Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Vývoj webových aplikací řízený testy

Bakalářská práce

Autor: Ondřej Habr
Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Daniela Ponce Ph.D.

Hradec Králové duben 2020

Prohlášení:

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

vlastnoruční podpis

V Hradci Králové dne 30.4.2020 Ondřej Habr

Poděkování:

Děkuji vedoucí bakalářské práce Mgr. Deniele Ponce, PhD. za metodické vedení práce, cenné rady, připomínky a doporučení při tvorbě této práce.

Anotace

Tato práce se zabývá použitím metody vývoje řízeného testy při tvorbě webové aplikace za použití vhodného testovacího nástroje. Teoretická část popisuje vývoj řízený testy a různé druhy testů, které lze během vývoje provádět včetně příkladu ve dvou nejpoužívanějších programovacích jazycích. Praktická část popisuje použití testovacího nástroje během vývoje aplikace pro vybrané druhy testů na různých částech aplikace. V samotném závěru jsou popsány výhody a přínosy testování v průběhu vývoje a důvody proč by se testování nemělo při vytváření aplikace opomíjet.

Annotation

Title: Test driven development of web applications

The bachelor thesis deals with the use of test-driven development method when building a web application using a suitable testing tool. The theoretical part describes test-driven development and various types of tests that can be performed during development, including an example in the two most used programming languages. The practical part describes the use of the testing tool during application development for selected types of tests on different parts of the application. The very conclusion describes the advantages and benefits of testing during development and reasons why testing should not be neglected when creating an application.

Obsah

Seznam zdrojových kódů

1 Úvod

Cílem práce je porovnat metodu vývoje řízeného testy s běžným postupem a zhodnotit tuto metodu praktickým vývojem webové aplikace.

Zásadním prvkem je vytvoření aplikace touto metodou a použití vhodného nástroje pro testování aplikace v programovacím jazyce PHP vyvíjené ve frameworku Nette, který byl zvolen díky zkušenostem z praxe a prací na různých aplikacích kde byl tento framework používán a také pro to, že je k dispozici testovací nástroj od stejných vývojářů, což umožňuje vysokou kompatibilitu. V závěru bude vhodné porovnat čitelnost a porozumění kódu, čas, který zabere otestování aplikace pokud se provede úprava nějaké funkce a porovnání času, jenž je potřeba k vytvoření metody, v závislosti na metodě vývoje.

Především při tvorbě rozsáhlých webových aplikací je pro stabilní chod a snadnou rozšiřitelnost důležité provádět testy a vzniklé chyby v průběhu vývoje, či pozdějších úprav, odhalit včas v nejlepším případě před samotným nasazením chybného kódu na ostrý web.

Je tedy dobré již od samotného začátku vytváření aplikace zahrnout testování jako součást projektu. Pro velké projekty, ale není problém i pro menší, je dobré použití automatického testování, které zvládne provádět několik testů najednou a za velmi krátkou dobu tak projde desítky, či stovky testů.

První část práce se zabývá teorií testů, testování aplikací a vývojem řízeným testy. Je zde popsán proces a postupy vývoje řízeného testy. Dále jsou rozebrány jednotlivé druhy testů, které lze použít pro dosažení co nejefektivnějších výsledků. V poslední části je popsán nejrozšířenější nástroj pro psaní jednotkových testů v jazyce PHP PHPUnit.

V druhé části bakalářské práce je stručně popsána vyvíjená aplikace a její požadavky a poté již vlastní vývoj aplikace řízený testy a praktické použití jednotlivých druhů testů na jednotlivé součásti aplikace.

V závěru práce jsou zmíněny poznatky z vývoje aplikace formou TDD a rozdíly oproti nasazení testů až po dokončení samotného vývoje.

2 Teoretická část

2.1 Co je to Test Driven Development?

Test Driven Development (dále TDD) je způsob vývoje softwaru. Čistý kód, který funguje, je hodnotným cílem TDD z mnoha důvodů: [1]

- Je to předvídatelný způsob vývoje. Víte, kdy jste skončili, aniž by bylo nutné starat se o dlouhou cestu s bugy
- Dává možnost naučit se všechny lekce, které kód nabízí. Pokud se řeší problém pouze prvním způsobem, který člověka napadne, pak už nikdy nebude mít čas přemýšlet o druhém, lepším řešení
- Zlepšuje to život uživatelům, kteří tento software následně používají

Ale jak dosáhnout čistého kódu, který funguje? Mnoho věcí odvádí od čistého kódu, a dokonce i od kódu, který funguje. Aby se předešlo zbytečným problémům, používá se řízení vývoje s automatickým testováním, jde o tzv. vývoj řízený testy (test-driven development).

2.1.1 Principy vývoje TDD

Při vývoji řízeném testy se:

- Píše nový kód pouze pokud automatický test skončí chybou
- Eliminují duplicity

Toto jsou dvě jednoduchá pravidla, která vytvářejí komplexní individuální a skupinová chování, jako například následující:

- Musí se navrhovat organicky, s běžícím kódem poskytujícím zpětnou vazbu mezi rozhodnutími
- Je potřeba psát své vlastní testy, protože nelze čekat několikrát za den až test napíše někdo jiný
- Vývojové prostředí musí zajistit rychlou reakci na malé změny
- Návrhy se musí skládat z mnoha vysoce soudržných, volně spojených komponent, aby bylo testování snadné

Tato tři pravidla znamenají pořadí úkolů programování:

1. Červená – napsání malého testu, který nefunguje a možná se zpočátku ani nezkompiluje
2. Zelená – udělat rychle test funkční
3. Refaktorizace – Odstranění všech duplicit vytvořených pouze za účelem zprovoznit test

Za předpokladu, že takový programovací styl je možný, může být dále možné dramaticky snížit množství defektů kódu a učinit předmět práce jasným všem zúčastněným. Pokud ano, pak tento kód, který vyžaduje neúspěšné testy, má také sociální důsledky.[1]

- Pokud lze množství chyb dostatečně snížit, pak může práce přejít z reaktivní na proaktivní
- Pokud se počet nepříjemných událostí sníží, projektoví manažeři mohou přesněji určit kdy zapojit skutečné zákazníky do každodenního vývoje
- Mohou-li být témata technických rozhovorů dostatečně objasněna, mohou softwaroví vývojáři spolupracovat každou minutu, na rozdíl od každodenní nebo týdenní spolupráce
- Opět, pokud může být množství chyb dostatečně sníženo, lze mít denně k dispozice software s novými funkcemi, což povede k novým obchodním vztahům se zákazníky

Vývojový cyklus:

1. Napsat test

První krok při tvorbě nové komponenty podle TDD je vytvoření testu, který ověřuje, jestli kód, který bude zajišťovat funkcionalitu, dělá to, co se od něj očekává. Takovéto základní napsání testů je vlastně definicí požadavků na onu funkcionalitu. Vývojář se ujistí napsáním testu, že přesně chápe požadavky a funkcionalitu na testovanou komponentu. Tímto se během psaní samotného kódu eliminuje odchýlení se od původního záměru a cíle. Testy mohou být psány podle use-case diagramů, user-stories, nebo jiných materiálů. Tento krok se odlišuje od přístupů, kde se testy píší až na konci, jelikož nutí vývojáře, aby se zaměřil na požadavky už před psaním samotného kódu. Tento rozdíl se může jevit jako bezvýznamný, nicméně je podstatou odlišností.[1]

