ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Testování integrací client-server

Testing of client-server integrations

Bc. Tomáš Markacz

Otevřená informatika
Softwarové inženýrství

Květen 2018

Vedoucí práce: RNDr. Ondřej Žára

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Markacz** Jméno: **Tomáš** Osobní číslo: **407044**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Testování integrací client-server

Název diplomové práce anglicky:

Testing of client-server integrations

Pokyny pro vypracování:

1. Nastudujte problematiku softwarového testování s detailním zaměřením na testování integrací client-server.
2. Popište a porovnejte existující způsoby testování integrací client-server, použitelné v kontextu mobilních a webových aplikací.
3. Vyhledejte existující nástroje umožňující popisované druhy testování (předpokládejte klienta ve formě webové aplikace napsané v Javascriptu a webový server využívající Node.js). Tyto nástoje vyzkoušejte, shrňte jejich výhody a nevýhody a zvažte, je-li na trhu prostor pro další takový software.
4. Navrhněte a implementujte knihovnu či nástroj umožňující testování podle zvoleného návrhu. Realizované řešení zdokumentujte (včetně automaticky generované dokumentace) a demonstrujte jeho použití na ukázkovém testu. Vzniklý produkt publikujte jako open-source software

Seznam doporučené literatury:

[1] DESIKAN, Srinivasan a Gopalaswamy RAMESH. Software testing principles and practices. Bangalore, India: Dorling Kindersley (India), 2006. ISBN 9788177581218.
[2] MOSLEY, Daniel J. Client-server software testing on the desktop and the Web. Upper Saddle River, NJ: Prentice Hall PTR, c2000. ISBN 9780131838802.
[3] GOGLIA, Patricia. Testing client/server applications. Boston: QED Pub. Group, c1993. ISBN 9780894354502.

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Ondřej Žára, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2018** Termín odevzdání diplomové práce: **09.01.2018**

Platnost zadání diplomové práce: **30.09.2019**

RNDr. Ondřej Žára podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání Podpis studenta

Poděkování / Prohlášení

V prvé řadě bych chtěl poděkovat panu RNDr. Ondřeji Žárovi za cenné podněty a zpětnou vazbu při vedení mé diplomové práce.

Dále bych chtěl poděkovat své rodině za podporu při studiu.

V neposlední řadě bych chtěl také poděkovat Ing. Martinu Šteklovi za odborné rady týkající se především implementace testovacího nástroje.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. 5. 2018

.....

Abstrakt / Abstract

Hlavním cílem této práce je popsat, jakými způsoby lze provádět testování integrací client-server. Práce se zaměřuje především na prostředí mobilních a webových aplikací. Součástí je také návrh nového způsobu testování integrací client-server a realizace nástroje, který tento způsob testování umožňuje.

Práce se zaměřuje na tři existující způsoby testování – Mock testing, Consumer-Driver Contracts testing a End-to-end testing. Je také navržen nový způsob synchronizovaného testování. Na základě shromážděných zdrojů jsou tyto způsoby charakterizovány, rozděleny do dvou skupin a detailněji popsány. Dále jsou způsoby porovnány pomocí šesti kritérií a výsledky zaneseny do tabulky. K existujícím způsobům testování jsou navrženy vhodné testovací nástroje. Tyto nástroje jsou v práci posány, je demostrováno jejich použití a shrnuty výhody a nevýhody.

Výsledkem praktické části této diplomové práce je funkční implementace nástroje, který umožňuje testování integrací client-server podle návrženého způsobu testování. Vytvořený nástroj lze použít pro přímé testování integrace webové aplikace napsané v JavaScriptu a webového serveru využívajícího Node.js.

Tato práce přináší ucelený zdroj informací zabívající se problematikou testování integrací client-server. Navržený způsob testování je chybějící alternativou, k již existujícím způsobům. Zavedení tohoto způsobu testování do praxe je podpořeno nástrojem, který vznikl v rámci praktické části této práce.

Klíčová slova: integrační testování, client-server, mobilní a webová aplikace, JavaScript, Node.js, mocking, consumer-driver contracts, end-to-end

The main goal of this thesis is to describe different approaches to client-server integration testing. The thesis' main focus are mobile and web applications. It also includes design of new approach to client-server integration testing and implementation of a tool which uses this testing approach.

The thesis deals with three existing testing approaches – Mock testing, Consumer-Driver Contracts testing and End-to-end testing. A new way of synchronized testing is also proposed. Based on the collected resources, these approaches are characterized, divided into two groups and described in more detail. All these approaches are compared using six criteria and the results are presented in a table. Appropriate testing tools are proposed for each of existing test approaches. These tools are described, examples of their use are shown, and their advantages and disadvantages are summarized.

The result of the practical part of this thesis is functional implementation of a tool that allows testing the client-server integration according to the proposed testing approach. The created tool can be used to test direct integration of web application written in JavaScript and web server powered by Node.js.

This thesis brings a comprehensive source of information covering the topic of client-server integration testing. The proposed test approach is the missing alternative to the existing ones. The practical application of this test approach is supported by the tool created in the practical part of this thesis.

Keywords: integration testing, client-server, mobile and web application, JavaScript, Node.js, mocking, consumer-driver contracts, end-to-end

Obsah /

Úvod

Automatizované testování je jeden z hlavních pilířů řízení kvality vyvíjeného softwaru. Díky němu mohou vznikat kvalitní aplikace, které neobsahují chyby mající za následek znemožnění jejich používání, ztrátu a únik citlivých dat nebo prezentování nepravdivých skutečností.

Mezi hlavní cíle této práce patří popis a porovnání existujících způsobů testování client-server integrací v kontextu mobilních a webových aplikací a shromáždění relevantních zdrojů zabývajících se touto problematikou. Dále pak vyhledání, vyzkoušení a popis existujících nástrojů umožňujících popsané druhy testování. Pro tyto účely se předpokládá integrace klienta ve formě webové aplikace napsané v JavaScriptu a webového serveru využívajícího Node.js. V neposlední řadě je cílem návrh a implementace nástroje, který umožní testovat integrace client-server podle chybějícího způsobu testování.

V současné době je většina webových a mobilních aplikací postavených na client-server architektuře. Klientská aplikace ve formě webové nebo mobilní aplikace komunikuje přes internet se serverem, který zajišťuje načítání, zpracovávání a ukládání dat. Testování webových a mobilních aplikací postavených na client-server architektuře však není triviální. Jedná se o druh integračního testování, který je specifický zmíněnou architekturou a svým zaměřením na technologie orientované na webový a mobilní vývoj. Téma zabývající se testováním integrací client-server je důležité, protože se k němu jednotlivci, vývojové týmy i firmy staví různými způsoby a chybí vhodný ucelený zdroj, který by popisoval a porovnával současné možnosti.

Na začátku práce se proto nejdříve zabývám obecnou problematikou softwarového testování a definuji důležité pojmy, které v práci používám. Tématu testování integrací client-server se pak věnuji ve druhé kapitole. Popsané způsoby testování jsem realizoval za pomoci existujících nástrojů vhodných pro testování mobilních a webových aplikací. Díky realizaci jsem si vytvořil ucelený pohled na jednotlivé nástroje a zjistil jejich výhody a nevýhody. Svoje poznatky popisuji ve třetí kapitole.

Během rešerše způsobů testování i existujících nástrojů jsem zjistil, že neexistuje způsob vhodný pro testování integrace mobilní aplikace a serveru, na které společně s kolegy z týmu vývojářů pracuji. Proto jsem se na základě této skutečnosti rozhodl pro návrh a implementaci nástroje, který by tento způsob testování umožnil realizovat. Popisem tohoto nástroje se zabývám ve čtvrté kapitole.

Nedílnou součástí této práce jsou také zdrojové kódy vzniklého nástroje společně s jeho dokumentací a ukázkovým testem demonstrujícím použití. Vzniklý nástroj jsem rovněž publikoval ve formě open-source softwaru.

Kapitola 1
Softwarové testování obecně

V této kapitole se zaměřuji na problematiku softwarového testování, která je nezbytná pro následující kapitoly. Hned na úvod bych rád zmínil definici toho, co softwarové testování vlastně je. Softwarové testování je proces vykonávání programu za účelem najít chyby [1]. V odborné literatuře lze najít různé, často složitější definice, tato však podle mě nejpřesněji vyjadřuje způsob, jak by se k testování mělo přistupovat, a co je jeho přidaná hodnota. Cílem testování není ukázat, že software neobsahuje chyby nebo to, že dělá, co se od něj očekává. Cílem testování je v programu, který obsahuje chyby, najít co nejvíce těchto chyb a odstranit je. Přidaná hodnota testování tedy tkví ve zvýšení kvality a spolehlivosti softwaru.

1.1 Způsoby provádění testů

Softwarové testování lze rozdělit na dva způsoby, podle toho, zda jej provádí člověk nebo stroj. Tyto dva způsoby se nazývají manuální a automatizované testování. S každým z nich jsou spojené výhody a nevýhody [2–3].

Manuální testování probíhá tak, že testeři provádějí testování vlastními silami. Pro nalezení chyb porovnávají očekávané chování programu se skutečným chováním. Manuální testování není ve své podstatě nic jiného než tester používající program stejným způsobem, jakým jej používá i běžný uživatel, s průběžnou kontrolou toho, že se program chová správně.

Mezi výhody manuálního testování patří:

- Lidský faktor. Získání stejné odezvy jako té od běžného uživatele. Skript nedokáže hodnotit vzhled uživatelského rozhraní nebo subjektivní rychlost.
- Levnější v krátkém časovém horizontu. Manuální testování má minimální počáteční náklady.
- Flexibilita. Automatizované testy vyžadují sestavení testovacích případů, napsaní testovacího kódu a jeho spuštění. Pokud je však potřeba jen malá změna, je jednodušší ji otestovat manuálně ihned.

Mezi nevýhody manuálního testování patří:

- Nelze aplikovat na vše. Některé případy mohou být pro člověka náročné na manuální testování. Například testování nízkoúrovňového rozhraní může být pracné, náročné na pozornost a hrozí přehlédnutí chyby. Automatizované testy mají pro tyto účely mnohem lepší předpoklady.
- Je únavné. Manuální testování je opakující se monotónní činnost. Ve výsledku může být pro testery náročné zůstat v pozoru a chyby v testovaném programu se mohou projevit bez jejich povšimnutí.
- Není znovupoužitelné. Pokud dochází k častým změnám, je potřeba manuální testy provádět častěji, což stojí čas i peníze. V případě automatizovaného testování lze získat výsledky mnohonásobně rychleji.

Automatizované testování používá specializovaný software, který spouští testovaný program a provádí předdefinované akce. Jeho úkolem je řídit vykonávání testů, sledovat testovaný program a porovnávat předpokládané výstupy programu s těmi skutečnými. Pokud nedojde ke shodě, je potřeba, aby člověk prověřil testovaný program, ale i samotný test, protože neshoda výstupů nemusí vždy nutně znamenat chybu v programu.

Mezi výhody automatizovaného testování patří:

- Efektivita a rychlost. Skripty jsou rychlejší než lidé, dokáží sledovat více věcí naráz a udrží více informací v paměti.
- Nižší náklady v dlouhodobém časovém horizontu. Automatizované testování vyžaduje vyšší počáteční náklady, avšak dlouhodobě vychází levněji. Nejen, že dokáže za stejný čas více než člověk, ale také dokáže rychleji objevit chyby.
- Znovupoužitelnost. I když prvotní implementace automatizovaných testů zabere nějaký čas, jakmile je vše připraveno, další změny lze realizovat rychle, protože úprava nebo znovupoužití testu je rychlejší než opakované manuální testování.

Mezi nevýhody automatizovaného testování patří:

- Nemožnost testovat neurčité vlastnosti. Nelze testovat uživatelský prožitek ani uživatelské rozhraní.
- Vyšší počáteční náklady. Automatizované testování vyžaduje vyšší počáteční náklady pro pořízení testovacích nástrojů, nastavení prostředí a napsání prvních testů.
- Není vhodné pro rychlou kontrolu. V průběhu vývoje může být pro vývojáře jednodušší provést jednoduchý a krátký manuální test než psát automatizovaný test.

Při praktické realizaci softwarového testování není nutné volit pouze jeden způsob testování a u něj setrvat po celou dobu životnosti projektu. Naopak je vhodné vytěžit co nejvíce z výhod obou způsobů a pro konkrétní test vždy použít ten lepší. Navíc může být výhodné měnit používané způsoby během vývoje projektu, podle toho, v jaké fázi se projekt nachází. Pokud dojde například ke stabilizaci podoby kódu během vývoje, je toto vhodný okamžik pro přehodnocení používaných způsobů testování.

Ve své diplomové práci se zaměřuji na automatizované testování.

1.2 Strategie hledání chyb

V praxi je téměř nemožné najít všechny chyby v programu. Zároveň je nepraktické se o to pokoušet, protože náklady na takové testování jsou zbytečně vysoké a nevyplatí se vzhledem k prostředkům ušetřeným díky bezchybnému programu. Z tohoto důvodu je důležité stanovit testovací strategie, které pomáhají odhalit většinu chyb s co nejmenšími náklady. Dvě z nejčastěji používaných strategií pro hledání chyb jsou *black-box* a *white-box* [1].

Black-box testování, překládané do češtiny jako testování černé skříňky, nazývané také jako testování řízené daty, je jedna ze základních testovacích strategií. Při používání této strategie je potřeba nahlížet na program jako na skříňku, do které není vidět. Cílem je kompletní odstínění od vnitřního chování a struktur, které program používá. Namísto toho se tato strategie zaměřuje na hledání okolností, za kterých se program nechová dle specifikace.

V tomto přístupu jsou testovací data odvozována pouze ze specifikace, nevyužívá se zde znalosti interní struktury programu. Pro nalezení všech chyb v programu by bylo

nutné otestovat všechny možné vstupní kombinace, což je až na triviální programy nemožné.

White-box testování, překládané do češtiny jako testování bílé skříňky, nazývané také jako testování řízené logikou, je druhá ze základních strategií pro testování. Tento přístup dovoluje zkoumat vnitřní strukturu programu. Testovací data jsou odvozována na základě zkoumání vnitřní logiky programu.

Cílem tohoto přístupu je vytvořit paralelu k testování všech možných vstupů v *black-box* přístupu, a to otestovat všechny možné průchody programem. Tím však není zajištěna bezchybnost programu, jako v případě otestování všech možných vstupů. Příkladem může být program na obrázku 1.1 rozhodující o tom, zda jsou od sebe dvě čísla vzdálena méně než určitá mez.

```
(a, b) => {
    if (a - b < 2) {
        console.log('distance less than two');
    } else {
        console.log('distance greather than or equal to two');
    }
}
```

Obrázek 1.1. Program pro kontrolu vzdálenosti dvou čísel obsahující chybu

Při použití vstupů `a=2, b=1` a `a=4, b=1` jsou otestovány všechny průchody testovaného programu. Program přesto obsahuje chybu, protože má být porovnávána absolutní hodnota rozdílu namísto prostého rozdílu. Chybu by odhalil například vstup `a=1, b=4`.

Z tohoto důvodu by se mohlo zdát testování vstupních kombinací vhodnější než testování možných průchodů pro nalezení co největšího množství chyb. Oba tyto způsoby jsou však nevhodné pro praktické testování, protože je potřeba testovat mnoho případů a strategie neříkají nic o tom, jaké testovací případy vybírat. Proto se v praxi používají pokročilejší testovací strategie založené na těchto základních. Kombinací prvků *black-box* a *white-box* testování je možné vytvořit smysluplnou testovací strategii vhodnou pro praktické použití, která využívá co nejmenší množinu testovacích případů, avšak i přesto maximalizuje pravděpodobnost odhalení co nejvíce chyb.

1.3 Úrovně testování

Testování software je možné rozdělit na úrovně podle toho, jaká část softwaru testem prochází. Každá z těchto úrovní využívá jiné metodologie, které řídí proces softwarového testování [4–6].

Unit testing, překládáno do češtiny jako jednotkové testování, je způsob, jakým jsou testovány malé části kódu – jednotky, typicky funkce nebo moduly, samostatně v izolaci. Psaní jednotkových testů bývá jednoduché a přímočaré. Ve své podstatě by takový test měl předat testované funkci vstupní data a ověřit, že je výstup funkce správný. V praxi se může stát, že pokud je kód špatně navržený, nemusí být jednoduché pro něj napsat jednotkový test. Díky tomu je *unit testing* také způsob, jak psát lepší kód, protože dobře testovatelný kód bývá čitelnější a lépe udržovatelný.

Pro izolaci testovaného kódu se používá *mocking*. Jedná se o techniku, pomocí které lze odstínit testovaný kód od jeho externích závislostí. Tuto techniku popisuji později v této kapitole. V jednotkových testech je důležité odstínit všechny asynchronní operace, jako je komunikace po síti nebo vstupně-výstupní operace. Díky

tomu jsou tyto testy velmi rychlé, jednoduché a zajišťují odhalení chyb přímo v implementaci jednotlivých částí softwaru. Pokud selhává jednotkový test, je zpravidla velmi jednoduché chybu odhalit, protože testuje pouze malý kus izolovaného kódu.

Tyto testy jsou základním stavebním kamenem automatizovaného testování, měly by sloužit jako nástroj pro vývojáře, který slouží k odhalení chyb v kódu ještě před jeho odevzdáním. Jednotkový test také výborně demonstruje vlastnosti a způsob použití testovaného kódu, čímž dále pomáhá k pochopení jeho funkce.

Integration testing, překládáno do češtiny jako integrační testování, je způsob, jak otestovat, že spolu různé části softwaru správně fungují. Integrační testy se používají v případech, kdy jednotkové testy nedostačují. Příkladem může být situace, kdy je potřeba otestovat ukládání dat z aplikace do databáze. V takovém případě by integrační test vyvolal kód odpovědný za uložení dat a poté pomocí dotazu do databáze provedl kontrolu jejího stavu.

Integrační testy jsou často pomalejší než jednotkové testy z důvodu vyšší složitosti. Navíc mohou vyžadovat nastavení a konfiguraci, jako například připojení k testovací databázi. Díky tomu je náročnější tyto testy napsat a dále udržovat. Integrační testy by neměly být používány pro účely, pro které by byly vhodnější jednotkové testy [6].

V některých případech může být jednodušší napsat pro složitý kód integrační test namísto testu jednotkového. V tomto případě může být vhodnější testovaný kód přepsat tak, aby se dal lépe testovat pomocí jednotkového testu a integrační test vůbec nepoužít.

Integrační testy mohou být často psány pomocí stejných nástrojů, jako testy jednotkové.

Functional testing, překládáno do češtiny jako funkční testování, je popisováno jako testování kompletní funkčnosti software. Funkční testování se realizuje pomocí nástroje umožňujícího ovládat program stejným způsobem, jako uživatel, tedy skrze uživatelské rozhraní. Jednotkové testy slouží k otestování malých izolovaných částí kódu, integrační testy slouží k testování spolupráce dvou a více částí, funkční testy se nacházejí na nejvyšší úrovni a slouží k testování celého programu jako celku. Vzhledem k tomu, že tyto testy pracují s celou aplikací od *frontendu* po *backend*, bývají také někdy označovány jako *end-to-end* (E2E). Jedná se o formu *black-box* testování, protože testy probíhají podle specifikace softwaru.

Funkčních testů se většinou píše omezené množství, ve srovnání s testy jednotkovými. To je dáno především tím, že funkční testy jsou náročné na napsání a údržbu z důvodu jejich vysoké komplexity. Jejich běh je velmi pomalý ve srovnání s jednotkovými testy, protože testují průchody celou aplikací. Rychlost průchodu aplikací ovlivňují faktory jako je rychlost síťové komunikace, načítání dat, přechodové animace a další. I když k tomu název „funkční“ navádí, není vhodné testovat pomocí funkčního testování jednotlivé funkce, ale spíše běžné uživatelské interakce. Díky tomu je zajištěno otestování důležitých a často používaných průchodů aplikací, příkladem může být registrace nového uživatele.

Jednotkové i integrační testy ověřují správnost programu pomocí porovnání výsledků přímo v kódu. Správnost programu by v případě funkčních testů měla být ověřována stejným způsobem, jakým by ji mohl ověřit uživatel provádějící ten samý test. V případě registrace nového uživatele by se mohlo jednat o kontrolu zobrazení zprávy informující o úspěšné registraci.

1.3.1 Efektivní složení testů

Testy poskytují vývojářům zpětnovazební smyčku, která je informuje, zda software funguje tak, jak má. Ideální smyčka má následující vlastnosti [7]:

- Je rychlá. Vývojáři nechtějí čekat na ověření funkčnosti jejich změny. Pokud změna navíc nefunguje, je potřeba smyčku opakovat. Rychlá smyčka umožňuje rychleji opravovat chyby v kódu.
- Je spolehlivá. Vývojáři nechtějí zbytečně trávit čas hledáním chyby na základě selhávajícího testu, který se nakonec ukáže jako nespolehlivý. Nespolehlivé testy snižují důvěru vývojářů v testy, což může vyústit k ignorování jejich selhání, i když takové testy dokáží ve výsledku objevit v softwaru skutečné problémy.
- Lokalizuje chyby. K opravě chyby je potřeba, aby vývojáři našli konkrétní řádky v kódu, které chybu způsobují. Pokud selhává příliš komplexní test, může být chyba ve velkém množství kódu a jejích hledán je téměř nemožné.

Jednotkové testy splňují vlastnosti ideální zpětnovazební smyčky. Jsou rychlé, protože testují pouze malou izolovanou část kódu. Dokáží být spolehlivé, protože se jedná o jednoduché testy, které neobsahují příliš logiky. Umí lokalizovat chybu, protože jejich selhání je způsobené pouze malou částí kódu, ve které bude třeba chybu hledat.

