další charakteristiky patří pravidelné standupy, plánování sprintů a vývojových iterací a retrospektiva. [16] Mezi základní metody agilního vývoje patří:

- Extreme programming
- Scrum
- Feature-driven Development (FDD)
- Kanban

Tyto metody jsou podrobněji popsány níže.

3.1.1 Agilní vs tradiční přístup

Ještě před popisem samotných agilních metodik objasňuje tato podkapitola rozdíl mezi agilním a tradičním přístupem k vývoji software.

V úvodu kapitoly č. 3 jsou popsány základní principy Agilního přístupu k vývoji software, z tohoto důvodu je v této podkapitole popsán pouze tradiční přístup následovaný rozdíly mezi těmito přístupy.

Tradiční přístup k vývoji software je sekvenční proces, v němž se postupuje od jedné fáze k druhé. Jedná se o čtyři fáze, které jsou popsané níže [17].

1) **Definice požadavků a stanovení časového harmonogramu**. Tato fáze zahrnuje také identifikaci potencionálních problémů během vývoje.
2) **Návrh architektury a plánování**. V této fázi se vytváří technická infrastruktura ve formě diagramů či modelů. Tyto diagramy a modely pomáhají identifikovat potencionální problémy, s nimiž se projekt může potýkat, a poskytují vývojářům plán pro implementaci.
3) **Vývoj**. Tato fáze zahrnuje samotné programování pro dosažení požadavků z předchozí fáze. Úkoly jsou rozděleny mezi týmy podle jejich odbornosti.
4) **Testování a vydání**. Fáze testování se může prolínat s třetí fází vývoje, aby se včas identifikovaly a odstranily problémy, které během vývoje nastanou. Jakmile se blíží dokončení projektu, zákazník je zapojen do testování a poskytuje zpětnou vazbu. Výsledný software je dokončený a vydaný v momentě, kdy je zákazník spokojený s výsledkem.

Agilní přístup oproti tradičnímu dokáže úspěšně dodat projekty s nejasně definovanými požadavky. Toho je dosaženo díky tomu, že agilní přístup na rozdíl od tradičního přístupu nevyžaduje úplné a konečné definování požadavků na začátku vývoje. Z tohoto hlediska je agilní přístup schopen dodat produkt, které lépe reflektuje požadavky zákazníka, jelikož tyto požadavky jsou od zákazníka získávány iterativně. [17]

Další rozdíly jsou krátce uvedeny v tabulce č. 1

Tabulka 1: Rozdíly mezi agilním a tradičním přístupem

	Agilní přístup	**Tradiční přístup**
Uživatelské požadavky	Iterativní, interaktivní získávání	Definovány před začátkem vývoje
Náklady na přepracování	Nízké	Vysoké
Směr vývoje	Snadno měnitelný	Pevný
Testování	Kontinuální	Po dokončení vývoje
Zapojení uživatelů	Vysoké	Nízké
Požadavky	Emergentní	Stabilní, předem známé
Vhodný rozsah projektu	Menší až středně velký	Velký
Kontrola kvality	Pravidelná kontrola, reflektování aktuálních požadavků, stálé testování	Složité plánování a pozdní testování

Zdroj: [17], [18], *upraveno autorem*

3.1.2 Extrémní programování

Extrémní programování se zaměřuje na zrychlení vývoje software a zlepšení kvality pomocí zlepšení komunikace a spolupráce mezi jednotlivými členy týmu. Techniky extrémního programování zahrnují: [19]

- Filozofii vývoje software založenou na hodnotách komunikace, zpětné vazby, jednoduchosti a respektu
- Krátké vývojové cykly vedoucí k brzké zpětné vazbě od uživatele
- Schopnost flexibilně plánovat implementaci funkcionalit s reagováním na měnící se potřeby zákazníka
- Využívání automatických testů, které umožňují dřívější odhalení chyb.
- Úzká spolupráce aktivně zapojených vývojářů
- Postupy, které jsou v souladu jak s krátkodobými, tak dlouhodobými cíli.

Extrémní programování se mimo jiné zabývá riziky na všech úrovních vývoje software. Řešení jednotlivých rizik pomocí extrémního programování je uvedeno v tabulce níže.

Tabulka 2: Rizika a jejich řešení pomocí extrémního programování

Plánování skluzů	Díky krátkým vývojovým cyklům je velikost časového skluzu omezená. Prioritní implementaci zajišťuje včasné dodání nejdůležitějších funkcionalit, tudíž časový skluz implementace méně důležitých funckionalit nebude tak velký.
Zrušení projektu	Extrémní programování vyžaduje od obchodní části organizace, aby vybrala nejmenší možnou verzi, která má největší obchodní smysl. V případě zrušení projektu organizace přijde o méně času a investic, než když by začala s vyvíjením celého projektu.
Pád aplikace	Extrémní programování vytváří a udržuje komplexní sadu automatizovaných testů, které se spouštějí a opakují po každé změně, aby byla zajištěna dostatečná úroveň kvality.
Business změny	Během vývoje může dojít k tomu, že bude zákazník požadovat změnit či přidat novou funkcionalitu. Díky krátkým vývojovým cyklům si vývojový tým ani nevšimne, zda pracuje na nové funkcionalitě, nebo na funkcionalitě definované před několika lety.
Fluktuace vývojářů	Extrémní programování žádá od vývojářů, aby přijali odpovědnost za časový odhad dokončení své práce. Organizace dále přebere tento odhad a zaplánuje jej. Tím se snižuje pravděpodobnost, že programátor bude frustrován tím, že se po něm bude požadovat zjevně nemožné.

Zdroj: [19]

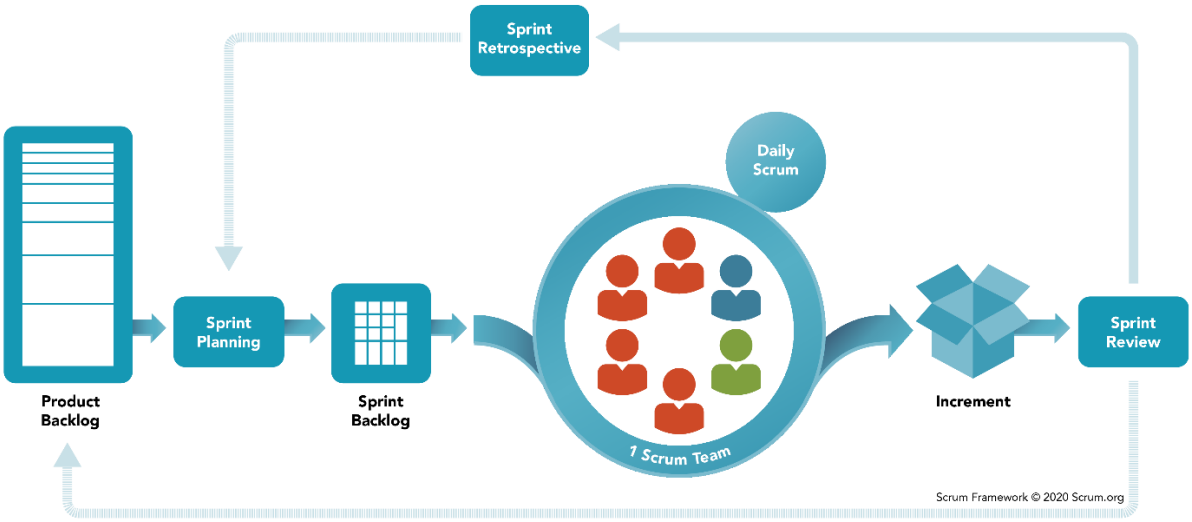
3.1.3 Scrum

Scrum je metodika vývoje software, která se zaměřuje na průběžné zlepšování procesů vývoje a komunikace v rámci týmu. Metodika Scrum rozděluje vývoj na krátké vývojové období nazývané sprint, které obvykle trvá od dvou do čtyř týdnů. Každý sprint k stávajícímu software přidá malý přírůstek (inkrement), který může být dodán koncovým uživatelům. Po konci každého sprintu následuje retrospektivní schůzka týmu, na které se

vývojáři spolu se scrum mastrem snaží najít způsoby, jak zefektivnit proces vývoje v nadcházejícím sprintu. [20]

Scrum definuje následující role: [20]

- **Scrum master** je osoba, která je odpovědná za dodržování metodik Scrumu a pomáhá týmu pracovat efektivněji. Scrum master není vedoucí týmu ani nadřízený, ale kouč, který týmu pomáhá aplikovat metody Scrumu.
- **Product owner** je osoba, která zastupuje zákazníka. Určuje priority vývoje a má na starosti, jaké funkcionality se budou v daném sprintu vyvíjet. Product owner není vedoucí týmu, ani nadřízený týmu.
- **Development team** je skupina vývojářů, která vyvíjí a testuje produkt. Tým je samostatně organizovaný a nemá týmového vůdce. Tým dělá rozhodnutí jako celek.



Obrázek 4: Scrum flow
Zdroj: [21]

Flow Scrum metodiky je následující: [20]

1) Proces začíná vytvořením vize projektu, která je následně formulována pomocí plánu zvaného Backlog. Jedná se o seznam funkčních a nefunkčních požadavků, jejichž realizací bude dosaženo vize projektu. Backlog je sestaven tak, aby

nejdůležitější položky byly v seznamu nahoře. Pořadí backlogu se může v průběhu vývoje měnit.

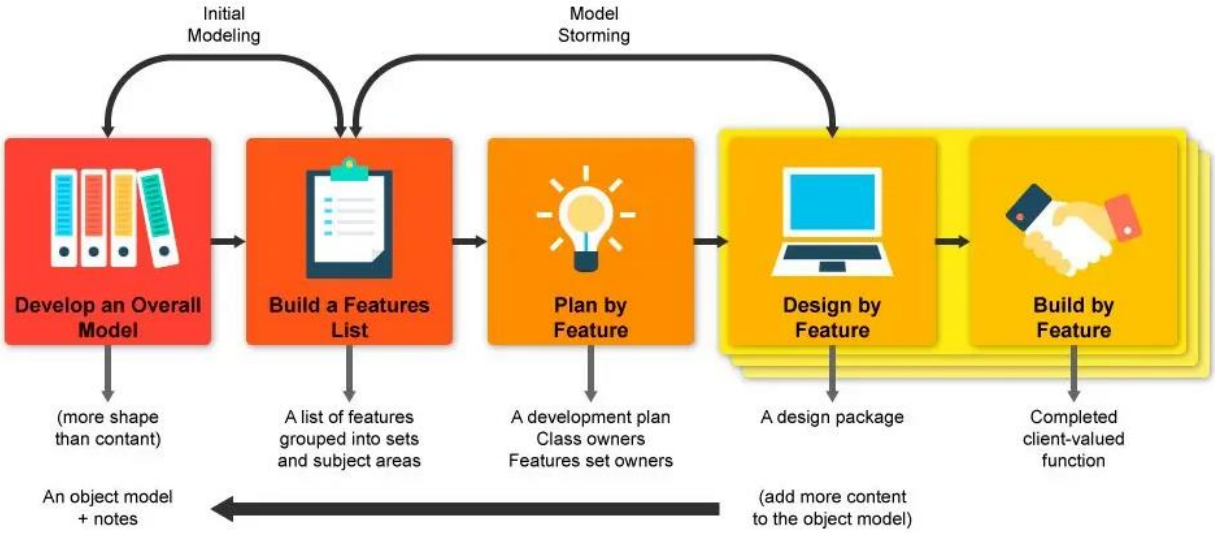
2) Projekt je rozdělen do jednotlivých sprintů, které jsou zahájeny plánovací schůzkou. Na této schůzce project owner představí nejdůležitější položky z backlogu a tým se rozhodne, kolik lze stihnout úkolů v nadcházejícím sprintu. Na základě tohoto rozhodnutí tým vytvoří seznam úkolů pro konkrétní sprint zvaný Sprint Backlog.
3) Tým má každý pracovní den společnou schůzi, na které každý člen informuje o svém pokroku od poslední schůze, plánu na daný den a o problémech, se kterými se aktuálně během vývoje potýká.
4) Na konci každého sprintu má tým schůzku, na které se představí dokončená práce project ownerovi a dalším zainteresovaným stranám. Poté následuje retrospektivní schůze, na které členové týmu diskutují o tom, co se během sprintu povedlo, a o tom, co by mohlo být v nadcházejícím sprintu zlepšeno.

3.1.4 Feature-driven development

Feature-driven development (FDD) je metodika vývoje software, která klade důraz na vývoj a dodávání malých postupných funkcí. Jedná se o agilní proces, jehož cílem je rychlé dodání funkčního software pomocí krátkých vývojových cyklů zvaných iterace. Při FDD se klade důraz na rozdělení procesu vývoje na malé zvládnutelné části, přičemž každá část představuje konkrétní funkci nebo schopnost, kterou lze dodat uživateli. Každá funkce je navržena, implementována a testována v samostatné iteraci, což umožňuje rychlejší dodávání a průběžné zlepšování software. FDD definuje následující role: [22]

- **Projektový manažer** je vedoucí pracovník vývojového týmu, který zodpovídá za vedení celého projektu. Je zodpovědný za řízení jednotlivých vývojářů a určuje rozsah a harmonogram projektu. Projektový manažer se dále zabývá rozpočtem projektu a s ním spjatými finančními záležitostmi.
- **Hlavní architekt** je osoba, která je zodpovědná za celkový design a návrh systému a jeho slovo je považováno za rozhodující. Mimo jiné také řídí a zajišťuje potřebná školení týmu.

- **Vývojový manažer** je osoba, která má za úkol řízení vývojového týmu. Dohlíží na to, aby všichni jeho členové plnili své úkoly. Mimo jiné také řeší konflikty mezi členy týmu.
- **Hlavní programátor** je osoba, která skutečně řídí vývoj. Vzhledem k nutnosti bohatých zkušeností z dané oblasti vývoje software hlavní vývojář dohlíží na kvalitu vývoje, ale také na to, aby proces vývoje probíhal hladce a efektivně.
- **Class owner** neboli „vlastník třídy" je vývojář, který odpovídá za návrh, kódování a testování sad funkcí přidělených hlavním programátorem. Tento vývojář pracuje pod dohledem hlavního programátora.
- **Doménový expert** může být osoba či více osob, které mají relevantní znalosti v oboru, pro který se daný software vyvíjí. Tato osoba je často odborník na danou problematiku či profesionál v oboru, který může poskytnout cenné informace o funkčních požadavcích na vyvíjený software. Může určovat směr vývoje díky poskytování svých znalostí, které přispívají k tomu, aby byl software relevantní pro zamýšlené uživatele. Musí mít dobré komunikační dovednosti, aby dokázal předat své požadavky vývojovému týmu.
- **Feature tým** je dočasná skupina vývojářů, kteří pracují na určité sadě funkcí v průběhu jedné iterace. Tento tým je rozpuštěn v momentě, kdy je daná sada funkcí naimplementována a přijata do hlavního buildu aplikace.



Obrázek 5: FDD flow
Zdroj: [23]

Flow FDD metodiky je následující: [22]

1) **Vypracování celkového modelu**. V této fázi pracuje tým na definování kontextu a rozsahu projektu. Za tímto účelem uspořádá tým schůzku, na které řeší vysokoúrovňové aspekty funkčností, které jsou potřeba pro splnění projektu. Následuje podrobnější analýza jednotlivých aspektů. Na základě této analýzy vzniká několik objektových modelů, které jsou následně přezkoumány a jeden z nich je vybrán jako model pro danou oblast domény. Tento model může být v pozdějších fázích vývoje dále zpřesňován.
2) **Sestavení seznamu funkcí.** Jedná se o funkce, které mají v software určitou obchodní vlastnost. Po vytvoření objektového modelu z první fáze je pro tým snazší takové funkce identifikovat a seskupit je do ucelených seznamů funkcí, které je třeba vyvinout. Jednotlivé funkce v seznamu by měly být implementovány maximálně do dvou týdnů. V opačném případě by měla být funkce rozdělena na menší úkoly. Celkový seznam funkcí je nakonec schvalován zákazníkem.

3) **Plánování na základě funkcí**. V této fázi se přiřazují priority jednotlivým funkcím na základě následujících faktorů:
    a. Závislosti mezi funkcemi
    b. Související rizika mezi funkcemi
    c. Složitost implementace dané funkce
    d. Pracovní vytížení týmů

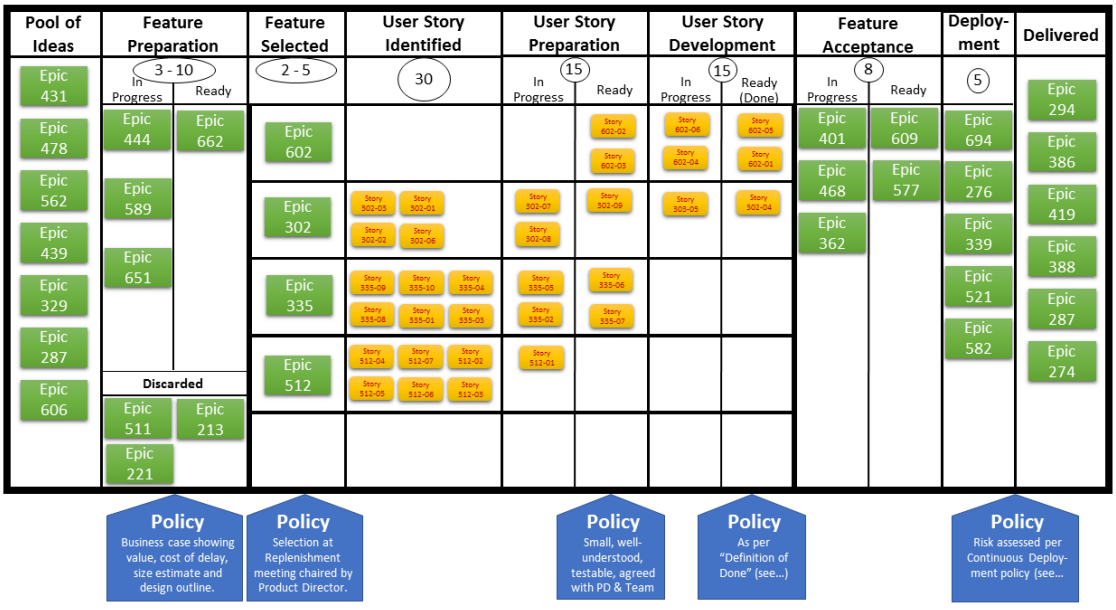
    Hlavní programátor následně přiřadí každou funkci konkrétnímu vývojáři.
4) **Designování na základě funkce**. Tato fáze probíhá v iterativních cyklech a trvá od několika dnů do maximálně dvou týdnů. V této fázi probíhá vývoj jednotlivých funkcí, jejichž výsledná implementace je následně přezkoumána a zkontrolována.
5) **Sestavení na základě funkce**. Stejně jako ve čtvrté fázi probíhá sestavení na základě funkce v iterativních cyklech. Po implementaci následují unit testy a integrační testování. Po dokončení testování jsou jednotlivé funkce zahrnuty do hlavního buildu aplikace a následuje další iterace s novými funkcemi. Každá funkce musí dosáhnout následujících milníků:
    a. **Projití domény** (přezkoumání návrhu konkrétní funkce)
    b. **Design**. V tomto milníku dokončuje tým návrh konkrétní funkce, přičemž zohledňuje zpětnou vazbu, která byla provedena v předchozím milníku.
    c. **Prověření designu**. Tento milník zahrnuje přezkoumání návrhu konkrétní funkce skupinou odborníku na danou doménu, aby se odhalily případné problémy s návrhem.
    d. **Kód**. V tomto milníku tým napíše implementaci funkce na základě konečného návrhu.
    e. **Prověření kódu.** Tento milník zahrnuje kontrolu kódu pro určitou funkci skupinou vývojářů s cílem identifikovat případné problémy s kódem.
    f. **Merge**. V této fázi dochází k začlenění vyvinuté funkce do hlavního buildu aplikace.

3.1.5 Kanban

Kanban je metoda řízení procesu vývoje software, která se snaží zefektivnit řízení vývoje. Kanban napomáhá porozumění a kontrole pracovního procesu, ale také identifikací tzv.

„bottleneck“[3]. Tradiční přístup vývoje software si lze představit jako řetězec, kdy jeden vývojář týmu předává výsledky své práce jinému, tudíž může docházet k prodlevám ve vývojovém cyklu. Pokud by například jeden vývojář týmu měl problémy, které souvisejí s prací jiného vývojáře, budou jeho úkoly opožděné. Kanban oproti tomu využívá „pull systém“, kdy všichni vývojáři týmu musí mít v určitém čase k dispozici pouze jeden úkol, na kterém pracují. Po dokončení daného úkolu může vývojář přejít na jiný úkol. [24]

Kanban přístup ve vývoji software rozděluje práci na malé části, které se nazývají „karty“, které představují jednotlivé úkoly nebo části práce. Tyto karty jsou umístěny na Kanban tabuli, která je rozdělena do sloupců, které představují jednotlivé fáze vývojového procesu. Jakmile je práce dokončena, karty se přesouvají zleva doprava. Toto vizuální znázornění práce pomáhá týmům na první pohled vidět stav jednotlivých úkolů a identifikovat případné „bottlenecks“ nebo jiná zpoždění ve vývojovém procesu. [25]



Obrázek 6: Kanban board
Zdroj: [26]

[3] Situace, kdy je rychlost dokončování práce zpomalena určitým omezením.

3.2 DevOps

DevOps je přístup ve vývoji software, který se zaměřuje na zlepšení propojení mezi vývojáři a IT operacemi. Slovo DevOps se skládá ze dvou pojmů *development* a *operations*. Pojem *operations* v tomto kontextu představuje operační tým, který se stará o nasazování a udržování software napříč prostředími. [27]

S potřebou rychlejšího doručování nových funkcionalit zákazníkům a zkracováním vývojových cyklů vzniká potřeba efektivnější spolupráce mezi vývojáři a týmy pro nasazování software. DevOps umožňuje tyto dva světy spojit a snaží se zajistit, aby byl software kvalitní, bezpečný a zároveň zlepšit efektivitu procesů vývoje, testování a nasazování. Toho je dosaženo pomocí automatizace procesů v kombinaci s kontinuální integrací a kontinuálním doručování.

3.2.1 Principy DevOps

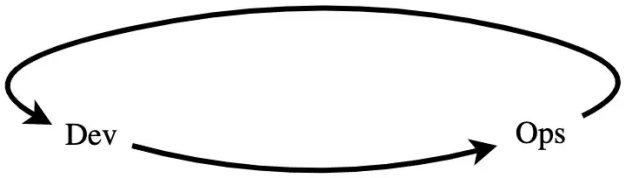
DevOps nemá jasně definované technologie či procesy. DevOps je spíše filozofií. Principy DevOps jsou založeny na tzv. *The Three Ways,* které byly představeny v knize *The Phoenix Project* v roce 2014 : [28]

- Princip první se zaměřuje na tok práce zleva doprava, a to od vývoje k operacím až k zákazníkovi. V rámci snahy o maximalizaci flow se snaží snížit objem rozpracované práce. Snaží se zabránit úniku defektů do pravé části (k zákazníkovi). Mezi nezbytné operace patří kontinuální integrace a doručování.