2. Spustit testy a ujistit se, že žádný neprojde

Jakmile jsou napsány nové testy, jsou spuštěny a všechny končí neúspěchem, jelikož zatím neexistuje žádný kód, který by zajistil jejich úspěšné dokončení. Tento krok funguje také jako jakási sebekontrola, která eliminuje možnost, že by nová funkcionalita byla dokončena dřív, než by začal vůbec někdo s její implementací. Pakliže by v tomto kroku prošly jako splněné, znamenalo by to, že jsou špatně napsané, nebo že budou pokaždé splněny a ztrácí tím tak svůj smysl.[1]

3. Napsat vlastní kód

Poté co jsou všechny nově napsané testy neúspěšně dokončeny, přichází na řadu psaní kódu. V tomto kroku se bude jednat o rychlé přírůstky, jejichž cílem je pouze zajistit to, aby při dalším spuštění nové testy prošly. Nedává se zde za cíl, aby byl kód hned co možná nejefektivnější a nejelegantnější. Efektivnost a elegance kódu je záležitostí dalších kroků, jedná se zde opravdu jen o úspěšné splnění testů.[1]

4. Kód automatickými testy prochází

Jestliže kód, který byl napsaný v předcházejícím kroku, dosáhne úspěšného splnění testů, pak to také znamená, že splňuje definované požadavky. Pokud jsou všechny testy splněné, lze přejít k poslednímu kroku.

5. Refaktorizace

Poslední krok je refaktorizace kódu. Předmětem tohoto kroku je efektivita a elegance dříve napsaného kódu. Refaktorizace kódu může také probíhat automaticky za pomocí různých nástrojů. Eliminují se duplicity v kódu a zajišťuje se úprava do co možná nejpřijatelnější podoby. Opakované spouštění automatických testů umožňuje v tomto kroku neustále ověřovat, že úpravami dříve napsaného kódu nevznikly chyby ve funkcionalitě a že se refaktorizace nestane spíše kontraproduktivní.[1]

Opakování

Přechozí kroky se pak neustále opakují. Na základě různých vstupů (use-case diagramy, user stories, …) je znovu definována funckionalita, která je vyjádřena pomocí automatických testů, které při prvním spuštění opět neprojdou. Poté je znovu za účelem splnění testů napsán kód, a pokud jsou testy splněny, kód se refaktoruje. Takovéto přírůstky by měly být co nejmenší, zhruba 1–10 úprav mezi opětovným testováním. Jestliže by skončil některý z předchozích testů neúspěchem, znamenalo by to návrat o pár kroků zpět a úpravu kódu.[1]

2.1.2 Přínosy

Používání automatických testů během vývoje přináší tu výhodu, díky které je možné opětovné spouštění velkého množství testů, které znovu a znovu ověřují všechny předem definované funkcionality a požadavky. Jestliže by nějaký zásah do kódu způsobil někde nějakou chybu, automatické testy chybu odhalí.

Podobná situace může nastat i při změně požadavků na dosavadní funkcionalitu. Jestliže je potřeba někde něco upravit, stačí upravit nebo napsat nové testy a následně i kód. Pokud by některé změny vedly k nefunkčnosti v jiných částech

programu, znovu lze jednoduše a rychle zjistit, kde je chyba. Protože tento přístup skládá program z malých částí, mnohem snadněji se lze vypořádat s různými vnějšími zásahy do návrhu programu.[10]

2.1.3 Omezení

Omezení tohoto přístupu spočívá v tom, že se v testech mohou nacházet chyby stejně tak jako v kódu. Jestliže jsou chyby časté, může to směřovat k demotivaci a zanevření na tento přístup. Testy také zvyšují množství kódu, který je potřeba napsat. Zkušenosti také ukazují, že ze začátku vede tento přístup k dočasnému snížení produktivity. Omezením je také například špatná testovatelnost některých částí programů, jako například uživatelské rozhraní. Tento přístup nemůže nahradit všechny testy. Testy používané zde jsou slabé a na jejich základě nelze zapisovat například akceptační protokoly nutné pro vypuštění nebo předání programu.[11]

2.2 Druhy testů

Při tvorbě aplikací se provádí testování na několika úrovních. U jednotlivých testů je možné kontrolovat celý systém nebo se může kontrolovat zvolená část funkce. Existuje pět úrovní testování. Každá úroveň nahlíží na testování jiným způsobem. Tyto úrovně vycházejí z postupu vývoje aplikace. V této práci je úroveň nazývána jako druh. [7]

2.2.1 Testování programátorem

Testování programátorem neboli Assembly tests se provádí tak, že napsaný zdrojový kód je zkontrolován programátorem. Hlavní pravidlo je, že by kód neměl kontrolovat jen člověk, který ho vytvořil. Tuto kontrolu mnoho vývojářů přehlíží. Většinou však stačí, když si kód projde další člověk a ten vidí, jestli by šel změnit, nebo zda je napsaný správně. Mohla by nastat situace, kdy by byl špatný kód předán testerovi, on by ho začal testovat a hned by narazil na chybu. Tu by nahlásil zpět autorovi, který by ji opravil a znovu předal kód testerovi. Takovýto proces je časově a finančně náročný, tudíž by měla být prováděna kontrola jiným programátorem.[7]

2.2.2 Testování jednotek

Tyto testy, již podle názvu, slouží k testování menších jednotek zdrojového kódu. Obecně lze říct, že programátor napíše kód (metodu) a následně pro ni napíše test. V případě TDD se nejprve napíše test a až poté se píše vlastní kód. Test by měl testovat kód za normálních podmínek, ale i v mimořádných situacích. Např. co se stane, dostane-li metoda do parametru *null*.[2]

S určitou podobou unit testů se lze setkat u každého vývoje. Jedná se i o situace kdy si programátor vypíše do konzole informace o proměnných a zkontroluje, zda mají správné hodnoty. V podstatě takto testuje svůj kód. Tento způsob je ale dost pomalý, neefektivní, nepřehledný, není automatický (hodnota se musí pokaždé kontrolovat ručně) a nezachovatelný – výpis je nutné poté smazat nebo dát do komentáře (kód je tím ještě méně přehledný). Pro vývoj větších projektů je jednoznačně nedostačující.[2]

Prvním důležitým krokem je oddělit testy od samotného zdrojového kódu. Pro každou vytvořenou metodu, která bude testována, vytvořit speciální metodu v odlišném balíčku a v ní testovat její funkčnost. Pro jednotlivé jazyky existují různé testovací frameworky, které podporují i samotné IDE a celý proces testování usnadňují. Pro PHP jsou to např. frameworky PHPUnit nebo Nette Tester a pro Javu existuje např. framework JUnit nebo TestNG.[12]

V následujícím přikladu je jednoduchá třída, která provádí dělení dvou zadaných čísel. První třída je psaná v jazyce PHP a druhá pro pozdější porovnání testů v Javě.

```php
class Calculator {

    public static function divide(int $divident, int $divisor): int
    {
        if (is_null($divident) || is_null($divisor)) {
            throw new IllegalArgumentException("cannot set value to null");
        }
        if ($divisor === 0) {
            throw new DivideByZeroException("cannot be divided by zero");
        }

        return $divident / $divisor;
    }
}
```

Zdrojový kód 1.1: Unit testy – testovací třída v PHP

```php
class Calculator {

    public static int divide(int divident, int divisor)
    {
        if (dividend == null || divisor == 0) {
            throw new IllegalArgumentException("cannot set value to null");
        }
        if (divisor == 0) {
            throw new DivideByZeroException("cannot be divided by zero");
        }

        return divident / divisor;
    }
}
```

Zdrojový kód 1.2: Unit testy – testovací třída v Javě

Následně jsou k těmto třídám vytvořené testy. Pro jednoduché testování v PHP není ani potřeba vytvářet třídu, ale lze použít pouze PHP skript. Test je důležité vytvořit pro každou (důležitou) funkcionalitu.