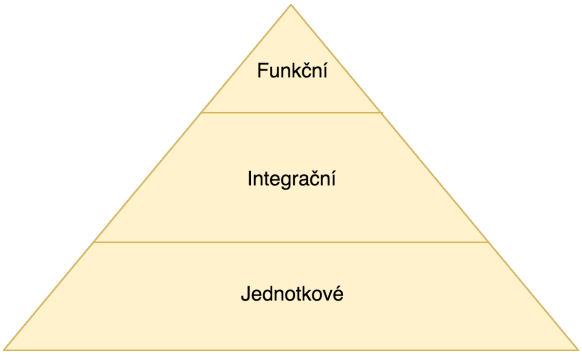
Funkční testy se zdají být nevhodné, protože nesplňují vlastnosti ideální zpětnovazební smyčky. V jejich případě může trvat sestavení produktu, jeho nasazení a spuštění všech funkčních testů velmi dlouho. Tyto testy bývají často nespolehlivé vzhledem k jejich komplexitě. Pokud takový test selže, může být těžké najít příčinu selhání v kódu, protože není chyba dostatečně lokalizována. I přes výše zmíněné nevýhody mají tyto testy v testovacím procesu své místo, protože dokáží simulovat skutečné scénáře, což je pro jednotkové testy vzhledem ke své jednoduchosti a izolovanosti nemožné.

Hlavní nevýhodou jednotkových testů je to, že probíhají v izolaci, což nezaručuje, že testovaný kód bude správně fungovat i jako součást celku. Z tohoto důvodu však není potřeba psát funkční testy. Proto je dobré, aby testovací proces obsahoval integrační testy. Tyto testy pracují s několika komponentami či moduly, často pouze dvěma, a testují jejich chování jako celku, čímž ověřují jejich správnou společnou funkci.

I s jednotkovými a integračními testy je často vhodné, aby testovací proces obsahoval malý počet funkčních testů sloužících k otestování softwaru jako celku. Pro vizualizaci vyváženého poměru mezi typy testů je možné použít testovací pyramidu z obrázku 1.2. Tato pyramida obsahuje tři úrovně testování používané v testovacím procesu [7]. Základ pyramidy obsahuje jednotkové testy, které jsou důležité pro úspěšné testování. Testy jsou se stoupáním v pyramidě složitější, rozsáhlejší a komplexnější, ale zároveň klesá jejich počet, který vyjadřuje šíře pyramidy. Toto pravidlo se dá aplikovat nejen pro tyto tři úrovně testování, ale také obecně pro libovolné testy.

Vývojářům v Googlu [7] se osvědčil poměr testů 70/20/10. Dle nich by se měl testovací proces skládat přibližně z 70 % jednotkových testů, 20 % integračních testů a 10 % funkčních testů. Vhodné zastoupení testů může být pro každý tým odlišné, jejich poměr by se však měl držet tvaru pyramidy. Je doporučeno vyhnout se následujícím tvarům:

- Zmrzlinový kornout má vespodu a uprostřed málo nebo žádné jednotkové a integrační testy. Tým převážně spoléhá na funkční testy, které bývají často prováděny pouze manuálně a integrační nebo jednotkové testy nepíše.



Obrázek 1.2. Testovací pyramida.

- Přesýpací hodiny mají vespodu mnoho jednotkových testů, nahoře mnoho funkčních testů, ale pouze několik málo nebo žádné integrační testy uprostřed. Tato situace nastává tak, že tým začne s mnoha jednotkovými testy a později v průběhu vývoje začne používat testy funkční pro lepší otestování produktu. Používá však často funkční testy v situacích, ve který by bylo vhodnější použít integrační testy.

Podobně jako ve skutečném světě jsou pyramidy velmi stabilní struktury, v testovacím procesu to platí také.

1.4 Mocking

Mocking je technika běžně používaná v automatizovaném testování. Umožňuje v testech použít objekty, které se chovají a vypadají stejně, jako objekty, které se reálně používají v produkci. Tyto objekty se nazývají *test doubles*. Umožňují snížit komplexitu testů a testovaný kód izolovat od jeho závislostí. *Test doubles* objekty jsou nezbytné pro realizaci jednotkového testování. Mezi důvody, proč používat jiné objekty během testovací fáze, patří [8]:

- Izolace závislostí. Díky tomu se zmenší rozsah testovaného kódu a lze lépe lokalizovat případné chyby.
- Rychlost. Skutečný objekt obsahuje pomalé algoritmy nebo náročné výpočty, které není vhodné během testů používat.
- Speciální stavy. Některé stavy nastávají ojediněle, je proto žádoucí je navodit například pro otestování *race conditions*, chyb sítě a dalších.
- Nedeterminismus. Testy by měly být co nejvíce deterministické, což usnadňuje hledání příčiny selhání testu. Z tohoto důvodu není vhodné v testech pracovat například s aktuálním datem a časem.
- Skutečný objekt neexistuje. Někdo jiný na něm zatím pracuje, ale ještě není k dispozici.
- Kontrola nepřímých výstupů. Lze zachytit a ověřit, zda testovaný kód správně předává své výstupy dalším závislostem.

Test doubles je obecný název zahrnující několik kategorií objektů, jejichž účelem je naplnit zmíněné požadavky. Tyto objekty mají stejné rozhraní jako objekty, které zastupují, avšak poskytují pouze omezené chování, které je jinak očekávané od původních objektů. Existují různé kategorizace *test doubles*, nejčastější je však následující dělení [9–10].

Dummy je hloupý objekt, jehož smyslem je pouze projít kontrolami během kompilace a spuštění v běhovém prostředí. Takový objekt se nepoužívá během testů. Důvodem existence těchto objektů je fakt, že některé metody v kódu mohou vyžadovat objekty s určitým rozhraním ve svých argumentech. Pokud však samotný test ani testovaný kód tento objekt nepotřebují, je možné předat pouze *dummy* objekt. *Dummy* objekt může být například prázdná reference, objekt bez implementace nebo literál.

Stub je objekt, který má v sobě předem definovaná data, která používá jako návratové hodnoty, pokud někdo volá jeho metody. Používá se v situacích, kdy není možné nebo je zbytečné používat objekty, které odpovídají skutečnými daty nebo jejichž volání má vedlejší efekt. *Stub* objekty mohou v testech spustit průchody testovaného programu, které by jinak nemohly být otestovány. Příkladem může být situace, ve které kód potřebuje pro svou další činnost načíst data z databáze. Namísto skutečného objektu je možné použít *stub* objekt vracející předem definovaná data.

Fake je objekt, který v sobě obsahuje fungující, avšak zjednodušenou implementaci. Cílem je, aby objekt vykazoval určité skutečné chování dle specifikace, nesplňuje ji však zcela. I když se *fake* objekty vytvářejí pro testy, nejsou používány pro řízení nebo kontrolu testovaného kódu. Používají se nejčastěji v situacích, ve kterých ještě není skutečný objekt k dispozici, je příliš pomalý nebo v testech nemůže být použit, protože způsobuje nežádoucí vedlejší efekty.

Mock je objekt, který sleduje volání svých metod. To umožňuje v testech kontrolovat nepřímé výstupy testovaného kódu. Tento objekt obsahuje připravený kód, který ověřuje správnost volání jeho metod. Kód zároveň popisuje, jaká volání a s jakými daty by měl objekt v konkrétním testu obdržet. *Mock* objekt bývá často zároveň *stub*, čímž umožňuje v testu vracet také předem definované hodnoty. Tento typ objektů se používá v situacích, ve kterých testovaný kód nevrací žádná data a není jednoduché zjistit, zda dochází k jeho správnému provedení. Příkladem může být kód, který v určitých situacích volá objekt představující službu pro odesílání e-mailů. Je nežádoucí odesílat emaily s každým spuštěním testů a zároveň není jednoduché v testu ověřit, zda kód prostřednictvím objektu odesílá správný e-mail. Řešení pomocí *mock* objektu ověří nepřímé výstupy testovaného kódu, tedy že služba odesílající e-maily byla správně zavolána.

Spy je objekt velmi podobný *mock* objektu. Slouží také ke sledování nepřímých výstupů testovaného kódu. Také bývá často zároveň *stub*, aby mohl testovanému kódu předávat návratové hodnoty. *Spy* objekt zaznamenává volání, která přijal, a ukládá je pro pozdější kontrolu v testu. I když se používá pro stejné účely jako *Mock*, styl psaní testů je odlišný, protože jsou kontroly volání prováděny v jiný okamžik a vně objektu.

Přestože existuje několik druhů *test doubles*, vývojáři často mívají sklon používat termín *mock* v obecném slova smyslu pro libovolný objekt s výše popsanými vlastnostmi, což může vést k nedorozumění a případně i chybné implementaci testů [8]. Je proto vhodné znát jejich charakteristiky a dokázat od sebe jednotlivé typy odlišit.

1.5 Test fixtures

Pro efektivní testování je nutné zajistit známé a fixní prostředí, ve kterém testy dávají opakovaně stejné výsledky. Toto prostředí se nazývá *test context*. Termínem *test fixtures* se označuje kód, jehož účelem je takové prostředí připravit. Tento kód může například vytvořit nebo nakonfigurovat *test doubles*, zkopírovat připravené testovací

soubory nebo nahrát připravená testovací data do databáze. Termín *test fixtures* může v přeneseném smyslu také označovat tato předem připravená data [9].

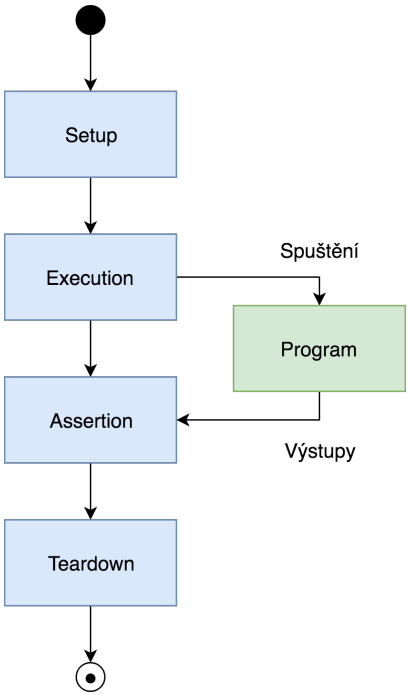
Pro lepší udržovatelnost testů je nutné *fixtures* vhodně strukturovat, pojmenovávat, udržovat je co nejmenší a zamezit jejich duplikaci napříč testy. K tomu také napomáhá vhodná struktura testů.

Test fixtures mohou být umístěné přímo v testu, ve kterém se používají. Díky tomu je vše nutné pro test udržováno na jednom místě. Problém může být zbytečný nárůst opakujícího se kódu, pokud se stejné nebo podobné *fixtures* duplikují ve více testech. Druhou možností je jejich umístění do pomocného modulu. To je zvlášť vhodné v případě velkých *fixtures*, kdy jejich vyčlenění do vlastního modulu udržuje testy malé, zvyšuje jejich čitelnost a umožňuje abstrahovat proces testování od použitých dat. Tento způsob zároveň umožňuje, aby více testů používalo stejné *test fixtures*. To však může být v některých situacích na škodu, protože změna ve *fixtures* nutná pro jeden test může jiné testy rozbít.

Test fixtures mají uplatnění pro všechny zmíněné úrovně testování, pokaždé však v trochu jiné formě. Pro jednotkové testy je jejich úkolem primárně vytváření a konfigurace *test doubles*. Naopak v případě funkčních testů budou převážně nahrávat testovací data do databáze a nastavovat připojení k externím službám.

1.6 Životní cyklus testu

Životní cyklus testu je proces rozdělený na několik navazujících kroků, které jsou společné pro většinu testovacích scénářů. Tento proces se zaměřuje pouze na samotné provádění testů a je součástí většího procesu, který obsahuje kroky jako plánování, analýza, návrh testů, reporting a další [9, 11]. Životní cyklus testu znázorňuje diagram na obrázku 1.3.



Obrázek 1.3. Životní cyklus testu.

Setup je prvním krokem životního cyklu testu. V tomto kroku dochází například k inicializaci testování, alokaci prostředků, spuštění *test fixtures* a dalších. Slouží pro

přípravu samotného testování. Většina testovacích *frameworků* umožňuje definovat *setup* kód, který bude vykonán před všemi nebo pouze některými testy.

Execution je krok, ve kterém probíhá samotné testování. V tomto kroku dochází ke spuštění kódu, který nějakým způsobem interaguje s testovaným programem. Způsob, jakým je realizována interakce záleží na typu testu. Testovaný program může být v případě funkčních testů ovládán skrze uživatelské rozhraní, v případě integračních a jednotkových testů je interakce zajištěna přímo pomocí volání funkcí a metod v kódu testovaného programu.

Assertion je krok, ve kterém dochází k porovnání očekávaných a skutečných výsledků interakce. Způsob, jakým je prováděno porovnání záleží na zvolené testovací úrovni. V případě funkčního testu je potřeba získat stav uživatelského rozhraní a porovnat jej s očekávaným stavem. V případě jednotkových a integračních testů je možné porovnat návratové hodnoty nebo zkontrolovat nepřímé výstupy programu pomocí *test doubles*.

Teardown je posledním krokem životního cyklu testu. V tomto kroku dochází k dealokaci všech prostředků použitých během testu a řešení problémů jako je znehodnocení dat. Dojít k němu může například tak, že jsou v databázi během testu upravena data, čímž dojde ke změně jejího stavu. Tento problém se týká i *test doubles* objektů, pokud jsou opakovaně používány napříč testy. Proto je nutné tyto změny vždy vrátit do původního stavu, aby byla zajištěna izolace testů.

Kapitola 2
Testování integrací client-server

Hlavní náplní této kapitoly je popis existujících způsobů testování integrací client-server, které lze použít v kontextu webových a mobilních aplikací. Pro to, abych mohl tyto způsoby popsat, nejdříve objasňuji pojem integrace client-server a další příbuzné pojmy. V kapitole se také krátce zmiňuji o souvisejícím tématu testování API a testovacích prostředích.

2.1 Integrace client-server

V oboru vývoje softwaru se integrací nazývá spojení dvou a více nezávislých komponent v jeden celek. Tento celek přináší nějakou funkcionalitu, kterou dílčí komponenty samy o sobě nemají [12]. Client-server architektura umožňuje vytvářet distribuované aplikace, které rozdělují odpovědnosti mezi klienta a server. Klient, často aplikace s grafickým uživatelským rozhraním, ke své činnosti využívá služby a prostředky serveru. Integrace client-server je tedy spojení klientské a serverové části do jedné aplikace za účelem poskytovat uživatelům jako celek konkrétní funkce [13].

Pro to, aby klientská aplikace mohla využít služeb a prostředků serveru, poskytuje server API. Application programming interface (API), do češtiny překládané jako rozhraní pro programování aplikací, je specifikace kontraktu mezi dvěma oddělenými částmi software. Díky API je možné propojit dvě a více části software do jednoho funkčního celku [14–15].

Kontrakt API je vždy definován mezi dvěma stranami. Jedna ze stran, nazývána poskytovatel (*provider*) API, nabízí v rámci kontraktu množinu operací. Poskytovatel v kontraktu zároveň definuje pravidla komunikace, jako například požadavky na formát dat či použitý protokol. Druhá ze stran, nazývána konzument (*consumer*) API, může využívat nabízené operace a má zaručeno, že se při dodržení pravidel definovaných poskytovatelem budou tyto operace chovat přesně tak, jak je uvedeno v kontraktu, bez nutnosti znát vnitřní implementaci poskytovatele [16].

V kontextu integrace client-server vystupuje klient v roli konzumenta a server v roli poskytovatele. V případě webových a mobilních aplikací patří mezi nejčastější realizace komunikace volání serveru klientem pomocí HTTP protokolu. Aplikace, které potřebují mezi klientem a serverem zajistit komunikaci v reálném čase, mohou použít například WebSocket protokol.

Výše popsané pojmy jsou klíčové pro správné pochopení toho, co v této práci popisuji. Způsoby testování integrací, na které se v této práci zaměřuji, používají výše zmíněné nejčastější způsoby komunikace klienta a serveru v souvislosti s webovými a mobilními aplikacemi.

2.2 API testing

API testing je proces, který zahrnuje přímé testování API i jeho nepřímé testování jako součást integračních testů za účelem ověřit, zda rozhraní splňuje požadavky na

funkcionalitu, spolehlivost, výkonnost a bezpečnost. Cílem testování API je, mimo jiné, odhalit chyby, nekonzistence v datech nebo odchylky od očekávaného chování v implementaci poskytovatele API [14].

Zajistit výše zmíněné požadavky na API lze i během jeho provozu pomocí kontinuálního testování API. Toto testování probíhá automatizovaně, v pravidelných intervalech mimo hlavní cyklus vývoje software. Tyto testy umožňují monitorovat stav API a zajistit tak rychlou zpětnou vazbu v případě vzniklých problémů [17].

Důležitost testování API je dána především tím, že API je místo, kde bývá implementována podstatná část doménové logiky, dochází zde k získávání a zpracovávání dat a k souběhu různých procesů. Testování API se snaží předejít chybám v API, protože tyto chyby ovlivňují funkčnost dalších částí software, které v rámci integrace API využívají.

Testování API je velmi podobné testování za pomoci grafického uživatelského rozhraní (GUI). V obou případech se jedná o přirozenou cestu, jak otestovat aplikaci ukrytou za rozhraním bez hlubší znalosti její vnitřní implementace. Ve své podstatě se tak jedná o způsob *black-box* testování. Testování pomocí GUI má ve vývoji software stále své místo, avšak v současné době agilního a kontinuálního vývoje roste důležitost testování API. To je dáno především tím, že se API používá pro integraci softwarových komponent, ve které jsou jeho časté změny nežádoucí. Změny API, kterým se nelze vyhnout, bývají pro zajištění zpětné kompatibility nějakým způsobem podchyceny. Tyto důvody dávají testování API lepší předpoklady pro efektivní hledání chyb ve srovnání s testováním skrze GUI [15].

Testování integrací client-server je možné chápat jako generalizaci testování API. V případě testování integrace client-server je cílem zajistit správnou funkčnost klienta a serveru jako celku. Testování API se zaměřuje pouze na správnou funkčnost implementace serveru ukrytého za rozhraním. Při testování API se většinou nebere v potaz klient API, protože se testování řídí pouze specifikací API. Z tohoto důvodu je kontraproduktivní zaměřovat se v API testech na konkrétní klienty a doporučuje se klientskou část zcela abstrahovat.

Protože je testování API velmi blízké testování integrací client-server, lze postupy a nástroje používané pro testování API použít i pro testování client-server integrací. Konkrétní způsoby, které vycházejí z principů používaných pro testování API, popisuji dále v této kapitole. Ve třetí kapitole se zaměřuji na nástroje použitelné pro API testování i testování integrace client-server.

2.3 Sandbox

Sandbox je v kontextu softwarového vývoje testovací prostředí, které slouží k izolaci neotestovaných změn v kódu od produkčního prostředí. Umožňuje tedy spustit testovanou aplikaci a bezpečně provést její otestování [18].

Tento termín je běžně používaný i pro vývoj webových služeb, v tomto případě představuje redundantní prostředí k prostředí produkčnímu. V přeneseném slova smyslu se toto označení používá i pro označení *sandbox* API [19]. Nejedná se o nic jiného, než o API, které je nabízeno webovou službou běžící v *sandbox* prostředí.

Takové prostředí umožňuje vývojářům implementujícím integraci s webovou službou zkoumat její chování či vůči ní testovat svůj kód. Díky tomu, že se jedná o izolované prostředí, jsou redukována rizika spojená s chybným voláním webové služby během vývoje a testování. Sandbox prostředí lze také použít pro simulaci různých chybových stavů, které se mohou během běžného provozu objevit. Díky tomu na

ně lze implementaci konzumenta připravit a otestovat jeho správnou reakci. Pomocí sandbox prostředí je možné simulovat i celou řadu dalších situací, například problém s odezvou serveru.

Sandbox prostředí je důležitý stavební kámen v testech integrací client-server. Umožňuje testovat serverovou implementaci i se svými závislostmi v prostředí, které se velmi blíží reálnému produkčnímu nasazení, avšak bez rizika jeho ovlivnění. Některé způsoby testování integrací client-server, které popisuji v následující sekci, využívají *sandbox* prostředí pro spuštění serverové implementace za účelem testování integrace s klientem.

2.4 Způsoby testování integrací client-server

Pro účely této práce a zjednodušení orientace rozděluji způsoby testování client-server integrací do dvou kategorií. Toto rozdělení dělám na základě toho, zda spolu klient a server komunikují napřímo nebo ne. Způsoby testování ve stejné kategorii mají podobné charakteristiky a sdílí některé výhody a nevýhody. V následujících sekcích toto rozdělení dále popisuji.

2.5 Nepřímé testování integrace

Pod pojem nepřímé testování integrace zařazuji testy, ve kterých se strany interakce, klient a server, netestují v přímé komunikaci. Pokud je komunikace mezi klientem a serverem dobře zadefinována, je možné testování integrace rozdělit na dvě nezávislé části, klientskou a serverovou. Každá část je odděleně testována vůči společné specifikaci. Když obě strany projdou všemi testy, lze předpokládat, že jejich přímá komunikace bude také fungovat.

Kvalita praktické realizace tohoto testování závisí na tom, jak přesná je specifikace API. Ta by měla definovat pro každou odpověď, kterou je možné ze serveru očekávat, množinu předem definovaných požadavků, které může klient zaslat. Forma vedení takové specifikace záleží na použitém způsobu testování. Pro nepřímé testování integrací se používají dvě sady testů, jedna pro testování klientské části a druhá pro testování serverové části.

Tento způsob testování dobře testuje rozhraní mezi klientem a serverem. Umožňuje otestovat všechny možné varianty požadavků, které například přijímá jeden *endpoint*. Dále je možné otestovat správné reakce serveru na neplatné vstupy či chyby, kdy je požadavek po syntaktické stránce chybný. Hlavní výhodou tohoto způsobu testování je, že umožňuje testovat klientskou a serverovou část odděleně. Nepřímé testování je v tomto ohledu více flexibilní, protože na těchto implementacích pracují často rozdílné týmy používající jiné technologie a postupy.