Obrázek 7: The first way
Zdroj: [29]

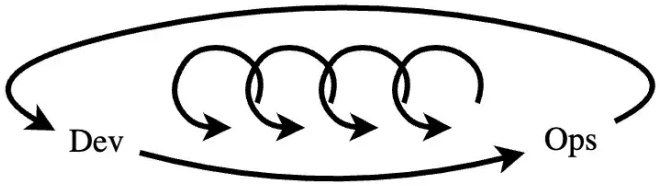
- Princip druhý je o neustálém toku zpětné vazby zprava doleva (od zákazníka), který se používá k opravám nalezených závad v systému. Vytvoří se tím koloběh

zpětné vazby, který umožňuje efektivní zlepšování prostřednictvím řešení nalezených problémů. Tento koloběh umožňuje lépe porozumět potřebám zákazníka.



Obrázek 8: The second way
Zdroj: [29]

- Princip třetí spočívá ve vytvoření kultury, která podporuje obě věci. Neustálé experimentování, které může vyžadovat riskování, a učení se z neúspěchů a úspěchů, což vede ke zlepšování organizace i jednotlivců. Dále pak k pochopení, že opakování je předpokladem tvorby lepšího software. V případě, že nějaký proces selže, neustálé opakování umožní mít dovednosti a návyky, které umožní vrátit se do předchozího funkčního stavu.



Obrázek 9: The third way
Zdroj: [29]

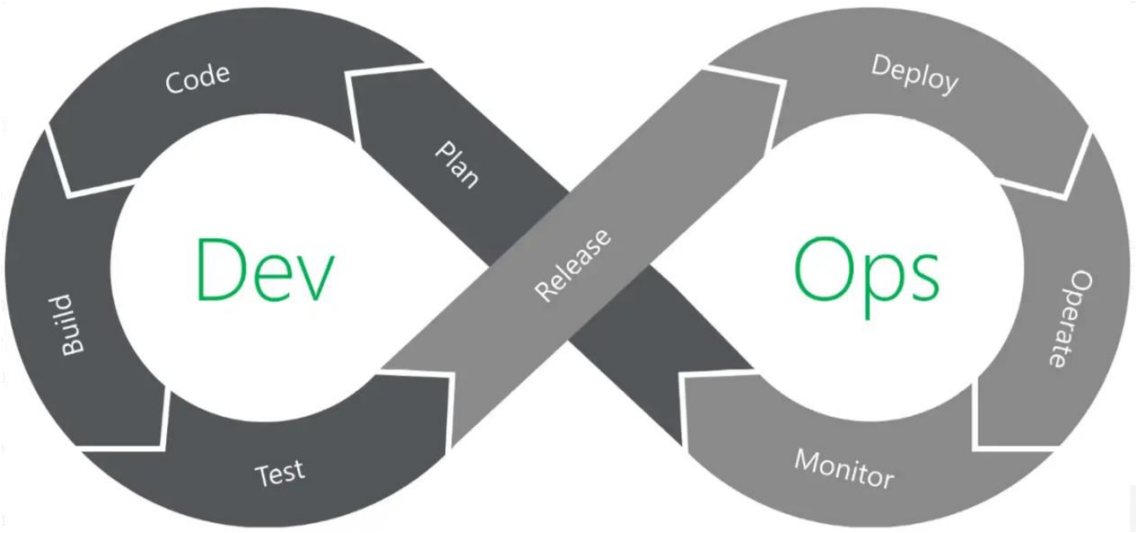
Přijetí DevOps v rámci vývoje software přináší následující benefity: [30]

- Výrazně zkrácená doba dodání software k zákazníkům díky lepší spolupráci mezi vývojářskými a operačními týmy
- Zkrácení vývojových cyklů
- Kontinuální doručování s integrací v kombinací s rychlým doručováním software vedoucí k automatizaci činností
- Podpora konstantní komunikace mezi jednotlivými týmy a podpora spolupráce

- DevOps zvyšuje kvalitu spolupráce a zlepšuje viditelnost procesů doručování software a snižuje rizika lepší analýzou v rámci jednotlivých sprintů[4].
- Automatizace procesů doručování, nasazování a testování
- Důraz na výkonnost celého systému
- Zvýšená viditelnost dat a procesů
- Jednoduchá komunikace mezi zákazníkem a dodavatelem pomocí zpětné vazby od zákazníka
- Nižší náklady během vývoje díky větší produktivitě a výkonu jednotlivých týmů

3.2.2 DevOps pipeline

DevOps pipeline je proces, kterým se organizace řídí při zavádění DevOps do svých procesů a metodik. Jedná se o iterativní kontinuální proces, který začíná plánováním, pokračuje vývojem, sestavením, testováním, vydáním, nasazením, provozem a monitorováním. Každá z těchto fází má svůj vlastní soubor činností, výstupů a cílů a jejich popis vychází z [31].



Obrázek 10: DevOps pipeline
Zdroj: [32]

[4] Sprint je krátké období, během kterého tým vyvíjí konkrétní funkce. Doba sprintu obvykle bývá od jednoho do čtyř týdnů.

- **Plánování** zahrnuje definování požadavků na projekt a jeho realizaci. Různé nástroje, jako je například JIRA, jsou využívány pro plánování a sledování problémů a správu projektu.
- **Vývoj** zahrnuje implementaci nových požadavků. Tato fáze by měla být v souladu s časovým plánem definovaným během plánovací fáze.
- **Sestavení** představuje celý softwarový balíček, který je potřeba ke spuštění software a obsahuje business logiku, softwarové závislosti a prostředí. Účelem sestavovací fáze je ověření správnosti softwarových artefaktů[5].
- **Testování** zahrnuje průběžné automatizované testování, které zajišťuje kvalitu vyvíjeného software.
- **Vydání** je fází, kdy je dokončeno manuální a automatické testování a software je připraven k nasazení do produkčního prostředí. V této fázi je možné zavést manuální schvalování, které umožňuje nasadit novou verzi aplikace do produkce pouze omezenému počtu lidí. To pomáhá zajistit, že nasazovaný software prošel důkladnou kontrolou a je připraven pro koncové uživatele.
- **Nasazení** je fází, ve které se řádně otestovaný software nasazuje do produkčního prostředí. Tato fáze bývá plně automatizovaná, aby se zabránilo chybám způsobeným lidským faktorem. Specifika nasazování se mohou lišit v závislosti na vlastnostech nasazované aplikace.
- **Provozování** se zaměřuje na údržbu a řešení problému aplikací po nasazení do produkčního prostředí. Cílím této fáze je zajistit spolehlivost, vysokou dostupnost a nulové výpadky. Výběr správného hardware nebo škálování jsou jedny z faktorů, které rozhodují o tom, jak bude aplikace provozuschopná v produkčním prostředí a jak bude zvládat vysokou zátěž.
- **Monitorování** je důležitou fází, která probíhá po nasazení aplikace. Jejím cílem je sledování stavu aplikace v produkčním prostředí. Může se jednat o různé incidenty či o monitorování výkonu aplikace.

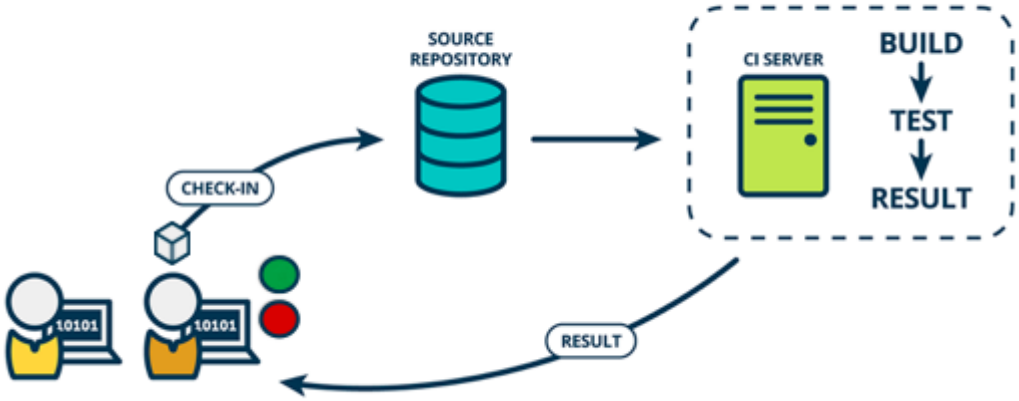
[5] Jedná se o různé součásti či soubory, které tvoří sestavovaný softwarový balíček (např. zdrojový kód, konfigurační soubory, datové soubory…)

3.2.3 Continuous integration

Kontinuální integrace (Continuous Integration – CI) je metoda, která se zaměřuje na častou integraci změn do hlavního buildu aplikace, kde se následně spustí automatizovaný build a testy. Mezí hlavní cíle CI patří: [33]

- Rychlá identifikace chyb
- Zlepšení kvality software
- Zkrácení doby doručování

Kontinuální integrace zahrnuje menší změny v kódu a jejich častější integraci s hlavním buildem aplikace. Vývojář nejdříve stáhne aktuální kód aplikace a sloučí jej se svojí verzí. Tuto sloučenou verzi následně odešle na build server, který spustí různé testy a buď přijme nebo odmítne nové změny. Kontinuální integrace tedy zahrnuje automatizaci procesu integrování a testování změn v kódu. [33]

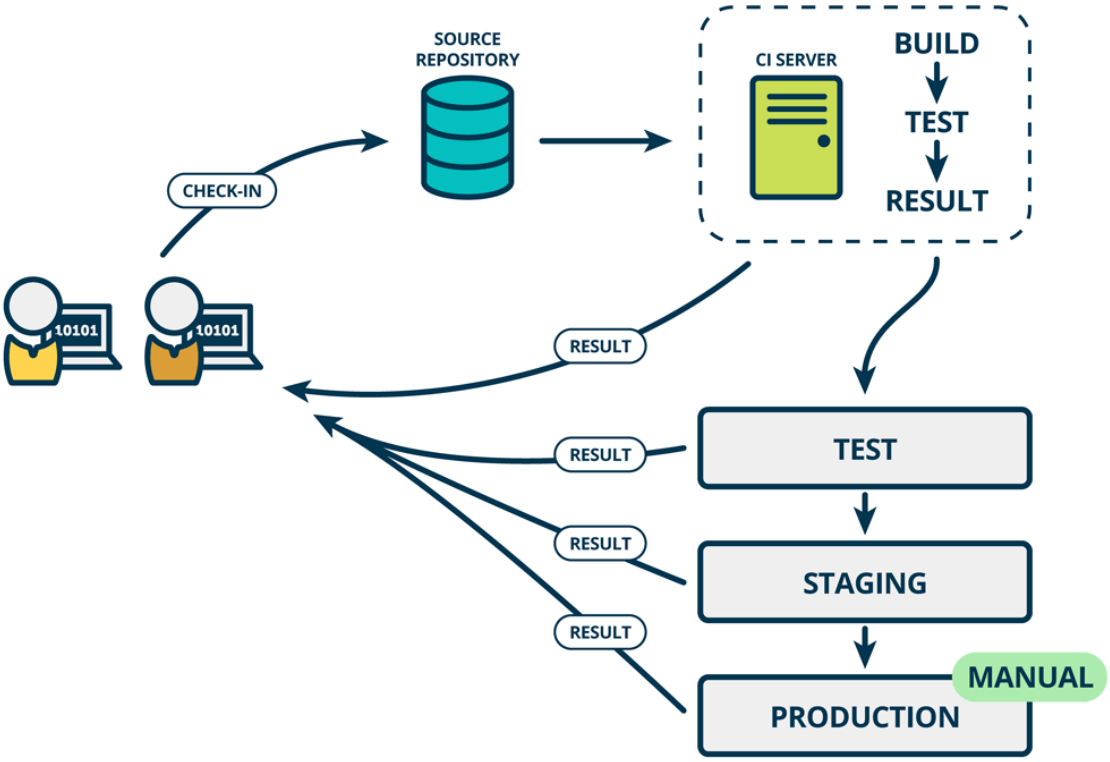


Obrázek 11: Continuous integration
Zdroj: [34]

3.2.4 Continuous delivery

Continuous delivery (CD) je metoda, při které jsou změny v kódu automaticky sestaveny, otestovány a připraveny k nasazení do produkčního prostředí. Tento proces rozšiřuje CI tím, že po úspěšném sestavení a otestovaní se aplikace automaticky nasazuje do testovacího nebo produkčního prostředí. Kontinuální doručování může být plně či částečně automatizováno s manuálními kroky ve významných fázích. V případě

správného nastavení a implementace procesů kontinuálního doručování mají vývojáři k dispozici build aplikace, který prošel všemi fázemi testování a je připraven k nasazení. [33]



Obrázek 12: Continuous delivery
Zdroj: [34]

Mezi výhody kontinuálního doručování patří: [33]

- **Automatizace procesu nasazování** umožňuje týmům efektivně a rychle dodávat na produkční prostředí kód, který byl automaticky sestaven, otestován.
- **Zlepšení produktivity vývojářů** díky jejich osvobození od procesu manuálního sestavování a nasazování aplikace do produkčního prostředí. Vývojáři se tak mohou více soustředit na implementaci potřebných funkcí.
- **Zlepšení kvality kódu** díky automatickým testům, které pomohou odhalit problémy v rané fázi procesu, kdy je jejich odstranění výrazně levnější a jednodušší. Rychlá zpětná vazba díky kontinuálnímu doručování také přispívá k stabilitě a bezpečnosti produkčního kódu.

- **Rychlejší doručování aktualizací.** Organizace, které aplikují CD, mohou rychleji doručovat nové funkce zákazníkům, protože CD zvyšuje rychlost, s jakou může organizace vydávat nové funkce a opravovat chyby. Zavedením kontinuální integrace může organizace rychleji reagovat na:
    - Změny trhu
    - Bezpečnostní problémy
    - Nové požadavky zákazníků

3.2.5 Continuous testing

Kontinuální testování je postup v testování software, při kterém se testování provádí průběžně po celou dobu vývoje software. Jedná se o klíčový aspekt DevOps, protože umožňuje vývojovým týmům rychle identifikovat a následně odstraňovat chyby v software, což vede k jeho zkvalitňování. [35]

Kontinuálním testováním během vývoje mohou týmy včas zachytit problémy a řešit je dříve, než bude jejich oprava složitější a nákladnější. To pomáhá zabránit technickému dluhu, což je hromadění menších, v danou chvíli přijatelných vad, které však mohou vést k závažnějším problémům v budoucnu. [36]

Kontinuální testování zahrnuje automatizaci co největší části procesu testování tak, aby se testy spouštěly automaticky při změnách v kódu. To napomáhá zajistit, aby nové změny kódu neporušily stávající funkcionalitu, nebo nezpůsobily nové chyby. Kontinuální testování lze aplikovat na různé typy testů včetně unit, integračních a akceptačních.

3.3 DevOps testování

V rámci DevOps je testování klíčovou součástí procesu vývoje a integrace software. Jeho cílem je zajistit, aby byl software spolehlivý, funkční a bez chyb.

V DevOps není testování pouze fází SDLC, která probíhá před dodáním software do provozu, jedná se o kontinuální proces, který je integrován do celého procesu vývoje a je do něj zapojen celý tým. Zapojením celého týmu do testování a jeho integrací do procesu

testování je možné budovat kvalitu software od samého počátku vývoje, což vede k vyšší kvalitě výsledného produktu. [37]

V následujících podkapitolách jsou popsány principy testování v určitých fázích DevOps pipeline (viz. kapitola 3.2.2). Východiskem je přitom „DevOps Testing Syllabus“ [36] vydaný mezinárodní asociací AT*SQA[6].

3.3.1 Testování během plánování

Testování ve fázi plánování zahrnuje:

- Shromáždění potřebných požadavků od uživatelů a dalších zainteresovaných stran
- Přesné zdokumentování získaných požadavků
- Zajištění testovatelnosti pomocí definování akceptačních kritérií na základě zdokumentovaných požadavků

V momentě, kdy máme shromážděné požadavky, následuje jejich revize. Jasně definované požadavky pomáhají zajistit, že nedojde k rozporu mezi tím, co vývojář vyvíjí, a tím, co uživatel očekává. Definovat tyto požadavky je užitečné zejména z toho důvodu, že se díky tomu předchází nedorozuměním mezi uživateli a vývojáři, a je tak zajištěno, že konečný produkt splňuje potřeby a požadavky uživatele. Díky jasnému vymezení požadavků vývojář ví, co přesně se od něj očekává, a může pracovat na tom, aby tato očekávání splnil, což pomáhá zlepšit celkovou kvalitu výsledného software a zvyšuje spokojenost zákazníka.

3.3.2 Testování během vývoje a sestavování

Tato fáze se zaměřuje na statickou analýzu kódu, unit testy a integrační testy. Je důležité provádět testování v každé fázi vývoje, aby bylo zajištěno, že do další fáze testování postoupí pouze kvalitní kód (např. přechod od unit k integračním testům).

[6] Association for Testing and Software Quality Assurance je mezinárodní asociace pro testování a zajišťování kvality software [38].

3.3.3 Testování během Staging fáze

Staging fáze je testování v simulovaném produkčním prostředí. Toto prostředí by mělo být věrnou napodobeninou produkčního prostředí, aby se daly odhalit možné problémy po nasazení do skutečného produkčního prostředí. Cílem této fáze je tedy zajistit, aby systém správně fungoval a splňoval všechny požadavky před jeho nasazením do produkčního prostředí. To obnáší funkční, nefunkční a systémové testy.

3.3.4 Testování během nasazování

Testování během nasazování je proces ověřování, zda je software připraven k použití v cílovém prostředí. To může zahrnovat opakované funkční testování z předchozí fáze a také kombinované testování, které má zajistit, aby různé podmínky v cílovém prostředí nezpůsobily chyby. Tyto podmínky se mohou týkat například cílového operačního systému nebo typu prohlížeče.

3.4 Agilní metodiky testování

Agilní testování software je koncept, který se věnuje testování aplikace v rámci agilních vývojových metodik. Tyto metodiky jsou popsané v podkapitole 3.1.

Agilní přístup k testování software musí odpovídat rychlému tempu a iterativnímu charakteru agilních projektů. Na testování v rámci agilních přístupů by se měli podílet všichni členové týmu včetně vývojářů a dalších zainteresovaných stran. Agilní tester v tomto případě přináší do testování specializované znalosti a zkušenosti zaměřené na agilní testování a funguje jako kouč pro zbytek týmu. Agilní metodiky testování popsané v této podkapitole vychází z knihy „Agile Testing Foundations – Agile Tester Guide“ [39] vydané mezinárodní organizací ISTQB[1]. Mezi nejznámější agilní metodiky testování podle této knihy patří:

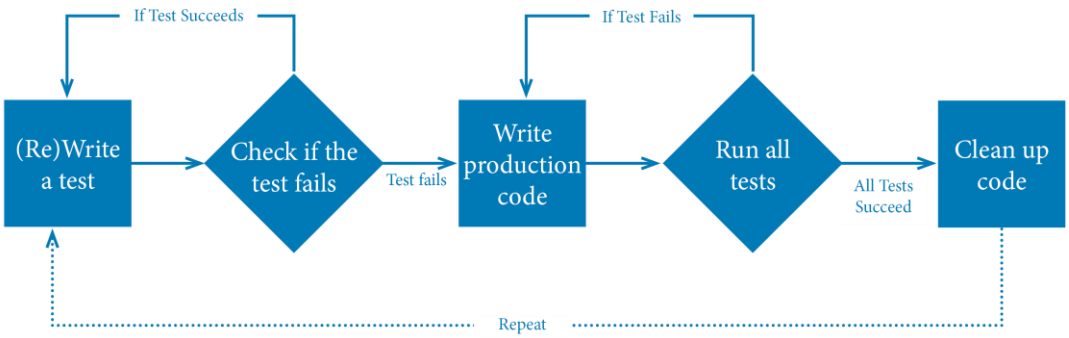
- Test-driven development
- Acceptance test-driven development
- Behaviour-driven development

3.4.1 Test-driven development

Test-driven development (TDD), neboli programování řízené testy, je metodika, při které se nejdříve napíšou unit testy a až poté se začne psát kód, který těmito testy projede. Smyslem TDD je vytvořit testy pro každou malou část funkce a následně psát kód, který testy projde. Tento postup pomáhá zajistit, aby se kód choval tak, jak má, a aby byl snadno udržovatelný a srozumitelný. S postupným růstem počtu testů a kódu je třeba kód refaktorovat. Refaktorování kódu je nezbytné k zachování kvality kódu. V opačném případě skončíme s tzv. „špagety“ kódem, který je nepřehledný a nesrozumitelný. Síla TDD spočívá v tom, že se píše pouze minimum nového kódu, které je nutné, aby prošly napsané testy. Díky tomu, že vývojář nejdříve napíše unit testy, ostatní vývojáři si mohou promyslet požadavky a návrh kódu ještě před tím, než jej začnou psát, a vyhnou se tím psaní často zbytečného kódu. Z těchto důvodů se TDD často využívá právě v agilním vývoji, jelikož podporuje krátké vývojové cykly, časté refaktorizace kódu a poskytuje vysoce kvalitní a funkční kód.

Postup při TDD programování je následující:

- Napíše se unit test pro určitou funkcionalitu a očekává se, že selže, jelikož zatím neexistuje žádný kód, proti kterému by daný test mohl být spuštěn.
- Napíše se minimum kódu k tomu, aby vytvořený unit test prošel. Refaktoruje se předchozí kód kvůli udržitelnosti a pustí se nový test spolu s ostatními dříve napsanými testy.
- Tato činnost se opakuje, dokud všechny testy neprojdou.



Obrázek 13: TDD flow
Zdroj:[40]

3.4.2 Acceptance test-driven development

Acceptance test-driven development (ATDD), neboli vývoj řízený akceptačními testy, je metodika ve vývoji software, kde se nejdříve napíšou akceptační testy na základě akceptačních kritérií, které určují, zda výstup projektu splňuje požadavky zákazníka. ATTD se využívá zejména v Agilním vývoji, protože v tomto přístupu se testy píšou iterativním způsobem. Začíná se definováním akceptačních kritérií na základě požadavku uživatele. Poté následuje samotné psaní testů a vývoj. Akceptační testy jsou vyšší úrovně než unit testy, ale stále se jedná o relativně malé testy, z nichž každý prověřuje jedno, nebo více akceptačních kritérií. Slouží k rychlému ověření toho, že výsledný software splňuje požadavky zákazníka. Využití ATDD umožňuje identifikovat chyby vyšší úrovně v rané fázi vývoje, díky čemuž ušetří finanční a časové zdroje.