Tyto testy navíc mohou sloužit jako dokumentace. Pokud programátor neporozumí přímo testované třídě, přečte si testy a pochopí ji. Testy totiž popisují každý důležitý aspekt dané třídy.

```php
<?php
require __DIR__ . "/Calculator.php";

$calculator = new Calculator();

try {
    $calculator->divide(null, null);
    echo "Null settings test failed - should throw a IllegalArgumentException!";
} catch (IllegalArgumentException $ex) {
    echo "Null settings test passed!";
} catch (Exception $ex) {
    echo "Null settings test failed - threw another exception!";
}

try {
    $calculator->divide(10, 0);
    echo "Divide by zero settings test failed - should throw a
DivisionByZeroException!";
} catch (DivideByZeroException $ex) {
    echo "Divide by zero settings test passed!";
} catch (Exception $ex) {
    echo "Divide by zero settings test failed - threw another exception!";
}

try {
    $calculator->divide(10, 2);
    echo "Divide test passed!";
} catch (Exception $ex) {
    echo "Divide test failed!";
}
```

Zdrojový kód 2.1: Unit testy – test v PHP

```
class CalculatorTest {

  private Calculator;

  private void prepare() {
        calculator = new Calculator();
  }


  public void testNullSettings() {
    try {
        calculator.divide(null, null);
        System.out.println("Null settings test failed - should throw a
IllegalArgumentException!");
    } catch (IllegalArgumentException ex) {
        System.out.println("Null settings test passed!");
    } catch (Exception ex) {
        System.out.println("Null settings test failed - threw another exception");
    }
  }


  public void testZeroSettings() {
    try {
        calculator.divide(10, 0);
        System.out.println("Divide by zero settings test failed - should throw a
DivisionByZeroException!");
    } catch (DivideByZeroException ex) {
        System.out.println("Divide by zero settings test passed!");
    } catch (Exception ex) {
        System.out.println("Divide by zero settings test failed - threw another
exception!");
    }
  }


  public void testNumbersSettings() {
    try {
        calculator.divide(10, 2);
        System.out.println("Divide test passed!");
    } catch (Exception $ex) {
        System.out.println("Divide test failed!");
    }
  }
}
```

Zdrojový kód 2.2: Unit testy – test v Javě

2.2.3 Integrační testování

V momentě, kdy programátor dokončí testy, přichází na řadu testovací tým. O integrační testy se tedy nestará programátor, ale především testovací tým. Integrační testy se někdy mohou označovat jako „testy vnitřní integrace." Je zapotřebí ověřit správnost komunikace mezi jednotlivými komponentami v aplikaci. [7]

Kromě komunikace mezi komponentami lze ověřovat také integraci mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. Tato fáze se tak zabývá testováním integrace doposud jednotlivě testovaných částí. Začíná se tak testovat integrace mezi dvěma komponentami a následně se přidávají další. Tyto testy mohou být manuální i automatické.[7]

Ve většině testovacích postupů je úroveň integračního testování určitým způsobem obsažena. U menších projektů se ale na tyto testy klade velmi malý důraz. Lze to logicky vysvětlit. Z testovacího cyklu lze integrační testy úplně vynechat. Na konečnou funkčnost softwaru to přitom nebude mít vliv, ovšem za předpokladu, že následující úrovně testování budou provedeny bezchybně.[14]

Pokud by se během integračních testů objevila nějaká chyba, tato chyba se projeví zcela určitě i během dalších úrovní testování. Během testování ovšem platí pravidlo: „čím dříve se chyba objeví, tím méně práce stojí její oprava." Z tohoto důvodu mají integrační testy smysl, ovšem jejich použití nelze nikterak přeceňovat.[14]

2.2.4 Systémové testování

Systémové testování je druh testování softwaru, který je prováděn na hotovém integrovaném řešení za účelem ověření vyhodnocení systému s odpovídajícími požadavky.[3]

V systémovém testování se jako vstup berou komponenty úspěšně prošlé integračními testy. Systémové testování detekuje chyby jednak v integrovaných jednotkách, ale i v celém systému. Výsledkem systémového testování je pozorované chování komponenty nebo systému v průběhu testování. Systémové testy zásadně provádí testovací tým, který je nezávislý od vývojářského týmu. Tím se zajistí nezávislé testování systému.[15]

Systémové testování se provádí v následujících krocích:

- Nastavení prostředí pro testování
  Vytvoření testovacího prostředí pro lepší kvalitu testování
- Vytvoření Test Case
  Generování případů pro testování
- Vytvoření testovacích dat
  Generování dat pro testování
- Spuštění Test Case
  Po vygenerování testovacích situací a testovacích dat jsou spuštěny Test Cases
- Hlášení chyb
  Jsou detekovány chyby v systému
- Regresivní testování
  Provádí se k testování vedlejších účinků procesu testování
- Oprava chyb
  V tomto kroku jsou opraveny všechny chyby
- Nový test
  Pokud nebyl test úspěšný je proveden nový test
  [3]

2.2.4.1 Druhy systémového testování

Systémové testování zahrnuje většinu druhů testování, protože jsou v něm obsaženy všechny hlavní typy testování. Nicméně zaměření na typy testování se může lišit v závislosti na produktu, organizačních procesech, časové ose a požadavcích.[4]

- Testování funkčnosti: Pro ověření funkčnosti produktu dle definovaných požadavků, v rámci možností systému
- Testování obnovitelnosti: Ke zjištění, jak se systém vzpamatuje při různých vstupních chybách a jiných situacích vedoucích k chybě

- Testování interoperability: Ujištění, zda systém správně operuje s produkty třetích stran
- Testování výkonu: Měření výkonu systému za různých podmínek z hlediska výkonnostních charakteristik
- Testování škálovatelnosti: Zajištění různých možností škálovatelnosti, jako je škálování uživatelů, geografické škálování nebo škálování zdrojů
- Testování spolehlivosti: Ověření, že systém může být provozován po delší dobu, aniž by došlo k poruchám
- Testování regrese: Kontrola stability systému při průchodu integrací různých subsystémů a údržbou
- Testování dokumentace: Ověření, zda je uživatelská příručka a ostatní nápovědy v pořádku a použitelné
- Testování bezpečnosti: Kontrola, že systém neumožní neautorizovaný přístup k datům a zdrojům
- Testování použitelnosti: Ujištění, že je systém jednoduchý k použití
  [4]

2.2.4.2 Příklad systémového testování

Výrobce automobilu nevyrábí auto jako celek. Každá jeho součást je vyráběna samostatně, jako např. sedadla, řízení, podvozek. Po vyrobení každé položky se nezávisle otestuje, zda funguje tak jak má – jednotkové testování.[4]

Poté, co je každá část sestavena s jinou částí, je tato kombinace zkontrolována, pokud sestavení nevytvořilo žádný vedlejší účinek na funkčnost jednotlivých komponent a zda obě komponenty spolupracují dle očekávání – integrační test.[4]

Poté co jsou všechny části sestaveny a auto je připraveno, ve skutečnosti ještě není zcela připraveno.