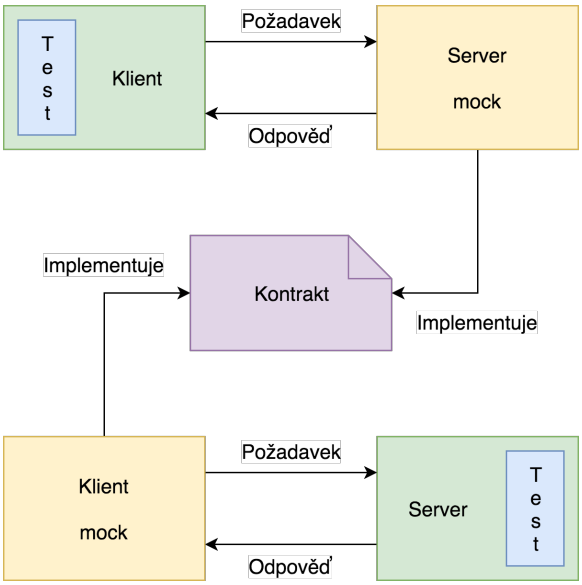
Hlavní nevýhodou tohoto způsobu testování je to, že lze otestovat pouze chyby v komunikaci. Složitější chyby vázané na sémantiku dat zůstávají často neodhaleny. Další nevýhodou je, že existující nástroje se primárně zaměřují na webová API využívající RESTful architektury. Ta využívá bezstavové operace, které se dají dobře testovat, což je jeden z důvodů, proč v současné době neexistují podobné nástroje pro testování komunikace například pomocí WebSocket protokolu.

Implementace těchto testů je vhodná pro malé i velké projekty jako základní stavební kámen testovacího procesu, protože plní podobnou funkci, jako unit testy.

2.5.1 Mocking

Mocking jsou testy, ve kterých nedochází k testování celého produktu, ale dochází ke spouštění malých izolovaných testů mezi klientem a *mock* serverem a mezi *mock* klientem a serverem [20].

V těchto testech jsou klient a server testovány vůči *mocku* svého protějšku, který používá požadavky a odpovědi splňující společný kontrakt. V první sadě testů je spouštěna klientská aplikace tak, aby odesílala požadavky na *mock* server. *Mock* server kontroluje, zda takové požadavky dle kontraktu očekává, a pokud ano, odpovídá pomocí připravených odpovědí splňující náležitosti v kontraktu. V druhé sadě testů odesílá *mock* klient požadavky podle kontraktu na testovaný server. *Mock* klient poté ověřuje, zda odpovědi ze serveru splňují požadavky na odpověď definované v kontraktu. Průběh *mocking* testu demonstruje obrázek 2.1.



Obrázek 2.1. Schéma *mocking* testování

Tento způsob testování lze jako jediný ze způsobů, které v této kapitole popisuji, použít pro testování klienta a serveru jen na jedné straně integrace. Pokud má například tým vyvíjející klientskou aplikaci jiné priority a nechce testovat společnou integraci, je stále možné použít tento způsob testování, bude se však používat jen serverová sada testů. Jedinou podmínkou je, že mezi týmy musí být dostatečně specifikovaný kontrakt.

Existují dva přístupy, jak prakticky realizovat *mocking* testování. Prvním způsobem je zadefinovat mezi týmy co nejpřesněji kontrakt. Každý tým pak provádí tento způsob testování nezávisle ve svém projektu. Pro realizaci se používají testy velmi podobné jednotkovým testům, ve kterých je využíván *mock* protistrany. V případě klienta se jedná o jednoduchý HTTP server, který je předprogramován pro konkrétní test tak, aby přijímal požadavky a vracel odpovědi v souladu se specifikací. Tento server je možné realizovat za využití vhodného nástroje či knihovny. V testech na serverové straně je situace o něco jednodušší. Používá se klientský *mock*, který může být realizován formou jednoduchého HTTP klienta. Pomocí něj pak testovaný server odesílá požadavky sám na sebe. Odpovědi přijaté klientským *mockem* se poté v testech porovnávají vůči specifikaci. Vzhledem k tomu, že tyto testy bývají relativně jednoduché a nevyžadují nutně specializované nástroje, není testování vázáno

na konkrétní technologie. Lze tedy jednoduše testovat například i komunikaci pomocí WebSocket protokolu.

Úspěšnost realizace tohoto přístupu velmi závisí na tom, jak přesně je kontrakt definován. Pokud jej každá strana pochopí jinak, může dojít k situaci, kdy má každý z týmů důkladně otestovanou svou část integrace, ale ve společné integraci se objevují chyby. Pokud však týmy dokáží eliminovat popsanou situaci, jedná se o velmi účinné testování, pomocí kterého je možné otestovat po malých částech všechny interakce mezi klientem a serverem.

Druhým způsobem realizace *mocking* testování je použití kontraktu definovaného ve strojově čitelném formátu. Ten typicky definuje pro každý *endpoint* nároky na požadavek a možné odpovědi. Pro účely tohoto testování je vhodné, pokud specifikace popisuje přímo ukázkové požadavky a k nim příslušné odpovědi, namísto obecných nároků na jejich formát, strukturu a další.

Testování probíhá stejným způsobem, jako v prvním přístupu, rozdíl je v tom, že *mock* server a *mock* klient může být vygenerován přímo z kontraktu. To odstraňuje problém se správnou implementací kontraktu. Nevýhodou tohoto přístupu však je, že se současné formáty pro popis API kontraktů zaměřují na API založené na RESTful architektuře.

Pro každý *endpoint* je možné definovat více požadavků i odpovědí, díky čemuž lze otestovat různé způsoby interakce s jedním *endpointem*. Pokud je však potřeba otestovat i složitější interakce s více *endpointy* skládající se z posloupnosti požadavků a odpovědí, nelze vazbu mezi těmito požadavky a odpověďmi ve formátu pro popis API kontraktu zachytit. Tento problém je možné demonstrovat na příkladu:

1. Požadavek `GET /users/1` – získání uživatele s identifikátorem 1
   Odpověď HTTP 404 Not Found – uživatel neexistuje
2. Požadavek `POST /users` – vytvoření uživatele s identifikátorem 1
   Odpověď HTTP 201 Created – uživatel vytvořen
3. Požadavek `GET /users/1` – získání uživatele s identifikátorem 1
   Odpověď HTTP 200 OK – vrátí vytvořeného uživatele

Ve formátu pro popis kontraktu lze definovat dvě možné odpovědi pro *endpoint* `GET /users/1`, už však nelze určit, která z nich se má v konkrétní situaci použít. Pro účely této interakce by bylo zapotřebí zachytit nějakým způsobem v kontraktu vztah mezi těmito voláními. Vzhledem k tomu, že současné formáty toto neumožňují, nelze pomocí tohoto přístupu testovat dynamické interakce, které mění stav na klientu nebo serveru. Tento přístup je tedy vhodný převážně pro testování syntaxe komunikace.

Předpokladem pro toto testování je fakt, že kontrakt musí být ve strojově zpracovatelném formátu. Vytvoření a především udržování takového kontraktu může být časově náročné. Proto nemusí být tento přístup testování vhodný pro projekty ve fázi prototypování či velmi agilní týmy z důvodu častých změn. Potřeba udržování aktuální specifikace API je ovšem i výhodou, protože taková specifikace funguje zároveň jako dokumentace pro vývojáře. Projekty, které mají rozsáhlejší API, nebo projekty, které nabízejí veřejné API nebo API pro externí partnery, jsou často z těchto důvodů nuceny dokumentaci API v nějaké formě vést. Rozdíl v náročnosti při tvorbě a udržování strojově čitelného kontraktu API použitelného i jako dokumentace API není oproti jinému vedení dokumentace příliš velký. Za tuto cenu je však k projektu vedena specifikace API použitelná i pro další účely, než jen integrační testování.

2.5.2 Consumer-Driven Contracts

Hlavní myšlenkou *consumer-driver contracts* (CDC) je definovat mezi konzumentem a poskytovatelem API kontrakt, pomocí kterého může být konzument a poskytovatel nezávisle otestován [21]. Definice kontraktu je odpovědností konzumenta, proto *consumer-driven*. Konzument v kontraktu definuje množinu požadavků s očekávanými odpovědmi ze serveru. Pomocí toho poskytovatel přesně ví, která volání API jsou od konzumenta požadována a jaké se od serveru očekávají odpovědi. Pomocí tohoto přístupu lze také zjistit, která volání jsou používána, a která mohou být bezpečně odebrána z API.

Vlastnosti dané tímto návrhovým vzorem lze použít pro způsob testování integrací klient-server [16]. Testovací proces se pak skládá z následujících kroků:

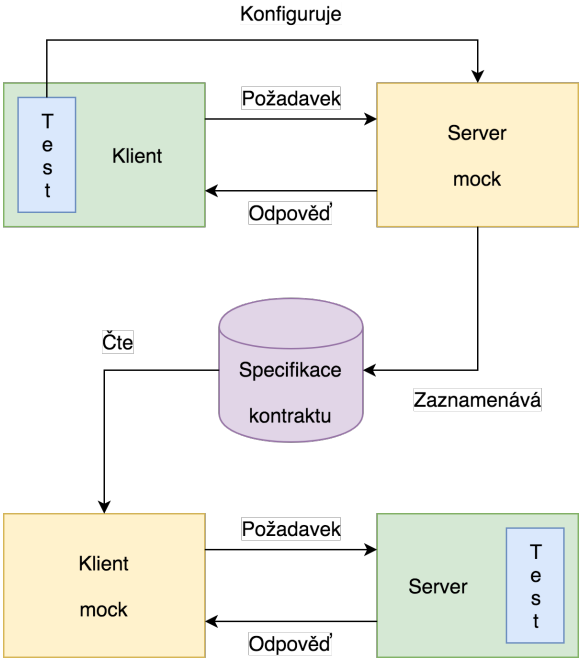
1. Klient vytvoří a udržuje kontrakt.
2. Klient pomocí testů ověří svou funkčnost vůči tomuto kontraktu.
3. Klient publikuje kontrakt.
4. Server pomocí testů ověří svou funkčnost vůči publikovanému kontraktu.

CDC testy jsou svým způsobem podkategorií *mocking* testů popsaných výše. V tomto způsobu testování jsou také dvě sady testů, ve kterých je klient a server testován vůči *mocku* svého protějšku. Důležitá odlišnost je však v tom, jakým způsobem je vytvořen kontrakt. Testy v klientské části používají *mock* server, který je konfigurován zvlášť pro každý test. Pomocí *test fixtures* jsou definovány mezi klientem a *mock* serverem interakce, které budou v testu prováděny. Každá interakce popisuje *endpoint*, který bude volán, a očekávanou odpověď, kterou vrátí *mock* server. Během testů klientské aplikace odesílá testovaná implementace požadavky na *mock* server. Tyto požadavky jsou ukládány do souboru. *Mock* server na ně odpovídá podle předem definovaných interakcí. Po skončení všech testů je veškerá proběhlá interakce zaznamenána v souboru. Tento soubor popisuje pomocí párů ukázkových požadavků a odpovědí kontrakt, který klientská aplikace zadefinovala ve svých testech. Pro otestování serveru je potřeba použít *mock* klienta, který bude číst zaznamenanou interakci ze souboru, jednotlivé požadavky odesílat na testovaný server a vrácené odpovědi porovnávat se zaznamenanými v souboru. Tím je zajištěno velmi přesné otestování interakce mezi serverem a klientem. Průběh CDC testu demonstruje obrázek 2.2.

Pro každou interakci definovanou klientem, je také možné doplnit identifikátor stavu, ve kterém se server musí nacházet před tím, než je na něj odeslán požadavek. Takovým stavem může například být „uživatel Jan Novák je přihlášen.“ Jedná se o mechanismus pro spuštění *test fixtures* na serveru. Testovací nástroj vždy před začátkem interakce volá kód asociovaný s tímto stavem, čímž je možné na serveru připravit například *test doubles* nebo nahrát testovací data do databáze.

Tento způsob testování kombinuje výhody obou způsobů realizace *mocking* testování. Automaticky generovaný a kontrolovaný kontrakt zabraňuje nepřesnostem v interpretaci, které mohou vznikat při prvním způsobu realizace *mocking* testování. Zároveň je možné otestovat složitější interakce s více *endpointy* skládající se z posloupnosti požadavků a odpovědí, které nebylo možné otestovat v druhém způsobu realizace *mocking* testování.

Nevýhodou tohoto způsobu testování je, že se současné nástroje zaměřují na API založené na RESTful architektuře. Další nevýhodou je, že hloubka a kvalita otestování interakce závisí primárně na klientské části. To je dáno tím, že kontrakt je definován pomocí interakcí popsaných v testech klientské aplikace. Vzhledem k tomu, že



Obrázek 2.2. Schéma consumer-driven contracts testování

tým realizující tento způsob testování na serveru pouze připravuje *test fixtures* pro interakce zadefinované týmem z klienta, nemá možnost ovlivnit rozsah a náplň testů.

Tento způsob testování také zjednodušuje vývoj integrace. Pokud například klientský tým potřebuje přidat na serveru *endpoint* z důvodu implementace nové funkčnosti, může tuto funkci vyvíjet nezávisle na serveru, pouze si nadefinuje požadované interakce s takovým *endpointem* v testech. Jakmile je jisté, že se kontrakt pro novou funkčnost již měnit nebude, je možné předat vygenerované specifikace kontraktu týmu vyvíjejícímu server. Tým tento *endpoint* přidá, má jej rovnou automaticky otestovaný a je ověřeno, že se *endpoint* chová tak, jak klientská aplikace očekává.

2.6 Přímé testování integrace

Pod pojem přímé testování integrace zařazuji testy, ve kterých se strany interakce, klient a server, testují v přímé komunikaci. Vzhledem k tomu, že klientská a serverová část spolu v testech komunikují stejným způsobem, jako v produkčním nasazení, lze předpokládat, že pokud v testovaných případech nebudou vykazovat chyby, bude ve stejných situacích zaručena jejich funkčnost i v produkci.

Pro tento druh testování je potřeba zajistit společný běh klientské a serverové aplikace. Testování je řízeno z jednoho místa, které může být zcela mimo klienta i server, nebo součástí jednoho z nich. Pro správnou funkčnost těchto testů je nutné zajistit správný stav na obou stranách. To lze zajistit různými způsoby, ať již tak, že testování bude probíhat z jedné strany a na druhé straně bude správný stav zajištěn ještě před spuštěním testů, nebo tak, že během testů bude docházet k jejich průběžné synchronizaci. V těchto testech se nachází pouze jedna sada testů společná pro obě strany komunikace.

Výhodou přímého způsobu testování integrací oproti nepřímému je, že dokáže lépe testovat integraci klienta a serveru, protože spolu interagují přímo. Je tak zachována sémantika vyměňovaných dat, která nejsou „vytržena z kontextu“. Tyto testy lépe simulují reálné prostředí než testy nepřímé.

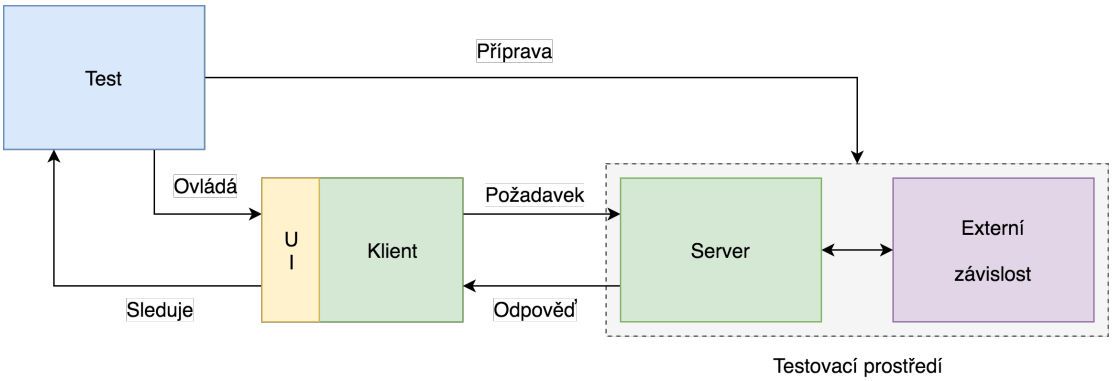
Pro přímé integrační testování je potřeba více komunikace mezi týmy implementujícími klientskou a serverovou aplikaci. Mezi vývojářskými týmy není jen jedna vazba ve formě aplikačního API, ale také další ve formě integračních testů. Oba týmy musí na jejich implementaci i udržování spolupracovat. Z tohoto důvodu jsou testy napsané tímto způsobem také více náchylné k rozbití, protože je ovlivňují změny v obou projektech.

Implementace těchto testů je tak spíše vhodná pro rozsáhlejší projekty jako doplňkový způsob testování k některému z nepřímých způsobů testování.

2.6.1 End-to-end

End-to-end testování je nejpřirozenější způsob, jak testovat rozhraní mezi klientskou aplikací a serverem. V těchto testech dochází k testování klienta a serveru jako celku stejným způsobem, jakým jej používá uživatel, tedy skrze uživatelské rozhraní [22].

Testy jsou realizovány pomocí interakce s klientskou aplikací skrze uživatelské rozhraní. Aplikace volá připravený server běžící v *sandboxu*. Vzhledem k tomu, že server v testech používá i všechny své závislosti, například databázi, musí být i tyto závislosti izolovány. Průběh *end-to-end* testu demonstruje obrázek 2.3.



Obrázek 2.3. Schéma *end-to-end* testování.

Klientské aplikace mají uživatelská rozhraní, která se skládají z hierarchicky členěných elementů. Nad těmito elementy jsou vyvolávány události podle toho, jak s nimi uživatel interaguje. Kód v klientské aplikaci pomocí událostmi řízeného programování tyto události zpracovává. Pro automatizované ovládání uživatelského rozhraní je potřeba nějakým způsobem tyto elementy adresovat, měnit jejich atributy a vyvolávat nad nimi události, které následně budou zpracovávány obslužným kódem v aplikaci. Tímto způsobem je zajištěno spuštění kódu v aplikaci, který volá API serveru. Server požadavek zpracuje a odesílá klientovi odpověď, která se promítne v podobě uživatelského rozhraní. Tyto změny v rozhraní je možné sledovat a ověřit, zda odpovídají tomu, co je očekáváno.

Pro *end-to-end* testování je důležité, aby se server před spuštěním testů nacházel ve známém stavu. To například znamená, že pokud se v testovacím scénáři nachází *endpoint*, pomocí kterého klient vyžádá data pro uživatele s konkrétním identifikátorem, musí server tato data znát, aby je mohl vrátit. Toto chování lze zajistit pomocí *test fixtures* popsaných v předchozí kapitole. Pokud bude v testu například testován server s databází, bude potřeba v tomto případě data vyžadovaná klientem vždy před testy nahrát do databáze.

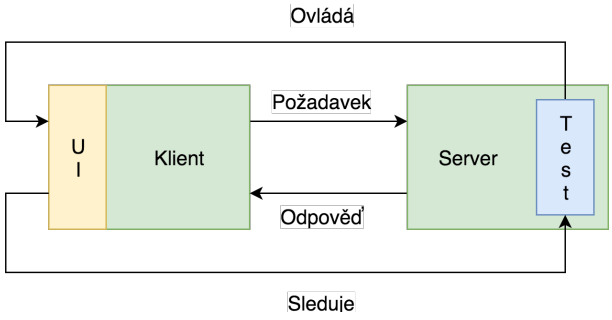
Může se stát, že některé interakce není možné pouze pomocí konfigurace a *test fixtures* otestovat. Jedná se o takové interakce, u nichž závisí i na jiných, často externích

stavech. Problémová může být například situace, ve které je činnost serveru závislá na odpovědi ze služby třetí strany, kterou nepůjde během testů vynutit. Problém může být také to, že se lze dostat do situace, kdy serverová implementace používá nějakou externí službu, která nenabízí oddělené *sandbox* API pro účely testování. Tento způsob testování lze však použít i v těchto případech, pokud během testování nedojde k ovlivnění stavu externí služby, například pokud budou data ze služby pouze čtena. Pokud nelze tyto předpoklady zajistit, není možné *end-to-end* testy použít.

Hlavní výhodou tohoto způsobu testování je fakt, že testováním prochází celý produkt, tedy kompletní integrace klienta, serveru, externích závislostí a služeb třetích stran a to navíc způsobem, který velmi věrohodně simuluje reálné produkční nasazení. Problémem je však složitost takového testování. Vzhledem k tomu, že se testuje několik rozdílných komponent, jsou tyto testy náročné na údržbu. Testy jsou také velmi náchylné na změny v grafickém uživatelském rozhraní. To způsobuje, že s každou změnou rozhraní je nutné upravit i testy, což může být problém v případě prototypování nebo častých redesignů. Vzhledem k nákladům na tvorbu a údržbu tohoto druhu testů je vhodné s jejich tvorbou počkat až do fáze, kdy je celý produkt stabilní a nebude již docházet k častým změnám. Zároveň je nutné zajistit, aby byly veškeré externí zdroje a služby oddělené od těch používaných v produkci. Nevýhodou tohoto způsobu je také nutnost spuštění všech komponent naráz, což může být výkonově i paměťově náročné. Vzhledem k tomu, že testování probíhá napříč několika komponentami skrze uživatelské rozhraní, jedná se o nejpomalejší způsob testování.

Pokud je potřeba testovat stabilní produkt, který nebude procházet masivním vývojem a lze zajistit prostředky a testovací prostředí pro všechny komponenty, může být vhodné použít tento způsob testování. Není příliš praktické ani žádoucí snažit se těmito testy pokrýt veškerou funkcionalitu produktu, je dobré zaměřit se na testování hlavních funkcí a na důležité průchody aplikací. Takové testy jsou pak velmi přínosné, protože zabraňují v případě dalšího vývoje zanesení kritických chyb, které by ovlivnily značnou část uživatelů.

Zajímavou variací *end-to-end* testování je řízení testů, poněkud netradičně, ze serveru. Testy se v tomto případě píší v projektu serveru. Využívá se nástroje umožňujícího ovládání uživatelského rozhraní mobilní či webové aplikace přímo z kódu v testech. Tyto testy při svém spuštění ovládají klientskou aplikaci pomocí uživatelského rozhraní, které následně volá skrze webové API implementaci serveru. Server v testech tak vlastně volá sám sebe, podobně jako při *mocking* testování, avšak ne přímo, ale skrze klientskou aplikaci. Schéma této variace *end-to-end* testování je možné vidět na obrázku 2.4.