3.4.3 Behaviour-driven development

Behaviour-driven development (BDD), neboli vývoj založený na chování, je metodika vývoje software, která se zaměřuje na definování chování software a využívá tyto definice k vytvoření testů, které ověřují, zda se sytém chová správně. Klíčovou myšlenkou BDD je, že zainteresované strany (zákazníci, obchodní zástupci, vývojáři…) snáze pochopí požadované chování než konkrétní testy.

Při tomto přístupu se na začátku vývoje pořádá schůzka, na které jsou účastníci požádáni, aby se zamysleli nad tím, jak by se měl systém chovat. Častým výstupem z takových schůzek je seznam testovacích případů ve formátu „Given-When-Then“ který dané testy popisuje jako posloupnost následujících kroků:

- Poskytnutí kontextu
- Uskutečnění nějaké události
- Reakce na danou událost

Na základě takovýchto případů, které jsou jednoduché a srozumitelné, jsou následně vytvořeny testy, které vývojáři průběžně spouštějí při vývoji. Tyto testy mají výborné pokrytí a zajišťují, že výsledný software bude poskytovat funkčnost, jakou zákazník očekává.

4 Automatické testování software

Automatizace testování software se stala oblíbeným přístupem v procesu jeho vývoje, protože je známo, že urychluje projekt, šetří zdroje a zvyšuje efektivitu testování. Fáze testování může alokovat značné množství zdrojů určených pro projekt, a proto je automatizace považována za nákladově efektivní řešení. Organizace se často potýkají s problémy jako jsou měnící se požadavky a napjaté termíny, což může vést ke zpoždění a dalším nákladům. Automatizované testování umožňuje organizacím důkladněji testovat své softwarové systémy a zároveň minimalizovat čas a zdroje, což z něj činí pro mnoho organizací žádanější variantu. [41]

Automatizace testů sice může zpočátku vyžadovat značné investice, ale v konečném důsledku může ušetřit signifikantní množství času a úsilí, které by jinak bylo vynaloženo na manuální testování. Vhodně provedené automatizované testování umožňuje provádět větší množství testů, což vede k rozsáhlejšímu pokrytí požadavků, rychlejšímu provedení testů a stabilnějším výsledkům. [42]

Automatizace testování oproti manuálnímu testování software zahrnuje využití skriptů a specializovaných nástrojů. Účelem automatizování testování je zvýšit účinnost a efektivitu testování a zároveň snížit možnost lidské chyby.

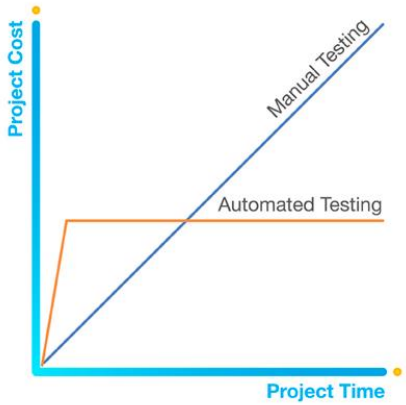
4.1 Charakteristiky automatizovaných projektů

Ne každý software je vhodný automaticky testovat. Vysoké počáteční náklady spojené s automatickým testováním mohou značně navýšit rozpočet na projekt, pro který by se automatizace testování nevyplatila. Automatizace testování se vyplatí pouze tehdy, pokud ušetřený čas při využití manuálního testování převýší náklady na vybudování a údržbu automatizace.

Následující podkapitoly prezentují charakteristiky projektů, na kterých se vyplatí automatizaci provádět. Vychází z „Test Automation Syllabus“ [42] od mezinárodní asociace AT*SQA.

4.1.1 Předpoklad dlouhodobého používání

Vzhledem k tomu, že vývoj automatizace testů je nákladný proces zejména kvůli úsilí na vytvoření testovacích skriptů a cen nástrojů určených k automatizaci testování, je třeba výsledné testy spustit vícekrát, aby se náklady vrátily. Odhaduje se, že testovaný software by měl zůstat v produkčním prostředí dva až pět let, aby se náklady na investici vrátily.



Obrázek 14: Náklady na manuální vs automatické testování
Zdroj: [43]

Na obrázku č. 14 je patrné, že s rostoucím časem a velikostí projektu rostou i náklady na manuální testování v porovnání s automatickým testováním. Ovšem automatické testování vyžaduje větší prvotní investici.

4.1.2 Stabilní funkčnost a rozhraní

Nákladově efektivnější je vytvářet automatizaci testů, jakmile cílový systém dosáhne stabilního stavu se základní sadou funkcí, které se nebudou výrazně měnit. Tím se sníží náklady na údržbu způsobené změnami ve skriptech automatizace testů.

Kromě výše uvedeného je také důležité, aby programová rozhraní byla pevně stanovena, protože jakékoliv změny rozhraní budou vyžadovat aktualizace skriptů automatizace testů, které prostřednictvím tohoto rozhraní komunikují s testovaným systémem.

4.1.3 Časté regresní testování

Nejpřínosnější automatické testy jsou ty, které je třeba spouštět často. Pokud testovaný software vyžaduje pravidelné regresní testování, lze využít automatické testování k efektivnímu vykonávání těchto testů. Čím častěji jsou testy prováděny, tím vyšší je finanční návratnost investice.

4.1.4 Adekvátní podpora nástrojů

Ne vše lze nebo je třeba automatizovat a mohou nastat situace, kdy není vhodný nástroj k dispozici. V některých případech dokonce není jiná cesta, než si takový nástroj vytvořit. V takovém případě je důležité zvážit, kolik úsilí bude potřeba vynaložit na jeho vývoj.

4.1.5 Dostatečné dovednosti v týmu

Vývoj automatizace testů je stále vývoj software a k jako takovému by se k němu mělo přistupovat. Takový vývoj představuje kroky jako návrh, plánování, volba architektury, vývoj, testování a dokumentaci. Přesto, že některé nástroje (většinou za poplatek) nabízí přívětivější uživatelské rozhraní, ve většině případů je však ke spojení skriptů pro automatizaci testů s dalšími funkcemi nutné mít programátorské znalosti. Je důležité mít tým, který disponuje odpovídajícími dovednostmi pro efektivní vývoj automatizace testů.

4.1.6 Podpora vedení

Automatizace testů je nákladná záležitost, která vyžaduje podporu a pochopení ze strany vedení a také odpovídající zdroje v rozpočtu na projekt. Automatizace testování vyžaduje specifické programátorské dovednosti, což může vést k nutnosti najmout specializované vývojáře. Pro zajištění úspěšné implementace automatizovaného testování musí mít vedení jasnou představu o potřebném úsilí a potencionálním dopadu na časový plán.

4.2 Výhody a nevýhody automatizace

V předchozí podkapitole jsou shrnuty faktory, které je třeba vzít v úvahu, pokud zvažujeme automatizaci testů. Tato kapitola popisuje výhody a nevýhody automatického testování, rozhodneme-li se pro automatizaci testování.

Automatické testování je zpočátku časově náročný a nákladný proces zejména při vytváření. Jakmile je však automatické testování úspěšně implementováno do SDLC, mohou být takové testy spouštěny autonomně, což snižuje potřebu neustálého dohledu ze strany testerů. Zároveň zkracuje doby dodávek nových verzí software k zákazníkovi. V případě dlouhodobého kontinuálního vývoje lze očekávat vysokou návratnost prvotní

investice do automatického testování. Automatické testování má pozitivní vliv na kvalitu software a zvyšuje efektivitu testování v rámci SDLC. [44]

Automatizace tetování sebou přináší několik výhod, z nichž některé jsou uvedeny níže: [45]

- Urychluje vydávání software díky zrychlení testování.
- Umožňuje častější testování.
- Po počáteční investici do napsání skriptů snižuje náklady na testování.
- Zvyšuje spolehlivost testování.
- Umožňuje testovat software méně kvalifikovanými pracovníky.

Níže v této kapitole jsou popsány některé nevýhody, které je třeba zvážit, rozhodneme-li se pro automatizaci testování. Popis těchto nevýhod vychází z [46].

- Pokud se automatizace testování neudržuje, investice do jejího vývoje nepřinese požadované výhody, protože její hodnota se bude s časem snižovat s tím, jak se bude měnit kód testovaného software. Aby tomu bylo možné předejít a plně využít potenciál automatizace testování, je třeba průběžně aktualizovat testovací skripty, aby pokrývaly potřeby testovaného software.
- Automatizované testování software obvykle neodhalí nové chyby testovaného software. To platí zejména proto, že k odhalení nových chyb je třeba kombinace zkušeností, dovedností a inteligence, což jsou vlastnosti, které se v automatickém testování těžko reprodukují.

4.3 Proces automatizace testování

Automatizace testování software není pouze jeden úkol, ale soubor procesů, činností a nástrojů, které vyžadují plánování, aby byly úspěšné. Podle [47] lze proces automatizace testování rozdělit na čtyři fáze:

- Selekce
- Modelování
- Exekuce
- Analýza

Každá z těchto fází je detailněji popsaná v podkapitolách níže a jejich popis vychází z [47].

4.3.1 Selekce

V této fázi je třeba učinit několik rozhodnutí ještě před tím, než se začne s automatizací testování. Patří sem rozhodnutí jako:

- Výběr testů vhodných pro automatizaci
- Výběr vhodného automatizačního nástroje
- Určení potřebných lidských zdrojů

Výsledkem této fáze je soubor testů určených k automatizaci a popsaných v přirozeném jazyce, který bude třeba převést do formátu srozumitelného počítači.

4.3.2 Modelování

Tato fáze se zaměřuje na převod testovacích artefaktů, jako jsou testovací případy a data, do formátu, který lze výpočetně zpracovat zvoleným automatizačním nástrojem. Tento proces je ovlivněn technickými parametry podporovanými daným nástrojem a může probíhat ručně, automaticky nebo kombinovaně. Hlavním výstupem z této fáze je obvykle testovací model.

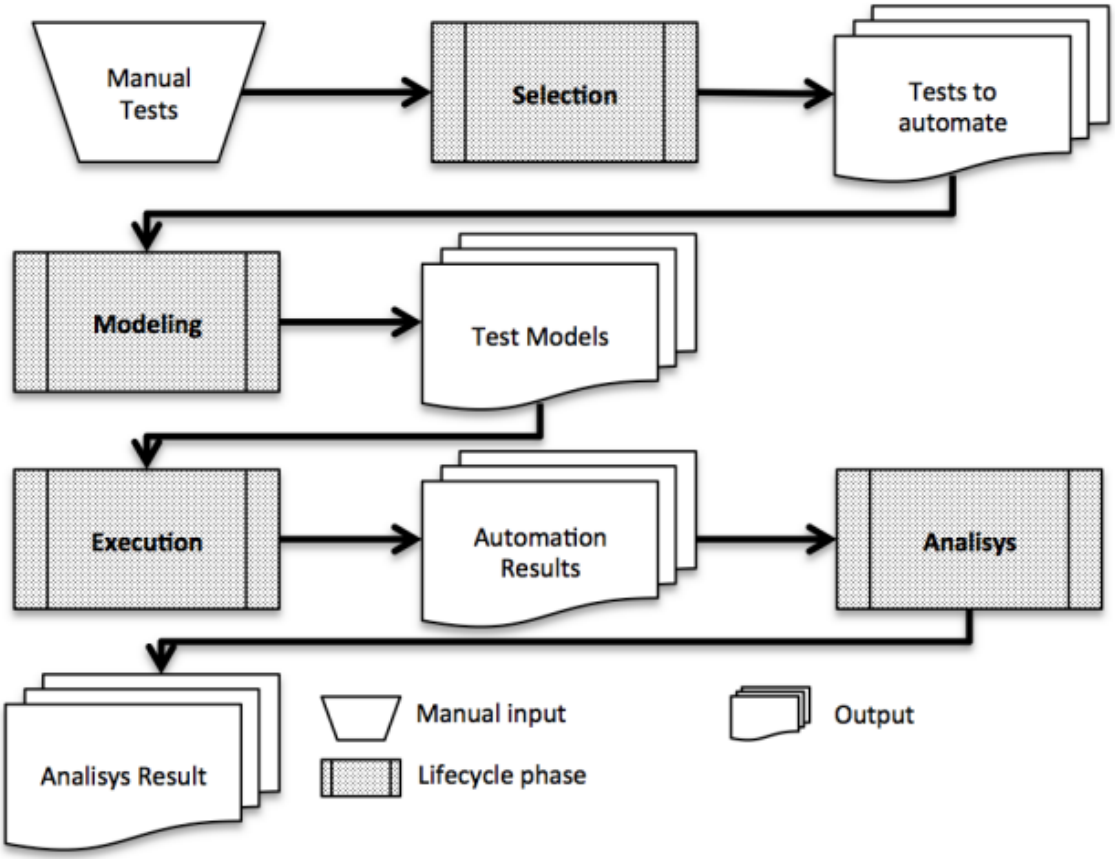
4.3.3 Exekuce

Tato fáze bývá považována za nejdůležitější. Během této fáze automatizační nástroj interpretuje instrukce z modelu (skriptů) a využívá testovací data k vyhodnocení testů. Hlavním výstupem z této fáze jsou výsledky testů, které ukazují, zda automatizace dosahuje svých cílů.

4.3.4 Analýza

V této části se vyhodnocuje účinnost automatizovaného testování s cílem prokázat hodnotu vynaložených investic. Výstupem této fáze je analýza, která poskytuje přehled o stavu procesu automatizace testování a identifikuje oblasti, které je třeba zlepšit.

Celý proces je znázorněn na obrázku níže.



Obrázek 15: Proces automatizace testování
Zdroj: [47]

4.4 Automatizace testovacích případů

Automatizace testovacích případů je proces používání softwarových nástrojů a skriptů k provádění automatizace testovaného systému. Testovací případy je třeba převést na sérií akcí, které se provádí na testovaném systému. Tyto akce mohou být zaznamenány v testovací proceduře, nebo naprogramovány v testovacím skriptu. Kromě akcí by automatizované testovací případy měly obsahovat potřebné údaje pro systém, na kterém se testuje, a měly by obsahovat ověřovací kroky, které potvrdí, že bylo dosaženo očekáváného výsledku. K vytvoření těchto posloupností akcí lze použít různé přístupy, které jsou blíže popsány níže.

Tato podkapitola vychází zejména ze sylabu „Advanced Level Syllabus – Test Automation Engineer“ vydaného organizací ISTQB dostupného na [48].

Mezi přístupy při vytváření posloupnosti akcí patří:

1) Tester implementuje testovací případy „napřímo“ do automatizačních testovacích skriptů. Tento přístup je nejméně doporučovaný, protože postrádá úroveň abstrakce a může vést k vyššímu objemu práce při údržbě.
2) Tester navrhne testovací postupy a poté je převede na automatizované testovací skripty. Tento přístup sice nabízí určitou formu abstrakce, ale nemusí zahrnovat plnou automatizaci pro generování testovacích skriptů.
3) Tester používá nástroj, který dokáže převést testovací postupy do automatizovaných testovacích skriptů. Tento přístup kombinuje jak abstrakci, tak automatizaci generování skriptů.
4) Tester používá nástroj, který dokáže automaticky generovat testovací postupy nebo přímo překládat testovací skripty z modelů. Tento přístup má nejvyšší stupeň automatizace.

Při volbě přístupu k automatizaci testovacích případů je nutné zvážit konkrétní kontext projektu. Jednou ze strategií, která může být účinná, je začít s méně pokročilou možností, která je v krátkodobém horizontu jednodušší na implementaci, z dlouhodobého hlediska však může být hůře udržitelná.

4.4.1 Přístup zachycení/přehrávání

Tato metoda využívá nástroje pro záznam interakcí s testovaným systémem při provádění definované sekvence akcí. Tyto interakce zahrnují vstupy a výstupy, které mohou být zaznamenány pro pozdější kontrolu. Během přehrávání lze výstupy systému ručně zkontrolovat, zreprodukovat na úrovni detailů, či kontrolovat určité hodnoty v jednotlivých krocích. Hlavní výhodou tohoto přístupu je, že jej lze snadno nastavit a použít pro systém jak na úrovni grafického rozhraní, tak na úrovni API. Mezi nevýhody patří obtížná údržba, vývoj a silná závislost na verzi systému, na kterém je tato metoda uplatňována.

4.4.2 Lineární skriptování

Lineární skriptování zahrnuje zpočátku ruční spouštění testovacích postupů. To může zahrnovat zachycení posloupnosti akcí prováděných testerem a viditelný výstup z testovaného systému, který má být zaznamenán testovacím nástrojem v podobě skriptu.

Skripty mohou být nástrojem automatizovány, což umožňuje snadné opakování testů. Mezi výhody lineárního skriptování patří minimální příprava před zahájením automatizace a jednoduchost pro testera po zvládnutí této metody. V podstatě jde o nahrání manuálního testu, jeho opakování a porovnání výsledků. Nevýhodou je však nižší efektivita při automatizaci mnoha testů nebo různých verzí software, protože aktualizace skriptů v reakci na změny systému vyžaduje vysoké náklady na údržbu.

4.4.3 Strukturované skriptování

Strukturované skriptování oproti lineárnímu je využití knihovny skriptů, která obsahuje skripty, které lze použít vícekrát k provedení řady instrukcí, které jsou často potřebné v dalších testech. Výhodou tohoto přístupu je snadná opětovná použitelnost. Již napsané skripty lze znovu použít pro nové testy místo psaní nových skriptů. Výsledkem je možnost automatizace více testů s menším úsilím, což má za následek nižší náklady na sestavení a údržbu těchto skriptů. Hlavní výhoda toho přístupu spočívá v úspoře nákladů, protože nevyžaduje velké investice do údržby díky opětovné použitelnosti jednotlivých skriptů. Další výhodou je snadnější psaní nových automatizovaných testů s využitím předešlých skriptů. Nevýhodou může být počáteční vytvoření skriptů. Tato investice však může z dlouhodobého hlediska vést k významným úsporám dalších nákladů na testování.

4.4.4 Testování založené na datech

Testování založené na datech neboli data-driven testing je metoda, která navazuje na strukturované skriptování. Hlavní rozdíl mezi oběma metodami spočívá ve způsobu zpracování vstupů. Při využití testování založeného na datech jsou vstupy testů extrahovány ze skriptů a umístěny do samostatných souborů, což umožňuje opakované použití hlavního testovacího skriptu pro více testů. Tento hlavní skript bývá označován jako řídící skript a obsahuje nezbytné kroky k provedení testů. Ačkoliv řídící skript lze použít pro více testů, nemusí stačit pro automatizaci širokého spektra testů, takže může být zapotřebí více řídících skriptů. Výhodou tohoto přístupu je výrazné ušetření nákladů při vytváření nových automatizovaných testů. Dále umožňuje automatizovat mnoho variant jednoho testu, což vede k prohloubení testování ve specifických oblastech. Nevýhodou je potřeba správného přístupu k managementu datových souborů, aby bylo zajištěno, že je automatizovaný systém dokáže přečíst.

4.4.5 Testování založené na klíčových slovech

Technika založená na klíčových slovech neboli keyword-driven testing je metoda, která navazuje na testování na datech. Dvěma hlavními rozdíly jsou:

- Datové soubory se zde nazývají „test definition“ (nebo podobná spojení)
- Existuje pouze jeden řídící skript

Datové soubory „test definition“ obsahují podrobný popis testů způsobem, který je pro testovací analytiky snadno srozumitelný. Tento formát je jednodušší než datové soubory, které nahrazuje. Tyto soubory obsahují data podobně jako datové soubory, ale také instrukce na vysoké úrovni (vyšší forma abstrakce), které se nazývají klíčová slova (keyword). Tato slova jsou vybírána tak, aby byla smysluplná pro testovací analytiky, testovanou aplikaci a popisované testy. Mezi benefity této metody patří, že umožňuje redukci výdajů na psaní nových automatizovaných testů v momentě, kdy je vytvořen řídící skript a asistenční skripty pro klíčová slova. To platí především proto, že testy jsou popsány soubory klíčových slov, což znamená, že testovací analytici mohou zadávat automatizované testy jednoduše tak, že je popíšou pomocí klíčových slov a souvisejících údajů. Může se jednat o spojení jako „vytvořit účet“ či „zadat objednávku“. Tyto testovací případy se snadněji udržují, čtou a píšou, protože složitost může být skryta v klíčových slovech. Díky klíčovým slovům tedy dochází k vysoké formě abstrakce. Nevýhodou tohoto přístupu je složitost napsání řídícího a podpůrných skriptů. U malých systémů mohou úsilí a náklady převážit nad výhodami. Také je třeba vybrat dobrá klíčová slova, která půjdou použít v mnoha testech. V opačném případě výběr špatných klíčových slov bude moci být použit pravděpodobně jen v malém počtu testů.

4.4.6 Testování na základě modelu

Testování založené na modelu se týká automatizovaného generování testovacích případů na rozdíl od automatizovaného provádění testovacích případů s využitím některé z dříve uvedených technik (kapitoly 4.4.1 až 4.4.5). Testování na základě modelu využívá modely, které představují chování systému a jsou odstíněny od skriptovacích technologií daného systému. Tuto metodu lze využít k vytvoření testů pro dříve uvedené techniky (kapitoly 4.4.1 až 4.4.5). Výhodou je vysoká míra abstrakce a lze se tak soustředit na podstatu testování (business logika, scénáře, konfigurace atd.). Další výhodou je

generování testů pro různé cílové systémy, díky čemuž se lze přizpůsobit změnám ve smyslu používaných technologiích.

4.5 Nástroje pro automatizaci

Automatizované testovací nástroje jsou softwarové programy, které automatizují proces testování a pomáhají vývojářům rychle identifikovat a opravovat chyby. Tato podkapitola takové nástroje popisuje. Rozdělení těchto nástrojů vychází z [49]. Nástroje jako Selenium, JMeter, JIRA a JUnit, které se využívají i ve společnosti Unicorn, jsou detailněji popsány v samostatné podkapitole.

4.5.1 Nástroje pro unit testy

Nástroje pro unit testování jsou nejpoužívanějšími nástroji pro automatizaci testů a jsou snadno integrovatelné jako framework přímo ve vývojovém prostředí. Mezi takovéto nástroje patří například JUnit, NUnit, TestNG a další.

4.5.2 Nástroje pro automatizaci funkčních testů

Nástroje pro funkční testování testují funkce tak, že jim zadávají vstupní údaje a zkoumají získaný výstup v porovnání se zadaným očekávaným výstupem v daném testovacím případu. Do této kategorie spadají i nástroje, které lze použít pro testování uživatelského rozhraní. Mezi nástroje pro funkční testování patří Selenium, TestComplete, Appiumm, JMeter a další.