Celý vůz musí být zkontrolován z různých hledisek podle definovaných požadavků, jako je to, zda jezdí hladce, brzdí, řadí a ostatní funkcionality fungují správně, auto neukazuje žádné známky únavy po souvislé jízdě v délce pěti kilometrů, barva auta je schválena, auto může jet po všech typech vozovky, celé toto úsilí o testování se nazývá systémové testování a nemá nic společného s testováním integrace.[4]

2.2.5 Akceptační testování

Akceptační testování je poslední fáze procesu testování softwaru. Během tohoto testování zkoušejí skuteční uživatelé tohoto softwaru, jestli je schopen poradit si se všemi požadavky v reálných scénářích. Je to jeden z konečných a kritických postupů, který musí proběhnout, než bude nově vyvinutý software uveden na trh.

Přestože akceptační testování je nezbytné, většinou jej nelze provést, dokud není aplikace z velké části dokončena.[5]

V akceptačním testování by měli být zahrnutí skuteční uživatelé, pro které je software určen.

Běžně se akceptační testování skládá ze čtyř kroků, může se však lišit v závislosti na tom, zda se aplikace dodává jednomu zákazníkovi, nebo zda se jedná o volně prodejný software, který si může kdokoli zakoupit.[5]

Nejprve se připraví scénáře, které vytvořil zákazník společně s dodavatelem. Testy se provádí u zákazníka v testovacím prostředí. Nalezené neshody mezi aplikací a specifikací jsou nahlášeny zpět vývojovému týmu. Po opravení chyb je kód nasazen na prostředí u zákazníka. V tento okamžik je velmi důležité definovat si dopředu jakým způsobem bude probíhat hlášení chyb od zákazníka a jak umožnit opravení těchto chyb v co nejkratším čase.[16]

2.3 Vybrané nástroje

Následující nástroj je vybrán jako alternativa k nástroji Nette Tester, který je použit a popsán v rámci praktické části. Jedná se o nejrozšířenější testovací nástroj pro jazyk PHP, se kterým se lze setkat u většiny projektů s testy.

2.3.1 PHPUnit

PHPUnit je framework pro psaní jednotkových (unit) testů v jazyce PHP. Tento framework je postavený na architektuře xUnit.

V PHPUnit by každá testovaná třída měla mít svůj „TestCase.“ To je třída, která zastává funkci takového kontejneru pro jednotlivé testovací metody. PHPUnit zde vytváří trochu nesoulad v termínu „Test Case,“ neboli testovací případ. Normálně se testovacím případem rozumí jedna metoda, testující danou funkcionalitu. V PHPUnit je testovacím případem celá sada několika testovacích případů.[6]

Pokud je vytvořena Test Case třída, musí dědit z třídy „PHPUnit_Framework_TestCase," ta je součástí balíčku PHPUnit. Její název by měl být tvořen z názvu třídy, která je testovacím případem testována a slova „Test." Název třídy v názvu Test Case není vyžadován, je to ale jmenná konvence v PHPUnit a je dobré ji dodržovat. Struktura Test Case tříd by měla odrážet strukturu testovaných tříd. Zlepší se tím přehlednost mezi jednotlivými testovacími případy, případně jejich spouštění v příkazové řádce.[6]
Metody v Test Case třídách:

- setUp()
  Všechny testovací metody se spouštějí odděleně od ostatních metod. Nelze tedy volat testovací metodu, která nastaví potřebný stav pro další metodu. Jelikož je ale potřeba mít někde možnost pro nastavení výchozího stavu testování jedné či více metod, je k tomu zde určena metoda setUp(). Ta se volá před každou testovací metodou a umožňuje tak nastavit prostředí, pro jednotlivé testovací metody.
- tearDown()
  Tato metoda se naopak volá při ukončení každé testovací metody a používá se pro nastavení prostředí testovacích metod do původního stavu.
- testMetoda()
  Metoda pro provedení samotného testu. V hlavičce metody by neměly být žádné vstupní parametry a název metody musí obsahovat prefix „test."
- Asertační metody
  Metoda k ověření předpokladu. Klasickým předpokladem, který se kontroluje, je ověření, zda metoda testované třídy vrací očekávanou hodnotu. Typickými příklady asertačních metod jsou: assertEquals(), assertTrue(), assertFalse(), assertInstanceOf().[13]

PHPUnit rozlišuje hned několik výsledků testu:

- . (tečka) – Úspěšné provedení testovací metody
- F – Selhání asertační metody uvnitř testovací metody
- E – V případě, že se během provádění testovací metody vyskytne chyba

- R – V případě, že test byl označen jako riskantní
- S – V případě, že test byl přeskočen
- I – V případě, že test byl označen jako nekompletní

PHPUnit je komplexním nástrojem pro testování PHP aplikací a pro svou rozšířenost je dobré mít alespoň základní přehled o jeho fungování a použití. Vytváření testů pomocí nástroje Nette Tester má velmi podobný základ jako PHPUnit, jak bude patrné v příkladech jeho použití během vývoje.

3 Analýza

Aplikace je určena pro pořádání sportovních akcí, na které se budou moci uživatelé přihlašovat, a poté sledovat svou historii výsledků. Také bude možné přidávat různé články a fotogalerie z konaných událostí.

3.1 Požadavky

3.1.1 Funkční požadavky

Seznam funkčních požadavků na webovou aplikaci:

- V aplikaci budou uživatelé rozděleni do těchto rolí – administrátor, moderátor, registrovaný uživatel a neregistrovaný uživatel.
- Aplikace umožní vkládání novinek. Ty bude moci přidávat pouze administrátor. Přihlášení i nepřihlášení uživatelé si budou moci novinky zobrazit. Novinky bude možné přiřazovat do kategorií.
- V aplikaci bude možné vytváření události, které bude vytvářet administrátor. Všichni uživatelé si budou moci události zobrazit. Registrovaní uživatelé budou mít možnost registrovat se na jednotlivé události po spuštění registrací.
- Registrovaný uživatel bude mít možnost zobrazit si historii událostí, na které se registroval.
- Aplikace bude obsahovat kalendář s jednotlivými událostmi, který bude dostupný pro všechny uživatele.

3.1.2 Nefunkční požadavky

Seznam nefunkčních požadavků na webovou aplikaci je následující:

- MVC architektura, za použití PHP frameworku Nette
- Datová vrstva na platformě MySQL
- Autorizace přes uživatelské účty
- Oprávnění řešeno pomocí RBAC struktury

4 Vlastní vývoj webové aplikace

Během vývoje se programátoři potýkají s řešením mnoha různých chyb. Pokud se současně se zdrojovým kódem píší i samostatné testy dochází k eliminaci zdlouhavého hledání vzniklých chyb při budoucích úpravách, či rozšířeních aplikace. Pro tuto vyvíjenou aplikaci byl zvolen způsob vývoje řízený testy.