Obrázek 2.4. Schéma *end-to-end* testování ze serveru.

Díky tomu, že je v testech možné ovlivnit serverovou implementaci, lze tímto způsobem testování integraci lépe otestovat. Je možné například upravit chování ser-

veru nahrazením externích závislostí za *test doubles* a testovat interakce závisející na vnějších stavech těchto závislostí, které by jinak nebylo možné v běžných *end-to-end* testech provést. Další výhodou tohoto přístupu je, že pomocí *test doubles* externích závislostí je eliminována nutnost mít kompletní serverové testovací prostředí, což zjednodušuje celý testovací proces.

Tato variace *end-to-end* testování odstraňuje některé nevýhody původní formy *end-to-end* testování, avšak za cenu toho, že náplň těchto testů je součástí serverového projektu. Vzniká tak problém, že testy mají silnou vazbu na vnitřní implementaci serveru, ale také na uživatelské rozhraní klientské aplikace, které se serverem vůbec nesouvisí. Problematická může být také koordinace změn v klientské aplikaci. Pokud je klientská a serverová implementace udržována odlišnými týmy, je potřeba změny, které mají dopad na tyto testy, řešit s vývojáři ze serverového týmu.

Tato variace je vhodná spíše pro projekty, ve kterých je klient i server vyvíjen stejnými vývojáři. Komplikace vznikající v případě odděleného vývoje obou částí pravděpodobně nevyváží přínosy tohoto způsobu testování, a bude vhodnější použít původní formu *end-to-end* testů.

2.6.2 Synchronizované testování

Poslední způsob testování, který v práci popisuji, je způsob, který vyplynul z potřeb týmu vývojářů, se kterými pracuji. Při rešerši se mi nepodařilo najít žádné zmínky o alespoň podobném přístupu. Přesto se však jedná o zajímavý způsob testování, který se díky svým výhodám a nevýhodám pro určité případy testování hodí více, než zatím zmíněné.

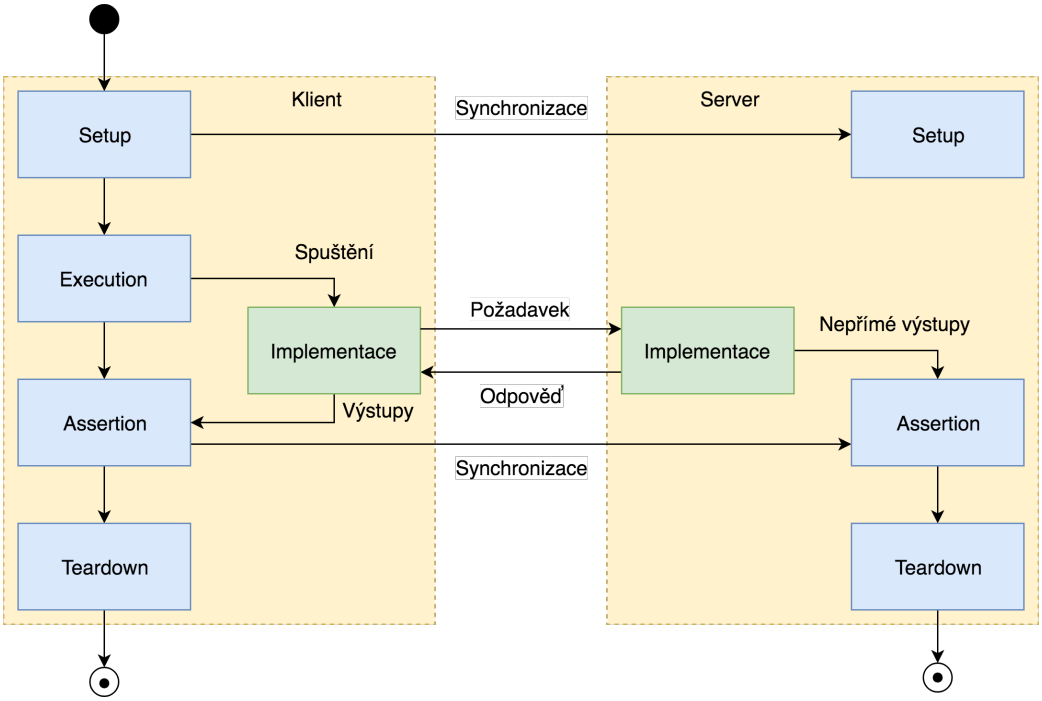
Tento způsob testování má jen jednu sadu testů na klientu, avšak testovací kód není jen na klientu, ale také na serveru. To umožňuje mít plnou kontrolu nad klientskou i serverovou aplikací během testů. Na obou stranách je možné použít *test doubles* a zaměřit se tak na testování integrace pouze určitých částí klienta a serveru. Pro realizaci tohoto způsobu testování je však potřeba zajistit synchronizaci testování mezi klientskou a serverovou částí.

Testovací proces je v případě těchto testů stejný, jako testovací proces popsaný v první kapitole. Důležité však je, že tento proces probíhá napříč klientem a serverem. Celý proces je iniciovaný a řízený z klienta. Kroky *setup*, *assertion* a *teardown* probíhají na klientu i serveru ve stejný okamžik. Krok *execution* probíhá pouze na klientu. V tomto kroku je spuštěna klientská implementace, která díky interakci s API nepřímo spustí také serverovou implementaci. Je nutné podotknout, že v kroku *assertion* na klientu je možné porovnávat přímé i nepřímé výstupy testované implementace, protože je volána přímo v kroku *execution*, avšak na serveru, vzhledem k tomu, že serverová implementace byla spuštěna skrze API, lze porovnávat pouze nepřímé výstupy. Schéma průběhu testu demonstruje obrázek 2.5.

I když se tento způsob testování může na první pohled zdát příliš složitý, není tomu tak, protože o spuštění a synchronizaci jednotlivých kroků se postará *framework*, a na vývojářích tak zůstává napsat samotnou náplň testů, která se příliš neliší například od psaní jednotkových testů.

Pro synchronizaci klienta a serveru v průběhu testovacího procesu je potřeba zajistit nějakou formu komunikace. Pro tyto účely je možné použít libovolnou formu meziprocesové komunikace, například webové API.

Výhodou tohoto způsobu testování je, že je možné mít připravenou běžící instanci testovacího serveru, a kdykoliv bude potřeba otestovat integraci, lze spustit inte-



Obrázek 2.5. Životní cyklus synchronizovaného testu.

grační testy z klienta. Testovací server může při své práci využívat i vývojář klientské aplikace a otestovat tak interakci se serverem v libovolný moment.

Při psaní testů na klientu je potřeba zajistit, aby na serveru existoval kód realizující serverovou část testování tak, jako očekává klient. To klade určité nároky na komunikaci mezi týmy realizující klientskou a serverovou část. Z tohoto důvodu je vhodnější tyto testy realizovat jako scénáře, ve kterých server splňuje nějaké předpoklady a vlastnosti. Ideální komunikace mezi týmy by pak měla vypadat tak, že týmy společně definují scénáře pro integrační testování. V každém scénáři se podrobně zaměří na definování chování serveru. Specifikování testovacích scénářů poté umožní oběma týmům implementovat svou část testů tak, aby splnila očekávání druhé strany, bez zbytečných nejasností a problémů během integračního testování.

Vzhledem k tomu, že je možné během testů ovlivňovat klientskou i serverovou implementaci, lze otestovat libovolné situace, které mohou při integraci v produkci vzniknout. Obě implementace je možné například pomocí *test doubles* odstínit od závislostí a zaměřit se tak na určitou část kódu. Výhodou tohoto přístupu je, že není potřeba spouštět celou klientskou aplikaci nebo celou serverovou infrastrukturu k otestování jejich integrace. Díky tomu jsou tyto testy rychlejší a robustnější než v případě *end-to-end* testování. Další výhodou je, že je možné testovat složitější interakce, například zasílání zpráv po WebSocket protokolu, simulovat chybové stavy na serveru a podobně.

Nevýhoda tohoto způsobu testování je v tom, že vyžaduje určitou úroveň disciplíny ze strany vývojářů. Je potřeba udržovat dokumentaci testovacích scénářů. Problém může být také jiná priorita v týmech, kdy má jeden tým implementovanou nebo upravenou svou část testů, ale druhý tým kvůli jiným prioritám se svou implementací či úpravou otálí. Zde pak vzniká problém, že nové testy nelze spustit, v projektu je tak nepoužívaný kód, a hrozí, že další vývoj tyto testy rozbije a bude ve výsledku nutné testy znovu přepisovat.

2.6.3 Srovnání způsobů testování

V této sekci popíši šest důležitých kritérií testovacího procesu [16] a porovnám s jejich pomocí popsané způsoby testování. Výstupem porovnání je tabulka na konci této sekce.

Složitost. Toto kritérium vyjadřuje, jak náročná je příprava prostředí, ve kterém budou probíhat testy. Například pro *end-to-end* testy je nutné spustit klientskou i serverovou aplikaci, připravit všechny jejich závislosti, vytvořit izolovanou databázi, nahrát do ní testovací data a vše pomocí konfigurace správně propojit. V současné době toto nástroje jako například Docker nebo Kubernetes velmi zjednodušují, avšak stále je potřeba napsat skripty pro automatizované nasazení jednotlivých komponent. Opačným příkladem jsou *mocking* testy, které potřebují pouze část testované komponenty a *mock* pro otestování integrace.

Izolace. V první kapitole popisuji efektivní složení testů a testovací pyramidu, kterou chci nyní použít pro popsaní kritéria na izolaci testů. Ve spodní části této pyramidy se nacházejí izolované jednotkové testy, výše v pyramidě jsou testy méně a méně izolované, až v samotném vrcholu pyramidy jsou testy zcela bez izolace, testující celý produkt. Čím níže je způsob testování v pyramidě, tedy čím více jsou testy izolovány, tím více pozitivních vlastností popsaných v první kapitole tento způsob má. *End-to-end* testy jsou ve vrcholu pyramidy, zcela bez izolace. Naopak *mocking* a CDC testy mohou být velmi izolované, proto spadají spíše do spodnější části pyramidy.

Stabilita. Vzhledem ke komplexitě, potenciálním chybám v testovacích datech a dalším faktorům mohou *end-to-end* testy selhat. Pokud takové testy selhávají, neznamená to nutně chybu v kódu. Může se například stát, že testovací prostředí bylo špatně nakonfigurováno, že je dočasně nedostupná externí služba, nebo že byla nějaká komponenta nasazena ve špatné verzi. To znamená, že *end-to-end* testy jsou méně stabilní, než lépe izolované testy, například *mocking* testy. Toto kritérium tedy vyjadřuje, jak často testy selhávají z jiných důvodů, než je chyba v kódu, který testují.

Rychlost zpětné vazby. Důležitým kritériem testů je čas, který je potřeba pro spuštění testů, získání jejich výsledků a zásahu do kódu opravující chybu nebo upravující test. Čím kratší čas, tím více jsou vývojáři produktivní. V případě *end-to-end* testů trvá tato zpětná vazba dlouho, protože nějaký čas zabere připravení kompletního testovacího prostředí a spuštění testů. *Mocking* testy mají mnohem rychlejší zpětnou vazbu, protože je lze spustit kdykoliv, často přímo z počítače vývojáře, a tím získat výsledky rychle. Oprava chyby nalezené v *mocking* testu je oproti například *end-to-end* testu rychlejší, protože je nutné chybu hledat pouze v malém množství kódu.

Test fixtures. Fixní prostředí pro testy je nutné řešit při realizování libovolného způsobu testování. V případě *end-to-end* testů je toto však obzvláště náročné, protože je zde potenciálně velké množství komponent obsahujících nějaký vnitřní stav. Každá z těchto komponent by se měla nacházet ve známém stavu, který bude zajištěn pomocí *test fixtures*, což nemusí být vždy jednoduché. Pokud toto není pro některé komponenty explicitně zajištěno, ovlivní to s nejvyšší pravděpodobností stabilitu takových testů. V případě *mocking* testů je situace o poznání jednodušší, protože stačí zadefinovat pouze *mock* protistrany a izolovat test od dalších závislostí pomocí *test doubles*. Toto kritérium tedy vyjadřuje náročnost zajištění fixního prostředí pro testy.

Testování sémantiky. Správný význam dat vyměňovaných mezi klientem a serverem je důležitý pro jejich další zpracování. *Mocking* testy však většinou kontrolují pouze syntaxi dat, například, že zadaná adresa je ve správném formátu, avšak už nekontrolují i to, že je možné na tuto adresu odeslat zásilku. Sémantika dat může být

testována například pomocí *end-to-end* testů, protože běží v kompletním prostředí zahrnujícím i doménovou logiku, která dokáže rozhodnout, na které adresy je možné zásilky odesílat. Redukování rozsahu testů degraduje testování významu dat pouze na testování jejich formy. Toto by mělo být vědomé rozhodnutí, protože již není zajištěno testování doménové logiky, ale testování se více zaměřuje na komunikaci mezi serverem a klientem. Testování doménové logiky by však nemělo být opomíjeno a mělo by být otestováno odděleně pomocí jiné formy testů.

Tabulka 2.1 srovnává jednotlivé způsoby testování integrace client-server podle kritérií popsaných výše. Z tabulky je patrné, že neexistuje způsob testování, který by byl použitelný pro všechny situace. Z tohoto důvodu je žádoucí tyto způsoby testování vhodně kombinovat. Každý projekt je jiný, a proto je důležité u každého projektu vždy individuálně zhodnotit nejvhodnější strategii testovacího procesu.

	Nepřímé		Přímé	
	Mocking	CDC	Synchronizované	End-to-end
Složitost	nízká	nízká	střední	vysoká
Izolace	vysoká	vysoká	střední	žádná
Stabilita	vysoká	vysoká	vysoká	nízká
Rychlost zpětné vazby	vysoká	vysoká	střední	nízká
Test fixtures	jednoduché	jednoduché	jednoduché	složité
Testování sémantiky	není	není	částečné	úplné

Tabulka 2.1. Srovnání způsobů testování client-server integrací.

Kapitola 3
Nástroje pro testování integrací client-server

V této kapitole představuji nástroje, s jejichž pomocí lze realizovat způsoby testování integrací client-server z předchozí kapitoly. Každý nástroj důkladně popisuji, zaměřuji se na jeho klíčové vlastnosti pro konkrétní způsob integračního testování, předvádím způsob použití nástroje na jednoduchém příkladě, a nakonec shrnuji jeho výhody a nevýhody.

Nástroje, které v této kapitole popisuji, se zaměřují na testování klienta ve formě webové aplikace napsané v JavaScriptu a webového serveru využívajícího Node.js. Některé ze zde uvedených nástrojů je však možné použít i pro testování klienta a serveru založených na jiných technologiích.

3.1 Mocking

V této sekci popisuji nástroje, se kterými je možné realizovat *mocking* testování integrací client-server. Jak jsem již zmínil v předchozí kapitole, tento způsob testování je možné realizovat dvěma způsoby.

V prvním způsobu vývojáři v testech vytvářejí a používají *mock* protistrany. Tento *mock* je vždy vytvářen pro účely konkrétního testu a měl by se chovat v souladu se specifikací API. Tyto testy se více zaměřují na menší části integrace, které je možné díky *test doubles* snadno izolovat a spravovat.

V testech klientské aplikace je možné pro vytvoření *mock* serveru použít nástroje Fetch-mock nebo Mockttp. Oba tyto nástroje umožňují pomocí pravidel popsat, na které požadavky má *mock* server vracet předdefinované odpovědi. Přijetí odpovědi klientem a její zpracování testuje pouze jeden směr interakce, proto je důležité, že tyto nástroje umožňují přijatý požadavek zkontrolovat vůči očekávanému požadavku, a testovat tak i opačný směr interakce.

Pro otestování serverové části integrace je potřeba použít HTTP klienta umožňujícího volat serverové API. Důležité je, aby takový klient uměl jednoduše definovat validační pravidla pro odpovědi. Mezi nástroje nabízející takové klienty patří SuperTest a Frisby.

Druhý způsob *mocking* testování využívá k testování integrace specifikaci API ve strojově čitelném formátu. Tyto formáty se zaměřují na API využívající RESTful architektury a umožňují definovat chování jednotlivých *endpointů*. Mezi nejznámější formáty patří API Blueprint a OpenAPI. Každý z těchto formátů je součástí ekosystému nástrojů a služeb, které s tím to formátem pracují. Proto se nezaměřím pouze na popis formátů, ale zmíním také služby spojené s těmito formáty použitelné pro testování integrací client-server.

Dále pak popíši nástroje, které umožňují podle specifikace API napsané v jednom z těchto formátů vytvořit *mock* klienta nebo serveru použitelného pro účely testování integrací client-server. Mezi nástroje vytvářející *mock* serveru pro použití v testech na klientu patří webová služba Apiary. Pro testy na serveru lze použít nástroj Dredd, který umožňuje testovat podle specifikace API serverovou implementaci.

Nyní se zaměřím na detailnější popis zmíněných nástrojů a formátů i s ukázkami použití pro testování client-server integrací.

3.1.1 Fetch-mock

Fetch-mock [23] je nástroj umožňující *mocking* HTTP požadavků prováděných pomocí Fetch API [24] dostupného v prohlížečích nebo ve formě knihoven implementujících tuto funkcionalitu. Fetch-mock funguje tak, že nahrazuje globální implementaci `fetch` funkce. To znamená, že v *mocking* testech při použití tohoto nástroje nedochází k žádnému HTTP spojení. *Mocking* serveru tak probíhá už na úrovni kódu. Fetch-mock lze použít v testech běžících přímo v prohlížeči i v Node.js.

Pro vrácení odpovědi z *mocku* je potřeba vytvořit podmínku, která rozhodne o tom, které požadavky budou odchyceny, a zároveň definovat odpověď, která bude na tyto požadavky vrácena. Všechna volání *mocku* tento nástroj zaznamenává a seskupuje je podle URL a HTTP metody. Pro získání zachycených požadavků nabízí rozhraní umožňující filtrovat požadavky podle různých kritérií. Díky tomu je možné po zavolání testovaného kódu zkontrolovat požadavky, které tento kód uskutečnil.

Vytvoření *mocku* serveru pro test přihlášení uživatele demonstruji na obrázku 3.1. Výhodou tohoto nástroje je velmi dobrá konfigurovatelnost. Fetch-mock má rozhraní, které podporuje několik způsobů vytváření podmínek zajišťujících odchytávání požadavků a filtrování zachycených požadavků pro jejich kontrolu.

```
1  const authenticator = require('./authenticator');
2  const config = require('./config');
3  const fetchMock = require('fetch-mock');
4
5  describe('authenticator', () => {
6    it('successfully logs user in', async () => {
7      fetchMock.mock(`${config.baseUrl}/login`, {
8        status: 200,
9        body: {
10         id: 42,
11         name: 'Tomáš Markacz',
12       },
13     });
14
15     // performs POST /login with given email and password and returns user info
16     const user = await authenticator.login('tomas@markacz.com', 'foobar');
17
18     expect(user).toEqual({
19       id: 42,
20       name: 'Tomáš Markacz',
21     });
22     expect(fetchMock.lastOptions(`${config.baseUrl}/login`)).toEqual({
23       method: 'POST',
24       headers: {
25         'Content-Type': 'application/json',
26       },
27       body: JSON.stringify({
28         email: 'tomas@markacz.com',
29         password: 'foobar',
30       }),
31     });
32   });
33 });
```

Obrázek 3.1. Jednoduchý klientský test používající nástroj Fetch-mock.

Nevýhodou tohoto nástroje je, že klientská implementace musí používat Fetch API, jinak pomocí Fetch-mock nelze odchytávat požadavky.

3.1.2 Mockttp

Mockttp [25] je nástroj sloužící k vytvoření serverového *mocku*. Jedná se o plnohodnotný server poslouchající na určitém portu. V testech tedy dochází, na rozdíl od předešlého nástroje, k HTTP komunikaci. Mockttp se skládá ze dvou částí, z Mockttp serveru, který přijímá požadavky a vrací odpovědi, a Mockttp klienta, který slouží ke konfiguraci serveru. Přímo v testech se používá jen Mockttp klient, pomocí kterého se před voláním testované implementace konfiguruje Mockttp server tak, aby na požadavky z testovaného kódu vracel předdefinované odpovědi. Mockttp server všechna volání zaznamenává, a proto je po zavolání testované implementace možné pomocí Mockttp klienta zkontrolovat, jaká volání byla provedena.

Pro testování přímo v prohlížeči je potřeba zajistit, aby byl Mockttp server před zahájením testů spuštěn. Při testování v Node.js není třeba spuštění serveru řešit, Mockttp jej automaticky nastartuje v aktuálním procesu. Tento nástroj má podporu i pro komunikaci přes HTTPS protokol. Je však nutné vytvořit certifikát a nastavit k němu cestu pomocí Mockttp klienta.

Mockttp umožňuje vytvářet pravidla pro zachycení požadavků pomocí *fluent interface*. Tímto způsobem vytvořená pravidla se vždy váží ke konkrétní URL adrese a HTTP metodě. Pro každé pravidlo je pak možné definovat odpověď, kterou Mockttp server vrátí. Vytvořené dvojice pravidlo–odpověď jsou předávány Mockttp serveru, který s jejich pomocí interaguje s testovanou implementací. Vzhledem k tomu, že nástroj komunikuje s Mockttp serverem, vracejí jeho metody instance `Promise`.