4.5.3 Nástroje pro pokrytí kódu

Nástroje pro pokrytí kódu jsou nástroje, které slouží k měření množství kódu, který byl otestován automatizovanými sadami testů. Pomáhají identifikovat části kódu, které možná nebyly dostatečně otestovány a vyžadují dodatečné otestování. Tyto informace jsou důležité pro identifikaci oblastí, kde se mohou vyskytovat chyby. Pokrytí kódu se obvykle měří v procentuálních hodnotách, přičemž vyšší hodnoty znamenají menší pravděpodobnost výskytu neodhalených chyb. Mezi nástroje pro měření pokrytí kódu patří Coberturta, CodeCover a další.

4.5.4 Nástroje pro správu testů

Nástroje pro správu testování pomáhají zefektivnit různé testovací činnosti, jako je vytváření testovacích případů, tvorba testovacích plánů a strategií či vykazování výsledků

testů. Pomáhají týmům efektivně řídit projekty tím, že poskytují centralizované a snadno udržovatelné místo pro všechny testovací činnosti. Různé testovací nástroje mají různé funkce, ale všechny mají za cíl zefektivnit proces testování a poskytnout rychlý přístup k analýze dat a snadnou komunikaci mezi projektovými týmy. Mezi takovéto nástroje patří JIRA, Test Manager, TETware a další.

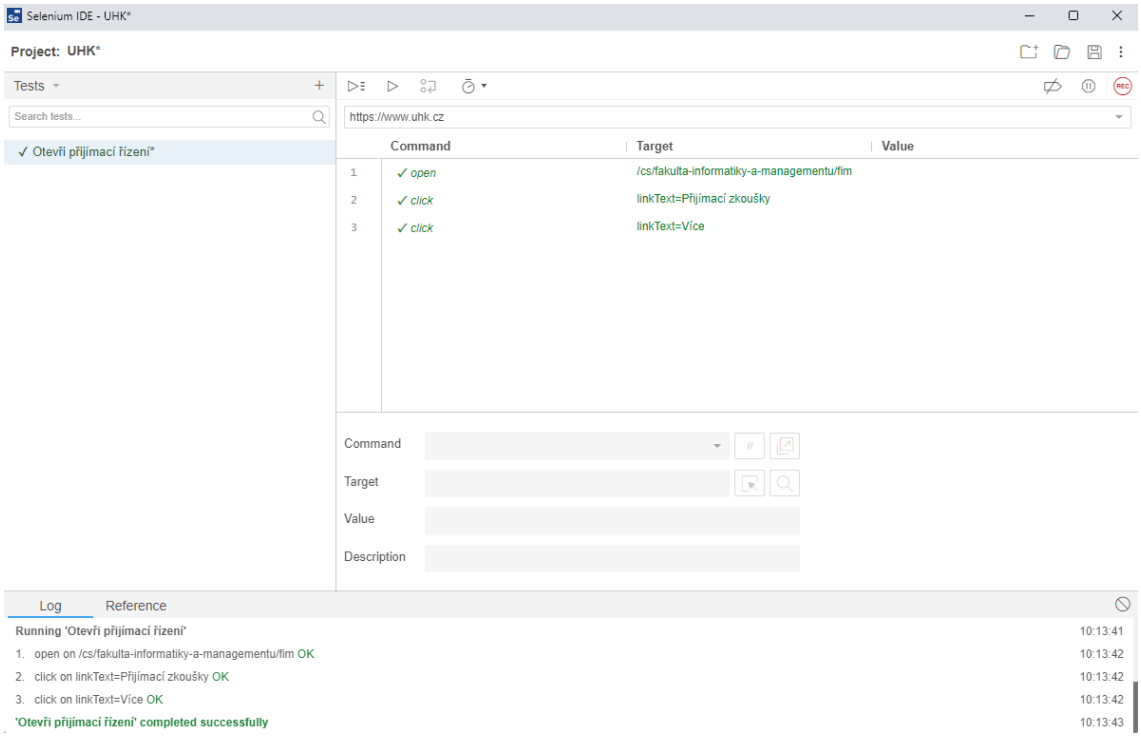
4.5.5 Selenium

Selenium je široce používaný open-source nástroj pro automatizaci webových prohlížečů. Umožňuje vývojářům skriptovat interakce s webovými stránkami a lze jej snadno použít k testování webových aplikací. Selenium se však dá využít například i při web-scrapingu, nemusí být využíváno pouze jako testovací nástroj. Tato podkapitola vychází z oficiální dokumentace Selenia, která je dostupná na [50].

Selenium se skládá z různých komponent, přičemž stěžejní komponentou je Selenium WebDriver, která automatizuje webové prohlížeče. Další komponentou je Selenium Grid, která umožňuje paralelní provádění testů na více počítačích v různých prohlížečích. Selenium obsahuje také své integrované vývojové prostředí (IDE), které slouží mimo jiné jako jednoduchý nástroj pro generování testovacích případů. Selenium IDE funguje jako jednoduché rozšíření pro prohlížeče Firefox a Chrome a umožňuje uživatelům zaznamenávat akce prohlížeče, které vytvářejí příkazy pro Selenium IDE.

API rozhraní Selenium WebDriver podporuje širokou škálu programovacích jazyků včetně Java, C# a mnoho dalších, takže je snadno přístupné vývojářům bez ohledu na preferenci programovacího jazyka.

Jak už bylo naznačeno v úvodu této podkapitoly, tak jednou z klíčových vlastností Selenia je jeho schopnost automatizovat testy webových aplikací. Selenium lze integrovat s řadou testovacích frameworků, jako jsou JUnit a TestNG a lze jej použít k provádění funkčních i regresních testů.



Obrázek 16: Ukázka Selenium IDE
Zdroj: Vlastní zpracování

Na obrázku č.16 je ukázka Selenium IDE v prohlížeči Chrome. Tento „test“ otevře web Univerzity Hradec Králové, Fakulty informatiky a managementu a přejde na informace o přijímacím řízení.

4.5.6 JUnit

JUnit je populární open-source framework pro psaní a spouštění unit testů v jazyce Java. Nejnovější verze JUnit 5 se skládá (podobně jako předchozí verze) ze tří modulů: [51]

- **JUnit Platform** je základ pro spouštění testovacích frameworků v JVM. Definuje API rozhraní nazvané TestEngine pro vývoj testovacího frameworku, který běží na platformě JUnit. Tato platforma nabízí Console Launcher pro spouštění z příkazového řádku a JUnit Platform Suite Engine pro spuštění vlastní testovací sady pomocí jednoho nebo více „test enginů“[7].
- **JUnit Jupiter** je modul, který usnadňuje vytváření testů tím, že poskytuje různé metody a anotace pro porovnávání. Tento modul také umožňuje vývojářům psát rozšíření za pomocí extension modulu.

[7] Komponenta, která je zodpovědná za vyhledání a exekuci testů.

- **JUnit Vintage** je modul pro spouštění testů JUnit 3 a JUnit 4 a umožňuje vývojářům postupně přejít na testy využívající JUnit 5, aniž by museli přepisovat všechny testy najednou.

4.5.7 JMeter

Informace obsažené v této podkapitole vychází z oficiálních návodů a dokumentace.
JMeter, které jsou dostupné na [52]. JMeter je výkonný open-source nástroj, který se používá k testování výkonu a zátěže webových aplikací. JMeter lze použít k simulaci velkého zatížení serveru, sítě nebo objektu a otestovat tak jeho sílu nebo analyzovat celkový výkon při různých typech zátěže. Lze ho použít k testování funkčnosti a zátěže webových aplikací, služeb nebo databází. JMeter podporuje širokou škálu protokolů včetně HTTP, HTTPS, FTP, JDBC, JMS a LDAP. JMeter také poskytuje řadu vestavěných posluchačů (listeners), které lze použít k zobrazení výsledků testů. Tito posluchači mohou zobrazovat výsledky testů v různých formátech včetně grafů a stromů. Kromě toho JMeter umožňuje exportovat výsledky testů do různých formátů včetně CSV, XML a JSON. Tato široká podpora protokolů, exportovaných testů do různých formátů a analýza výsledků testování z něj činí ideální nástroj pro automatické testování a integraci s CI/CD. JMeter také podporuje distribuované testování, které umožňuje testerům spouštět testy na více zařízeních současně. Testeři tak mohou simulovat mnohem větší zatížení webové aplikace, než by bylo možné na jednom stroji.

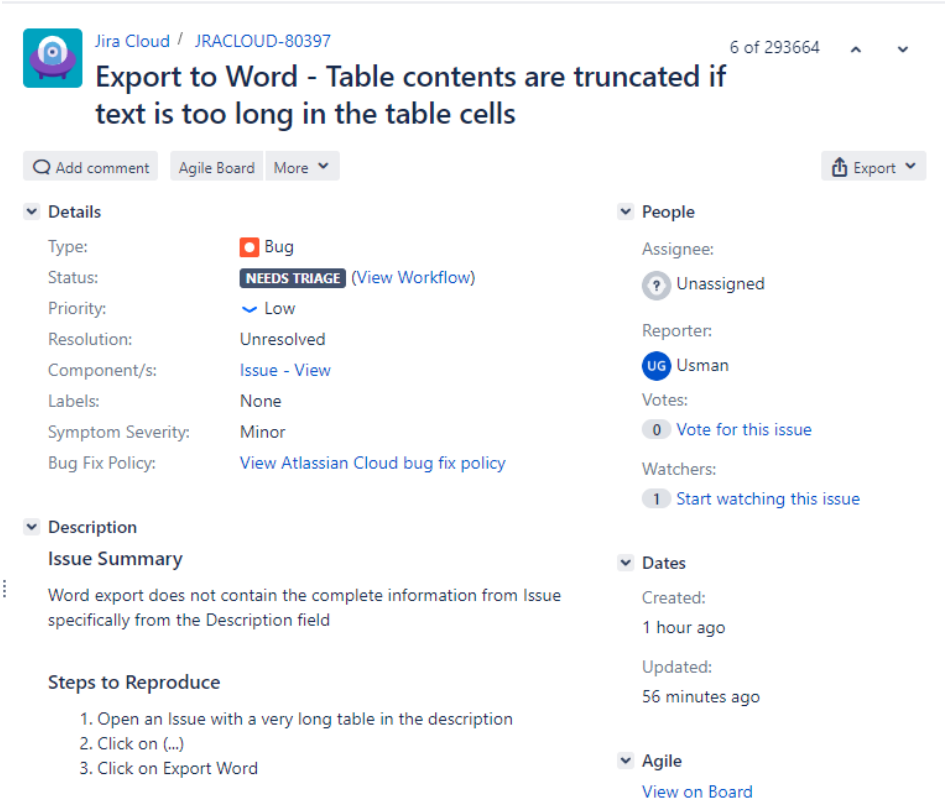
4.5.8 JIRA

JIRA je populární nástroj používaný pro řízení projektů a sledování problémů při vývoji software. JIRA umožňuje sledovat a spravovat proces testování software, včetně vytváření a provádění testovacích případů, sledování chyb a koordinaci jednotlivých testů v rámci týmu a jednotlivců. [53]
JIRA může být využita k:

- Sledování bugů
- Řízení projektů
- Řízení procesů
- Správě úkolů

JIRA umožňuje vytvářet testovací případy a propojit je s konkrétními případy či uživatelskými příběhy. Tyto testovací případy lze následně přiřadit konkrétním osobám a sledovat jejich provádění. Případy mohou být označeny různými statusy jako to-do, done, in-progress a další.

Základní komponentou při práci s JIRA software v kontextu vývoje software je issue. Issue představuje činnost, kterou je třeba provést. Může se jednat o úkol na projektu, bug případně uživatel si může vytvořit vlastní typ issue. Podle těchto typů může například analytik filtrovat kritické úkoly, které je třeba prioritizovat. Ukázka issue je na obrázku níže.



Obrázek 17: Ukázka JIRA issue
Zdroj: [54]

4.6 Údržba automatizovaných testů

Pokud není sada automatických testů průběžně udržovaná, může to pro všechny, kteří na projektu pracují, představovat nemalé riziko. Neaktualizované testy mohou vést k nepřesným výsledkům a nekonzistentnímu chování. Jestliže tým nevěnuje údržbě testů dostatek času, je obtížné identifikovat a následně řešit vzniklé problémy, které mohou přetrvávat a časem jich může přibývat.

Na druhou stranu dobře udržované automatické testy pomáhají řešit nevyhnutelné problémy s automatizací testů, které se mohou objevit. Testy se tak snáze sledují, aktualizují a rozšiřují. Pokud některý test začne selhávat, zabere méně času problém izolovat a opravit díky aktualizovaným testům. Další výhodou v případě dobře udržovaných automatických testů je, že přinášejí okamžitou hodnotu s tím, jak aplikace roste a rozšiřuje se. [55]

Automatické testy a automatizaci testování je třeba dobře navrhnout už od začátku automatizace, jinak se může stát, že tým místo vyvíjení a psaní nových testů bude dokola udržovat ty staré. Následující tipy pro návrh automatizace testování jsou převzaty z [56].

- **Rozhodnutí o tom, jaké případy automatizovat.** Automatizace všech testů není praktická, je důležité rozhodnout, které testy automatizovat jako první. Výhodou automatizovaného testování je, že lze testy mnohokrát opakovat. Proto by měly být automatizovány případy, které se provádějí často a vyžadují velké množství dat. Dále je vhodné automatizovat testy, ve kterých tester snadno udělá chybu, či testy které vyžadují mnoho času a úsilí, jsou-li prováděny manuálně.
- **Volba vhodného nástroje.** Výběr nástroje pro automatizaci testování je zásadní. Na trhu existuje velké množství nástrojů pro automatizované testování a je důležité vybrat takový nástroj, který nejlépe vyhovuje požadavkům. Při výběru nástroje pro automatizaci je mimo jiné třeba vzít v potaz následující:
    - Podpora pro platformy a technologie, které tým využívá.
    - Flexibilita pro testery s různou úrovní dovedností.
    - Integrace se stávajícím ekosystémem (Jenkins Gitlab Azure…).
- **Kvalitní data**. Pro automatizaci testování je důležité mít k dispozici kvalitní testovací data. Tato data zajišťují, že testy simulují reálné scénáře, což vede k spolehlivějším výsledkům testů.

5 Integrace testování s CI / CD nástroji

Integrace testování s pomocí CI/CD nástrojů do procesu vývoje je klíčovým krokem k zajištění kvality a spolehlivosti softwarových aplikací. Díky automatizaci procesu testování a jeho integrací s CI/CD nástroji mohou týmy rychle identifikovat a opravit problémy dříve, než se stanou závažnými, a snížit tak riziko selhání software.

5.1 GitLab

GitLab CI/CD je nástroj pro zefektivnění vývoje software zejména v kontextu agilního vývoje software. Tento nástroj umožňuje automatizovat fáze DevOps pipeline, konkrétně se jedná o fáze sestavení, testování a nasazení pomocí tzv. GitLab pipeline, která je klíčovým konceptem GitLab CI/CD.

GitLab pipeline je soubor činností (job), které se provedou v reakci na nějakou událost v GitLab projektu. Běžně se jedná o nový kód v projektu či spojování jednotlivých větví projektu. Jednotlivé činnosti jsou uloženy v souboru *gitlab-ci.yml*. Tento soubor je rozdělen na konkrétní činnosti, které obsahují informaci o tom, k jaké etapě (stage) náleží skript, který se má v rámci činnosti vykonat, a další konfigurace. Název činnosti musí být unikátní, názvy jednotlivých etap jsou GitLabem předdefinovány jako *pre, build, test, deploy* a *post*. Jednotlivé činnosti v rámci etapy běží paralelně. Selhání jediné činnosti má za následek selhání celé pipeliny. Jakmile dojde k dokončení všech činností v rámci jedné etapy, pipeline se přesune do etapy následující.

Exekuci samotných etap má na starosti tzv. Gitlab runner. Jedná se o separátní aplikaci, která může být nainstalovaná lokálně či v cloudu. Gitlab umožňuje využít buď Specifický nebo Sdílený runner. Sdílený runner (shared-runner) je runner připravený GitLabem a po udělení povolení v projektu využívat sdílený runner není nutná žádná další konfigurace. Speciální runner vyžaduje vlastní zařízení, na kterém může běžet v podobě binárního souboru nebo docker kontejneru. [57]

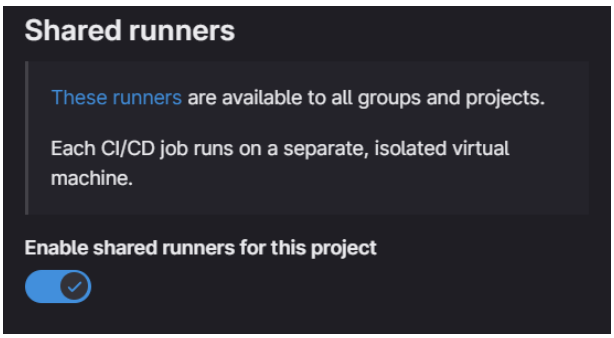
5.1.1 Ukázka unit testů

V této podkapitole je demonstrován nástroj GitLab CI/CD pro sestavení projektu a spouštění automatických unit testů. Jedná se o jednoduchou generickou aplikaci, která je

napsaná v Javě s použitím frameworku SpringBoot. Aplikace je sestavena pomocí nástroje Gradle a pro unit testy využívá knihovnu JUnit 5.

5.1.1.1 Nastavení runnerů

Prvním krokem, chceme-li využít nástroj Gitlab CI/CD, je udělit povolení využívat *shared runner*, jelikož nemáme vlastní infrastrukturu pro *specifický runner*. Toto povolení se uděluje v nastavení projektu v sekci CI/CD a podsekci Runners.



Obrázek 18: Povolení runnerů pro GitLab projekt
Zdroj: vlastní zpracování

5.1.1.2 Nastavení gitlab-ci.yml

Dalším krokem je vytvoření souboru *gitlab-ci.yml* v kořenovém adresáři projektu, do kterého bude zapsána konfigurace GitLab pipeline. Jako první musíme definovat docker image, ve kterém poběží naše etapy. Jelikož aplikace využívá Gradle jako sestavovací nástroj, chceme, aby byl součástí daného image. Z těchto důvodů volíme následující docker image.

```
1  image: gradle:alpine
```

Ukázka kódu 1: Nastavení docker image
Zdroj: vlastní zpracování

Tento image obsahuje Alpine Linux společně spolu s Gradle nástrojem pro sestavení našeho projektu. Dále je třeba nadefinovat jednotlivé etapy (stages). Cílem této pipeline je sestavit a otestovat projekt, tudíž jsou použity etapy build a test.

```
1  stages:
2    - build
3    - test
```

Ukázka kódu 2: Definice etap
Zdroj: vlastní zpracování

Jak je zmiňováno v úvodu této kapitoly, tyto etapy jsou předdefinovány a není nutné je explicitně definovat, pro větší přehlednost však zmíněny jsou. Následuje definice jednotlivých činností v rámci jednotlivých etap, tedy určení toho, co se má v jaké etapě vykonat. Tyto činnosti jsou popsány v následujících dvou podkapitolách 5.1.1.3 a 5.1.1.4.

5.1.1.3 Nastavení build etapy

První etapou je build etapa, která zajišťuje sestavení projektu.

```
1  build:
2      stage: build
3      script: gradle assemble
4      artifacts:
5          when: on_failure
6          paths:
7          - build/libs/*.jar
```

Ukázka kódu 3: build etapa
Zdroj: vlastní zpracování

Význam jednotlivých řádků je následující:

1) Označuje název činnosti, cílem této činnosti je sestavení projektu, tudíž je použít název „build“.
2) Stage určuje, k jaké etapě tato činnost náleží. V tuto chvíli se snažíme projekt prvotně sestavit, tudíž je použita etapa build.
3) Představuje skript, který se má vykonat. V našem případě chceme sestavit aplikaci, proto volíme „gradle assemble“, který nám projekt sestaví. Záměrně zde není zvolen skript „gradle build“, který oproti “assemble“ provádí dodatečné kontroly jako je například spouštění testů. Ty se budou spouštět v následující etapě.
4) Artifacts jsou adresáře a soubory, které mohou být v rámci dané etapy uloženy.
5) Podmínka, kdy se mají soubory uložit. Hodnota *on_failure* znamená, že soubory se uloží pouze v případě, že daná činnost selže. Dalšími možnostmi jsou *always* a *on_succes*, to je výchozí hodnota, k uložení dojde v případě, že daná činnost skončí úspěšně.
6) Řádky 6 a 7 specifikují cestu, ze které se mají soubory uložit. Gradle při sestavování projektu ukládá výsledné .jar soubory do adresáře na cestě build/libs.

Hvězdička před koncovkou souboru nám říká, že chceme uložit všechny soubory s příponou .jar.

5.1.1.4 Nastavení test etapy

Následuje test etapa, která spouští unit testy.

```
1  unit_tests:
2      stage: test
3      script: gradle test
4      artifacts:
5          when: always
6          repotrts:
7              junit: build/test-results/test/**/TEST-*.xml
8          paths:
9          - build/test-results/test/**/TEST-*.xml
```

Ukázka kódu 4: Test etapa
Zdroj: vlastní zpracování

Význam jednotlivých řádků je následující:

1) Název činnosti, v této činnosti chceme spustit unit testy, proto název „unit_tests“.
2) Nyní se nacházíme v etapě testování, proto hodnota test.
3) Tímto skriptem spustíme testy v rámci našeho projektu.
4) Artefakty testovací fáze se liší od sestavovací fáze a jsou popsány v následujících řádcích.
5) V této fázi chceme mít vždy výsledky testů v podobě .xml souboru, se kterým se pracuje dále, proto hodnota *always*.
6) Tento řádek představuje konfiguraci cesty, kde má pipeline hledat výsledky testů. Tyto výsledky pak gitlab přehledně zobrazuje v rámci každé pipeline.
7) Tento řádek představuje samotnou cestu pro vygenerované soubory. JUnit generuje soubory ve formátu TEST + relativní cesta k testovací třídě + .xml. Chceme-li mít k dispozici výstup ze všech souborů, je nutné použít symbol hvězdičky coby regex symbolu.
8) Řádky 8 a 9 stejně jako v případě build etapy představují adresáře a složky, které budou po dokončení dané činnosti k dispozici ke stažení. V tomto případě chceme mít k dispozici všechny .xml reporty obsahující výsledky testů.