4.1 Použité technologie

4.1.1 Nette Framework

Jedná se o český open source framework pro tvorbu webových aplikací v PHP. Podporuje AJAX, DRY, KISS, MVC a znovupoužitelnost kódu. Používá událostmi řízené programování a je založen na používání komponent.

4.1.2 Nette Tester

Pro instalaci Testeru 2.3 je vyžadována minimální verze PHP 7.1.

Tester lze jednoduše nainstalovat v adresáři s aplikací přes Composer příkazem: *composer require –dev nette/tester.[9]*

Tester se spouští z příkazové řádky. Spuštění bez parametrů vypíše nápovědu. Každý test je samostatně spustitelný soubor. Je spouštěný jako samostatný proces. Tím je zajištěna dokonalá izolace. Testy jsou spouštěny paralelně v zadaném počtu vláken. To celé testování dramaticky zrychlí.

Tester může vygenerovat přehlednou mapu pokrytí kódu testy neboli code coverage.[9]

4.2 Metodiky vývoje

Jelikož je vývoj řízený testy založen na prvotním vytvoření testu a až následně se píše samotná metoda, je vývojář celou dobu veden k tomu, aby nevynechal žádné testy. Tyto testy se především zaměřují na testování modelových tříd, které mají nějakou vnitřní logiku a pracují s databází, kde je potřeba předcházet výskytu chyb.

Je ale možné, že při vytváření presenteru, jehož testováním se zabývají integrační testy, které zde navíc spíše řeší úspěšný průběh kódu než samotnou logiku, mohlo by se stát, že si vývojář vytvoří v modelu nějakou funkci a opomene na testy. S tímto dokáže pomoci samotný Nette Tester, který umí vygenerovat přehled, kolik zdrojového kódu testy pokrývají, pro jednotlivé třídy.

4.3 Vývoj aplikace

Před samotným psaním testů je ideální vytvořit nějaký bootstrap soubor, který se bude starat o počáteční nastavení pro všechny testy např. načtení autoload souborů, definice konstant a inicializaci společných proměnných.

V ukázce vytvořeného bootstrapu se nejprve načtou autoload soubory, nastaví se prostředí testeru voláním metody *setup()*, kterou obsahuje přímo Tester. Poté se nastaví konstanty pro lepší přehlednost a efektivitu vytváření a úpravy testů.

Nakonec se zde vytvoří instance třídy ConnectionHelper, pro práci s testovací databází.

```php
<?php declare(strict_types=1);

use App\Tests\Database\ConnectionHelper;

require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/autoload.php';

Tester\Environment::setup();
date_default_timezone_set('Europe/Prague');

define('ITEM_GET_ID', 1);
define('ITEM_UPDATE_ID', 2);
define('ITEM_DELETE_ID', 3);
define('ITEM_OUT_OF_RANGE_ID', 20);

define('TMP_DIR', __DIR__ . '/temp');
@mkdir(TMP_DIR);


$connectionHelper = new ConnectionHelper();
$database = $connectionHelper->getContext();
```

Zdrojový kód 3.1: Soubor bootstrap.php

4.3.1 ConnectionHelper

Třída ConnectionHelper obsahuje pouze využití traity ConnectionTHelper. Tato třída slouží pouze k použití u tohoto typu testování za pouhých PHP skriptů. V pozdější fázi, kde se bude využívat TestCase tříd, se používá pouze traita ConnectionTHelper.

V samotné traitě pak již dochází ke konfiguraci a připojení k databázi. V privátních proměnných je uložena adresa testovací databáze a testovací přihlašovací údaje, díky kterým má aplikace práva pracovat pouze s testovací databází a zamezí se nechtěné úpravě či smazání dat z ostré databáze.

První metoda vytvoří připojení k databázi a ze souboru se schématem databáze vytvoří potřebnou strukturu a následně vloží testovací data z odděleného souboru.

Druhá metoda vytvoří potřebné závislosti a vrátí instanci třídy Nette\Database\Context, která umožňuje pohodlnou práci z databází.

```
trait ConnectionTHelper
{
    private string $dsn = 'mysql:host=127.0.0.1;dbname=sport_test';
    private string $user = 'tester';
    private string $password = 'tester123';


    /**
     * @return Connection
     */
    public function getConnection(): Connection
    {
        $database = new Connection($this->dsn, $this->user,
                                   $this->password);

        try {
            Helpers::loadFromFile($database, __DIR__ .
'/../../database.sql');

            Helpers::loadFromFile($database, __DIR__ .
'/testing_data.sql');
        } catch (\Exception $ex) {
            echo $ex->getMessage();
        }

        return $database;
    }


    /**
     * @return Context
     */
    public function getContext(): Context
    {
        $storage = new FileStorage(TMP_DIR);
        $connection = $this->getConnection();
        $structure = new Structure($connection, $storage);
        $conventions = new DiscoveredConventions($structure);

        return new Context($connection, $structure, $conventions,
storage);
    }
}
```

Zdrojový kód 3.2: ConnectionTHelper

4.3.2 Testování Modelu – základní Unit testy

Pro testy první modelové třídy *Articles*, je využito PHP skriptů s příponou .phpt a doporučené struktury. Testy této třídy se nacházejí v adresáři *tests/model/articles* a názvy jsou tvořeny způsobem NázevTestovanéTřídy.názevTestovanéMetody.phpt (např. Articles.createArticle.phpt).

Prvním souborem v tomto adresáři je klasický .php soubor Articles.bootstrap, který zde načte obecný bootstrap soubor a inicializuje proměnnou *articles*, kterou využijí všechny následující testy. V této první fázi vývoje jsou testovány a vytvářeny metody sloužící pro CRUD operace (CRUD – Create, Read, Update Delete).

```php
<?php declare(strict_types=1);

use App\Model\Articles;

require __DIR__ . '/../../bootstrap.php';

$articles = new Articles($database);
```

Zdrojový kód 3.3: Articles.bootsrap

4.3.2.1 CRUD – Metoda createArticle

Tato metoda slouží pro vytvoření nového článku. Jelikož se jedná o metodu s přímým zásahem do databáze, provede se testování v transakci a po ukončení testu se vrátí změny do předchozího stavu.

Pokud metoda skončí úspěšně, je očekávána návratová hodnota *true.* Pro porovnání očekávaných hodnot se skutečnými Tester nabízí třídu *Assert,* kde se nyní dá využít metoda *true*. Ta ověří, zda je testovaná hodnota skutečně *true*.

V tento okamžik, pokud je spuštěn test, dostaneme výjimku, že volaná metoda neexistuje. Je tedy potřeba vytvořit tuto metodu, poté už se vypisuje výjimka o špatné návratové hodnotě *void*, zatímco je očekávaná hodnota *true.* Nyní již stačí v metodě vrátit hodnotu *true* a test dopadne úspěšně.

V dalším kroku je vytvořena instance entity *Article,* kterou by metoda měla vložit do databáze. Když je tato proměnná předána jako argument testované funkci, test znovu skončí neúspěšně. Po implementaci parametru této funkci, je test znovu úspěšný.