Vytvoření *mocku* serveru pro test přihlášení uživatele demonstruje obrázek 3.2. Hlavní výhodou tohoto nástroje je, že umožňuje lépe otestovat komunikaci, protože v testovaném kódu dochází k HTTP komunikaci, nástroj také volitelně podporuje HTTPS protokol. Použití samostatného *mock* serveru neklade žádné požadavky na způsob realizace volání z klientské aplikace. Mockttp ve svém klientském rozhraní poskytuje metody pro časté úkony, například práci s JSON tělem požadavku.

```
1  const authenticator = require('./authenticator');
2  const mockttp = require('mockttp').getLocal(); // server within current process
3
4  describe('authenticator', () => {
5    beforeEach(() => mockttp.start(8000)); // port to listen to
6    afterEach(() => mockttp.stop());
7
8    it('successfully logs user in', async () => {
9      const loginMock = await mockttp.post('/login').thenJSON(200, {
10       id: 42,
11       name: 'Tomáš Markacz',
12     });
13
14     // performs POST /login with given email and password and returns user info
15     const user = await authenticator.login('tomas@markacz.com', 'foobar');
16
17     expect(user).toEqual({
18       id: 42,
19       name: 'Tomáš Markacz',
20     });
21     const requests = await loginMock.getSeenRequests();
22     expect(requests).toHaveLength(1);
23     expect(requests[0].body.json).toEqual({
24       email: 'tomas@markacz.com',
25       password: 'foobar',
26     });
27   });
28 });
```

Obrázek 3.2. Jednoduchý klientský test používající nástroj Mockttp.

Nevýhodou tohoto nástroje je složitější způsob fungování daný tím, že používá samostatný *mock* server.

3.1.3 SuperTest

SuperTest [26] je nástroj umožňující odesílání HTTP požadavků s kontrolou přijatých odpovědí. SuperTest z části staví na knihovně SuperAgent, což je populární HTTP klient pro prohlížeče i Node.js využívající *fluent interface*. SuperTest rozhraní tohoto klienta rozšiřuje o další metody umožňující kontrolu přijatých odpovědí.

Instanci SuperTest klienta je možné vytvořit pomocí funkce, která obalí instanci `http.Server` nebo libovolnou funkci přijímající instance `http.IncomingMessage` a `http.ServerResponse`, což může být například i Express nebo Koa aplikace. SuperTest instanci serveru automaticky přiřadí port, pomocí kterého s ní poté komunikuje. Není tak potřeba v testech specifikovat *hostname* ani port.

SuperTest umožňuje pomocí stejného *fluent interface*, jako v knihovně SuperAgent, interagovat s testovaným serverem. SuperTest k tomuto rozhraní přidává metody umožňující kontrolu stavového kódu, hlaviček a těla odpovědi. Pro složitější kontroly je možné předat *callback*, který je volán s instancí odpovědi.

Na obrázku 3.3 je možné vidět ukázkový test využívající nástroj SuperTest k otestování *endpointu* sloužícího pro přihlášení uživatele. Nejdříve je pomocí SuperTest funkce obalena instance aplikace, poté je na *endpoint* pro přihlášení odeslán požadavek, a následně je v tentýž zřetězeném volání vrácená odpověď zkontrolována.

```
1  const app = require('./app'); // express or koa application
2  const supertest = require('supertest');
3
4  describe('login endpoint', () => {
5    it('successfully logs user in', async () => {
6      await supertest(app)
7        .post('/login')
8        .send({
9          email: 'tomas@markacz.com',
10         password: 'foobar',
11       })
12       .expect(200, {
13         id: 42,
14         name: 'Tomáš Markacz',
15       });
16   });
17 });
```

Obrázek 3.3. Jednoduchý serverový test používající nástroj SuperTest.

Výhodou tohoto nástroje je jeho jednoduchost. Tím, že staví na populární knihovně SuperAgent, je pro vývojáře jednodušší jej v testech použít, protože pro odesílání požadavků nabízí stejné rozhraní. Navíc přidává pouze dvě metody sloužící ke kontrole odpovědí. Celé rozhraní je ve formě *fluent interface*, což umožňuje psát přehledné testy.

Nevýhodou však je, že projekt, ač vyzrálý a funkční, není dlouhodobě spravován. Poslední *commit* v oficiálním repozitáři je z ledna 2017. To však nemění nic na tom, že se jedná o stabilní a používaný nástroj pro testování HTTP komunikace – ke dni psaní tohoto textu má SuperTest přes 360 tisíc stažení za týden.

3.1.4 Frisby

Frisby [27] je Node.js nástroj pro testování webových API postavený nad *frameworkem* Jasmine. Tento nástroj obsahuje pro nejčastější operace prováděné nad vráce-

nou odpovědí předpřipravené funkce, které je možné řetězit pomocí *fluent interface*. Frisby umožňuje, stejně jako nástroj SuperTest, odeslat testovací HTTP požadavek a následně zkontrolovat přijatou odpověď. Frisby však na rozdíl od SuperTestu pracuje s celou URL požadavku, není tedy vázán na konkrétní instanci serveru či aplikace.

Nástroj nabízí základní funkce, pomocí kterých lze kontrolovat stavový kód, hlavičky i konkrétní položky v JSON odpovědi. Frisby je možné rozšířit o další takové funkce, což umožňuje tento nástroj specializovat na určitý způsob testování. Pokud je potřeba provést složitější kontrolu odpovědi, je možné předat *callback*, který bude zavolán s instancí odpovědi.

Na obrázku 3.4 je ukázkový kód používající nástroj Frisby pro otestování *endpointu* sloužícího k přihlášení uživatele. Nejdříve je aplikace spuštěna na některém z volných portů. Poté je na *endpoint* pro přihlášení odeslán požadavek a přijatá odpověď je pomocí zabudovaných funkcí `status` a `json` zkontrolována. Případné rozšiřující funkce se volají stejným způsobem.

```
1  const app = require('./app');
2  const config = require('./config');
3  const frisby = require('frisby');
4
5  beforeAll(() => app.listen(config.port));
6  afterAll(() => app.close());
7
8  describe('login endpoint', () => {
9    it('successfully logs user in', async () => {
10     await frisby
11       .post(`${config.baseUrl}/login`, {
12         email: 'tomas@markacz.com',
13         password: 'foobar',
14       })
15       .expect('status', 200)
16       .expect('json', {
17         id: 42,
18         name: 'Tomáš Markacz',
19       });
20   });
21 });
```

Obrázek 3.4. Jednoduchý serverový test používající nástroj Frisby.

Frisby je nástroj převážně pro testování API postavených na RESTful architektuře, nenabízí proto tolik možností jako SuperTest, který staví na obecně zaměřeném klientu SuperAgent. Frisby například nemá přímou podporu pro požadavky s `multipart/form-data` obsahem.

Výhodou tohoto nástroje je možnost rozšíření o vlastní funkce pro kontrolu odpovědí. Testy interakce se serverem pomocí Frisby jsou psány ve formě jednoho zřetězeného volání funkcí, což napomáhá čitelnosti kódu. Tento nástroj je na rozdíl od SuperTestu aktivně vyvíjen a udržován, je proto lepší jej použít u projektů s delší dobou životnosti.

3.1.5 API Blueprint

API Blueprint [28] je jazyk sloužící k dokumentaci webových API. Vznikl v roce 2013 v kancelářích společnosti Apiary. Pro zápis využívá Markdown syntaxe, výsledným dokumentem je strojově zpracovatelná dokumentace API. API Blueprint je open-source formát šířený pod MIT licencí. Vývoj jazyka probíhá pomocí návrhů ve formě RFC dokumentů.

API Blueprint používá elementy z Markdownu pro popis jednotlivých částí API. Základní dokument napsaný v API Blueprint využívá nadpisy pro členění dokumentace na skupiny, *resources* a akce, a odrážkové seznamy pro popis jednotlivých požadavků, odpovědí i datových struktur.

Jazyk umožňuje vyčlenit datové struktury používané napříč API na jedno místo v dokumentaci. Tato vlastnost velmi zjednodušuje údržbu dokumentace rozsáhlejších API. Pro popis datových struktur je možné použít MSON (Markdown Syntax for Object Notation). Pomocí této notace lze popisovat datové struktury v Markdown syntaxi nezávisle na výstupním formátu (JSON či XML). Umožňuje popsat zároveň strukturu objektu i jeho ukázková data.

Příklad definice datových struktur demonstruji na obrázku 3.5. V definici `Author` je popsána jak struktura, kterou je možné převést na JSON Schema [29] uvedené na obrázku 3.6, tak ukázková data, která jsou na obrázku 3.7. Definice *endpointu* v API Blueprint, pomocí kterého lze získat všechny příspěvky na blogu, je ukázána na obrázku 3.8. Tato definice používá dříve definované datové struktury `Blog Post` a `Author`.

```
# Data Structures

## Blog Post (object)
+ id: 42 (number, required)
+ text: Hello World (string)
+ author (Author) - Author of the blog post.

## Author (object)
+ name: Boba Fett (string, required) - Name of the author.
+ email: fett@intergalactic.com (string, required)
```

Obrázek 3.5. Ukázka definice datových struktur za pomoci MSON notace.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "name": {
      "description": "Name of the author",
      "type": "string"
    },
    "email": {
      "type": "string"
    }
  },
  "required": ["name", "email"]
}
```

Obrázek 3.6. JSON Schema vygenerované z MSON objektu `Author`.

```
{
  "name": "Boba Fett",
  "email": "fett@intergalactic.com"
}
```

Obrázek 3.7. Ukázkový JSON získaný z MSON objektu `Author`.

Výhodou tohoto jazyka je, že se jedná o velmi přehledný formát, což je dané použitou Markdown syntaxí. Popsání *endpointu*, schémat a ukázkových dat je při použití MSON notace velmi efektivní.

```
# Blog Posts [/posts]

## Retrieve All Posts [GET]
+ Response 200 (application/json)
    + Attributes (array[Blog Post])
```

Obrázek 3.8. Definice *endpointu* s použitím datové struktury v odpovědi.

Nevýhodou je ovšem nutná detailní znalost struktury API Blueprint dokumentu. Struktura dokumentu není triviální a bez její znalosti nelze napsat validní API Blueprint dokument.

Mezi další výhody API Blueprint patří podpora mnoha open-source nástrojů. Na webových stránkách API Blueprint lze nalézt seznam čítající přes 70 nástrojů. Jedná se o nástroje generující HTML výstupy pro dokumentační účely, pluginy do nejpoužívanějších editorů, parsery a lexery jazyka, nástroje pro testování API a spouštění *mock* serverů nebo konvertory z a do jiných formátů.

3.1.6 OpenAPI

OpenAPI [30] vzniklo v roce 2016 odštěpením od *frameworku* Swagger společnosti SmartBear software. OpenAPI definuje standard pro popis webových API. Původně vychází ze Swagger specifikace, jeho další vývoj nyní řídí konsorcium OpenAPI Initiative spadající pod The Linux Foundation. Společnost SmartBear nadále vyvíjí své nástroje pod značkou Swagger jako ekosystém nástrojů založených na tomto formátu.

OpenAPI dokument může být reprezentován v JSON nebo YAML formátu. Na rozdíl od API Blueprint je OpenAPI více orientované na specifikaci API, než na jeho dokumentaci.

Dokument je hierarchicky členěn dle jednotlivých *endpointů*. Definice *endpointu* obsahuje popis a příslušné požadavky s odpověďmi, jejichž strukturu je možné v dokumentu vyčlenit do zvláštní sekce `components`. Pro popis struktury používá OpenAPI JSON Schema [29]. Na schéma je možné odkazovat z libovolného místa v dokumentu pomocí JSON Reference [31].

OpenAPI umožňuje, stejně jako API Blueprint, definovat pro každý *endpoint* ukázková data. Tato data pak mohou použít testovací nástroje pro odesílání požadavku testované implementaci nebo při kontrole přijaté odpovědi.

Příklad definice schématu odpovědi demonstruji na obrázku 3.9. Na takto vytvořenou definici lze pak uvnitř i vně dokumentu odkázat. Definici *endpointu* sloužícího k získání dat ze serveru ukazuji na obrázku 3.10. URL *endpointu* obsahuje parametr `id`, který je také validován pomocí JSON schématu. V definici odpovědi pro úspěšné volání (stavový kód HTTP 200 – OK) je pomocí reference odkázáno na schéma definované dříve. Zároveň jsou v odpovědi také definována ukázková data.

Vzhledem k tomu, že se OpenAPI zaměřuje na specifikaci API, má díky tomu více prostředků pro jejich popis. Výhodou tohoto formátu oproti API Blueprint je jednodušší syntaxe, což přispívá k rychlejší tvorbě takové specifikace, a zároveň k jednoduššímu parsování, ovšem za cenu toho, že výsledný dokument je obsáhlejší a méně čitelný než srovnatelný dokument v API Blueprint.

Specifikace OpenAPI je aktivně vyvíjena, ke dni psaní této práce se nachází ve třetí verzi. Jedná se pravděpodobně o nejrozšířenější standard pro popis webových API založených na RESTful architektuře, což je dané také tím, že součástí OpenAPI Initiative jsou velcí hráči na trhu, jako například Google, Microsoft, Adobe nebo třeba IBM. OpenAPI je podporováno mnoha nástroji a službami, což z něj dělá všestranný

```yaml
components:
  schemas:
    Pet:
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
```

Obrázek 3.9. Definice schématu dle OpenAPI.

```yaml
paths:
  /pets/{id}:
    get:
      description: Returns a pet based on a single ID
      operationId: find pet by id
      parameters:
        - name: id
          in: path
          description: ID of pet to fetch
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: Pet response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
              example:
                id: 42
                name: Joseph
```

Obrázek 3.10. Definice *endpointu* s využitím schématu a ukázkových dat v odpovědi.

formát nejen pro specifikaci API, ale také pro tvorbu jeho dokumentace, validaci požadavků a odpovědí přímo v aplikaci, nebo testování.

3.1.7 Apiary

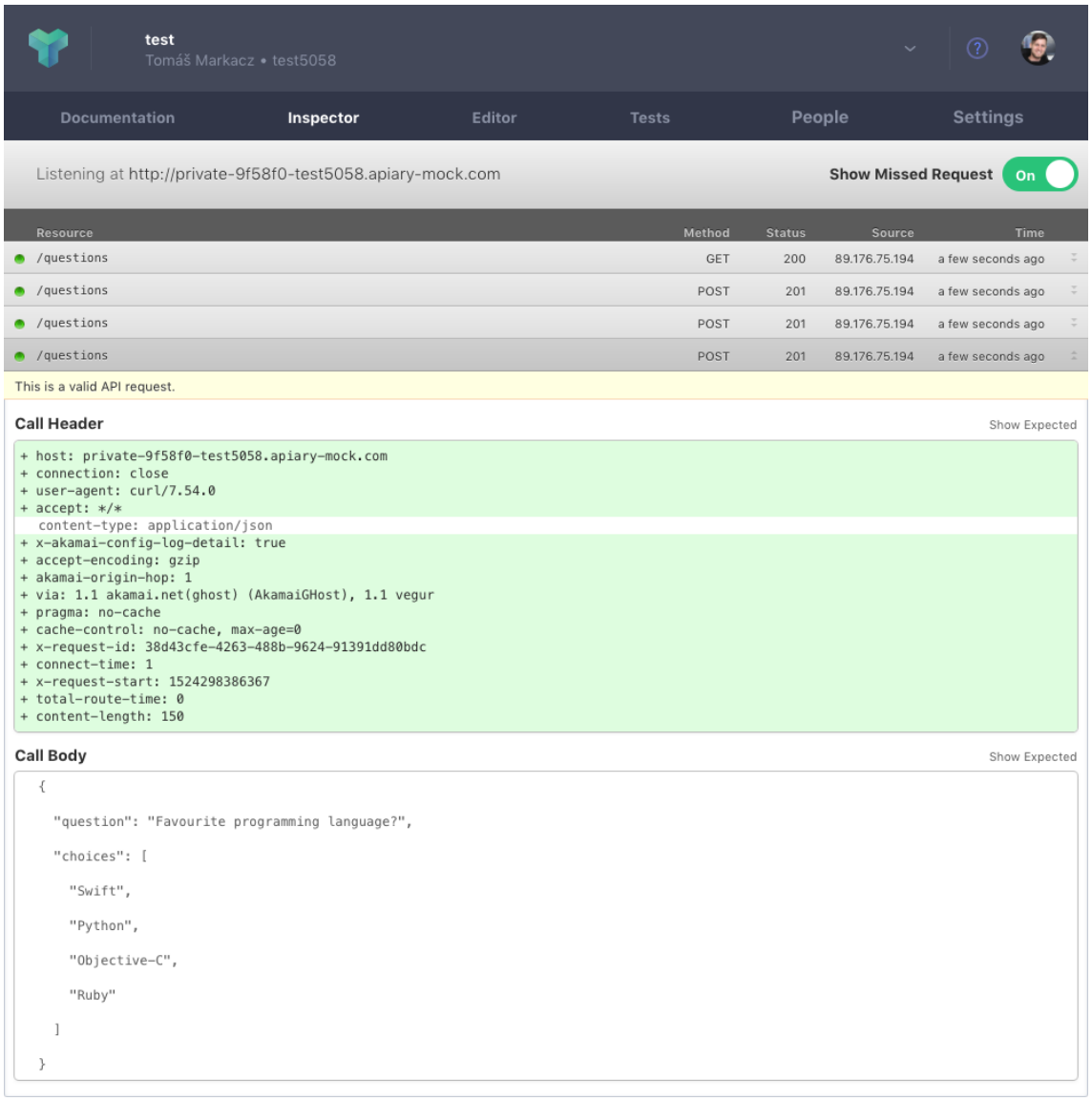
Apiary [32] je webová platforma pro návrh, vývoj a dokumentaci API. Apiary poskytuje webové rozhraní pro editaci API Blueprint dokumentů přímo v prohlížeči. Díky tomu je umožněna spolupráce více uživatelů v rámci projektu. Apiary nabízí několik funkcí, ze kterých je pro testování integrací client-server nejdůležitější *mock* server. Každý projekt má automaticky k dispozici URL, na kterou je možné odesílat požadavky z klienta. Pokud je přijatý požadavek v dokumentaci API nalezen, je vrácena odpověď, která je u tohoto požadavku uvedena.

Apiary umožňuje pracovat také s OpenAPI specifikací. Je však podporována pouze druhá, starší verze jazyka, původně nazývaná Swagger. Projekty používající OpenAPI mohou být také ochuzeny o některé funkce, které Apiary nabízí pouze pro projekty používající API Blueprint.

Testování klientské aplikace pomocí služby Apiary pak probíhá za využití zmíněného *mock* serveru. Jeho URL je během testů používána namísto produkčního serveru v testech. Veškerou příchozí i odchozí komunikaci zachycenou *mock* serverem

je možné zobrazit v nástroji Inspector. Ten může být velmi nápomocný při hledání problémů během testování integrace.

Na obrázku 3.11 je možné vidět nástroj Inspector zobrazující proběhlou komunikaci v rámci integračního testu. Tento nástroj zobrazuje všechny příchozí požadavky v přehledném seznamu. Jednotlivá volání je možné rozbalit, a zobrazit tak celý požadavek klienta i odpověď ze serveru včetně hlaviček a těla. Pokud v dokumentaci není nějaký požadavek nalezen, vrací *mock* server odpověď se stavovým kódem HTTP 404 – Not Found. I tyto neznámé požadavky je pomocí nástroje Inspector možné zobrazit.



Obrázek 3.11. Detail příchozího požadavku na *mock* server za pomoci nástroje Inspector.

Apiary je webová služba nabízející mnoho zajímavých funkcí. Pro testování client-server integrací jsou nejdůležitější nástroje Editor a Inspector. Výhodou je, že Apiary nabízí podporu nejen pro vlastní formát popisu API, ale také pro částečně konkurenční formát OpenAPI. Webový editor dokumentu umožňuje i okamžitý náhled vygenerované dokumentace. Tuto dokumentaci je možné hostovat na serverech Apiary a sdílet napříč týmy, obchodními partnery nebo ji publikovat veřejně. Nevýhodou je, že

Apiary zatím nepodporuje nejnovější OpenAPI specifikaci. V případě odstraňování problémů během testování integrace je možné narazit na to, že v nástroji Inspector není možné filtrovat ani vyhledávat jednotlivé požadavky a odpovědi.

3.1.8 Dredd

Dredd [33] je open-source nástroj vyvíjený společností Apiary umožňující validaci serverové implementace vůči specifikaci API. Dredd pracuje se specifikacemi ve formátech API Blueprint nebo OpenAPI. Funguje tak, že načte dokument se specifikací a na základě něj vytvoří interní reprezentaci HTTP požadavků a seznam pravidel, kterým musí odpovědi vrácené ze serveru vyhovět. Poté provádí testování tím způsobem, že postupně odesílá všechny požadavky na server. Vrácené odpovědi porovnává s pravidly, která vytvořil na základě načtené specifikace API. Výsledky z testování jsou poté vypsány na výstup.

Dredd je možné používat dvěma způsoby. První možností je spustit jej jako konzolovou aplikaci, která pomocí argumentů na příkazovém řádku nebo hodnot v konfiguračním souboru provede testování serverové implementace. Tento způsob je jednoduchý na zprovoznění. Dredd navíc obsahuje příkaz `init`, pomocí kterého lze automaticky vytvořit konfigurační soubor se správnými hodnotami, které získá od uživatele pomocí série jednoduchých otázek přímo v rozhraní příkazového řádku.

Druhý způsob, jakým lze spustit Dredd, je přímo v Node.js. Pro to stačí pouze vytvořit instanci Dreddu, která v konstruktoru obdrží konfiguraci. Vytvořená instance obsahuje metodu `run`, která přijímá *callback*. Dredd tento *callback* volá po skončení testů a předává případné chyby a statistiky z testování. Je pak jen na vývojářích, jak s těmito daty naloží.

Dredd také nabízí možnost použít *hook* skripty. Tyto skripty Dredd spouští v různých fázích testovacího cyklu. Náplní skriptů může být například nahrání databázových *fixtures*, vrácení změn na serveru mezi jednotlivými požadavky, nastavení přihlášení, předávání dat mezi požadavky, úprava požadavku před jeho odesláním, úprava vygenerovaných pravidel pro kontrolu odpovědí a další. Tyto skripty je možné psát v jednom ze sedmi programovacích jazyků, které Dredd podporuje. Na obrázku 3.12 je možné vidět ukázkový *hook* skript napsaný v JavaScriptu sloužící k nahrání *fixtures* do databáze před odesláním prvního požadavku na server.