5.1.1.5 Nastavení exportu JUnit reportů

Generování samotných reportů do .xml souborů je třeba explicitně povolit v konfiguračním Gradle souboru *build.gradle*.

```
tasks.named('test') {
	useJUnitPlatform()
	reports {
		junitXml.enabled = true
	}
}
```

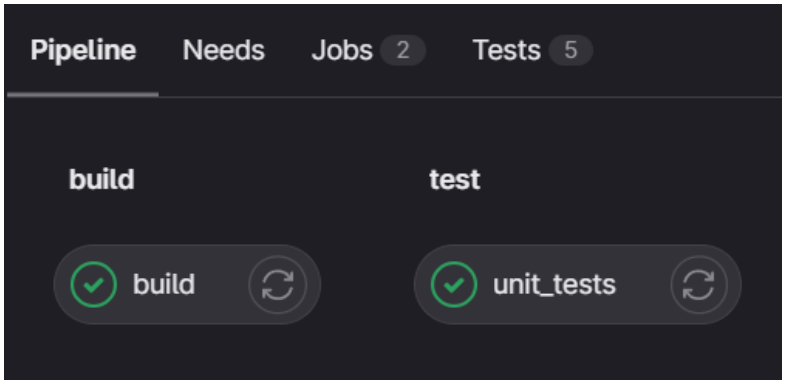
Ukázka kódu 5: Konfigurace testů
Zdroj: vlastní zpracování

Tento kód je napsán v jazyce DSL nástroje Gradle a konfiguruje úlohu s názvem „test“ následovně:

1) Začátek bloku.
2) Nastavení JUnit platformy pro spouštění testů.
3) Inicializace bloku reports, který nastavuje typy reportů.
4) Samotné nastavení typu reportu. V tomto případě se jedná o povolení generování JUnit XML.

5.1.1.6 Výsledky

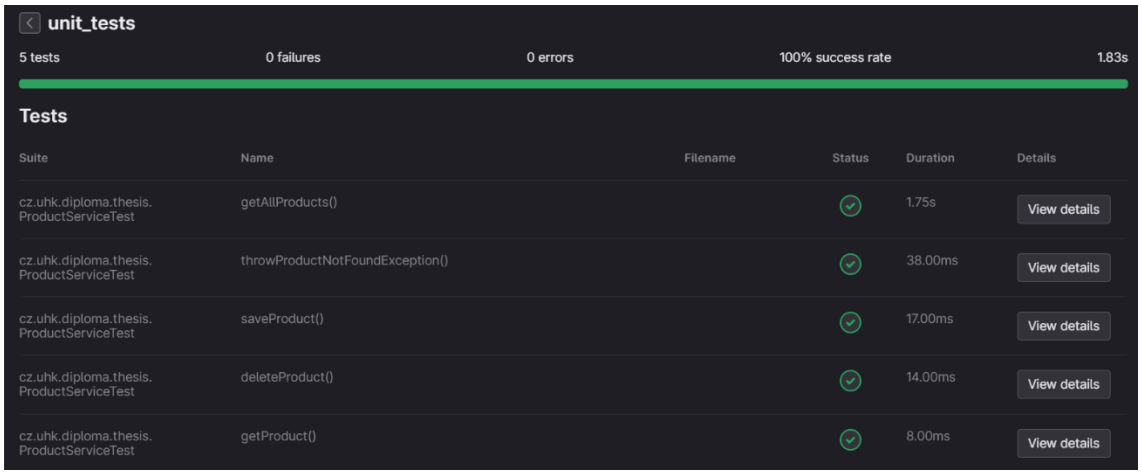
GitLab pipeline, která je popsaná v předchozích podkapitolách, je znázorněna na následujícím obrázku.



Obrázek 19: Ukázka GitLab pipeline
Zdroj: vlastní zpracování

Na obrázku je vidět rozdělení do jednotlivých etap a činnosti, které k dané etapě náleží. Pokud bychom přidali například činnost pro spouštění integračních testů, zobrazovala by se pod činnosti „unit_test“.

Následující obrázek zobrazuje kartu „tests“ z předchozího obrázku. Jedná se o detail výsledků jednotlivých testů. Tyto výsledky pochází z vygenerovaných .xml souborů, jejichž export je popsán v podkapitole 5.1.1.5.



Obrázek 20: Seznam provedených testů
Zdroj: vlastní zpracování

Pokud by nějaký test selhal, je možné pomocí tlačítka „View details“ zobrazit dodatečné informace o konkrétním testu.

5.2 Jenkins

Jenkins je populární open-source automatizační server, který umožňuje automatizovat procesy sestavení, testování a nasazování. Díky podpoře více než sta pluginů umožňuje jednoduchou integraci s dalšími nástroji. Jenkins umožňuje integraci s populárními testovacími nástroji jako je například Selenium, JUnit, TestNG a Cucumber. Následující podkapitoly popisují, jak lze Jenkins využít při automatizaci v procesu testování.

5.2.1 Ukázka

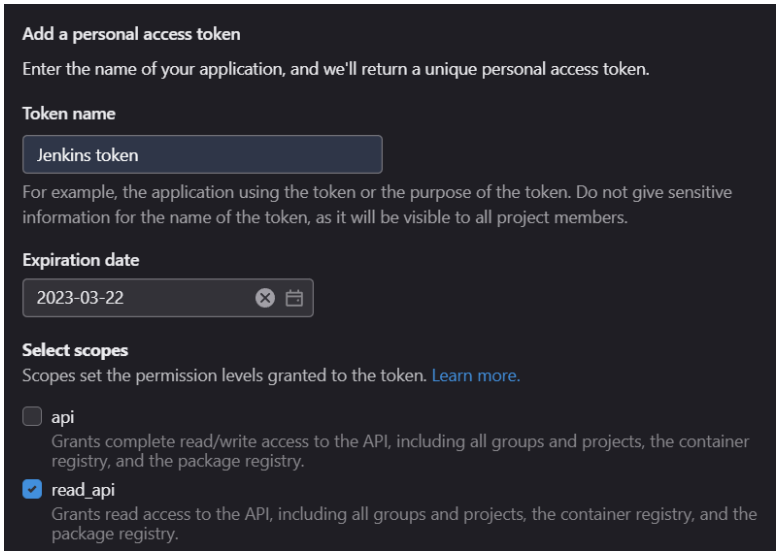
V této podkapitole je popsaná ukázka automatizace JUnit testů za pomocí nástroje Jenkins. Pro ukázku byla použita již napsaná aplikace, která je popsaná v kapitole 5.1.

5.2.1.1 Nastavení Jenkins

Prvním krokem je nainstalovat Jenkins. Jenkins byl nainstalován na lokálním zařízení s výchozím nastavením. Lokální stroj používá Windows 11, disponuje 32 GB operační pamětí a 6 jádrovým procesorem AMD Ryzen 5 5600 procesorem. Po instalaci je Jenkins vystaven na portu 8080. Po instalaci je třeba nainstalovat několik dodatečných pluginů. Prvním je GitLab plugin, který nám umožní sledovat změny v repositáři a stahovat nový kód. Druhým pluginem je JUnit, který umožňuje zobrazovat výsledky testů z .xml souborů. Pluginy se přidávají v sekci *Administration => Manage plugins => Available plugins => název pluginu*. Po přidání pluginů je vhodné Jenkins restartovat.

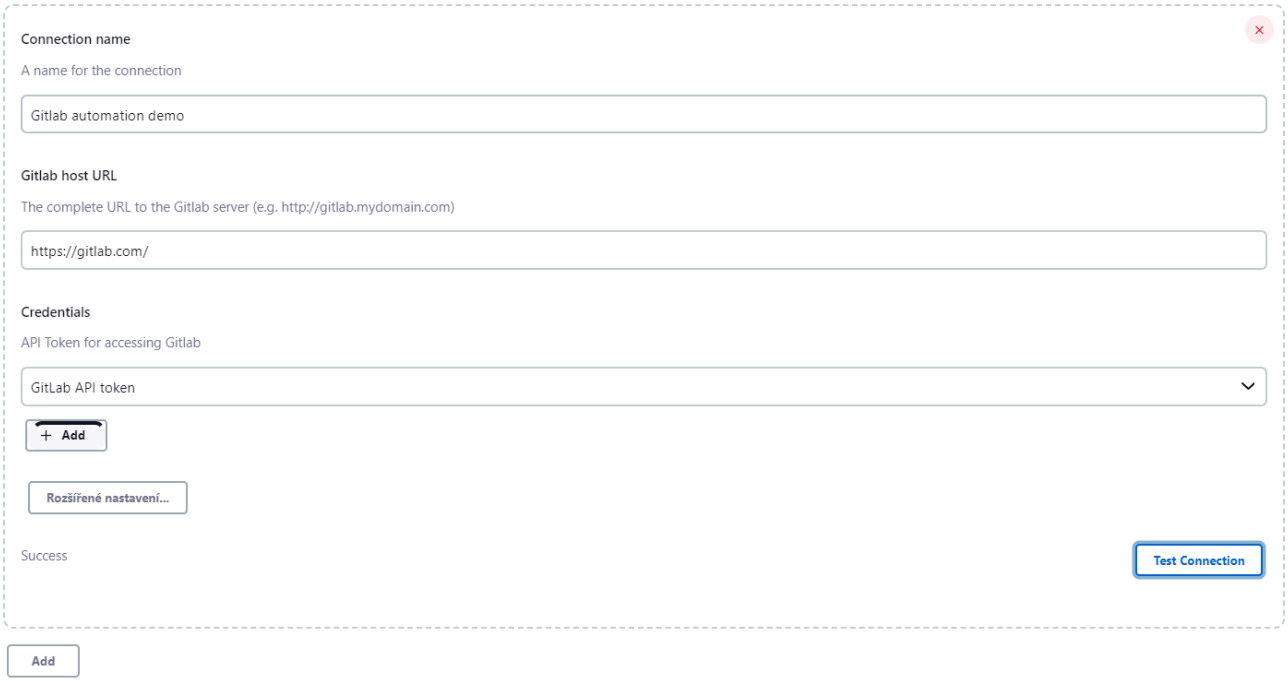
5.2.1.2 Nastavení GitLab

Další konfigurací je konfigurace spojení Jenkins a GitLab. Toto spojení se nachází v *Administration => Configure System => GitLab*. Sekce GitLab je zde díky GitLab pluginu, který jsme přidali v předchozím kroku. Zde musíme nakonfigurovat: název připojení, GitLab adresu a přidat Access token. Access token se generuje v GitLab Preferences => Access Tokens. Zde se zadá název tokenu, datum expirace tokenu a zvolí se oprávnění pro daný token.



Obrázek 21: GitLab vytvoření tokenu
Zdroj: vlastní zpracování

V tomto případě potřebujeme pouze sledovat změny v repositáři, tudíž možnost *read_api* je v tomto případě vyhovující. Po kliknutí na tlačítko „Create personal access token“ je vygenerován Access token, který můžeme následně použít.

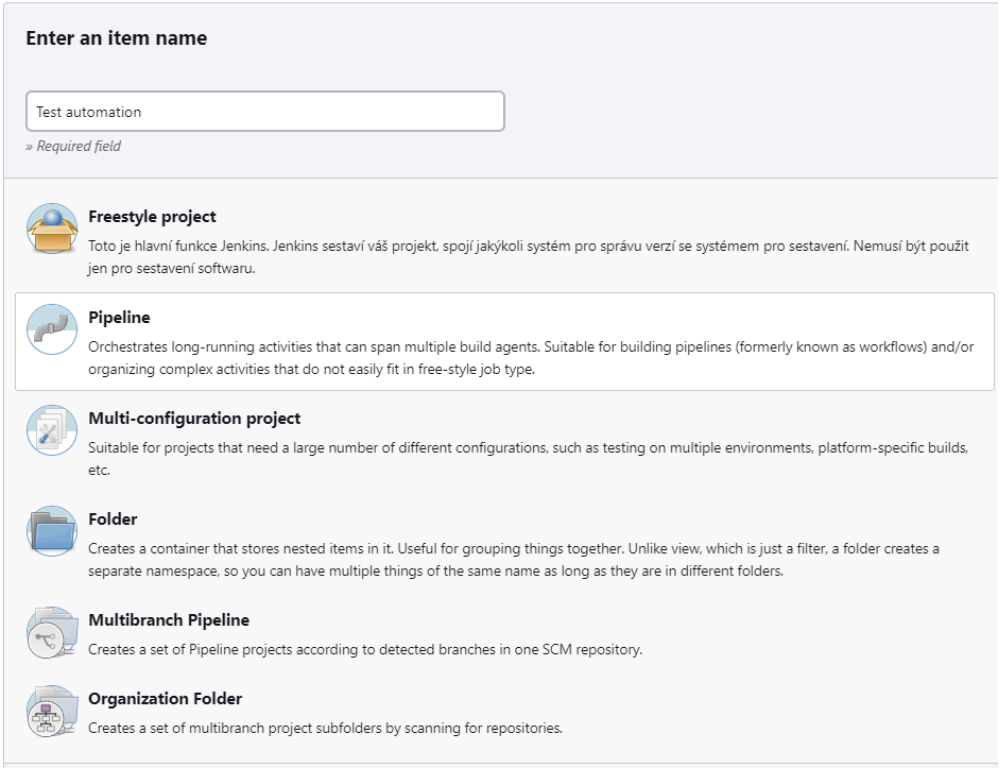


Obrázek 22: Ukázka GitLab konfigurace
Zdroj: vlastní zpracování

Vygenerovaný token uložíme pomocí tlačítka „+Add“ na obrázku výše. Na závěr komunikaci mezi Jenkins a GitLab ověříme pomocí tlačítka „Test Connection“.

5.2.1.3 Založení Jenkins pipeline

Prvním krokem je založení pipeline. Jenkins podporuje hned několik typů produktů viz. Obrázek níže. Byla zvolena varianta Pipeline, která umožňuje jednoduchou konfiguraci pro automatické sestavení a spouštění testů.



Obrázek 23: Vytvoření Jenkins pipeline
Zdroj: vlastní zpracování

Po pojmenování a výběru typu projektu Jenkins projekt vytvoří a odkáže na konfigurační stránku daného projektu.

5.2.1.4 Konfigurace Jenkins pipeline

Konfigurace Jenkins pipeline mimo jiné závisí na dodatečně nainstalovaných pluginech. Pro účely tohoto projektu byl dodatečně nainstalován plugin GitLab. V konfiguraci pipeline se volí GitLab připojení, které bylo nadefinováno v podkapitole 5.2.1.2.

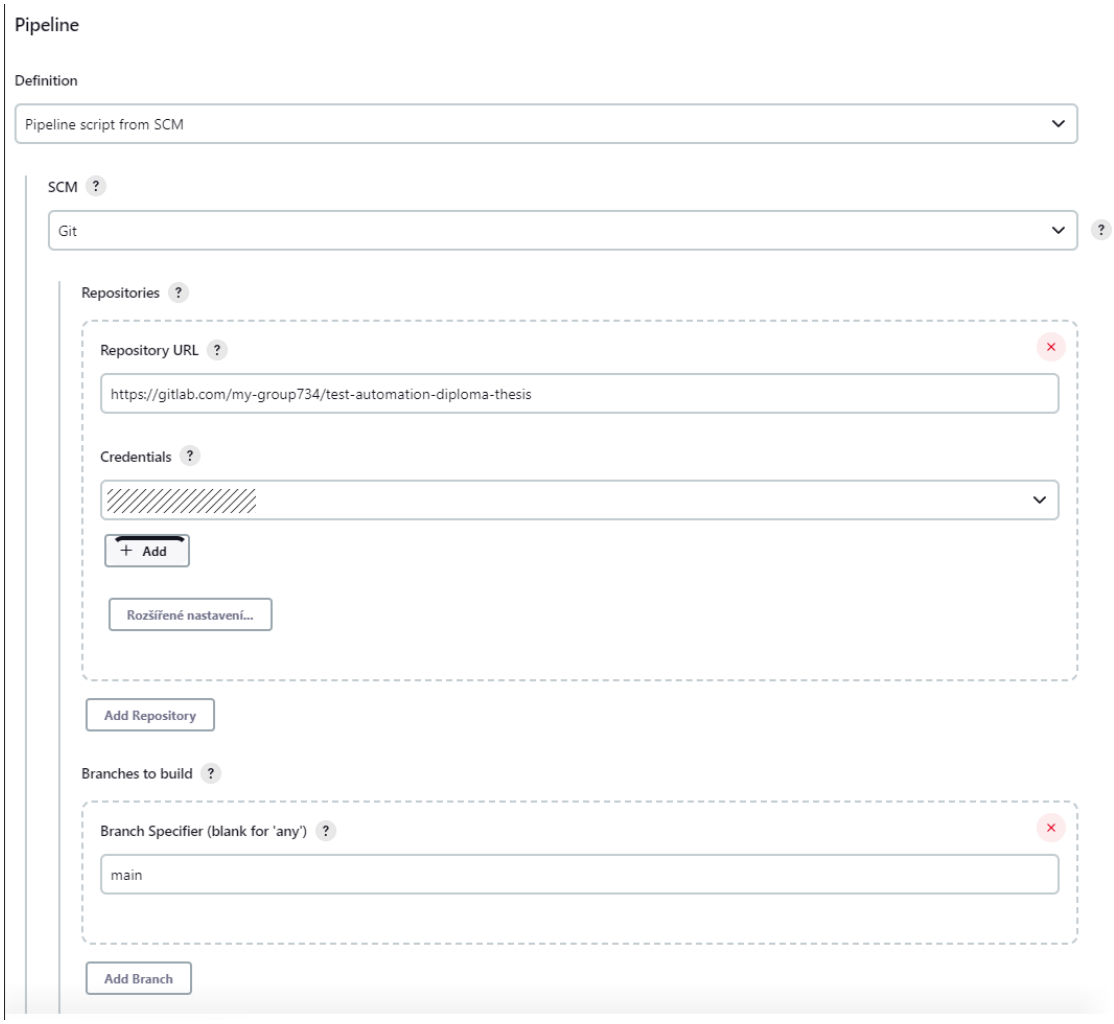
Obrázek 24: Volba GitLab připojení
Zdroj: vlastní zpracování

Další důležitou konfigurací Jenkins pipeline je záložka „Build Triggers“, ve které se nastavují kritéria, která určují, kdy se celá Jenkins pipeline automaticky spustí. Možnosti pro nastavení automatického spouštění jsou zobrazeny na obrázku níže.

Obrázek 25: Nastavení automatického spouštění Jenkins pipeline
Zdroj: vlastní zpracování

Vybrána byla varianta Poll SCM, která kontroluje nové změny v GitLab repositáři v pravidelných intervalech. V tomto případě se jedná o kontrolu jednou za 24 hodin. Pokud jsou zjištěny nové změny, pipeline je automaticky spuštěna.

Následuje definice samotné Jenkins pipeline. Definice Jenkins pipeline může být uložena přímo v Jenkins nebo v GitLab repositáři, kterého se daná pipeline týká. Ukázková konfigurace Jenkins pipeline je znázorněna na obrázku níže.



Obrázek 26: Jenkins konfigurace pipeline
Zdroj: vlastní zpracování

Jednotlivé bloky jsou vysvětleny v následujícím seznamu:

- **Definition** je místo, ve kterém jsou uloženy samotné kroky Jenkins pipeline. Může být uložena přímo v Jenkins, nebo v GitLab repositáři. V tomto případě je vybrána možnost *Pipeline script from SCM* (source control management), která znamená, že konfigurace je uložena v GitLab repositáři.
- **SMC** je nastavení zdrojového repositáře. Demo aplikace je uložena pomocí nástroje Git.
- **Repository URL** je adresa repositáře, ve kterém je uložena definice Jenkins pipeline v podobě textového souboru, který se nazývá Jenkinsfile. Tento soubor bývá uložen v kořenovém adresáři projektu, případně může být uložen pomocí dodatečné konfigurace i jinde.

- **Credentials** představují přihlašovací údaje ke GitLab účtu, který má přístup k danému projektu.
- **Branches to build** představuje seznam větví, které mají být sestaveny, v tomto případě je specifikována hlavní větev *main*.

5.2.1.5 Konfigurace Jenkinsfile

Samotná konfigurace Jenkinsfile, která je uložena v GitLab repositáři, je následující.

```
pipeline {
    agent any
    tools {
        jdk 'jdk11'
    }
    stages {
        stage('Build') {
            steps {
                bat 'gradle assemble'
            }
        }
        stage('Test') {
            steps {
                bat 'gradle test'
            }
            post {
                always {
                    junit 'build/test-results/test/*.xml'
                }
            }
        }
    }
}
```

Ukázka kódu 6: Konfigurace Jenkinsfile
Zdroj: vlastní zpracování

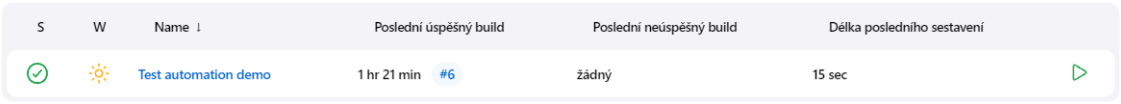
Význam jednotlivých řádků je následující:

1) Definice pipeline bloku.
2) Volba agenta, na kterém pipeline poběží. Any znamená, že pipeline poběží na jakémkoliv dostupném agentovi. V případě, že Jenkins běží lokálně jako v tomto případě, znamená to, že pipeline poběží na našem vlastním zařízení. To je třeba brát v úvahu, používáme-li příkazy, které nejsou v běžném systému nainstalovány. Může se jednat například o Gradle.

3) Blok *tools* definuje, jaké nástroje budou během exekuce pipeline k dispozici. Lokální stroj disponuje více verzemi nainstalovaného JDK a demo aplikace využívá JDK 11. Z tohoto důvodu je zde explicitně definována verze JDK 11.
6) Blok *stages* definuje jednotlivé fáze Jenkins pipeline. Každá fáze představuje krok v Jenkins pipeline. V této ukázce jsou nadefinovány dvě fáze *Build* a *Test*.
7) Blok definice fáze *Build*.
8) Blok *steps* představuje jednotlivé kroky, které se v rámci dané fáze mají vykonat.
9) Definice skriptu, který se má vykonat. Pipeline běží na operačním systému Windows, který má zabudovaný příkaz *bat* pro exekuci Shell příkazů. V tomto případě se jedná o příkaz *gradle assemble* který sestaví daný projekt. Gradle byl na lokální stroj dodatečně doinstalován.
14) Definice skriptu, který spustí testy.
16) Post blok definuje akce, které se mají vykonat po dokončení fáze, ve které je tento blok definován. V tomto případě se jedná o fázi *Test*.
17) Podmínka, kdy se má daný blok vykonat. Hodnota *always* představuje exekuci po každém dokončení jednotlivých kroků definovaných na předchozích řádcích.
18) Jednotlivé kroky, které se v rámci *post* mají vykonat. Hodnota junit pochází z dodatečně nainstalovaného pluginu JUnit. Tento krok slouží k reportování výsledků JUnit do Jenkins. Jako parametr je uvedena cesta k .xml souborům, které JUnit vygeneroval.