Dále je potřeba otestovat, zda byl článek opravdu vytvořen a zda všechny jeho atributy odpovídají předpokladům. Zde se implementuje vložení do databáze v metodě a přidají se další testy. Po vložení článku se z databáze načte poslední řádek, vytvoří se z něj nová entita *Article* a provedou se další testy. Tentokrát je použita metoda *Assert::same,* kde je jako první argument očekávaná hodnota a jako

druhý testovaná hodnota. Jelikož má *Article* dva atributy (title a content), je test proveden pro oba.

Po spuštění testu se ukázalo, že vše funguje, jak má a nenastala žádná chyba.

```
require __DIR__ . '/Articles.bootstrap.php';

$article = new Article(0, "Nový článek", "Test vytvoření nového článku");

$database->beginTransaction();

Assert::true($articles->createArticle($article));

$lastRow = $database->table('articles')->order('id DESC')->limit(1)->fetch();
if ($lastRow) {

    $lastArticle = new Article((int)$lastRow->id, (string)$lastRow->title,
(string)$lastRow->content);

    Assert::same($article->getTitle(), $lastArticle->getTitle());
    Assert::same($article->getContent(), $lastArticle->getContent());
}

$database->rollBack();
```

Zdrojový kód 4.1: Soubor s testem Articles.createArticle

```
/**
 * Create article
 * @param Article $article
 * @return bool
 */
public function createArticle(Article $article): bool
{
    $values = [
        "title" => $article->getTitle(),
        "content" => $article->getContent(),
    ];

    try {
        $this->db->table('articles')->insert($values);
    } catch (\Exception $ex) {
        unset($ex);
        return false;
    }

    return true;
}
```

Zdrojový kód 4.2: Metoda createArticle

4.3.2.2 CRUD – Metoda getArticle

Při vytváření této metody a testovacího souboru je postup stejný jako při předešlé metodě. Jelikož zde ale nedochází k zásahu do databáze, není potřeba použití transakce.

Zde je využita konstanta pro ověření metod typu get, která zastupuje id článku. Vrácená hodnota se následně testuje, zda odpovídá třídě *Article.* Poté se testují atributy, zda jsou shodné s hodnotami vkládanými během počátečního nastavení databáze.

Nachází se zde ještě jeden test, a to test *Assert::exception,* který jako první argument přijímá callback funkci, ve které očekává vyhození výjimky, která se určí jako druhý argument. Tato funkce ověřuje, zda metoda getArticle vyhodí výjimku *FileNotFoundException*, pokud hodnota id předána této funkci neodpovídá žádnému záznamu v databázi.

```
require __DIR__ . '/Articles.bootstrap.php';

$article = $articles->getArticleById(ITEM_GET_ID);
Assert::type(Article::class, $article);

Assert::same(ITEM_GET_ID, $article->getId());
Assert::same('První článek', $article->getTitle());
Assert::same('První článek svého druhu na této stránce', $article-
>getContent());

Assert::exception(function() use($articles) {
    $articles->getArticleById(ITEM_OUT_OF_RANGE_ID);
}, FileNotFoundException::class);
```

Zdrojový kód 4.3: Soubor s testem Articles.getArticle

```
/**
 * Return article by id
 * @param int $id
 * @return Article
 * @throws Nette\FileNotFoundException
 */
public function getArticleById(int $id): Article
{
    $result = $this->db->table('articles')->get($id);

    if (!$result) {
        throw new Nette\FileNotFoundException();
    }

    return new Article((int)$result->id, (string)$result->title,
(string)$result->content);
}
```

Zdrojový kód 4.4: Metoda getArticleById

4.3.2.3 CRUD – Metoda updateArticle

U této metody je postup velmi podobný metodě createArticle. Test se provádí v transakci, jelikož dochází k úpravě jednoho z původních záznamů.

Na začátku se zde také vytváří instance entity *Article*, ale jako id se použije konstanta určená pro testování update metod.

Návratová hodnota update metody je testována oproti hodnotě true. Dále je nalezen záznam, který má shodné id s upravovaným článkem a je porovnáno metodou *Assert::same,* zda současné hodnoty odpovídají hodnotám určeným k updatu.

Nakonec se také testuje vyhození výjimky, pokud se předá metodě *updateArticle* hodnota id neodpovídající žádnému záznamu.

```
require __DIR__ . '/Articles.bootstrap.php';

$article = new Article(ITEM_UPDATE_ID, 'Založení webu – edit', 'Založení webu
sportjaromer.cz');

$database->beginTransaction();

Assert::true($articles->updateArticle($article));

$row = $database->table('articles')->get($article->getId());
$updatedArticle = new Article((int)$row->id, (string)$row->title,
(string)$row->content);

Assert::same($article->getTitle(), $updatedArticle->getTitle());
Assert::same($article->getContent(), $updatedArticle->getContent());

Assert::exception(function() use($articles) {
    $badArticle = new Article(ITEM_OUT_OF_RANGE_ID, 'Založení webu – edit',
'Založení webu sportjaromer.cz');
    $articles->updateArticle($badArticle);
}, FileNotFoundException::class);
```

Zdrojový kód 4.5: Soubor s testem Articles.updateArticle

```php
/**
 * Update article
 * @param Article $article
 * @return bool
 * @throws Nette\FileNotFoundException
 */
public function updateArticle(Article $article): bool
{
    $result = $this->db->table('articles')->get($article->getId());

    if (!$result) {
        throw new Nette\FileNotFoundException();
    }

    $values = [
        "title" => $article->getTitle(),
        "content" => $article->getContent(),
    ];

    try {
        $result->update($values);
    } catch (\Exception $ex) {
        unset($ex);
        return false;
    }

    return true;
}
```

Zdrojový kód 4.6: Metoda updateArticle

4.3.2.4 CRUD – Metoda deleteArticle

Tato poslední vybraná metoda sloužící k odstranění článku z databáze je opět testována v transakci a zkouší se odebrat jeden z předem vytvořených záznamů. Nejprve se však testuje, zda je tento prvek opravdu v databázi a po odstranění, zda je vyhozena výjimka, že daný záznam neexistuje. Na závěr se ještě otestuje, zda při odstranění neexistujícího záznamu dojde k vyhození výjimky.

```php
require __DIR__ . '/Articles.bootstrap.php';

$database->beginTransaction();

Assert::type(Article::class, $articles->getArticleById(ITEM_DELETE_ID));

Assert::true($articles->deleteArticle(ITEM_DELETE_ID));

Assert::exception(function() use($articles) {
    $articles->getArticleById(ITEM_DELETE_ID);
}, FileNotFoundException::class);

Assert::exception(function() use($articles) {
    $articles->deleteArticle(ITEM_OUT_OF_RANGE_ID);
}, FileNotFoundException::class);

$database->rollBack();
```

Zdrojový kód 4.7: Soubor s testem Articles.deleteArticle

```php
/**
 * Delete article
 * @param int $id
 * @return bool
 * @throws Nette\FileNotFoundException
 */
public function deleteArticle(int $id): bool
{
    $result = $this->db->table('articles')->get($id);

    if (!$result) {
        throw new Nette\FileNotFoundException();
    }

    try {
        $result->delete();
    } catch (\Exception $ex) {
        unset($ex);
        return false;
    }
    return true;
}
```

Zdrojový kód 4.8: Metoda deleteArticle

4.3.3 TestCase

U jednoduchých testů lze mít aserce jednu za druhou. Občas se ale stává výhodnější zabalit aserce do jedné testovací třídy, a tak je strukturovat. Tato třída musí být potomkem *Tester\TestCase.* Ve třídě musí být metody s prefixem „test," které jsou poté spouštěny jako testy. Dále v této třídě mohou být metody *setUp()* a *tearDown(),* které jsou volány před, respektive za každou testovací metodou a lze v nich nastavit testovací prostředí společné pro všechny testovací metody této třídy.