```
1  const hooks = require('hooks');
2  const database = require('./database.js');
3  const testHelpers = require('./testHelpers.js');
4
5  hooks.beforeAll(async (transactions, done) => {
6    await database.connect();
7    await testHelpers.loadDatabaseFixtures();
8    done();
9  });
10
11 hooks.afterAll(async (transactions, done) => {
12   await database.close();
13   done();
14 });
```

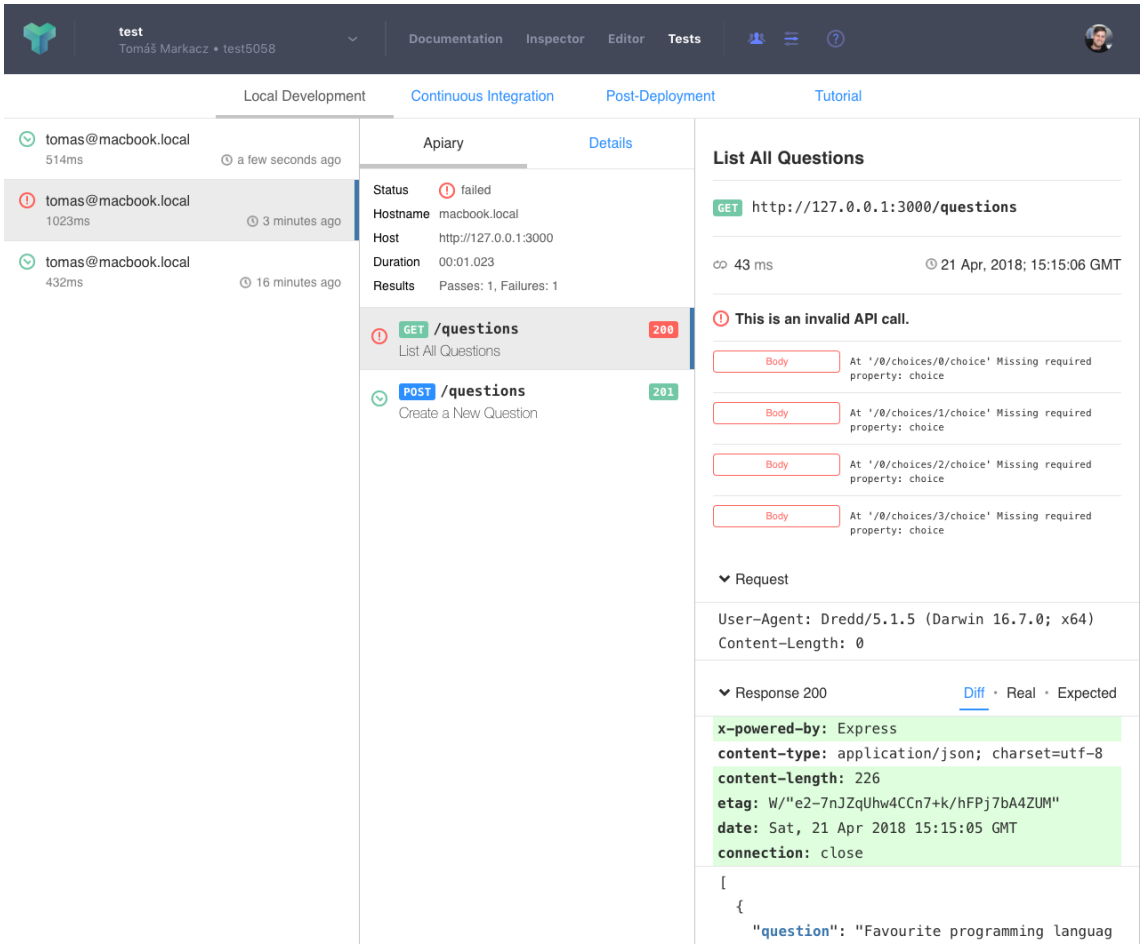
Obrázek 3.12. Ukázka *hook* skriptu nahrávajícího *fixtures* do databáze.

Pravidla pro kontrolu vrácených odpovědí Dredd automaticky generuje ze specifikace API. Kontrolovány jsou jak hlavičky, tak i tělo odpovědi. Dredd pro kontrolu struktury dat v odpovědi používá JSON Schema. Pokud je ve specifikaci toto schéma uvedeno, je rovnou použito. Pokud je specifikace ve formátu API Blueprint a datové

struktury jsou popsány pomocí MSON notace, použije Dredd schéma vygenerované z popisu těchto struktur. V ostatních případech Dredd odvozuje z ukázkových odpovědí pouze základní pravidla tak, aby kontroloval například datové typy.

Dredd umožňuje prezentovat výsledky z průběhu testů několika způsoby. Nejjednodušší způsob je přímo na výstup příkazové řádky při použití konzolové aplikace. Mnohem zajímavější je však použít integraci s Apiary. Dredd dokáže pomocí API klíče odesílat výsledky testů přímo do služby Apiary. V Apiary se pak dají tyto výsledky zobrazit. Z každého testování jsou k dispozici jednotlivé požadavky a odpovědi se zvýrazněním rozdílů oproti pravidlům, která Dredd vygeneroval z dokumentace API. Tyto výsledky jsou navíc v Apiary zařazeny přímo pod specifikaci, ke které patří.

Na obrázku 3.13 je možné vidět výstup testu ve službě Apiary. Testy byly spuštěny celkem třikrát, z toho podruhé selhal test pro načtení všech otázek. V tomto případě odeslal server chybnou odpověď, která není v souladu se specifikací API.



Obrázek 3.13. Výsledek selhaného testu realizovaného pomocí nástroje Dredd zobrazený ve službě Apiary.

Výhodou nástroje Dredd je jeho všestrannost. Je velmi dobře konfigurovatelný. Pokud je v některých situacích výstup nedostatečný, je možné Dredd spustit z Node.js a předat *callback*, který s výsledky testů může udělat cokoliv, co je potřeba. Podpora *hook* skriptů je velmi silný nástroj umožňující zasahovat do běhu testů. Implementace těchto skriptů navíc není omezena na jeden skriptovací jazyk. Užitečná je také integrace se službou Apiary. Nevýhodou tohoto nástroje je, že zatím nepodporuje

OpenAPI specifikaci ve třetí verzi. Konzolový výstup tohoto nástroje by v případě selhaného testu mohl lépe indikovat, kde se ve vrácené odpovědi nachází chyba. V případě výsledku zobrazeného ve službě Apiary je toto mnohem lépe vyřešené.

3.2 Consumer-driven contracts

V této sekci popíši nástroje, se kterými je možné realizovat *consumer-driven contracts* testování. Jedná se o velmi specifický druh testování, proto jsou takové nástroje použitelné pouze na tento způsob testování.

Podařilo se mi nalézt celkem tři nástroje, pomocí kterých lze provádět *consumer-driven contracts* testování. První z nich, Janus, je nástroj umožňující specifikovat kontrakty mezi službami v Clojure pomocí testů serverové implementace [34]. Janus však neumí otestovat klienta pomocí *mock* serveru. Navíc se jedná o opuštěný projekt, který není dlouhou dobu vyvíjen. Poslední příspěvek v oficiálním repozitáři je z listopadu roku 2013.

Druhý nástroj, Pacto, slouží k testování obou stran integrace [34]. Bohužel se opět jedná o opuštěný projekt, na který původní autoři již nemají čas. Přímo v oficiálním repozitáři doporučují autoři použít nástroj Pact.

Z těchto důvodů se ve své práci zaměřím pouze na třetí nástroj – Pact, který je oproti nástrojům Janus a Pacto aktivně vyvíjen, má podporu pro několik programovacích jazyků a nabízí mnohem více funkcí. Bohužel se mi nepodařilo najít žádný nástroj, který by nebyl omezen pouze na testování HTTP komunikace.

3.2.1 Pact

Pact [35] je *framework* umožňující *consumer-driven contracts* testování. K testování integrací využívá kontrakt, což je množina dohodnutých pravidel pro komunikaci mezi klientem (konzumentem) a serverem (poskytovatelem). Pact je nástroj, který umožňuje tato pravidla zapsat a zaručit, že jsou protistranou splněna.

Pact funguje tak, že v testech klienta jsou ještě předtím, než dojde k zavolání testované implementace, definovány interakce, které testovaný kód během svého spuštění provede. Testovaný kód pak tyto interakce provádí vůči *mock* serveru. Ten vrací odpovědi tak, jak bylo v interakcích zadefinováno. Požadavky přijaté *mock* serverem a kritéria na odpověď definovaná v interakci jsou zapisovány do Pact souboru.

Interakce zachycené v Pact souboru jsou později zopakovány v testech serverové implementace. Pact odesílá zaznamenané požadavky na testovaný server a přijaté odpovědi kontroluje vůči kritériím pro aktuální interakci z Pact souboru. Tímto způsobem je zajištěno ověření toho, že serverová implementace splňuje kritéria zadefinovaná klientem během svých testů. Je také důležité zmínit, že kritéria v Pact souboru nejsou uložena ve formě schémat, ale konkrétních požadavků a odpovědí. Schéma popisuje všechny možné varianty požadavku, Pact však využívá principu definice kontraktu pomocí ukázek. Každá interakce proto validuje jen jednu konkrétní variantu požadavku a odpovědi.

Pact také umožňuje ke každé interakci definovat, v jakém stavu se má server před zahájením interakce nacházet. Pro tyto účely může klient ve svých testech přiřadit k jednotlivým interakcím identifikátor požadovaného stavu. V testech na serveru je poté potřeba připravit kód, který na základě tohoto identifikátoru provede nastavení serveru do odpovídajícího stavu. Pomocí těchto stavů je možné, mimo jiné, testovat různé serverové odpovědi na tentýž klientský požadavek v závislosti na testované situaci.

Pact je souhrnné označení pro Pact specifikaci, konkrétní implementace knihoven v různých programovacích jazycích sloužící k testování integrací, a další pomocné nástroje. Pact specifikace popisuje formát dat v Pact souboru a způsob, jakým mají být porovnávány skutečné odpovědi vůči kritériím uvedeným v souboru. Knihovny pro testování klientské a serverové části integrace jsou k dispozici pro Ruby, JVM, .NET, JavaScript, Go, Python, Swift a PHP. Vzhledem k tomu, že jsou Pact soubory standardizovány, je možné testovat integraci klientské a serverové implementace, i když je každá napsána v jiném programovacím jazyce.

Pact soubor obsahuje data ve formátu JSON popisující konzumenta a poskytovatele, obsahující seznam interakcí a další metadata. Pro opakované testování interakce je potřeba tento soubor sdílet mezi klientem a serverem. Pro tyto účely je součástí Pact *frameworku* nástroj Pact Broker. Tento nástroj umožňuje jednoduše sdílet kontrakty mezi projekty používajícími *consumer-driven contracts* testy. Pact Broker poskytuje RESTful API pro publikování a získávání Pact souborů, obsahuje webovou aplikaci pro správu jednotlivých integrací, umožňuje vizualizaci závislostí pomocí diagramů, umí zobrazit výsledky z integračního testování, dokáže zobrazit matici kompatibilních verzí klienta a serveru nebo umožňuje integraci do existujících *continuous integration* procesů pomocí konzolového klienta nebo *webhooků*. Pact Broker je aplikace napsaná v Ruby, je možné ji provozovat na vlastních serverech přímo nebo ve formě Docker kontejneru. Existuje také hostované řešení od sponzora DiUS [36], které je zcela zdarma. Knihovny pro testování integrací se umějí automaticky napojit na Pact Brokera a zajistit tak načítání a publikování Pact souborů i export výsledků testování. Lze se však obejít i bez tohoto Brokera a zajistit sdílení Pact souborů mezi projekty jiným způsobem.

Pro účely této práce se dále zaměřím na knihovnu Pact.js sloužící k testování klientské i serverové implementace napsané v JavaScriptu. Na obrázku 3.14 je možné vidět jednoduchý test na straně klienta sloužící pro definici kontraktu a otestování práce s odpovědí ze serveru. Před zahájením testování je spuštěn *mock* server, který naslouchá na portu uvedeném v konfiguraci. V samotném testu je nejdříve definována interakce, která obsahuje i požadovaný stav na serveru. Odpověď je možné definovat přímo s konkrétními daty nebo je možné použít pravidla, která definují pouze požadavky na hodnoty. Tato pravidla umožňují ověřit například datový typ, zkontrolovat položky pole nebo otestovat formát řetězců pomocí připravených funkcí či pomocí vlastního regulárního výrazu. Definovaná interakce je pak realizována zavoláním testovací implementace. Ta zašle požadavek *mock* serveru, který na něj odpoví tak, jak bylo zadefinováno v interakci. Testování klientské implementace se v tomto případě výrazně neliší od *mocking* testů popsaných v předchozí sekci.

Na obrázku 3.15 je ukázán test serverové implementace. Pact pro nastavení serveru do požadovaného stavu používá HTTP požadavku na URL, která je uvedena v konfiguraci. V těle požadavku se nachází JSON s požadovaným stavem. Pomocí stavového kódu HTTP 200 – OK v odpovědi je signalizováno úspěšné nastavení stavu na serveru. V odkazované ukázce je na serveru definován pouze jeden stav, který zajišťuje existenci konkrétního uživatele v databázi. Na připravený testovaný server je pak odeslán požadavek z interakce zachycené v Pact souboru. Cesta k němu je uvedena v konfiguraci. Pact o průběhu testování informuje na standardním výstupu. Pokud testování selže z důvodu obdržení odpovědi v rozporu s tou definovanou v kontraktu, vypisuje Pact velmi přehledně v jaké interakci k chybě došlo a barevně zvýrazňuje jednotlivé rozdíly v přijaté a očekávané odpovědi.

```
const authenticator = require('./authenticator');
const config = require('./config');
const {Pact} = require('@pact-foundation/pact');

const pact = new Pact({
  consumer: 'client',
  provider: 'server',
  port: config.port,
});

beforeAll(() => pact.setup());
afterAll(() => pact.finalize());

describe('authenticator', () => {
  it('successfully logs in user', async () => {
    await pact.addInteraction({
      state: 'user #42 Tomáš Markacz exists',
      uponReceiving: 'a request for log in',
      withRequest: {
        method: 'POST',
        path: '/login',
        headers: {'Content-Type': 'application/json'},
        body: {
          email: 'tomas@markacz.com',
          password: 'foobar',
        },
      },
      willRespondWith: {
        status: 200,
        headers: {'Content-Type': 'application/json'},
        body: {
          id: 42,
          name: 'Tomáš Markacz',
        },
      }
    });

    // performs POST /login with given email and password and returns user info
    const user = await authenticator.login('tomas@markacz.com', 'foobar');

    expect(user).toEqual({
      id: 42,
      name: 'Tomáš Markacz',
    });
  });
});
```

Obrázek 3.14. Ukázka klientského testu využívajícího Pact pro definici kontraktu a vrácení odpovědi.

Nastavení serveru do konkrétního stavu je možné využít nejen k nahrání *test fixtures* do databáze, ale umožňuje také vytvořit *test doubles*, které budou použity v implementaci během testů. Tím je možné testy více izolovat a zaměřit se pouze na určitou část implementace. *Test doubles* navíc umožňují odstínit testovanou implementaci od externích závislostí a služeb nebo simulovat složitě dosažitelné stavy.

Výhodou tohoto *frameworku* je podpora mnoha programovacích jazyků. Standardizovaný formát kontraktů umožňuje testovat integraci nezávisle na použitých technologiích. Velmi užitečná funkce tohoto nástroje je možnost nastavit stav na serveru pomocí *test fixtures* kódu. To velmi zjednodušuje testování serveru a umožňuje zaměřit se v integračních testech pouze na kód týkající se interakce s klientem a zbytek implementace pokrýt jiným druhem testů. Užitečná je také možnost využít připravený nástroj Pact Broker pro sdílení a správu kontraktů mezi klienty a serverem. Pact je aktivně vyvíjen a sponzorován hned několika komerčními subjekty, což vede k rozši-

```
const app = require('./app');
const testHelpers = require('./testHelpers.js');
const {Verifier} = require('@pact-foundation/pact');

const verifier = new Verifier();

// custom URL used in tests for state setup
app.post('/_setup', async (request, response) => {
  switch (request.body.state) {
    case 'user #42 Tomáš Markacz exists':
      await testHelpers.loadUsersFixtures();
      break;

    default:
      response.status(400);
  }
  response.end();
});

beforeAll(() => app.listen(config.port));
afterAll(() => app.close());

describe('contract', () => {
  it('should correspond with client expectations', async () => {
    await verifier.verifyProvider({
      provider: 'server',
      providerBaseUrl: config.baseUrl,
      providerStatesSetupUrl: `${config.baseUrl}/_setup`,
      pactUrls: config.pactFilePaths, // pact file generated during client tests
    });
  });
});
```

Obrázek 3.15. Ukázka serverového testu sloužícího ke kontrole implementace API vůči kontraktu.

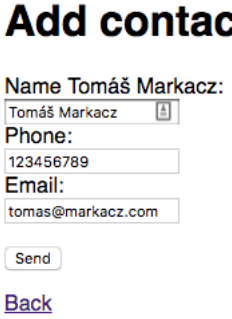
řování nástroje o nové funkce a možnosti. Nevýhodou tohoto nástroje je v současnosti silná vazba na testování bezstavové komunikace. Díky tomu je testování limitováno pouze na API využívající RESTful architektury a nelze například testovat interakce probíhající pomocí WebSocket protokolu.

3.3 End-to-end

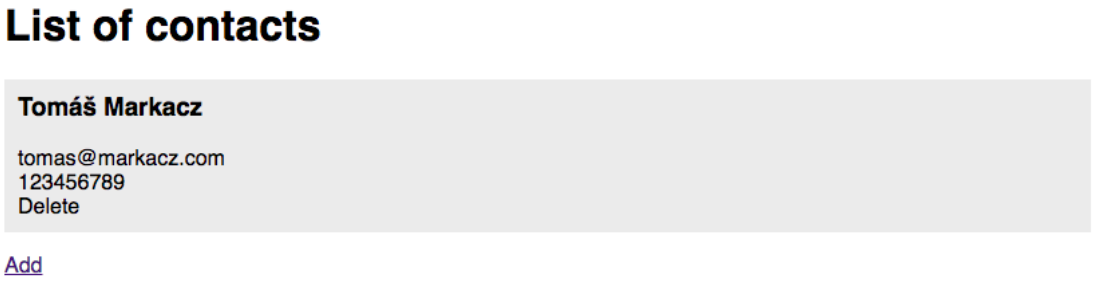
Pro *end-to-end* testování webových aplikací existuje mnoho nástrojů, které se dají použít i pro účely integračního testování. Synonymem pro *end-to-end* testování ve světě webových aplikací je nástroj Selenium. Proto se na něj i já ve své diplomové práci zaměřím, a popíši způsob, jakým jej lze použít pro *end-to-end* testování integrací client-server.

Selenium a jemu podobné nástroje používají k ovládání webové aplikace instanci běžícího prohlížeče, se kterou komunikují pomocí ovladače. Pro účely *end-to-end* testování integrací je možné použít standardní prohlížeč v módu bez grafického rozhraní. V anglických textech se tento mód nazývá *headless*. Je možné použít také specializované nástroje, které v sobě obsahují odlehčený prohlížeč, často zcela bez vykreslovacího jádra. Díky tomu lze pak testy používající tento prohlížeč spouštět efektivně například v *continuous integration* prostředí. Výhodou specializovaných nástrojů je jejich rychlejší provoz a menší nároky na systémové zdroje ve srovnání s nástroji používající k testování plnohodnotné prohlížeče. Pomocí nástroje Selenium je možné spustit jak běžné prohlížeče v *headless* módu, tak použít i zmíněné nástroje obsahující implementaci zjednodušeného prohlížeče.

Nyní se zaměřím na detailnější popis Selenium *frameworku* a jeho použití pro účely testování integrací client-server. Pro vyzkoušení tohoto nástroje jsem použil jednoduchou webovou aplikaci sloužící ke správě kontaktů. Na obrázku 3.16 je možné vidět obrazovku s formulářem sloužícím pro přidání nového kontaktu. Na obrázku 3.17 se nachází obrazovka s nově přidaným záznamem. V ukázce použití nástroje Selenium testuji právě tuto aplikaci.



Obrázek 3.16. Obrazovka sloužící k přidání nového kontaktu z testované webové aplikace.



Obrázek 3.17. Obrazovka výpisu všech uložených kontaktů z testované webové aplikace.

3.3.1 Selenium

Selenium [37] je *framework* pro testování webových aplikací. Skládá se z několika komponent. Selenium IDE je vývojové prostředí pro rychlé prototypování testovacích skriptů. Jedná se o plugin do webového prohlížeče Firefox. Selenium WebDriver, známé jako Selenium 2, je nástroj, pomocí kterého lze provádět interakce s webovým prohlížečem. Využívá nativní podpory pro automatizované ovládání prohlížečů. Selenium WebDriver používá ovladače pro nejpoužívanější webové prohlížeče na osobních počítačích i mobilních zařízeních. Selenium Grid umožňuje škálovat rozsáhlé testovací sady pomocí paralelizace nebo spouštět testy na více prostředích zároveň. Selenium Remote Control, známé jako Selenium 1, je server přijímající příkazy pomocí HTTP API. Umožňuje tak testovat webové aplikace nezávisle na programovacím jazyce. Pro interakci s prohlížečem používá JavasScript spouštěný přímo v prohlížeči. Selenium Remote Control se již nedoporučuje používat a není dále aktivně vyvíjen, neboť je jeho funkčnost nahrazena nástrojem Selenium WebDriver.

Pro účely testování integrací client-server lze použít JavaScriptovou implementaci nástroje Selenium WebDriver. Ta umožňuje získávat elementy uživatelského rozhraní v prohlížeči pomocí lokátorů. Lokátor je předpis adresující konkrétní elementy na stránce. Umožňuje adresovat podle identifikátoru, třídy, *tagu*, názvu nebo například

textu v odkazu. Elementy lze adresovat také pomocí CSS selektorů nebo XPath výrazů. Nad takto získanými elementy je možné vyvolávat události, jakými mohou být stisky kláves nebo kliknutí. Dále je možné číst atributy elementů nebo například získat jeho celý textový obsah. Selenium také umožňuje jednoduše testovat dynamické webové aplikace napsané v JavaScriptu. Tyto aplikace používají asynchronní volání serveru, které nepřekresluje celou stránku. Z tohoto důvodu je nutné nějakým způsobem zajistit během testu čekání na doručení a zpracování odpovědi ze serveru. Selenium WebDriver nabízí mechanismy, pomocí kterých je možné v testu čekat, dokud není splněna definovaná podmínka.

JavaScriptová implementace Selenium WebDriver umí pracovat s prohlížeči Chrome, Internet Explorer, Edge, Firefox a Safari. Pro účely testování integrací client-server je možné se zaměřit pouze na testování v jednom prohlížeči, protože cílem není testovat kompatibilitu aplikace s rozdílnými prohlížeči, ale integraci webové aplikace se serverem.