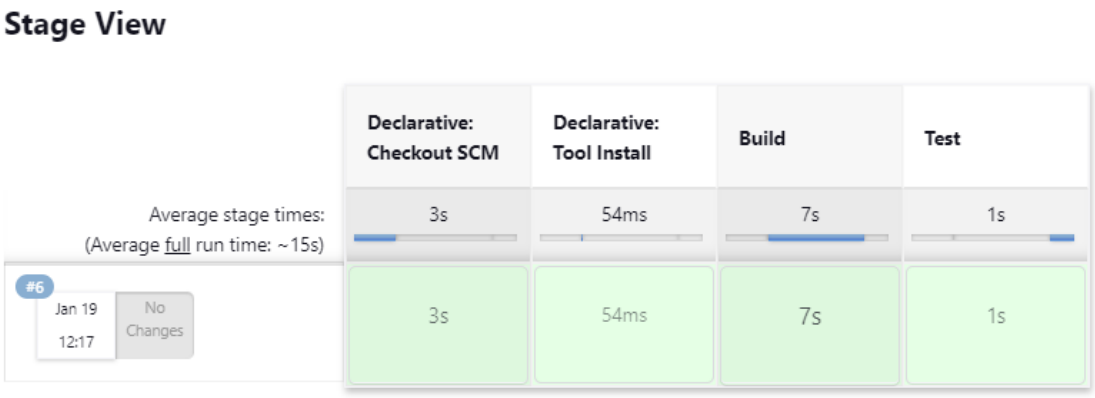
5.2.1.6 Výsledky

Pipeline, která byla popsána v předchozích podkapitolách se na hlavní stránce Jenkins Dashboard zobrazuje následovně.



Obrázek 27: Jenkins pipeline
Zdroj: vlastní zpracování

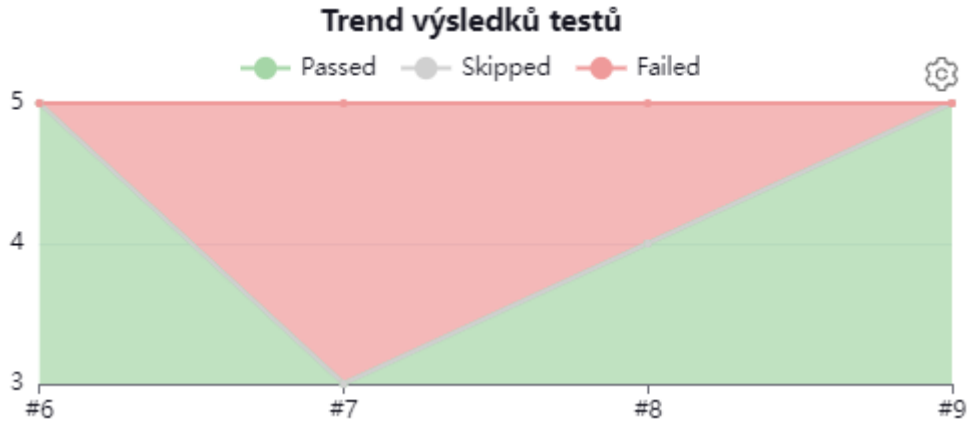
Proklikem na název Pipeline se dostaneme na detail dané Pipeline. Následující obrázek zobrazuje jednotlivé etapy námi nadefinované Jenkins Pipeline.



Obrázek 28: Stage view Jenkins pipeline
Zdroj: vlastní zpracování

Z obrázku je patrné, že jednotlivých etap je více, než bylo definováno v souboru Jenkinsfile. Etapa „Devlarative: Checkout SCM“ označuje proces stažení zdrojového kódu. Vyprší-li například Access token, tak tato etapa selže. Další etapou „navíc“ je „Declarative: Tool Install“. Jedná se o fázi, ve které se instalují potřebné nástroje pro běh pipeline. Jedná se o *tools* blok, který je nastaven v Jenkinsfile (viz. předchozí podkapitola). V tomto případě je jdk11 lokálně nainstalována, takže Jenkins nemusí stahovat a instalovat tuto verzi. Ostatní etapy představují etapy, které jsou nadefinovány v Jenkinsfile.

Pro lepší ilustraci grafu výsledků testů byly testy několikrát upraveny, aby jich určité množství v jednotlivých sestaveních selhalo.



Obrázek 29: Výsledky testů za využití JUnit pluginu.
Zdroj: vlastní zpracování

Obrázek výše ukazuje jednoduchý test report výsledků JUnit pluginu v rámci jednotlivých sestavení. Osa X představuje číslo sestavení, zatímco osa Y představuje počet testů. Na samotný graf lze kliknout a zobrazit si dodatečné detaily. Na obrázku níže je zobrazen detail sestavení #7.

Obrázek 30: Detail sestavení
Zdroj: vlastní zpracování

Detail sestavení ukazuje výsledek sestavení, datum a čas sestavení, odkaz do GitLab repositáře na commit, který s daným sestavením souvisí. Dále ukazuje výsledky jednotlivých testů, na které se lze prokliknout a zobrazit další podrobnosti.

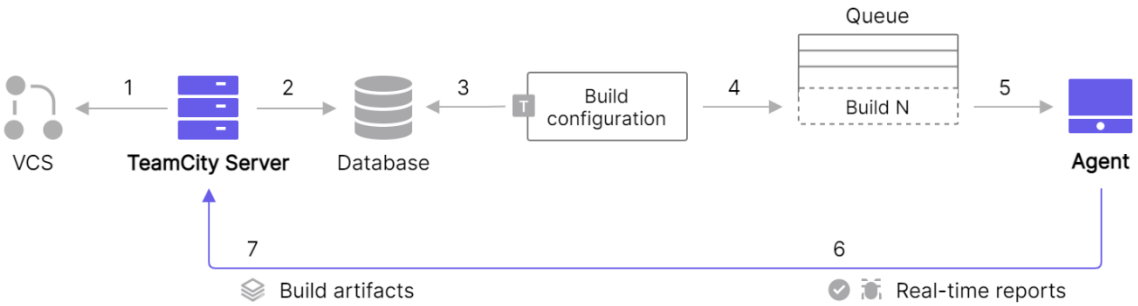
5.3 TeamCity

Popis TeamCity vychází z oficiální dokumentace TeamCity dostupné na [58]. TeamCity od společnosti JetBrains je výkonný CI/CD server, který umožňuje automatizovat procesy sestavení, testování a nasazení. TeamCity se skládá ze dvou hlavních částí:

- **TeamCity server** sám neprovádí sestavení ani testy, ale spravuje připojené agenty a přiřazuje jim jednotlivá sestavení na základě kompatibility. Server je zodpovědný za sledování průběhu sestavení a hlášení výsledků. Všechna data, která souvisí se sestavením jako je historie sestavení, změny ve správě verzí, agenti, uživatelské účty a oprávnění jsou uloženy v databázi. Server sám o sobě neuchovává protokoly sestavení a artefakty.

- **Build agent** je program, který je zodpovědný za provádění úkolů v rámci procesu sestavení. Tento agent bývá obvykle nainstalován na samostatném zařízení odděleně od TeamCity serveru. Použití agentů umožňuje testovat software na různých platformách a v různých prostředích současně, což vede k rychlejší zpětné vazbě pro vývojáře a přesnějším výsledkům testování.

Základní workflow TeamCity je zobrazena na obrázku níže.



Obrázek 31: TeamCity workflow
Zdroj: [58]

1) Server detekuje změnu v kořenovém úložišti VCS[8].
2) Server tuto změnu uloží do databáze.
3) Trigger připojený ke konkrétní konfiguraci sestavení zjistí příslušnou změnu a spustí sestavení.
4) Spuštěné sestavení se dostane do fronty sestavení.
5) Sestavení je přiřazeno agentovi, který je dostupný a kompatibilní.
6) Agent provede jednotlivé kroky sestavení. Během provádění kroků agent hlásí průběh sestavování na TeamCity server. Průběžně odesílá všechny zprávy protokolu, zprávy o testech a výsledky pokrytí kódu, takže může sledovat proces sestavování v reálném čase.
7) Po dokončení sestavení odešle agent artefakty sestavení na server.

TeamCity nabízí tři varianty tohoto nástroje. Jedná se o TeamCity Cloud, TeamCity Enterprise a TeamCity Proffesional. První dvě varianty jsou placené a v této práci

[8] Version control system. Jedná se o systém používaný ke správě a sledování změn v kódu. Typickým představitelem takového systémů je Git.

nebudou dále uvažovány. Oproti tomu TeamCity Professional je zdarma a praktická ukázka tohoto nástroje je popsána v následující podkapitole.

5.3.1 Ukázka

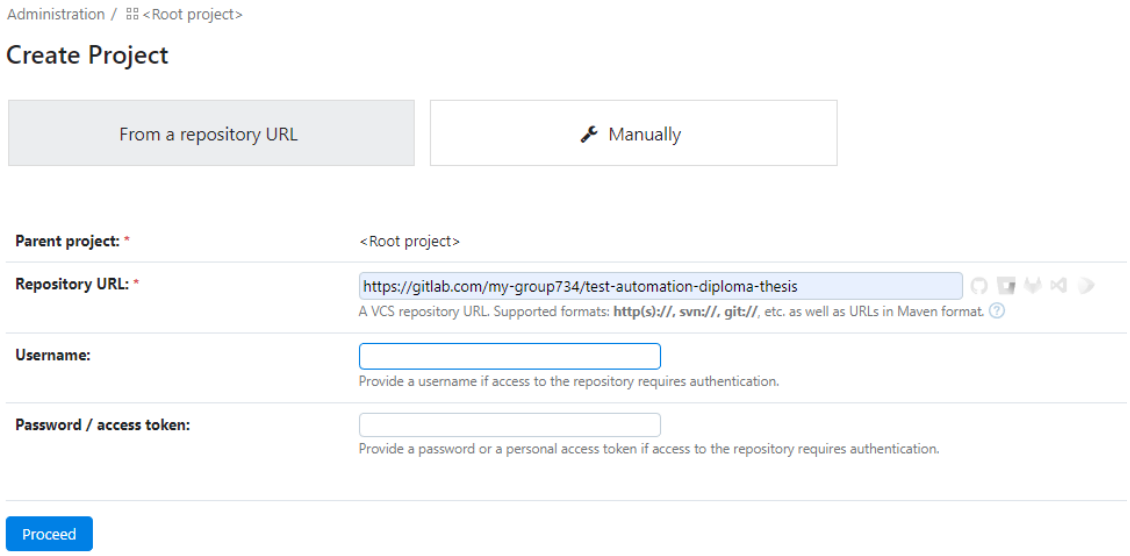
V této podkapitole je demonstrace nástroje TeamCity ve verzi Professional pro automatické spouštění unit testů. Stejně jako v předešlých kapitolách je využita napsaná aplikace, která je popsaná v podkapitole 5.1.1.

5.3.1.1 Spuštění TeamCity

Po instalaci na lokální stroj je třeba TeamCity spustit pomocí příkazové řádky. TeamCity instalační adresář obsahuje adresář s názvem „bin“, ve kterém jsou skripty pro manipulaci s TeamCity serverem. Pro spuštění serveru je zde umístěn skript s názvem *startup*. Skripty jsou dostupné pro Windows v podobě *.bat* souborů a pro systémy založené na UNIX v podobě *.sh* souborů. Po spuštění je TeamCity dostupné na portu 8111. Dále je třeba vytvořit účet pro lokální instanci nainstalovaného TeamCity. Toho lze dosáhnout pomocí webového rozhraní, které TeamCity vystavuje na portu 8111. Následně už lze přistoupit k vytvoření projektu.

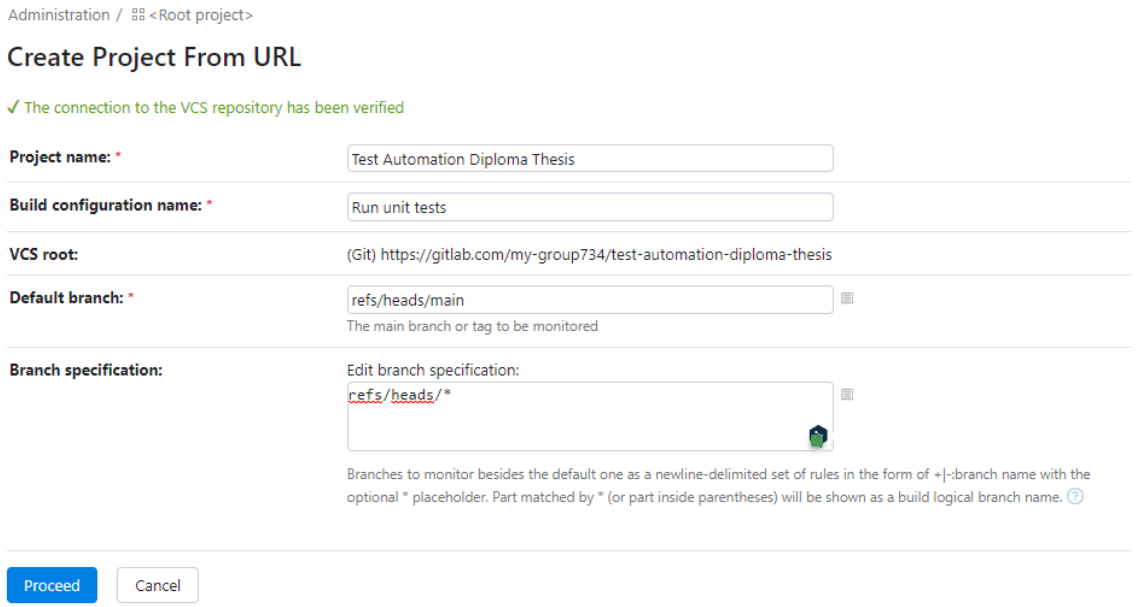
5.3.1.2 Vytvoření projektu

Po přihlášení je k dispozici uvítací obrazovka, která nám umožňuje rovnou projekt vytvořit. Případně pro vytvoření projektu je po levé straně tlačítko „Create project“.



Obrázek 32: Vytvoření projektu v TeamCity
Zdroj: vlastní zpracování

Pro vytvoření projektu je potřeba poskytnout údaje o repositáři, který chceme v daném projektu využívat. Musíme poskytnout URL pro daný repositář a údaje o účtu který má k danému projektu přístup. Kromě přihlašovacího jména můžeme poskytnout buď přístupový token, nebo heslo. Po kliknutí na tlačítko „Proceed“ TeamCity ověří údaje a v případě, že jsou údaje platné, přesměruje na další konfiguraci, kde lze nastavit název projektu, název konfigurace a specifikovat větev, která se má monitorovat.

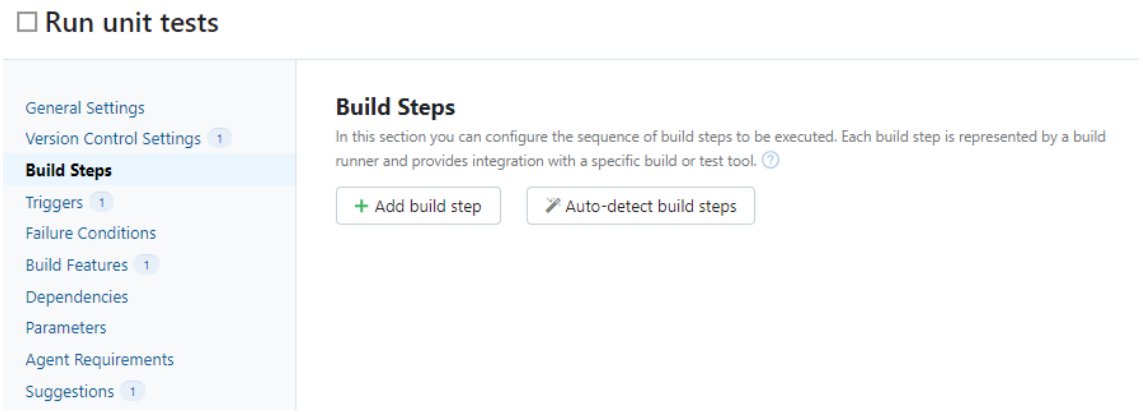


Obrázek 33: Dodatečná konfigurace vytvoření TeamCity projektu
Zdroj: vlastní zpracování

Kliknutím na tlačítko „Proceed“ se dokončí založení projektu.

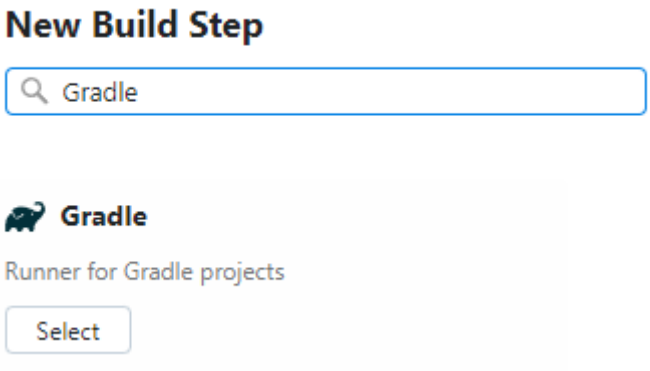
5.3.1.3 Konfigurace projektu

Dalším krokem je konfigurace samotných kroků, které se provedou, bude-li tato konfigurace spuštěna. Seznam projektů a jejich konfigurací je zobrazen na panelu v levé části. Po kliknutí na název konfigurace je k dispozici tlačítko „Edit configuration“, které umožňuje nakonfigurovat jednotlivé kroky. Na obrázku níže je v levé části navigační menu spjaté s konkrétní konfigurací a uprostřed tlačítko „Add build step“ pro přidání jednotlivých kroků.



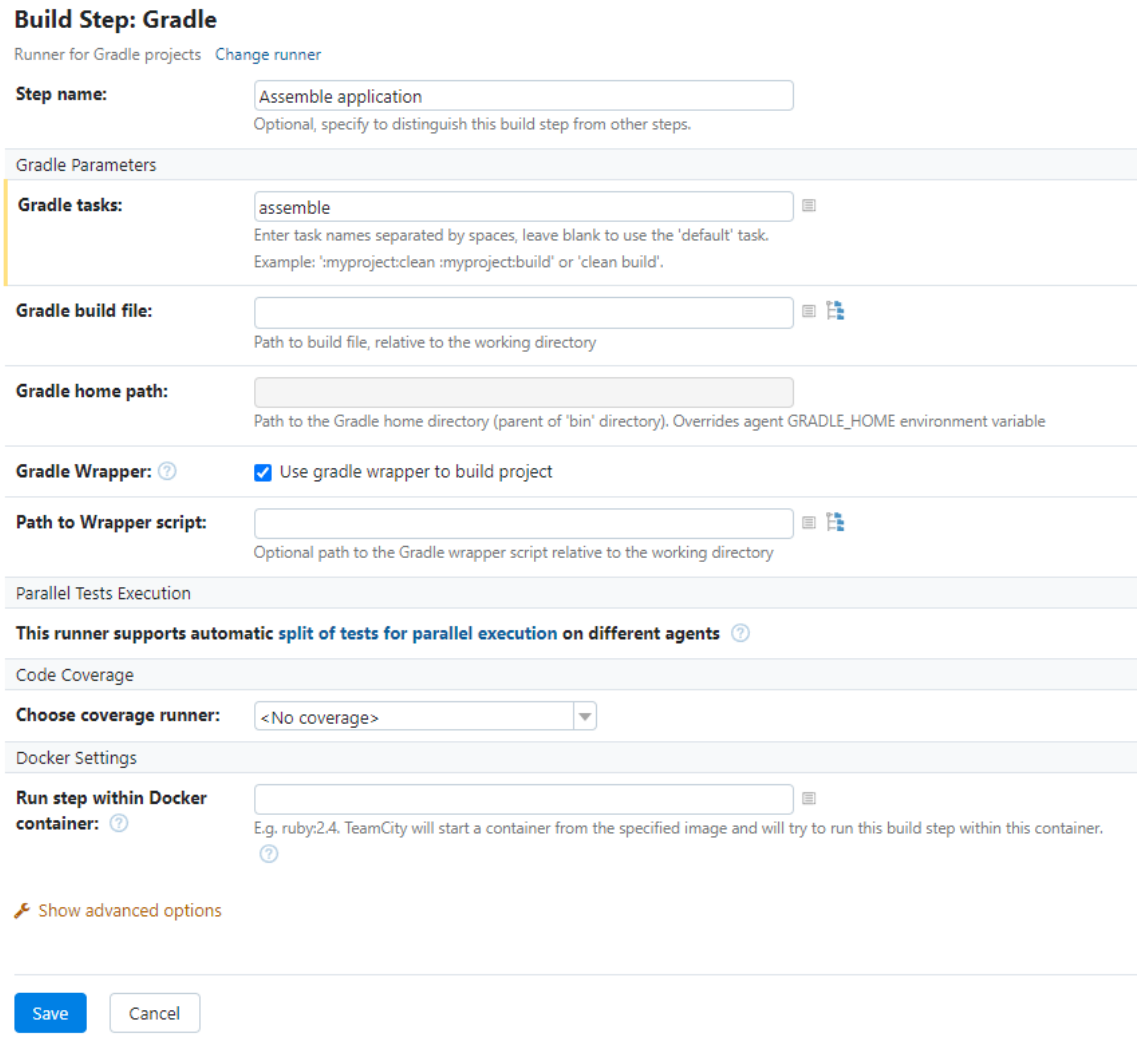
Obrázek 34: Detail konfigurace
Zdroj: vlastní zpracování

Při kliku na tlačítko „Add build step“ se nejdříve volí typ runneru. TeamCity nabízí desítky takových runnerů včetně Gradle, .NET, C#, Python, Maven, Ant, Docker a mnoho dalších. Ukázková aplikace využívá Gradle, tudíž je zvolena tato hodnota.



Obrázek 35: Volba typu runneru
Zdroj: vlastní zpracování.

Následuje již samotná konfigurace daného kroku. Tato konfigurace je zobrazena na obrázku níže.



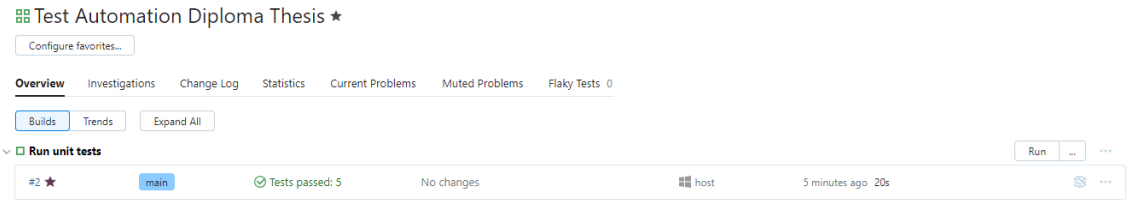
Obrázek 36: Konfigurace kroku pro sestavení projektu
Zdroj: vlastní zpracování

TeamCity umožňuje bohaté možnosti konfigurace. V tomto případě jsou využity konfigurační možnosti pro název konkrétního kroku a specifikace Gradle. Dalším krokem je spuštění testů. Konfigurace tohoto kroku je stejná vyjma jména a Gradle příkazu, který má nově hodnotu *test*.