Za použití TestCase je vytvořena další modelová třída Tournaments, která slouží pro správu turnajů, a v ní jsou pro každou CRUD operaci samostatné metody. Ty mají velmi podobný obsah jako předchozí jednotlivé soubory. Lze takto testovat jednu celou třídu v jednom PHP souboru s třídou dědící z TestCase.

Testování tímto stylem se tak stává mnohem přehlednější. Také zde není bootstrap soubor pro tuto konkrétní třídu, ale vytvoření instance modelové třídy, se provede v metodě *setUp()*.

Na konci každého souboru s TestCase třídou je poté potřeba vytvořit novou instanci této třídy a nad ní zavolat metodu *run().*

Také je zde zcela předělán původní bootstrap soubor na abstraktní třídu dědící z TestCase a jednotlivé testovací třídy již dědí z této bootstrap třídy.

V této abstraktní třídě jsou definovány konstanty pro testování, je zde použita dříve zmíněná traita *ConnectionTHelper*, díky které se zde načítá správa databáze.

Podstatná změna je v konstruktoru této třídy, kde se za využití Nette Frameworku použije vlastní bootstrap aplikace, který vytváří testovací kontejner a všechny potřebné závislosti.

Pro pozdější využití pro integrační testy se zde vytváří také instance třídy *PresenterFactory*.

Je zde využita *setUp()* metoda, ve které se získá přístup k databázi a také se zde zahájí transakce, tudíž všechny prováděné testy se automaticky provádí v ní. V metodě *tearDown()* se už pouze ukončí transakce a databáze se vrátí do stavu před zahájením.

TestCase třída *TournamentsTest,* je zde v ukázce omezena pouze na jednu testovací metodu.

```
abstract class BootstrapTestCase extends TestCase
{
    use ConnectionTHelper;

    const ITEM_GET_ID = 1;
    const ITEM_UPDATE_ID = 2;
    const ITEM_DELETE_ID = 3;
    const ITEM_OUT_OF_RANGE_ID = 20;

    /** @var Context */
    protected Context $db;

    /** @var Container */
    protected Container $container;

    /** @var PresenterFactory */
    protected PresenterFactory $presenterFactory;


    public function __construct()
    {
        $this->container = Bootstrap::bootForTests()->createContainer();
        $presenterFactory =  $this->container-
>getByType('Nette\Application\IPresenterFactory');
        if ($presenterFactory instanceof PresenterFactory) {
            $this->presenterFactory = $presenterFactory;
        }
    }


    protected function setUp()
    {
        $this->db = $this->getContext();
        $this->db->beginTransaction();
    }


    protected function tearDown()
    {
        $this->db->rollBack();
    }
}
```

Zdrojový kód 5.1: Třída BootstrapTestCase

```php
class TournamentsTest extends BootstrapTestCase
{

    /** @var Tournaments */
    private Tournaments $tournaments;


    protected function setUp()
    {
        parent::setUp();
        $this->tournaments = new Tournaments($this->db);
    }


    public function testGetTournaments()
    {
        $tournaments = $this->tournaments->getTournaments();

        Assert::type('array', $tournaments);

        foreach ($tournaments as $tournament) {
            Assert::type(Tournament::class, $tournament);
        }
    }
}

(new TournamentsTest())->run();
```

Zdrojový kód 5.2: TestCase třída TournamentsTest

4.3.4 Autorizace

Autorizace je řešena vytvořením třídy DbAuthenticator, která implementuje rozhraní Nette\Security\IAuthenticator. Z tohoto rozhraní je implementována metoda *authenticate(),* která přijímá pole s přihlašovacími údaji a vrací instanci třídy Nette\Security\Identity.

Pro tuto třídu a implementaci autentizační metody je vytvořena TestCase třída *DbAuthenticatorTest*, která je potomkem třídy *BootstrapTestCase.* Tato testovací třída obsahuje pouze jedinou metodu s testem *testAuthenticate*.

Jelikož metoda *authenticate()* není volána v aplikaci přímo, je potřeba na začátku vytvořit instanci třídy Nette\Security\User, k čemuž pomůže vytvořený kontejner v bootstrap třídě. Tato instance umožňuje volání metody *login(),* ta má první parametr pro přihlašovací email a druhý pro heslo a tyto údaje následně předá metodě *authenticate()*. Pokud vše funguje správně je uživatel přihlášen, to ověřuje test *Assert::true*, v kterém je volána metoda *isLoggedIn(),* na instanci třídy User, použité k přihlášení. Dále je testováno, zda uživatelské jméno, které obsahuje

reference na identitu ve třídě User. To je porovnáno s jménem a příjmením přihlašovaného uživatele.

Dále je testováno zadání správného emailu a špatného hesla, ke zjištění, zda je správně vyhozena výjimka a uživatel není přihlášen a taky špatného emailu a stejné výjimky, když není uživatel v databázi nalezen.

```
class DbAuthenticatorTest extends BootstrapTestCase
{


    public function testAuthenticate()
    {
        $user = $this->container->getByType('Nette\Security\User');
        $user->login('admin@sportjaromer.cz', '12345');

        Assert::true($user->isLoggedIn());
        Assert::same('Dominik Mojžíš', $user->getIdentity()->username);
        $user->logout();

        Assert::exception(function() use($user) {
            $user->login('admin@sportjaromer.cz', 'pass');
        }, AuthenticationException::class);

        Assert::exception(function() use($user) {
            $user->login('bad@email.com', 'pass');
        }, AuthenticationException::class);
    }

}

(new DbAuthenticatorTest())->run();
```

Zdrojový kód 6.1: Test třídy DbAuthenticator

```
class DbAuthenticator implements NS\IAuthenticator
{

    /** @var Context */
    private Context $db;

    /** @var Users */
    private Users $users;


    public function __construct(Context $db, Users $users)
    {
        $this->db = $db;
        $this->users = $users;
    }


    function authenticate(array $credentials): IIdentity
    {
        list($email, $password) = $credentials;

        try {
            $user = $this->users->getUserByEmail($email);

            $passwords = new NS\Passwords(PASSWORD_BCRYPT, ['cost' => 12]);

            if (!$passwords->verify($password, $user->getPasswordHash())) {
                throw new \UnexpectedValueException();
            }

            return new NS\Identity($user->getId(), $user->getRoleId(),
['username' => $user->getFirstname() . ' ' . $user->getLastname()]);

        } catch (\Exception $ex) {
            unset($ex);
            throw new NS\AuthenticationException('Přihlašovací jméno nebo
heslo bylo zadáno špatně');
        }

    }
}
```

Zdrojový kód 6.2: Třída DbAuthenticator

4.3.5 Integrační testy

Další fáze testování jsou integrační testy. Zde se takto dají testovat jednotlivé metody presenteru se svými závislostmi. Testy opět využívají třídy TestCase, kde se v metodě *setUp()* vytvoří instance testovaného presenteru pomocí třídy *PresenterFactory*, v tomto případě *HomepagePresetner*.