```
1  require('chromedriver');
2  const {Builder, By, until} = require('selenium-webdriver');
3
4  let driver;
5  beforeAll(async () => {
6    driver = await new Builder().forBrowser('chrome').build();
7  });
8  afterAll(() => driver.quit());
9
10 describe('user adding a new contact', () => {
11   beforeEach(() => driver.get('http://localhost:3000/add'));
12
13   it('submits form and checks for result', async () => {
14     await driver.findElement(By.name('name')).sendKeys('Tomáš Markacz');
15     await driver.findElement(By.name('phone')).sendKeys('123456789');
16     await driver.findElement(By.name('email')).sendKeys('tomas@markacz.com');
17     await driver.findElement(By.css('input[type=submit]')).click();
18
19     await driver.wait(until.titleIs('Contact list'));
20
21     const name = await driver.findElement(By.css('.Record .name'));
22     const phone = await driver.findElement(By.css('.Record .phone'));
23     const email = await driver.findElement(By.css('.Record .email'));
24
25     expect(await name.getText()).toMatch('Tomáš Markacz');
26     expect(await phone.getText()).toMatch('123456789');
27     expect(await email.getText()).toMatch('tomas@markacz.com');
28   });
29 });
```

Obrázek 3.18. Ukázka *end-to-end* testu pro přidání kontaktu za pomoci nástroje Selenium.

Na obrázku 3.18 je možné vidět ukázkový test pro přidání nového kontaktu do seznamu kontaktů. Tento test využívá prohlížeč Chrome. Pro interakci s ním je potřeba mít nainstalovaný příslušný ovladač. Nutnost instalovat ovladač do systému, na kterém bude testování probíhat, lze v případě ovladače pro prohlížeč Chrome obejít použitím NPM balíčku `chromedriver`. Tento balíček je načítán na prvním řádku v ukázkovém kódu. Obsahuje nejnovější verzi ovladače, který je automaticky spuštěn v podprocesu testu. V testech pak dochází pouze ke konfiguraci použitého ovladače. Pomocí něj je načtena stránka s formulářem pro přidání nového kontaktu. Reference na elementy pro jednotlivá formulářová pole jsou získána pomocí lokátorů. Dále jsou nad těmito poli vyvolány události představující stisky kláves. Pomocí události stisku tlačítka jsou následně data z formuláře odeslána na server. Poté je využito funkce

zajišťující čekání na změnu titulku stránky, který se změní s načtením seznamu kontaktů po odeslání formuláře. Nakonec je v testech ověřeno, zda byl kontakt úspěšně vypsán v seznamu na webové stránce.

Výhodou Selenium *frameworku* je jeho obecná rozšířenost. Z tohoto důvodu s ním má zkušenost nemalá část vývojářů webových aplikací. Selenium WebDriver je implementován v Javě, C#, Pythonu, Ruby, PHP, Perlu a JavaScriptu. Díky tomu, že používá standardní prohlížeč, je testování velmi blízké skutečnému produkčnímu nasazení. Nevýhodou tohoto nástroje může být jeho příliš univerzální rozhraní, čímž se časté úkony, například vyplňování formulářů, skládají ze zbytečně velkého množství kódu.

3.4 Synchronizované testování

Způsob synchronizovaného testování integrací client-server vyplynul z potřeb týmu vývojářů, se kterými pracuji. Tento způsob by našemu týmu umožnil spolehlivě a automatizovaně testovat integraci s klientskou aplikací. V tomto případě se jednalo o integraci klienta ve formě mobilní aplikace a serveru napsaného v Node.js. Komunikace klientské aplikace a serveru probíhala pomocí WebSocket protokolu. Vzhledem k povaze výsledného produktu se jednalo o netriviální komunikaci s mnoha stavy a několikakrokovými interakcemi mezi klientem a serverem.

Z těchto důvodů jsme byli nuceni pro zajištění kvality výsledného produktu provádět testování integrace z části manuálně, vždy ve spolupráci s vývojářem z druhého týmu. Vzhledem k tomu, že se v tomto případě navíc jednalo o více než pouze jednu klientskou aplikaci, hledali jsme jiné efektivnější způsoby testování integrací client-server.

Způsob synchronizovaného testování, který popisuji v předchozí kapitole, by nám umožnil integraci spolehlivě a automatizovaně otestovat. Ačkoliv se z mého pohledu jedná o zajímavý způsob testování, který má i v běžném použití své místo, nepodařilo se mi najít žádné zmínky o alespoň podobném způsobu testování.

Z tohoto důvodu jsem se na základě rešerše existujících způsobů testování integrací client-server použitelných v kontextu webových a mobilních aplikací a na základě rešerše existujících nástrojů vhodných pro realizaci těchto způsobů testování rozhodl pro vlastní implementaci nástroje umožňující synchronizované testování.

Realizaci tohoto nástroje jsem provedl na základě návrhu, který popisuji v následující kapitole. Součástí kapitoly je také popis výsledného nástroje společně s demonstrací jeho použití v ukázkovém testu.

Kapitola 4
Realizace nástroje pro testování integrací client-server

Tato kapitola se zabývá popisem vzniklého testovacího nástroje. Nástroj vznikal v několika fázích, od definice požadavků, návrhu, prototypování, až po výslednou implementaci.

4.1 Popis nástroje

Samotný nástroj je také postaven na architektuře client-server. Skládá se ze dvou částí, klienta, který se používá v testech klientské implementace, a serveru, který řídí testování serverové implementace. Životní cyklus integračního testu je vždy iniciován a řízen z klientské strany. Klientská a serverová část nástroje spolu komunikují pomocí HTTP požadavků a odpovědí obsahujících data ve formátu JSON. Schéma komunikace je definované nezávisle na konkrétní implementaci. Z tohoto důvodu je možné používat klientskou část nástroje napsanou v jiném programovacím jazyce, než v jakém je napsána serverová část. Pro účely této práce se zaměřuji pouze na JavaScriptovou implementaci obou částí, avšak tento nástroj je připraven pro implementaci i v dalších jazycích.

Testování integrací je založeno na scénářích. Scénář popisuje průběh testu konkrétní interakce klienta a serveru. Skládá se z celkem tří fází: *setup*, *step* a *teardown*. Každá z těchto částí scénáře je asociována s odpovídajícím kódem na serveru. *Setup* fáze je povinná a slouží ke spuštění serverové implementace a jejímu uvedení do stavu požadovaného scénářem. V této fázi je možné například vytvořit *test doubles* nebo nahrát testovací data do databáze. *Step* fáze jsou nepovinné a může jich být libovolný počet. Je možné je použít pro situace, ve kterých je v průběhu testu integrace potřeba na serveru nějakým způsobem upravit chování, změnit stav nebo provést další kontroly. Tyto fáze umožňují na serveru spustit kód, který ovlivní serverovou implementaci v průběhu testu integrace. Příkladem může být úprava chování *mock* objektu. *Teardown* fáze je nepovinná a může sloužit ke dvěma účelům. Primárním účelem je korektní ukončení testované implementace. Může se jednat například o uvolnění prostředků alokovaných serverovou implementací nebo vrácení změn v databázi. Vedlejším účelem je porovnání nepřímých výstupů serverové implementace s těmi očekávanými. Jde například o kontrolu volání *mock* objektů v rámci serverové implementace.

Testování integrace probíhá z klienta za pomoci libovolného testovacího *frameworku* umožňující zápis ve stylu jednotkových testů. Před spuštěním klientských testů je potřeba zajistit, aby byl spuštěn testovací server prostřednictvím konzolové aplikace serverové části nástroje. Tento server je možné spouštět s každým testováním nebo jej nechat běžet trvale, čímž lze integrační testy provádět v libovolný okamžik. Samotný test pak začíná spuštěním *setup* fáze vybraného scénáře prostřednictvím klientské části nástroje. Ten volá API serverové části, která vytvoří podproces a

spustí v izolovaném kontextu *setup* kód asociovaný s vybraným scénářem. Úkolem tohoto kódu je spustit serverovou implementaci a zajistit, aby se server nacházel ve stavu požadovaném scénářem. Po skončení této fáze dochází ke spuštění testované klientské implementace. Ta v rámci testu interaguje s připravenou serverovou implementací. Pokud to scénář nabízí, je možné na serveru spustit skrze klienta jeho další kroky. Po skončení klientského testu je nutné ukončit integrační test prostřednictvím klienta, který na serveru zajistí spuštění *teardown* kódu asociovaného s aktuálním scénářem, a ukončení podprocesu, ve kterém byl testovaný server spuštěn.

Vzhledem k tomu, že nástroj umožňuje paralelní běh více testů, je potřeba zajistit směrování požadavků z klientské implementace na testovaný server, který byl vytvořen v rámci společného testu. Nástroj nabízí dvě možnosti, jak tyto požadavky směrovat. První možností je přiřadit každému testu unikání port, pomocí kterého bude klient se serverem komunikovat. Druhou možností je používat pro každý test unikátní hodnotu ve speciální HTTP hlavičce, pomocí které server nástroje provede přeposlání požadavku na odpovídající testovanou implementaci. První způsob by měl být vhodný pro většinu situací. Vzhledem k tomu, že v případě směrování pomocí HTTP hlavičky vzniká nezanedbatelná režie spojená se zpracováním hlaviček a přeposláním požadavků a odpovědí, měl by být způsob směrování pomocí portu preferován. Směrování pomocí hlaviček nástroj nabízí zejména pro situace, kdy z nějakého důvodu není možné, aby webový server nástroje vystavoval dynamicky více portů, například z důvodu použité serverové architektury.

Způsob komunikace mezi klientskou a serverovou částí nástroje popisuji v následující sekci. Pro úplnost uvádím kompletní seznam požadavků na serverovou část nástroje, který jsem sestavil před samotným návrhem a implementací:

- Jedná se o konzolovou aplikaci napsanou v JavaScriptu pro Node.js.
- Konfigurace je možná pomocí sekce v souboru `package.json`.
- Pro řízení scénářů je poskytováno webové API.
- Scénáře se zapisují v JavaScriptu za pomoci globálních funkcí podobným způsobem, jako testy v nástrojích Jasmine nebo Mocha.
- Kód scénářů je izolován pomocí *virtual machine context*, který je k dispozici v Node.js.
- Nástroj je od testované implementace izolován takovým způsobem, aby byl nebyl náchylný na její pády.
- Je umožněn paralelní běh několika testů najednou, tyto testy jsou navzájem izolovány.
- Před spuštěním serveru jsou procházeny zadané adresáře, ve kterých jsou hledány soubory se scénáři, jejichž název vyhovuje zadanému regulárnímu výrazu. Scénáře definované v těchto souborech je možné spustit v rámci požadavku na řídící API.
- Směrování komunikace ke správnému testovanému serveru je zajištěno pomocí unikátního portu nebo identifikátoru v hlavičce požadavku.
- Testovaný serverový kód automaticky používá upravenou implementaci třídy `http.Server`, která zcela automaticky umožňuje příjem příchozích spojení pouze v rámci jednoho testu na základě zvoleného způsobu směrování.
- V souborech se scénáři je k dispozici rozhraní pro tvorbu *test doubles* a kontrolu výstupů testované implementace.
- Je možné omezit počet současně běžících instancí testovaného serveru s limitem času, po kterou může test běžet.
- Standardní výstup a standardní chybový výstup každého testu je zaznamenán do souboru.

Pro klientskou část nástroje jsem obdobným způsobem definoval následující požadavky:

- Jedná se o knihovnu napsanou v JavaScriptu, která je použitelná v prohlížeči nebo v Node.js.
- Klient je konfigurovatelný pomocí konstruktoru.
- Nabízí rozhraní, pomocí kterého je možné řídit fáze integračního testu (`setup`, `step` a `teardown`). Toto rozhraní je asynchronní, jeho metody vracejí instance `Promise`, které čekají na dokončení kódu na serveru.

Takto definované požadavky mi byly podkladem pro návrh obou částí nástroje. Nejdříve jsem střídavě prováděl prototypování a testování, abych zjistil, jakým způsobem bude možné splnit požadavky, které jsem na nástroj definoval. Následně jsem na základě kompletního návrhu a funkčních prototypů implementoval obě části nástroje.

4.2 Způsob komunikace

Pro zajištění komunikace mezi klientskou a serverovou částí nástroje slouží webové API. Toto API jsem navrhl tak, aby bylo univerzální a co nejjednodušší. Cílem tohoto API je vzájemně abstrahovat klientskou a serverovou část nástroje, díky čemuž může být každá napsána v jiném programovacím jazyce a pro běh využívat jiné platformy. Toto API se nazývá řídící, protože slouží k řízení integračního testu. Skládá se celkem ze tří *endpointů*:

- `POST /{prefix}/setup`
- `POST /{prefix}/step`
- `POST /{prefix}/teardown`

Pro zamezení konfliktů s *endpointy* testované implementace v případě použití sdíleného portu a směrování pomocí HTTP hlavičky jsou *endpointy* řídícího API prefixovány. Tento prefix je možné pomocí konfigurace změnit.

Integrační test začíná požadavkem na *endpoint* `POST /{prefix}/setup`. Server vytvoří podproces, ve kterém spustí *setup* fázi zadaného scénáře. Požadavek obsahuje verzi řídícího API, kterou klientská implementace vyžaduje. Server v závislosti na této hodnotě může s klientem komunikovat podle starší specifikace API, čímž je zajištěna zpětná kompatibilita serverové části se staršími verzemi klientů. Požadavek dále obsahuje identifikátor právě spouštěného testu, který je vždy generován klientem, název scénáře, podle kterého bude integrační test probíhat, a typ směrování požadavků mezi testovanou klientskou a serverovou implementací. Tento typ může nabývat dvou hodnot, `header` pro směrování pomocí hodnoty v hlavičce požadavku nebo `port` pro směrování prostřednictvím portu. Ukázkový *setup* požadavek je možné vidět na obrázku 4.1.

V odpovědi na *setup* požadavek server v případě směrování pomocí portu vrací číslo portu alokovaného pro tento test, v případě směrování pomocí hodnoty v hlavičce vrací server název hlavičky, kterou bude testovaná klientská implementace ve svých požadavcích používat. Dále je v odpovědi uveden časový limit v sekundách, po jehož překročení bude integrační test ukončen ze strany serveru. Časový limit je v odpovědi z toho důvodu, aby mohl být test v klientské části automaticky ukončen po jeho překročení. Díky tomu se zabrání dalším zbytečným požadavkům na server, které by selhaly, protože byl test již ukončen. Časový limit se nastavuje v konfiguraci na

```
{
  "version": 1,
  "test": {
    "id": "f92f01df-032b-4336-ab17-b5889b9025eb",
    "scenario": {
      "name": "login:successful"
    },
    "binding": {
      "type": "port"
    }
  }
}
```

Obrázek 4.1. Ukázkový *setup* požadavek.

```
{
  "test": {
    "binding": {
      "port": 43849
    },
    "timeout": 60
  }
}
```

Obrázek 4.2. Ukázková *setup* odpověď.

serveru a začíná se počítat od okamžiku spuštění testu. Ukázková *setup* odpověď je demonstrována na obrázku 4.2.

Pokud scénář obsahuje *step* fáze, lze serverový kód asociovaný s těmito fázemi spustit pomocí požadavku na *enpoint* `POST /{prefix}/step`. Tento požadavek obsahuje pouze identifikátor testu. Serverová část na základě tohoto požadavku v odpovídajícím podprocesu spustí *step* fázi definovanou ve scénáři. Ukázkový požadavek je možné vidět na obrázku 4.3. Server na tento požadavek nevrací žádná data, úspěšné provedení fáze je indikováno pomocí stavového kódu HTTP 204 – No Content.