Takto nastavený projekt se spustí s každou novou změnou v main větvi projektu v GitLabu. Toto nastavení lze změnit v sekci „Triggers“ pro konkrétní konfiguraci.

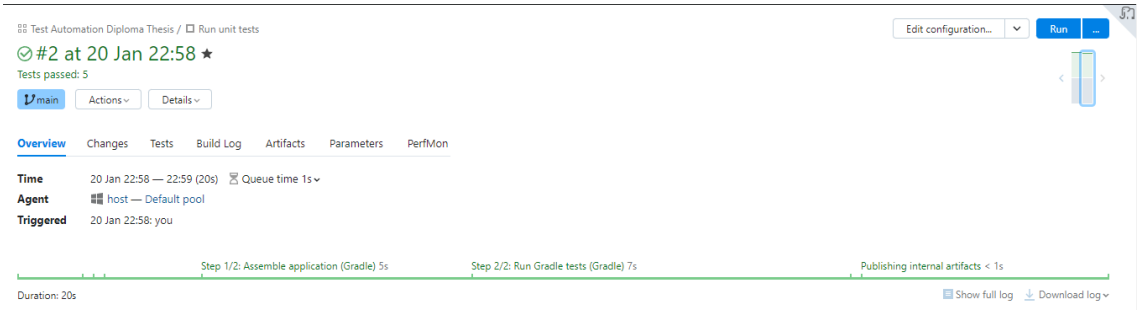
5.3.1.4 Výsledky

Projekt a jednotlivé konfigurace lze zobrazit v hlavní sekci TeamCity.



Obrázek 37: Přehled konfigurací projektu
Zdroj: vlastní zpracování

Samostatný běh dané konfigurace si lze detailně zobrazit proklikem na danou konfiguraci. Detail konfigurace z obrázku výše je zobrazen na obrázku níže.



Obrázek 38: Detail konfigurace
Zdroj: vlastní zpracování

Kromě všech kroků, které byly nakonfigurovány, umožňuje TeamCity zobrazit mimo jiné změny v repositáři, které se daného buildu týkají, build log nebo jednotlivé testy.

5.4 Srovnání nástrojů

V této podkapitole autor srovnává nástroje, které byly popsány v předchozích podkapitolách. Informace v následující tabulce vychází z oficiálních stránek každého nástroje a z autorových zkušeností s daným nástrojem.

Tabulka 3: Srovnání nástrojů pro integraci s CI/CD
Zdroj: vlastní zpracování

	GitLab	*Jenkins*	*TeamCity*
Placený	Ano (bezplatná verze k dispozici)	Ne (Open-source)	Ano (bezplatná verze k dispozici)
Licence	MIT Licence + vlastní	MIT licence	Vlastní
Podpora cloudu	Ano	Ne (vlastní hostování)	Ano (vlastní i cloudová podpora)
CI/CD	Zabudované	Pomocí pluginů	Zabudované
Podpora pro docker	Ano	Ano	Ano
Podpora SCM	Ano	Ano	Ano
Složitost nastavení	Snadné	Složitější	Složitější
Velikost komunity	Vysoká	Velmi vysoká	Střední

GitLab je dobrou volbou pro týmy, které preferují řešení „vše v jednom“. Jenkins je ideální pro týmy, které vyžadují vysoce přizpůsobitelný a rozšiřitelný nástroj. TeamCity je vhodný nástroj pro ty, kteří hledají snadno použitelné, komerčně podporované řešení CI/CD.

6 Automatizace testování ve společnosti Unicorn

Diplomová práce vznikla v rámci spolupráce se společností Unicorn a tato kapitola obsahuje ukázku automatizace testování na projektu Certigy v podobě testu, který je popsán v podkapitole 6.3.

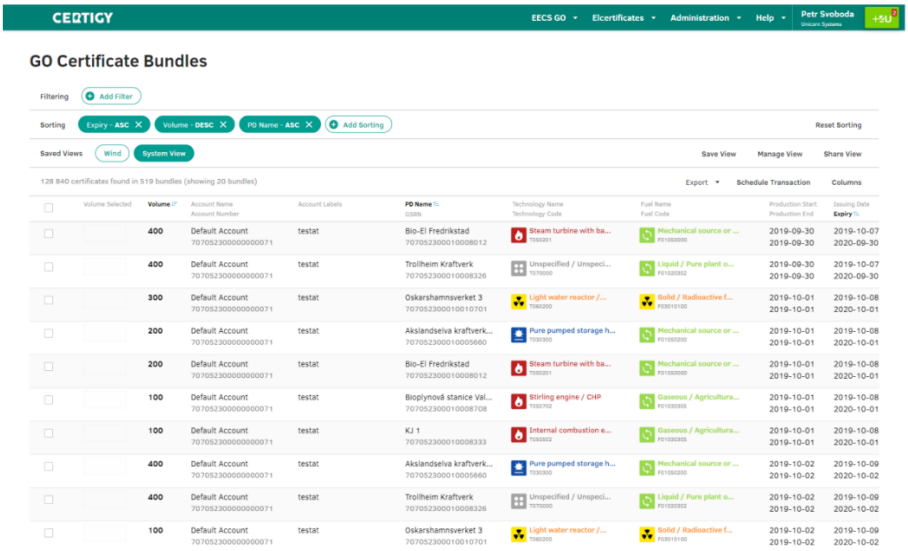
6.1 Představení společnosti Unicorn

Unicorn je evropská společnost založená v roce 1990, která poskytuje informační systémy a řešení z oblasti informačních technologií. Mezi její zákazníky patří přední firmy z různých odvětví jako jsou bankovnictví, pojišťovnictví, energetika a utility, komunikace a média, výroba, obchod a veřejná správa. Kromě toho společnost Unicorn provozuje internetovou službu Plus4U, kde nabízí široké portfolio služeb pro malé a střední podniky i jednotlivce. Společnost disponuje detailními znalostmi z celého spektra podnikatelských odvětví a rozumí principům jejich fungování a specifickým potřebám svých zákazníků. [59]

6.2 Představení projektu Certigy

Certigy je aplikace, která umožňuje zjednodušit proces certifikace pro výrobce zelené energie[9]. Díky přívětivému uživatelskému rozhraní a pokročilým možnostem sledování Certigy umožňuje jednoduchou správu a vydávání certifikátů. Uživatelské rozhraní aplikace Certigy je na obrázku níže

[9] Zelená energie je energie, která nezpůsobuje škodlivé emise a pochází z obnovitelných zdrojů jako jsou slunce, voda, vítr, biometan či geotermální energie.



Obrázek 39: Ukázka Certigy aplikace
Zdroj: Unicorn Systems a.s., interní dokumentace

Aplikace je navržena tak, aby se dala snadno integrovat se stávajícími systémy a lze ji přizpůsobit potřebám jednotlivých společností od malých začínajících firem až po nadnárodní korporace. Vzhledem k tomu, že poptávka po obnovitelných zdrojích energie stále roste, je Certigy připravena hrát významnou roli při utváření budoucnosti tohoto odvětví.

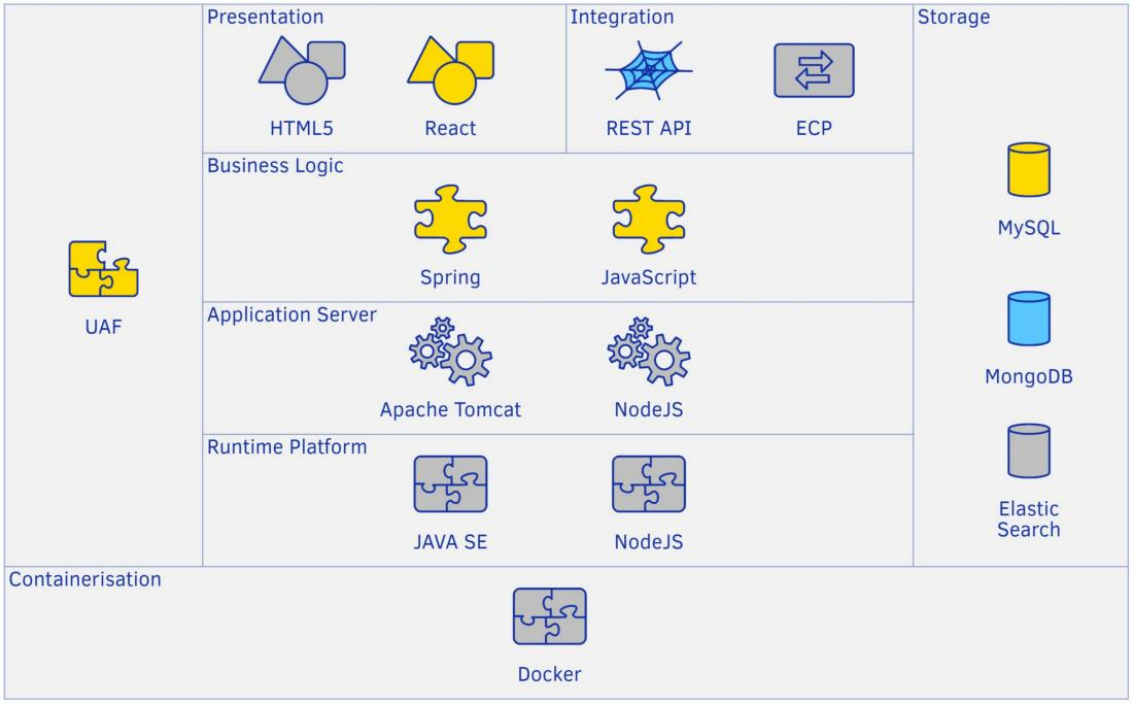
6.2.1 Architektura aplikace

Certigy používá mikroservisní architekturu, kdy jednotlivé komponenty pracují nezávisle na ostatních a komunikují mezi sebou pomocí REST API. Díky tomu je zajištěno, že Certigy může být jednoduše nasazena do cloudu nebo jako on-premise[10]. Mikroservisní architektura je v tomto případě vhodná z důvodu velkého zatížení systému. Mezi statistiky zatížení aplikace patří následující:

- Přes 500 milionů vydaných certifikátů ročně
- Přes jeden milion transakcí za měsíc
- Aplikace je využívána více než stovkou organizací

Použité technologie včetně konkrétní oblasti jejich použití jsou na obrázku níže.

[10] Lokálně k zákazníkovi do infrastruktury

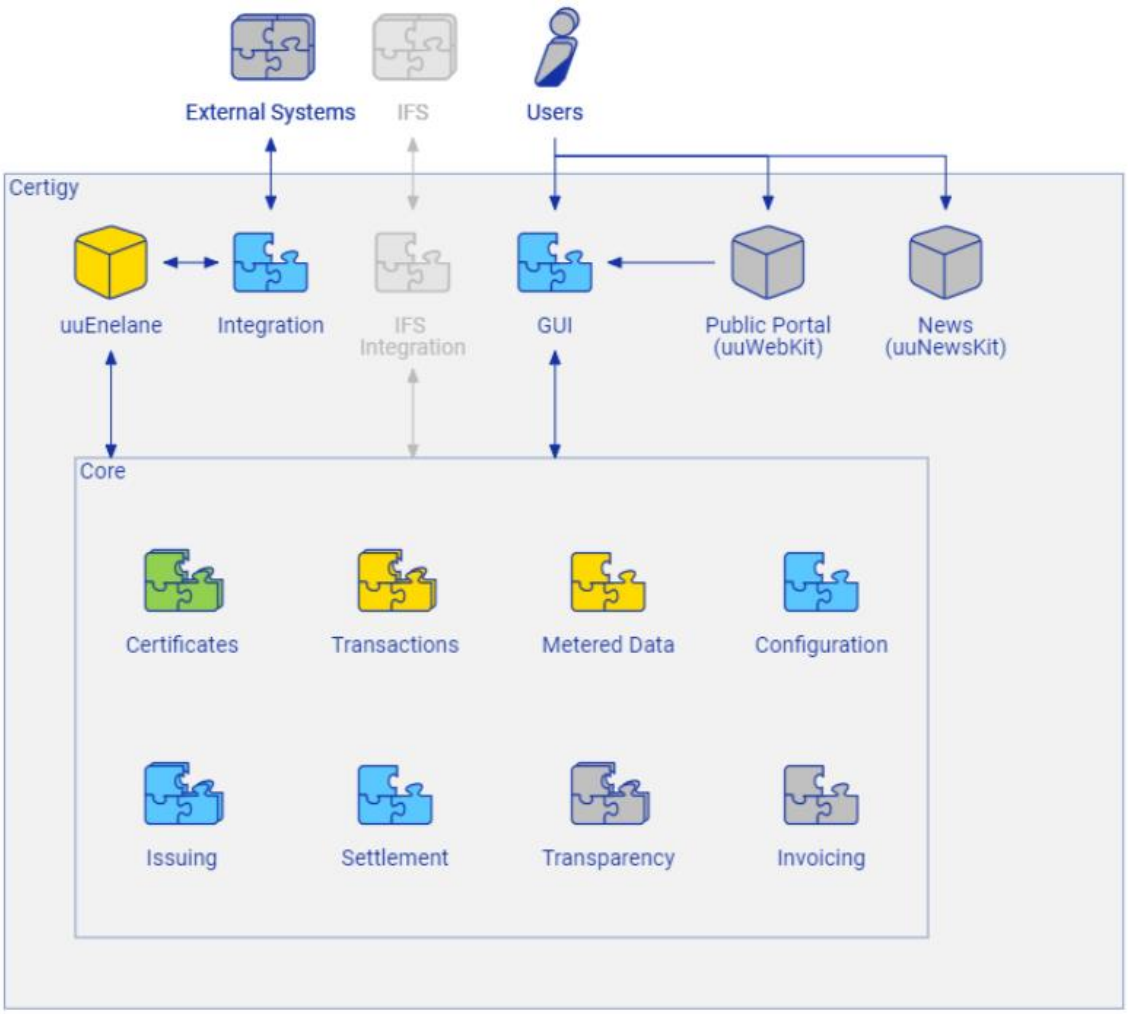


Obrázek 40: Technologie použité v projektu Certigy
Zdroj: Unicorn Systems a.s., interní dokumentace

V následujícím seznamu jsou uvedeny jednotlivé komponenty a jejich stručný popis, ze kterých se Certigy skládá:

- **Certificates** je modul zodpovědný za centrální správu certifikátů. Tento modul řídí životní cyklus jednotlivých typů certifikátů (akce na týdenní / měsíční / čtvrtletní bázi).
- **Transactions** komponenta poskytuje historii všech transakcí provedených v aplikaci včetně pohledů na tyto transakce.
- **Metered Data** komponenta je zodpovědná za ukládání a správu agregovaných produkčních dat.
- **Configuration** je modul zodpovědný za správu hlavních datových struktur.
- **Issuing** komponenta slouží jako primární zdroj certifikátů, které jsou vydávány pravidelně nebo ručně v procesu zvaném „Issuing Run“.
- **Settlement** komponenta poskytuje zprávy o vyúčtování.
- **Transparency** modul je hlavním komunikačním nástrojem a rozhraním pro neautorizované uživatele.
- **Invoicing** komponenta slouží k fakturaci všech držitelů účtu v systému NECS.

Schéma těchto komponent je na obrázku níže v sekci Core.



Obrázek 41: Certigy komponenty
Zdroj: Unicorn Systems a.s., interní dokumentace

6.3 Automatické testy

V rámci této podkapitoly jsou popsány automatické testy, které vznikly ve spolupráci se společností Unicorn. Jelikož se jedná o komerční aplikaci pod licencí, z bezpečnostních důvodů nejsou v této podkapitole některé konfigurace a kroky ukázány.

Certigy projekt stejně jako mnoho dalších produktů od společnosti Unicorn využívá k automatizaci testování nástroj TeamCity, který je blíže popsán v podkapitole 5.3. Následující podkapitola obsahuje popis testu, který byl napsán pro společnost

Unicorn v rámci této práce. Jedná se API test pomocí nástroje Postman, který je následně exportován do TeamCity.

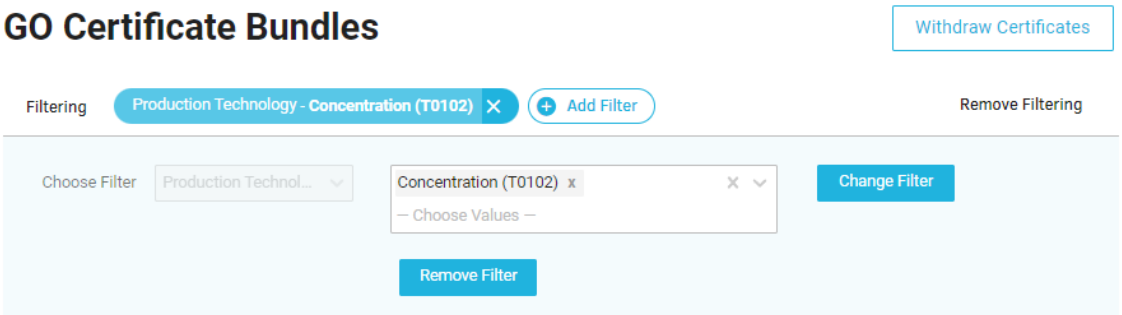
6.3.1 Test API komunikace

Tento test je realizován pomocí nástroje Postman, který může být použit mimo posílání requestů právě k testování API. Testování v této podkapitole se zaměřuje na testování pomocí nástroje Postman dle zadání společnosti Unicorn a integraci testu s nástrojem TeamCity. Konkrétně se jedná o testování na komponentě **Electrificate Bundles,** která zobrazuje jednotlivé certifikáty.

V této komponentě bylo za úkol otestovat filtrování certifikátů podle typu použité technologie. Typy použitých technologií jsou následující:

- Solární energie
- Větrná energie
- Hydro-elektrická instalace
- Mořská energie
- Termální energie
- Nukleární energie
- Nespecifikovaná energie

Přičemž každá z těchto kategorií je dále rozdělena na detailnější typy dané energie. Na obrázku níže je zobrazeno nastavení filtrování přes grafické webové rozhraní.



Obrázek 42: Nastavení filtrování Go Certificate Bundles
Zdroj: vlastní zpracování

Obrázek níže zobrazuje JSON strukturu jednoho certifikátu.

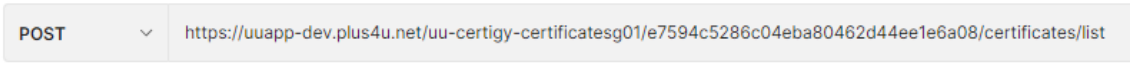
```
{
    "id": 20397,
    "awid": "e7594c5286c04eba80462d44ee1e6a08",
    "accountHolder": "08XG72PT9F",
    "accountHolderName": "Washington Mills AS",
    "accountName": "Mirda EL test 27-01-2023 163043",
    "accountNumber": "707052300000000323",
    "accountLabels": [
        "Test",
        "Transfer Account"
    ],
    "productionDeviceName": "ISSUING TEST LICENCE",
    "productionDeviceId": "707052300000000026",
    "typeOfInstallation": "T010000",
    "productionPeriodStart": "2022-12-01T23:00:00.000Z",
    "productionPeriodEnd": "2022-12-02T23:00:00.000Z",
    "volume": 55,
    "issuedDate": "2023-01-26T23:00:00.000Z",
    "startCertificateNumber": "708000340067400000000000015272",
    "endCertificateNumber": "708000340067400000000000015326",
    "earmarkFlag": "PRODUCTION",
    "issuingBody": "Statnett",
    "issuingBodyCode": "08",
    "countryOfIssue": "NO",
    "electricalCapacity": 40.0,
    "dateOperational": "2023-01-16T23:00:00.000Z",
    "energySource": "F00000000",
    "productionSupportDescription": "",
    "productionDeviceLongitude": "85",
    "productionDeviceLatitude": "85",
    "productionDeviceCoordinateCode": "WGS-84",
    "products": [
        {
            "name": "elcert",
            "purposes": [
                "SUPPORT"
            ],
            "types": [
                {
                    "value": "SOURCE",
                    "type": "driver"
                }
            ],
            "competentAuthorities": [
                {
                    "code": "NO01",
                    "name": {
                        "en": "Statnett"
                    }
                }
            ]
        }
    ]
},
```


Obrázek 43: JSON struktura certifikátu

Zdroj: vlastní zpracování

Pro účely tohoto testu je důležitý atribut s názvem *typeOfInstallation,* který určuje, o jaký typ zdroje se jedná. Pokud je například potřeba filtrovat certifikáty, které používají solární energii, musí všechny filtrované položky v atributu *typeOfInstallation* obsahovat textový řetězec, který začíná jako „T01“. Samotné testy v nástroji Postman se píšou v jazyce JavaScript. Cílem testů v této konkrétní situaci je prověřit, zda server odesílá certifikáty, které odpovídají požadovaným typům na základě použité technologie.

Prvním krokem je vytvořit kolekci v nástroji Postman, pod kterou budou sdruženy všechny testy. Název kolekce je ve formátu *UNI.NAZEV_PROJEKTU.NAZEV_KOMPONENTY,* v tomto případě se tedy jedná o *UNI.CERTIGY.EB*, jelikož testy se týkají projektu Certigy a jeho komponenty Electrificate Bundles. Úkolem bylo napsat testy pro filtrování na základě použité technologie. Jelikož těchto testů je větší množství, byl pro ukázku vybrán test pro kontrolu filtrování podle solární technologie certifikátů. Adresa post requestu pro testovací prostředí je zobrazena na obrázku níže.



Obrázek 44: Adresa testování
Zdroj: vlastní zpracování

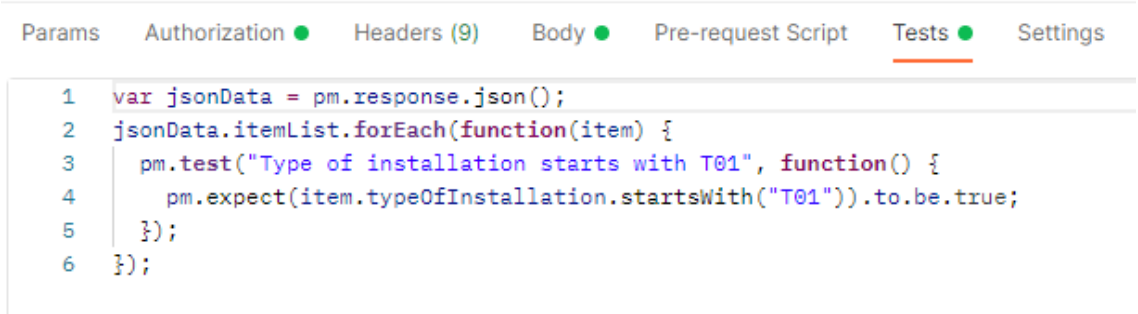
JSON obsah, který se na server odesílá, je zobrazen níže. Součástí requestu je i autorizační token, který se odesílá v autorizační hlavičce daného requestu.

```
{
  "filterMap": {
    "typeOfInstallation": ["T01"]
  },
  "sorterList": [],
  "pageInfo": {
    "pageSize": 100,
    "pageIndex": 0
  }
}
```

Ukázka kódu 7: POST request body
Zdroj: vlastní zpracování

Dalším krokem je definice samotného testu. Ta se provádí v záložce Tests pro konkrétní request. Test je ukázán na obrázku níže.