```
{
  "test": {
    "id": "f92f01df-032b-4336-ab17-b5889b9025eb"
  }
}
```

Obrázek 4.3. Ukázkový *step* a *teardown* požadavek.

K ukončení integračního testu slouží *endpoint* `POST /{prefix}/teardown`. Zavolání tohoto *endpointu* zapříčiní spuštění *teardown* fáze scénáře v odpovídajícím podprocesu. Výsledkem této fáze by mělo být ukončení podprocesu. Pokud do určitého časového limitu nedojde k jeho ukončení, je násilně ukončen z řídícího procesu. Tento požadavek, stejně jako požadavek pro spuštění *step* fáze, obsahuje pouze identifikátor testu a server na něj nevrací žádná data. V testech klientské implementace je důležité zajistit, aby byl tento *endpoint* vždy po skončení testu zavolán, nehledě na jeho úspěšné či neúspěšné provedení. Díky tomu se lze na serveru vyhnout situaci, ve které dochází k hromadění podprocesů nepoužívaných testů. Takové testy zbytečně zabírají systémové zdroje a jsou ukončeny až po uběhnutí časového limitu.

4.3 Popis implementace

Výsledkem praktické části této diplomové práce je funkční implementace nástroje. Implementace vznikla na základě požadavků, které jsem popsal v předchozí sekci. Jedná se o nástroj napsaný v JavaScriptu. Využívá nejnovějších vlastností jazyka

podle specifikace ECMAScript 8. Pro zajištění zpětné kompatibility je použit nástroj Babel, který kód transpiluje do starší verze jazyka. Implementace je staticky typována pomocí Flow. Její jednotná úprava je zajištěna pomocí nástroje ESLint.

Serverová část nástroje z části staví na *frameworku* Jest [38]. Jedná se o populární a aktivně vyvíjený nástroj od vývojářů z Facebooku určený pro testování JavaScriptového kódu. Jest je modulární, skládá se z necelých 40 komponent, které je možné používat i samostatně. Z tohoto důvodu jsem strávil nezanedbatelné množství času procházením zdrojových kódů *frameworku*, abych pochopil, jak funguje. Díky tomu jsem zjistil, jaké komponenty mohu použít v implementaci svého nástroje, a zároveň získal inspiraci, jak nový nástroj koncipovat. V implementaci nástroje jsem použil celkem čtyři komponenty z Jest *frameworku*: `jest-environment-node`, `jest-resolve`, `jest-runtime` a `expect`.

Balíček `jest-environment-node` slouží ke spouštění skriptů v izolovaném prostředí, které obsahuje stejné globální objekty jako Node.js. Využívá k tomu `vm` modulu v Node.js, který umožňuje kompilovat a spouštět kód v kontextech, které nabízí V8 Virtual Machine. Tyto kontexty slouží ke spouštění kódu v *sandbox* prostředí tak, aby neovlivnil své okolí. Jest *framework* pomocí tohoto mechanismu izoluje testovaný kód od svého vlastního.

Balíček `jest-resolve` slouží k dohledávání modulů, ať už se jedná o uživatelské moduly, Node.js *core* moduly nebo moduly v adresáři `node_modules`. Tento balíček je potřeba pro realizaci systému modulů, který v testovaném kódu není k dispozici, protože `vm` v Node.js slouží pouze ke spouštění zcela izolovaného JavaScriptového kódu. Takto izolovaný kód nemůže například načíst jiný modul, protože nemá přístup k funkci `require`.

Balíček `jest-runtime` využívá předešlé dva balíčky a slouží ke spouštění modulů v prostředí věrně kopírujícím Node.js. Pomocí tohoto balíčku lze načíst modul stejným způsobem, jako při jeho spuštění pomocí programu `node`. Toto prostředí však nabízí další funkce navíc, které lze využít v testech. Balíček umožňuje například vytvořit *mock* modulu, čímž nedojde k načtení reálné implementace. V prostředí je také globálně přístupný objekt `jest`, pomocí kterého lze vytvářet *test doubles* objekty pro použití v testované implementaci. Díky tomu, že v nástroji používám tento balíček, lze v integračních testech používat všechny pokročilé funkce, které Jest nabízí.

Poslední balíček z Jest *frameworku*, který je v nástroji použit, je `expect`. Tento balíček nabízí funkce pro porovnávání očekávaných a skutečných výsledků testované implementace. Nabízí mnoho pokročilých funkcí včetně práce s *test doubles* objekty.

Díky tomu, že jsou v nástroji použity výše zmíněné balíčky z *frameworku* Jest, je testovací prostředí téměř shodné s testovacím prostředím, které nabízí Jest. Pro uživatele tohoto *frameworku* tak odpadá nutnost učení se nových věcí a mohou s minimálním úsilím psát integrační testy používající vzniklý nástroj. Zároveň také odpadá velké množství práce, které by muselo být vynaloženo pro vývoj a údržbu obdobné funkčnosti pro účely nově vzniklého nástroje.

Klíčovou vlastností nového nástroje je, že je běh testované serverové implementace pomocí podprocesu zcela izolován od hlavního procesu, který řídí a spravuje jednotlivé podprocesy a poskytuje řídící API. Díky tomu není hlavní proces náchylný na pády způsobené testovanou serverovou implementací. Pro vytváření podprocesů, jejich správu a komunikaci s nimi využívám v implementaci rozhraní, které nabízí Node.js. Primárně se jedná o modul `cluster` [39]. S takto vytvořenými podprocesy je možné komunikovat pomocí zpráv, které jsou zasílány skrze IPC kanál. Jedná se o velmi primitivní rozhraní založené na události `message`. Vytvořil jsem tedy zno-

vupoužitelný modul, který umožňuje pokročilejší komunikaci napříč procesy. Rozhraní tohoto modulu vrací instance `Promise` a umožňuje zavolat z jednoho procesu asynchronní kód v procesu jiném a počkat na jeho *resolve* nebo *reject*. Komunikace s podprocesy se díky tomuto modulu velmi zjednodušila.

Výzvou v definovaných požadavcích bylo směrování komunikace mezi testovanými implementacemi za pomocí hodnoty v hlavičce požadavku. Vzhledem k tomu, že řídící i testovaná komunikace probíhá po stejném portu, musí obě komunikace přijímat hlavní proces. V Node.js existuje mechanismus, pomocí kterého lze předat spojení z hlavního procesu do podprocesu, ovšem za předpokladu, že z tohoto spojení ještě nebyla přečtena žádná data [40]. To bohužel není v tomto případě možné, protože je potřeba z otevřeného spojení nejdříve přečíst hlavičky, na základě kterých hlavní proces zjistí, kam má požadavky směrovat. Tento problém jsem obešel tak, že podproces používá falešné spojení, které implementuje všechny náležitosti reálného spojení. Takto vytvořené spojení je možné od Node.js verze 8 předat instanci HTTP serveru vyvoláním události `connection` [41]. Instance serveru toto spojení poté zpracovává stejným způsobem jako spojení, které vzniklo například poslechem na portu.

Celá komunikace probíhá tak, že hlavní proces nejdříve přečte z nově otevřeného příchozího spojení první požadavek. Na základě hodnoty v hlavičce zjistí, kterému podprocesu je komunikace po tomto spojení určena. Pro toto spojení vygeneruje identifikátor, pod kterým si na spojení uloží referenci. Celý požadavek znovu zakóduje a odešle jej pomocí IPC kanálu odpovídajícímu podprocesu zároveň s identifikátorem spojení. Všechna další data, která po otevřeném spojení na server dorazí, už jen přeposílá stejným způsobem podprocesu. Pokud podprocesu dorazí data s identifikátorem spojení, které zatím nezná, vytvoří si pro něj falešné spojení. Všechna příchozí data určená tomuto spojení předává instanci falešného spojení. V případě vytváření nové instance falešného spojení ji zároveň předá pomocí události `connection` HTTP serveru, který vytvořila testovaná serverová implementace. Tímto způsobem se dostane příchozí komunikace do testované serverové implementace. Obdobným způsobem je zajištěna komunikace v opačném směru, kdy testovaná implementace zapisuje do falešného spojení. Tato data jsou pomocí IPC kanálu odesílána hlavnímu procesu, který je za pomoci přiloženého identifikátoru spojení odešle prostřednictvím odpovídajícího otevřeného spojení. Tímto způsobem je možné zpracovávat komunikaci nejen prostřednictvím HTTP a HTTPS protokolu, ale také pomocí WebSocket protokolu.

Nástroj směrování požadavků, ať již pomocí hodnoty v hlavičce, nebo unikátního portu, provádí zcela automaticky, bez nutnosti úprav testované serverové implementace. Tato vlastnost je zajištěna aplikací *patche* v běhovém prostředí testované implementace. Tento *patch* upraví implementaci tříd `http.Server` a `https.Server` a funkcí `http.createServer` a `https.createServer` tak, aby neposlouchaly na portu, který je jim předán, ale v případě směrování pomocí portu poslouchali na tomto portu, nebo v případě směrování pomocí hodnoty v hlavičce neposlouchali vůbec a příchozí spojení přijímala pouze skrze zmiňovanou událost `connection`.

Pro řízení životního cyklu testu podle scénáře podproces přidává do testovacího prostředí čtyři globální funkce: `scenario`, `setup`, `step` a `teardown`. Tyto funkce slouží k získání reference na funkce obalující kód scénáře a jednotlivých fází. Takto zachycené funkce lze zavolat vně testovacího prostředí a přesto vykonat jejich kód v testovacím prostředí.

Klientská část obsahuje pouze jednoduché rozhraní, které umožňuje vytvořit instanci integračního testu a nad ní volat jednotlivé fáze serverového testu: *setup*, *step* a

teardown. Funkce spouštějící tyto fáze vrací instance `Promise`. Tento způsob integračního testování bohužel klade určité nároky na testovanou klientskou implementaci. Ta musí být napsána takovým způsobem, aby bylo možné během testu upravit port odchozího požadavku nebo do něj vložit hlavičku. Tyto informace však většinou bývají definované v konfiguračním souboru, jehož hodnoty je možné během testu změnit, čímž je tento předpoklad splněn. Pro nastavení portu nebo hlavičky nabízí klient *callback* funkci, která tuto konfiguraci zjednodušuje. Její použití je demonstrováno v ukázkovém testu v další sekci.

Vzniklý nástroj jsem uvolnil jako open-source software pod MIT licencí. Jeho zdrojové soubory jsou k dispozici na serveru GitHub[1]. Nástroj jsem také publikoval ve formě NPM balíčku s názvem `vindaloo`[2]. Součástí repozitáře je také dokumentace v anglickém jazyce, automaticky vygenerovaná API dokumentace klienta a ukázkové testy.

Nástroj zatím vydávám v nulté verzi (0.1.0), protože jej chci důkladně vyzkoušet na některém z projektů, na kterých spolupracuji. Předpokládám, že díky tomu odhalím chyby, které se v implementaci určitě nacházejí, získám zpětnou vazbu a také nápady na vylepšení. Zároveň chci na nástroji dále pracovat, mám v plánu další vylepšení, chci jeho implementaci více pokrýt automatizovanými testy, připravit CI prostředí a rozšířit dokumentaci. Všechny tyto úkony jsou předpokladem pro uvedení první verze nástroje.

4.4 Příklad použití

Nyní se zaměřím na ukázkové použití vzniklého nástroje pro integrační test mezi klientem a serverem. Cílem je otestovat serverovou implementaci, kterou je možné vidět na obrázku 4.4. Tato implementace exportuje jedinou funkci `start`, která slouží pro vytvoření aplikace s využitím *frameworku* Koa. Dochází zde k zaregistrování jediného *endpointu* sloužícího k přihlášení uživatele. V obslužné funkci *endpointu* dochází k volání modulu `authenticator`. V tomto konkrétním případě je nežádoucí zahrnovat i tento modul do integračního testu, v testech tedy bude vyměněn za *test double*. Důležité také je, že na řádku 22 je zahájeno poslouchání na portu. V testovacím prostředí, ve kterém bude nástroj tuto implementaci spouštět, nebude poslouchání zahájeno a serveru budou směrovány požadavky z hlavního procesu.

Součástí integračního testu bude také klientská implementace, kterou je možné vidět na obrázku 4.5. Jedná se o jednoduchý modul poskytující funkci `login` sloužící pro přihlášení uživatele. V tomto modulu je pouze konfigurován požadavek, který je následně odeslán na server. Podle stavového kódu odpovědi je rozhodnuto o úspěšnosti přihlášení. V případě úspěšného přihlášení vrací funkce informace o přihlášeném uživateli. Důležité je, že se pro sestavení požadavku používá konfigurace z modulu sloužícího pouze pro tento účel. Díky tomuto přístupu je možné v integračním testu přidat do požadavku hlavičku nebo změnit port.

Serverovou část testu demonstruje obrázek 4.6. V tomto modulu jsou definovány dva scénáře. Scénář `login:successful` obsahuje ve své *setup* sekci vytvoření *spy* funkce `login`, která vrací testovací data. Díky tomu je možné se v testu integrace zaměřit pouze na části kódu odpovědné za vzájemnou integraci a například logiku přihlašování otestovat důkladněji pomocí jednotkového testu. Na řádku 11 pak dochází ke spuštění kódu z obrázku 4.4. V *teardown* sekci scénáře dochází k obnovení

[1] https://github.com/markatom/vindaloo
[2] https://www.npmjs.com/package/vindaloo

```
const authenticator = require('./authenticator');
const bodyParser = require('koa-bodyparser');
const Koa = require('koa');
const route = require('koa-route');

const start = () => {
    const app = new Koa();

    app.use(bodyParser());
    app.use(route.post('/login', (ctx) => {
        try {
            ctx.response.body = authenticator.login(
                ctx.request.body.email,
                ctx.request.body.password
            );

        } catch (error) {
            ctx.response.status = 401; // unauthorized
        }
    }));

    app.listen(12345); // port number does not matter
};

module.exports = {
    start,
};
```

Obrázek 4.4. Modul `app.js` serverové implementace.

```
import configuration from './configuration';

const login = async (email, password) => {
    const url = `http://${configuration.api.host}:${configuration.api.port}/login`;

    const response = await fetch(url, {
        method: 'POST',
        headers: Object.assign(
            {'Content-Type': 'application/json'},
            configuration.api.extraHeaders
        ),
        body: JSON.stringify({email, password}),
    });

    if (response.status === 401) { // unauthorized
        throw new Error('Invalid email or password.');
    }

    return await response.json();
};

export {
    login,
};
```

Obrázek 4.5. Modul `authenticator.js` klientské implementace.

původní implementace `login`. Ve scénáři `login:failed` funkce `login` vyhazuje výjimku, která signalizuje chybné přihlašovací údaje.

Klientská část testu je na obrázku 4.7. Nejdříve je na řádku 5 vytvořena instance klienta sloužící pro řízení integračního testu. V konstruktoru je zvoleno směrování požadavků pomocí hlavičky a také zaregistrovaná pomocná funkce, kterou klient volá vždy se zahájením integračního testu. Tato funkce s každým testem, v závislosti na způsobu směrování požadavků, obdrží název a hodnotu hlavičky nebo číslo portu. V tomto případě je upravena konfigurace klientské aplikace. Dále následuje běžný test, zde konkrétně využívající *framework* Jest. Na řádcích 13 a 28 dochází k vytvoření instance testu podle předaného scénáře. Test je pak ukončen na řádcích 24 a 38. Přímo v testu se nachází pouze volání testované implementace a kontrola výstupu.

```
const app = require('../../src/server/app');
const authenticator = require('../../src/server/authenticator');

scenario('login:successful', () => {
    setup(() => {
        jest.spyOn(authenticator, 'login').mockImplementation(() => ({
            id: 42,
            name: 'Tomáš Markacz',
        }));

        app.start();
    });

    teardown(() => {
        jest.restoreAllMocks();
    });
});

scenario('login:failed', () => {
    setup(() => {
        jest.spyOn(authenticator, 'login').mockImplementation(() => {
            throw new Error('Invalid password.');
        });

        app.start();
    });

    teardown(() => {
        jest.restoreAllMocks();
    });
});
```

Obrázek 4.6. Modul s definicí scénářů pro testování serverové implementace.

Díky tomuto nástroji je možné s minimálním úsilím připravit integrační test, který spolehlivě otestuje přímou interakci klienta a serveru podle předem domluveného scénáře. Nástroj nabízí pokročilé funkce pro tvorbu *test doubles* a kontrolu výstupů serverové implementace. Lze proto otestovat libovolné situace, které mohou v integraci vzniknout.

Zdrojové kódy nástroje, jeho dokumentace a ukázkový test jsou k dispozici na přiloženém CD.

```
import * as authenticator from '../../src/client/authenticator';
import {Client} from 'vindaloo';
import configuration from '../../src/client/configuration';

const client = new Client({
    bindingType: 'port',
    setupBinding: (port) => {
        configuration.api.port = port;
    },
});

describe('successful user authentication', () => {
    const test = client.createTest('login:successful');

    beforeAll(() => test.setup());
    it('returns authenticated user', async () => {
        const user = await authenticator.login('tomas@markacz.com', 'foobar');

        expect(user).toEqual({
            id: 42,
            name: 'Tomáš Markacz',
        });
    });
    afterAll(() => test.teardown());
});

describe('failed user authentication', () => {
    const test = client.createTest('login:failed');

    beforeAll(() => test.setup());
    it('throws an error', async () => {
        const promise = authenticator.login('tomas@markacz.com', 'foobar');

        await expect(promise).rejects.toEqual(
            new Error('Invalid email or password.')
        );
    });
    afterAll(() => test.teardown());
});
```

Obrázek 4.7. Klientský integrační test používající serverové scénáře.

Závěr

Prvním cílem této práce bylo popsat existující způsoby testování client-server integrací v kontextu mobilních a webových aplikací a shromáždit relevantní zdroje zabývající se touto problematikou. Pro shromáždění relevantních zdrojů jsem se nejdříve musel seznámit s obecnou problematikou softwarového testování integrací. Na základě shromážděných zdrojů jsem poté existující způsoby obecně charakterizoval, rozdělil do dvou skupin a následně detailněji popsal. Tyto způsoby jsem porovnal pomocí šesti kritérií, jejichž charakteristiku a přínosy pro testování jsem důkladně popsal. Výstupem porovnání je přehledná tabulka se slovním ohodnocením pro každý ze způsobů testování dle jednotlivých kritérií. Z výsledné tabulky bylo patrné, že neexistuje způsob testování, který by byl použitelný pro všechny situace. Z tabulky také vyplývá, že pro naplnění všech zvolených kritérií je žádoucí popsané způsoby testování vhodně kombinovat.

Druhým cílem této práce bylo vyhledat, vyzkoušet a popsat existující nástroje umožňující popsané druhy testování. Záměrem bylo zaměřit se na testování integrace klienta ve formě webové aplikace napsané v JavaScriptu a webového serveru využívajícího Node.js. Nejdříve jsem na základě požadavků daných konkrétním způsobem testování nalezl vhodné nástroje. Tyto nástroje jsem následně vyzkoušel, abych ověřil, zda jsou použitelné pro účely testování integrací client-server. Vyzkoušené nástroje jsem popsal, na ukázkových testech předvedl jejich použití a shrnul jejich výhody a nevýhody. Zároveň jsem zmínil poznatky, které jsem tímto získal.

Posledním cílem této práce bylo navrhnout a implementovat nástroj, který umožní testovat integrace client-server podle navrženého způsobu testování. Nejdříve jsem definoval požadavky na nástroj, poté navrhl schéma komunikace a provedl prototypování za účelem vyzkoušení některých mechanismů potřebných pro jeho implementaci. Na základě těchto kroků jsem nástroj úspěšně navrhl a implementoval v jazyce JavaScript. Navržený způsob testování demonstruji za využití vytvořeného nástroje na ukázkovém testu. Nástroj je zdokumentován včetně automaticky generované dokumentace a publikován ve formě open-source softwaru.

Na vzniklém nástroji budu dále aktivně pracovat a rozšiřovat jej o další funkce. Navržený způsob testování chci aktivně propagovat a pomáhat s implementací integračních testů využívajících vytvořený nástroj. Vzhledem k tomu, že se jedná o obecně navržený princip testování, který je realizován pomocí přesně definovaného schématu komunikace a implementací v konkrétním jazyce, je možné na tuto práci navázat vývojem dalších implementací v jiných programovacích jazycích, a tím tento způsob testování rozšířit na další platformy.

Literatura

[1] Glenford Myers. *The art of software testing*. John Wiley & Sons, 2012. ISBN 1118031962.

[2] Inc. Base36. Automated vs. manual testing: The pros and cons of each. [Online], březen 2013. Cit. 31. 3. 2018.
Dostupné z: http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/.

[3] Carey Wodehouse. Upwork: Manual testing vs. automated software testing. [Online]. Cit. 31. 3. 2018.
Dostupné z: https://www.upwork.com/hiring/development/manual-testing-vs-automated-testing/.

[4] Tutorials Point. Software testing - levels. [Online]. Cit. 1. 4. 2018.
Dostupné z: https://www.tutorialspoint.com/software_testing/software_testing_levels.htm.

[5] Eric Elliott. Sitepoint: Javascript testing: Unit vs functional vs integration tests. [Online], duben 2016. Cit. 1. 4. 2018.
Dostupné z: https://www.sitepoint.com/javascript-testing-unit-functional-integration/.

[6] Jani Hartikainen. Codeutopia: What are unit testing, integration testing and functional testing? [Online], duben 2015. Cit. 1. 4. 2018.
Dostupné z: https://codeutopia.net/blog/2015/04/11/what-are-unit-testing-integration-testing-and-functional-testing/.

[7] Mike Wacker. Google testing blog: Just say no to more end-to-end tests. [Online], duben 2015. Cit. 1. 4. 2018.
Dostupné z: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

[8] Steve Hostettler. Fakes, stubs, dummy, mocks, doubles and all that... [Online], květen 2014. Cit. 2. 4. 2018.
Dostupné z: http://www.hostettler.net/blog/2014/05/18/fakes-stubs-dummy-mocks-doubles-and-all-that/.

[9] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. ISBN 0131495054.

[10] Michal Lipski. Pragmatists: Test doubles – fakes, mocks and stubs. [Online], březen 2017. Cit. 2. 4. 2018.
Dostupné z: https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da.

[11] SmartBear Software. Qacomplete: The software testing lifecycle - an introduction. [Online]. Cit. 2. 4. 2018.
Dostupné z: https://qacomplete.com/resources/articles/the-software-testing-lifecycle-an-introduction/.

[12] Techopedia Inc. System integration. [Online]. Cit. 24 5. 2018.
Dostupné z: https://www.techopedia.com/definition/9614/system-integration-si.

[13] Alex Berson. *Client/server architecture*. McGraw-Hill, 1996. ISBN 0070056641.

[14] Inflectra. Rapise: What is api testing? [Online]. Cit. 10. 4. 2018.
https://www.inflectra.com/rapise/highlights/api-testing.aspx.

[15] Stephen Feloney. Blazemeter: Functional api testing - how to do it right. [Online], březen 2018. Cit. 9. 4. 2018.
https://www.blazemeter.com/blog/functional-api-testing-how-to-do-it-right.

[16] Tom Hombergs. Reflectoring: 7 reasons to choose consumer-driven contract tests over end-to-end tests. [Online], listopad 2017. Cit. 8. 4. 2018.
Dostupné z: https://reflectoring.io/7-reasons-for-consumer-driven-contracts/.

[17] SmartBear Software. Soapui: Api testing 101: Learn the basics. [Online]. Cit. 9. 4. 2018.
https://www.soapui.org/resources/api-testing/article/api-testing-101.html.

[18] Margaret Rouse. Techtarget: What is sandbox? [Online]. Cit. 8. 5. 2018.
Dostupné z: https://searchsecurity.techtarget.com/definition/sandbox.

[19] SmartBear Software. What is an api sandbox? [Online]. Cit. 8. 5. 2018.
https://smartbear.com/learn/api-testing/what-is-an-api-sandbox/.

[20] SmartBear Software. Soapui: Api mocking. [Online]. Cit. 8. 4. 2018.
Dostupné z: https://www.soapui.org/resources/mocking/article/what-is-api-mocking.html.

[21] Ian Robinson. Consumer-driven contracts: A service evolution pattern. [Online], červen 2006. Cit. 8. 4. 2018.
Dostupné z: https://martinfowler.com/articles/consumerDrivenContracts.html.

[22] Screenster. What is end-to-end testing and is there a smarter way to automate it? [Online], říjen 2017. Cit. 9. 4. 2018.
Dostupné z: https://screenster.io/end-to-end-testing/.

[23] Rhys Evans. Fetch-mock. [Software].
Domovská stránka: http://www.wheresrhys.co.uk/fetch-mock/.

[24] Mozilla. Mdn: Fetch api. [Online], březen 2018. Cit. 15. 4. 2018.
Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[25] Tim Perry. Mockttp. [Software].
Domovská stránka: https://github.com/pimterry/mockttp.

[26] TJ Holowaychuk. Supertest. [Software].
Domovská stránka: https://github.com/visionmedia/supertest.

[27] Vance Lucas. Frisby. [Software].
Domovská stránka: https://www.frisbyjs.com.

[28] Apiary Inc. Api blueprint. [Software].
Domovská stránka: https://apiblueprint.org.

[29] JSON Schema Organization. Json schema. [Software].
Domovská stránka: http://json-schema.org.

[30] OpenAPI Initiative. The openapi specification. [Software].
Domovská stránka: https://github.com/OAI/OpenAPI-Specification.

[31] Paul Bryan and Kris Zyp. Json reference. [Software].
Domovská stránka: https://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03.

[32] Oracle. Apiary. [Software].
Domovská stránka: https://apiary.io.

[33] Apiary. Dredd. [Software].
Domovská stránka: https://dredd.readthedocs.io.

[34] BBVA-labs. Consumer driven contract tests. [Online], listopad 2017. Cit. 22. 4. 2018.
https://www.bbva.com/en/consumer-driven-contract-tests/.

[35] Pact Foundation. Pact. [Software].
Domovská stránka: https://docs.pact.io.

[36] Pact Foundation. Hosted pact broker. [Software].
Domovská stránka: https://pact.dius.com.au.

[37] Selenium team. Selenium. [Software].
Domovská stránka: https://www.seleniumhq.org.

[38] Facebook Inc. Jest. [Software].
Domovská stránka: https://facebook.github.io/jest/.

[39] Node.js Foundation. Node.js documentation. [Online]. Cit. 20. 5. 2018.
https://nodejs.org/api/cluster.html.

[40] Node.js Foundation. Node.js documentation. [Online]. Cit. 20. 5. 2018.
Dostupné z: https://nodejs.org/api/child_process.html
#child_process_example_sending_a_socket_object.

[41] Node.js Foundation. Node.js documentation. [Online]. Cit. 20. 5. 2018.
Dostupné z: https://nodejs.org/docs/latest-v8.x/api/http.html
#http_event_connection.

Příloha A
Seznam použitých zkratek

API	Application Programming Interface
CDC	Consumer-Driven Contracts
CI	Continuous Integration
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MSON	Markdown Syntax for Object Notation
NPM	Node Package Manager
PHP	PHP: Hypertext Preprocessor
REST	Representational State Transfer
RFC	Request For Comments
SSL	Secure Sockets Layer
URL	Uniform Resource Locator
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Příloha B
Obsah přiloženého CD

- `implementation/` – Implementace nástroje
  - `docs/` – Automaticky generovaná dokumentace API
  - `examples/` – Ukázkové testy
  - `src/` – Zdrojové soubory
  - `readme.md` – Dokumentace ve formátu Markdown
- `thesis/` – Tato diplomová práce
  - `sources/` – Zdrojové soubory ve formátu TEX
  - `thesis.pdf` – Tisknutelná podoba
- `readme` – Popis obsahu CD