Stěžejní je zejména řádek číslo čtyři, na kterém se do pole *typeOfInstallation* nastavují jednotlivé parametry pro filtrování na základě typu použité technologie. V tomto případě se jedná o hodnotu „T01“, která označuje solární typ použité technologie.

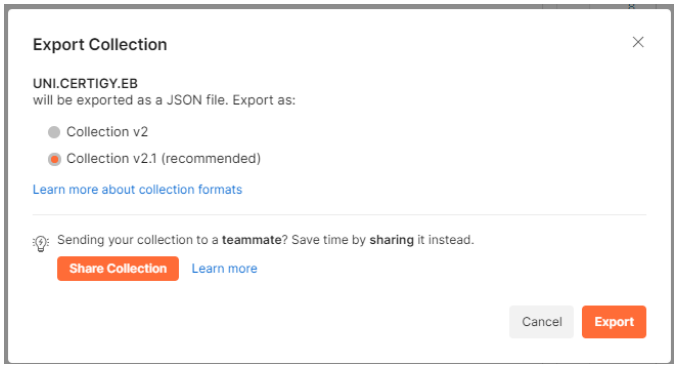


Obrázek 45: Ukázka testu filtrování
Zdroj: vlastní zpracování

Význam jednotlivých řádků je následující:

1) Uloží obsah ve formátu JSON do proměnné s názvem *jsonData*.
2) Endpoint vrací seznam certifikátů v poli s názvem *itemList,* přes který test iteruje.
3) Postman (*zkráceně pm*) je globální objekt, který poskytuje přístup informacím o daném requestu, zejména k jeho odpovědi. V tomto případě vytváříme funkci, respektive test, který ověří, zda atribut typeOfInstallation začíná hodnotou „T01“, která označuje solární typ použité technologie.

Dalším krokem je export celé kolekce, která obsahuje všechny testy, do souboru a následný commit do repositáře, který souvisí s danou komponentou. Export kolekce se provádí pravým klikem na název kolekce a zvolením možnosti export.



Obrázek 46: Export Postman kolekce
Zdroj: vlastní zpracování

Následuje konfigurace v TeamCity pro spouštění Postman testů. Kroky potřebné pro konfiguraci projektu v TeamCity jsou popsané v podkapitole 5.3 a z bezpečnostních důvodů zde není popsána konfigurace Gitu a dalších nastavení. Krok, který spouští Postman testy, je typu *Command Line* a obsahuje následující sktipt:

```
docker run
 --rm
-v "%system.teamcity.build.checkoutDir%"/postman:/etc/newman
postman/newman run
--reporters junit
--reporter-junit-export report.xml
collection.json
```

Jedná se o jeden řádek, v této ukázce je skript rozdělen do více řádků pro větší přehlednost. Význam jednotlivých řádků je následující:

1) Spuštění docker kontejneru
2) Flag –-rm automaticky odstraní kontejner po skončení.
3) Flag -v slouží k připojení externích soborů přímo do Docker image. V tomto případě se složka /postman, která je v kořenovém adresáři Git repositáře, překopíruje do adresáře etc/newman v Docker image. Na této cestě knihovna newman hledá .xml soubory s testy.
4) Postman/newman run spustí Docker image, který v sobě má zakomponovanou knihovnu newman, která dokáže spouštět Postman testy.
5) Flag –reporters junit specifikuje formát výsledků testů. V tomto případě se jedná o JUnit xml formát.

6) Flag –reporter-junit-export určuje název souboru, do kterého se ukládají výsledky testů. V tomto případě se cílový soubor jmenuje report.xml.
7) Název souboru, který obsahuje exportované Postman testy. Tento soubor je uložen na cestě etc/newman v běžícím Docker kontejneru.

Takto nakonfigurovaný step v TeamCity spouští Postman testy v Dockeru a exportuje jejich výsledky do nástroje TeamCity.

6.4 Výsledky spolupráce

Tato diplomová práce vznikla v rámci spolupráce se společností Unicorn. V této kapitole je popsán projekt Certigy, na kterém autor přispěl novými testy, které pomáhají udržet kvalitu daného projektu. Autor diplomové práce si v rámci spolupráce vyzkoušel práci v reálném prostředí, která mu přinesla nové zkušenosti a praktické znalosti. Autor si mohl ověřit teoretické vědomosti, kterých nabyl během studia a psaní této diplomové práce v pracovních podmínkách. Díky této spolupráci společnost Unicorn získala nové automatické testy, které využívá k testování svého projektu Certigy, jenž byl popsán v podkapitole 6.2.

7 Výsledky

Cílem diplomové práce bylo poskytnout přehled o automatizaci testování během vývoje softwaru, zejména v rámci agilního vývoje. Naplnění cílů práce započalo popisem cílů testování softwaru a klasifikací testů. Práce popisuje rozdíly mezi automatickým a manuálním testováním a mezi dynamickým a statickým testováním. Jsou zde uvedeny techniky testování softwaru a test management. Následuje část o agilních metodách a DevOps, které nabývají na popularitě. Dále je popsána automatizace testování včetně charakteristiky automatických testů. Jsou diskutovány výhody a nevýhody automatického testování a proces automatizace testování včetně popisu nástrojů pro automatizaci testování. Poté následuje kapitola o integraci automatického testování s CI/CD nástroji, jako jsou GitLab, Jenkins a TeamCity včetně praktické ukázky těchto nástrojů pro inspiraci při zavádění automatizace testování na projektu. Poslední část je věnována spolupráci s firmou Unicorn, kde je popsán projekt Certigy, na kterém autor přidal nové testy. Jeden test je vybrán a detailněji popsán v rámci této kapitoly. Konkrétně se jedná o API test, který je realizován v nástroji Postman, který je následně spouštěn v nástroji TeamCity. Tento test se zaměřuje na filtrování certifikátů podle typu použité technologie.

Kromě nových testů pro firmu Unicorn přináší tato diplomová práce také komplexní pohled na problematiku automatizace testování softwaru a iterativního vývoje, který je stále oblíbenější. Práce nabízí srozumitelný teoretický základ pro automatizaci testování softwaru včetně příkladů nástrojů, které lze pro automatické testování využít.

8 Závěr

Diplomová práce se zaměřuje na shrnutí procesu tvorby automatizovaných testů pro vývoj softwaru a jejich využití, zejména v souvislosti s agilním vývojem softwaru. V rámci této práce jsou vysvětleny základní koncepty testování, DevOps a agilní metodiky, stejně jako automatizované testování a nástroje, které se používají k jeho realizaci.

Automatické testování software je nezbytnou součástí agilního vývoje, DevOps a CI/CD procesů, které získávají na popularitě. Díky automatizaci testování mohou vývojáři snáze identifikovat a opravit chyby. Integrace automatického testování do CI/CD procesu umožňuje automatické spouštění testů př každé změně kódu. Tato integrace urychluje detekci chyb, ale také umožnuje vývojářům se zaměřit na psaní kódu, zatímco nástroje pro kontinuální testování se starají o validaci jeho kvality. I přes to, že automatizace procesu testování přináší mnoho výhod, je důležité si uvědomit, že nemůže zcela nahradit manuální testování. Proto je vhodné provádět průběžnou automatizaci testování v kombinaci s manuálním testováním, aby bylo možné zajistit komplexní pokrytí testování a odhalit chyby, které by mohly uniknout pouze automatickým testům.

Automatizovaný proces testování je klíčovým prvkem pro zajištění kvality a efektivity agilního vývoje software. Správně implementovaný proces automatického testování vede ke snížení rizika chyb, zvýšení spokojenosti zákazníků a konkurenceschopnosti produktu na trhu.

Tato diplomová práce poskytuje ucelený pohled na současné praktiky v oblasti testování software a automatizaci testování software. Je zřejmé, že automatizace testování je klíčovým prvkem úspěšného agilního vývoje a DevOps pipeline. Význam automatického testování software bude nadále růst, a proto je důležité, aby vývojáři, testeři a manažeři byli obeznámeni s nejnovějšími nástroji, metodami a postupy, které jim umožní zvládnout výzvy spojené s rychlým vývojem a nasazením kvalitního software.

9 Seznam tabulek

10 Seznam obrázků

11 Seznam ukázek kódů

12 Seznam použité literatury

[1] QUADRI, S.M.K a Sheikh Umar FAROOQ. Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications* [online]. 2010, **6**(9), 7–10. ISSN 09758887. Dostupné z: doi:10.5120/1343-1448

[2] ISTQB. *ISTQB-CTFL Syllabus* [online]. 2018. Dostupné z: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf

[3] HLAVA, Tomáš. *Statické a dynamické testy | Testování softwaru* [online]. 12. srpen 2011 [vid. 2023-01-27]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/staticke-a-dynamicke-testy/

[4] ZELINKA, Bořek. *Testování softwaru* [online]. 10. duben 2013. Dostupné z: https://d3s.mff.cuni.cz/legacy/teaching/commercial_workshops/previous/1213/zelinka-zajisteni_kvality_softwarovych_produktu.pdf

[5] VERMA, Akanksha, Amita KHATANA, Sarika CHAUDHARY, a DEPARTMENT OF COMPUTER SCIENCE, AMITY UNIVERSITY, GURGAON, INDIA. A Comparative Study of Black Box Testing and White Box Testing. *International Journal of Computer Sciences and Engineering* [online]. 2017, **5**(12), 301–304. ISSN 23472693. Dostupné z: doi:10.26438/ijcse/v5i12.301304

[6] EHMER, Mohd a Farmeena KHAN. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications* [online]. 2012, **3**(6) [vid. 2023-01-15]. ISSN 2158107X, 21565570. Dostupné z: doi:10.14569/IJACSA.2012.030603

[7] HUSSAIN, Taraq a Dr Satyaveer SINGH. A Comparative Study of Software Testing Techniques Viz. White Box Testing Black Box Testing and Grey Box Testing. 2015, (01).

[8] OIVO a HOFMANN. *Product Focused Software Process Improvement*. B.m.: Springer Berlin Heidelberg, 2002. ISBN 978-3-662-18645-9.

[9] HILL, T Adrian. Stress Testing Embedded Software Applications. In: *Embedded Systems Conference* [online]. 2007. Dostupné z: https://messenger.jhuapl.edu/Resources/Publications/Hill.2007.pdf

[10] ALAM, Mottahir. Risk-based Testing Techniques: A Perspective Study. *International Journal of Computer Applications* [online]. 2013, **65**. Dostupné z: https://www.researchgate.net/publication/235751299_Risk-based_Testing_Techniques_A_Perspective_Study

[11] HERTZUM, Morten. *Usability Testing: A Practitioner's Guide to Evaluating the User Experience* [online]. Cham: Springer International Publishing, 2020

[vid. 2023-01-03]. Synthesis Lectures on Human-Centered Informatics. ISBN 978-3-031-01099-6. Dostupné z: doi:10.1007/978-3-031-02227-2

[12] ITKONEN, J. a K. RAUTIAINEN. Exploratory testing: a multiple case study. In: *2005 International Symposium on Empirical Software Engineering, 2005.*: *2005 International Symposium on Empirical Software Engineering, 2005.* [online]. Queensland, Australia: IEEE, 2005, s. 82–91 [vid. 2023-01-04]. ISBN 978-0-7803-9507-7. Dostupné z: doi:10.1109/ISESE.2005.1541817

[13] *ISTQB Glossary* [online]. [vid. 2023-01-08]. Dostupné z: https://glossary.istqb.org/en/search/

[14] Certified Tester Advanced Level - Test Manager (CTAL-TM). *ISTQB not-for-profit association* [online]. [vid. 2023-01-08]. Dostupné z: http://istqb.org/certifications/test-manager

[15] *DevOps vs. agilita | Microsoft Azure* [online]. [vid. 2023-01-04]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/devops-vs-agile/

[16] GHIMIRE, Dipendra a Stuart CHARTERS. The Impact of Agile Development Practices on Project Outcomes. *Software* [online]. 2022, **1**(3), 265–275. ISSN 2674-113X. Dostupné z: doi:10.3390/software1030012

[17] LEAU, Yu Beng, Wooi Khong LOO, Wai Yip THAM a Soo Fun TAN. Software Development Life Cycle AGILE vs Traditional Approaches. In: *International Conference on Information and Network Technology*. 2012.

[18] STOICA, Marian, Marinela MIRCEA a Bogdan GHILIC-MICU. Software Development: Agile vs. Traditional. *Informatica Economica* [online]. 2013, **17**(4/2013), 64–76. ISSN 14531305, 18428088. Dostupné z: doi:10.12948/issn14531305/17.4.2013.06

[19] BECK, Kent a Cynthia ANDRES. *Extreme programming explained: embrace change*. 2nd ed. Boston, MA: Addison-Wesley, 2005. ISBN 978-0-321-27865-4.

[20] SCHWABER, Ken. *Agile project management with Scrum*. Redmond, Wash: Microsoft Press, 2004. ISBN 978-0-7356-1993-7.

[21] The Scrum Framework Poster. *Scrum.org* [online]. [vid. 2023-01-05]. Dostupné z: https://www.scrum.org/resources/scrum-framework-poster

[22] ANWER, Faiza, Shabib AFTAB, Usman WAHEED a Syed Shah MUHAMMAD. Agile Software Development Models TDD, FDD, DSDM, and Crystal Methods: A Survey [online]. 2017, **8**(2). Dostupné z: https://www.researchgate.net/publication/316273992

[23] Software Development Methodology - Feature Driven Development (FDD). *Tuan Nguyen's Blog* [online]. 22. červenec 2019 [vid. 2023-01-05]. Dostupné

z: https://www.tuannguyen.tech/2019/07/software-development-methodology-feature-driven-development-fdd/

[24] ALAIDAROS, Hamzah, Mazni OMAR a Rohaida ROMLI. The state of the art of agile kanban method: challenges and opportunities. *Independent Journal of Management & Production* [online]. 2021, **12**(8), 2535–2550. ISSN 2236-269X, 2236-269X. Dostupné z: doi:10.14807/ijmp.v12i8.1482

[25] DENNEHY, Denis a Kieran CONBOY. Going with the flow: An activity theory analysis of flow techniques in software development. *Journal of Systems and Software* [online]. 2017, **133**, 160–173. ISSN 01641212. Dostupné z: doi:10.1016/j.jss.2016.10.003

[26] *Kanban board* [online]. 2022 [vid. 2023-01-06]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Kanban_board&oldid=1130296826

[27] SHARMA, Sanjeev. *The DevOps Adoption Playbook: A Guide to adopting DevOps in a multi-speed IT Enterprise* [online]. Indianapolis, Indiana: John Wiley & Sons, Inc., 2017 [vid. 2023-01-04]. ISBN 978-1-119-31077-8. Dostupné z: doi:10.1002/9781119310778

[28] KIM, Gene, Kevin BEHR a George SPAFFORD. *The Phoenix project: a novel about it, DevOps, and helping your business win; [revised with new resource guide]*. Portland, Or: IT Revolution Press, 2014. ISBN 978-0-9882625-0-8.

[29] KAWAGUCHI, Stephen. Applying the Three Ways of DevOps to Accelerate Your Organization. *The Startup* [online]. 24. září 2020 [vid. 2023-01-04]. Dostupné z: https://medium.com/swlh/applying-the-three-ways-of-devops-to-accelerate-your-organization-702998ebf1c2

[30] ALMEIDA, Fernando, Jorge SIMÕES a Sérgio LOPES. Exploring the Benefits of Combining DevOps and Agile. *Future Internet* [online]. 2022, **14**(2), 63. ISSN 1999-5903. Dostupné z: doi:10.3390/fi14020063

[31] SUBRAMANYA, Rakshith, Seppo SIERLA a Valeriy VYATKIN. From DevOps to MLOps: Overview and Application to Electricity Market Forecasting. *Applied Sciences* [online]. 2022, **12**(19), 9851. ISSN 2076-3417. Dostupné z: doi:10.3390/app12199851

[32] PENNINGTON, Jakob. The Eight Phases of a DevOps Pipeline. *Taptu* [online]. 27. červenec 2020 [vid. 2023-01-10]. Dostupné z: https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba

[33] KHAN, Asif. Practicing Continuous Integration and Continuous Delivery on AWS Accelerating Software Delivery with DevOps [online]. 2017. Dostupné z: https://www.researchgate.net/publication/330988585

[34] PRINCE, Suzie. The Product Managers' Guide to Continuous Delivery and DevOps. *Mind the Product* [online]. 11. únor 2016 [vid. 2023-01-06]. Dostupné z: https://www.mindtheproduct.com/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/

[35] SHARMA, Sanjeev. *DevOps For Dummies® 3rd IBM Limited Edition*. 2017. ISBN 978-1-119-41588-6.

[36] *AT*SQA: DevOps Testing* [online]. [vid. 2023-01-09]. Dostupné z: https://atsqa.org/certifications/atsqa-devops-testing

[37] FABER, Frank. Testing in DevOps. In: Stephan GOERICKE, ed. *The Future of Software Quality Assurance* [online]. Cham: Springer International Publishing, 2020 [vid. 2023-01-08], s. 27–38. ISBN 978-3-030-29508-0. Dostupné z: doi:10.1007/978-3-030-29509-7_3

[38] *About AT*SQA* [online]. [vid. 2023-01-09]. Dostupné z: https://atsqa.org/about

[39] BLACK, Rex. *Agile Testing Foundations : An ISTQB Foundation Level Agile Tester Guide*. nedatováno. ISBN 1-78017-336-9.

[40] GEARY, Brian. *7 Steps of Test-Driven Development* [online]. [vid. 2023-01-10]. Dostupné z: https://www.andplus.com/blog/test-driven-development

[41] ANAND NAYYAR. *INSTANT APPROACH TO SOFTWARE TESTING*. S.l.: BPB PUBLICATIONS, 2019. ISBN 978-93-88511-16-2.

[42] *AT*SQA Micro-Credentials for Software Testing - Test Automation* [online]. [vid. 2023-01-11]. Dostupné z: https://atsqa.org/micro-credentials/test-automation

[43] ADMIN. What is automated testing? | IT Craft. *ITCraft* [online]. 25. říjen 2019 [vid. 2023-01-11]. Dostupné z: https://itechcraft.com/blog/automated-testing-how-it-works-and-how-it-affects-roi/

[44] KUMAR, Divya a K.K. MISHRA. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science* [online]. 2016, **79**, 8–15. ISSN 18770509. Dostupné z: doi:10.1016/j.procs.2016.03.003

[45] SULTANIA, Ashish Kumar. Developing software product and test automation software using Agile methodology. In: *2015 3rd International Conference on Computer, Communication, Control and Information Technology (C3IT)*: *Proceedings of the 2015 Third International Conference on Computer, Communication, Control and Information Technology (C3IT)* [online]. Hooghly, India: IEEE, 2015, s. 1–4 [vid. 2023-01-11]. ISBN 978-1-4799-4446-0. Dostupné z: doi:10.1109/C3IT.2015.7060120

[46] LAPLAZA, Irene a Simo SUURKUUKKA. Software testing automation tools and methods: The good, the bad & the ugly [online]. 2020 [vid. 2023-01-12]. Dostupné z: doi:10.13140/RG.2.2.21514.08645

[47] RODRIGUES, Anderson a Arilo DIAS-NETO. Relevance and Impact of Critical Factors of Success in Software Test Automation lifecycle: A Survey. In: *SAST '16: 1st Brazilian Symposium on Systematic and Automated Software Testing*: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing* [online]. Maringa Parana Brazil: ACM, 2016, s. 1–10 [vid. 2023-01-13]. ISBN 978-1-4503-4766-2. Dostupné z: doi:10.1145/2993288.2993302

[48] Test Automation Engineer. *ISTQB not-for-profit association* [online]. [vid. 2023-01-12]. Dostupné z: http://istqb.org/certifications/test-automation-engineer

[49] UMAR, Mubarak Albarka a Chen ZHANFANG. A Study of Automated Software Testing: Automation Tools and Frameworks [online]. 2019, **8**. Dostupné z: doi:10.5281/zenodo.3924795

[50] The Selenium Browser Automation Project. *Selenium* [online]. [vid. 2023-01-14]. Dostupné z: https://www.selenium.dev/documentation/

[51] *JUnit 5 User Guide* [online]. [vid. 2023-01-17]. Dostupné z: https://junit.org/junit5/docs/current/user-guide/

[52] *Apache JMeter - Apache JMeter™* [online]. [vid. 2023-01-25]. Dostupné z: https://jmeter.apache.org/

[53] ATLASSIAN. What is Jira Software used for? *Atlassian* [online]. [vid. 2023-01-25]. Dostupné z: https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for

[54] *[JRACLOUD-80397] Export to Word - Table contents are truncated if text is too long in the table cells - Create and track feature requests for Atlassian products.* [online]. [vid. 2023-01-25]. Dostupné z: https://jira.atlassian.com/browse/JRACLOUD-80397?filter=-4

[55] The Importance of Maintaining Your Automated Tests. *Telerik Blogs* [online]. 19. srpen 2021 [vid. 2023-04-05]. Dostupné z: https://www.telerik.com/blogs/importance-maintaining-automated-tests

[56] Test Automation Best Practices. *smartbear.com* [online]. [vid. 2023-04-07]. Dostupné z: https://smartbear.com/learn/automated-testing/best-practices-for-automation/

[57] *The scope of runners | GitLab* [online]. [vid. 2023-01-16]. Dostupné z: https://docs.gitlab.com/ee/ci/runners/runners_scope.html

[58] TeamCity Documentation Home | TeamCity On-Premises. *TeamCity On-Premises Help* [online]. [vid. 2023-01-25]. Dostupné z: https://www.jetbrains.com/help/teamcity/teamcity-documentation.html

[59] Unicorn. *Unicorn* [online]. [vid. 2023-01-29]. Dostupné z: https://unicorn.com/cs/unicorn.com

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. Jan Chaloupka**
Studium:	I2100059
Studijní program:	N1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název diplomové práce:	**Continuous Test Automation**
Název diplomové práce AJ:	Continuous Test Automation

Cíl, metody, literatura, předpoklady:

Cílem diplomové práce je shrnutí tvorby automatických testů při vývoji SW a jejich využití v agilním vývoji.

Osnova:

- Úvod
- Cíl práce
- Testování software
- Devops / agilní metodiky
- Automatické testování
- Integrace CI/CD
- Závěr
- Literatura

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Ing. Filip Malý, Ph.D.
Datum zadání závěrečné práce:	26.1.2021