V samotné testovací metodě, která zde testuje metodu *renderDefault()*, se vytvoří požadavek, kde je určen název volaného presenteru a metoda jakou má být požadavek zaslán. Nad instancí dříve vytvořeného presenteru se zavolá metoda *run()*, které se předá požadavek. Tato metoda vrací *Response*, u které je porovnán

správný návratový typ. V tuto chvíle není potřeba testovat nic dalšího, pokud by došlo k nějaké chybě v průběhu zpracovávání požadavku, test dopadne chybně a vypíše se error, či vyhozená výjimka.

4.3.5.1 Testy vykreslení šablony

Jelikož je občas dobré otestovat, zda je správně vykreslený front end. V již vytvořené metodě pro testování presenteru se z vrácené response získá DOM objekt, ve kterém lze například hledat vykreslené elementy pomocí identifikátorů.

Jeden ze způsobů využití může být kontrola, zda přihlášený a nepřihlášený uživatel vidí či nevidí to co je očekáváno. V metodě tedy přidáme funkci *Assert::true*, v které je nad objektem DOM volána funkce *has()*. V té je hledán html element, jenž má třídu *not-logged*. Jelikož uživatel není přihlášen, měl by být tento element nalezen. Naopak pokud se bude hledat element se třídou *logged,* měla by být vrácena hodnota false.

Pro otestování zobrazení přihlášenému uživateli, je vytvořena další testovací metoda *testDefaultLogged()*. Rozdíl je v tom, že na začátku metody se vytvoří objekt třídy User a nad ním se zavolá metoda *login()* s platnými údaji a uživatel je přihlášen. Poté dojde k vytvoření stejného požadavku a porovnání response. Krom toho se zde ověří, zda v aplikaci nedochází k chybě pro přihlášeného uživatele. Na konci je stejné otestování elementů, akorát jsou prohozeny názvy tříd a hodnota true je porovnávána vůči třídě *logged* a obráceně.

```php
class HomepagePresenterTest extends BootstrapTestCase
{

    /** @var IPresenter */
    private IPresenter $homepagePresenter;

    protected function setUp()
    {
        parent::setUp();
        $this->homepagePresenter =
                $this->presenterFactory->createPresenter('Homepage');

        $this->homepagePresenter->autoCanonicalize = false;
    }


    public function testDefault()
    {
        $request = new Request('Homepage', 'GET');
        $response = $this->homepagePresenter->run($request);

        $dom = $this->getDom($response);

        Assert::true($dom->has('.mb-0'));
        Assert::true($dom->has('.not-logged'));
        Assert::false($dom->has('.logged'));
    }


    public function testDefaultLogged()
    {
        $user = $this->container->getByType('Nette\Security\User');
        $user->login('admin@sportjaromer.cz', '12345');
        $request = new Request('Homepage', 'GET');
        $response = $this->homepagePresenter->run($request);

        $dom = $this->getDom($response);

        Assert::true($dom->has('.logged'));
        Assert::false($dom->has('.not-logged'));
    }


    private function getDom(IResponse $response): DomQuery
    {
        Assert::type(TextResponse::class, $response);
        Assert::type(Template::class, $response->getSource());

        $html = (string) $response->getSource();
        return @DomQuery::fromHtml($html);
    }
}

(new HomepagePresenterTest())->run();
```

Zdrojový kód 7: Test Homepage presenteru

5 Shrnutí výsledků

Cílem bakalářské práce je porovnat metodu vývoje řízeného testy s běžným postupem a zhodnotit tuto metodu praktickým vývojem webové aplikace Tento způsob a jednotlivé možnosti testování byly teoreticky popsány a byl představen nejpoužívanější testovací nástroj pro jazyk PHP.

Při vývoji samotné aplikace byl ale použit nástroj Nette Tester, pro svoji jednoduchost a vysokou kompatibilitu s frameworkem Nette. Pomocí tohoto nástroje byly provedeny všechny typy použitých testů pro všechny specifické části aplikace. Použité postupy byly popsány a doplněny zdrojovým kódem.

Jak bylo zmíněno tento způsob vede k tomu, aby vývojář nezapomínal na vytváření testů u všech důležitých tříd (tj. třídy, které obsahují vnitřní logiku aplikace).

Při porovnání s běžným vývojem aplikace z praxe je zřejmé, že vytvořené testy napomáhají k porozumění logice jednotlivých testovaných metod pro nezaujaté vývojáře. Při změně kódu v průběhu vývoje stačilo k ověření funkčnosti kódu pouze spustit automatické testy a v průběhu několika vteřin byl vidět výsledek. Bez těchto testů musí vývojář projít každou část aplikace zvlášť a ručně testovat její funkce, což může u větších aplikací zabrat desítky minut. Pokud se ale porovná čas strávený vytvořením nějaké metody, bývá vytvoření bez testu rychlejší, než pokud je psán ještě test, zejména u jednoduchých metod.

6 Závěry a doporučení

Vývoj aplikace je složitý proces, kde každá část musí řádně spolupracovat v rámci celku. Vývoj webových aplikací stojí nemalé peníze, jelikož je důležité mít spolehlivý a kvalitní vývojový tým. Protože čas strávený vytvářením aplikace stojí zákazníka, či samotnou firmu, která aplikaci vyvíjí, dost peněz, je důležité co nejvíce zkrátit dobu potřebnou pro řešení a hledání chyb a odhalovat chyby již v samotném počátku.

Používání testů by se tak nemělo opomíjet a brát na lehkou váhu a tým vývojářů by ho měl do vývoje zařadit v nějaké formě již od začátku, při vývoji způsobem TDD mohou testy odhalit chybu, či nekonzistentnost ihned po napsání nové části kódu nebo úpravě původních funkcionalit.

Napsání testu po vytvoření, či úpravě většího celku (např. před samotným pull requestem) také pomůže odhalit případné chyby, které se dají ještě celkem snadno nalézt a opravit.

Pokud jsou ovšem testy psány již na hotové aplikaci, může být hledání chyb obtížnější, ale především může vývojář zapomenout přesné funkce jednotlivých metod, respektive konkrétní chování v průběhu metody, a může tak dojít k opomenutí otestování některých důležitých situací, zvláště pak pokud testy píše někdo jiný, než kdo psal konkrétní metodu, či třídu.

Pro vyvíjenou aplikaci by bylo možné dále vytvářet i další druhy testů v jiných nástrojích (např. Selenium WebDriver) pro komplexnější otestování všech částí. Ideální však je vybrat si z nástrojů, které testují stejné oblasti, na začátku vývoje, aby nedošlo ke konfliktům a každý vývojář si mohl spouštět již vytvořené testy od někoho jiného.

Při použití této metody vývoje v jazyce PHP je velmi dobré používat pro jakoukoliv aplikaci testování pomocí TestCase tříd a vyhnout se jednoduchým PHP skriptům s testy. Testy jsou přehlednější a lze jednodušeji nastavit prostředí pro testovanou třídu.

Alternativou k použitému testovacímu nástroji Nette Tester by mohl být nástroj PHPUnit, pokud by se aplikace vyvíjela v Javě, lze použít JUnit, pro C# lze například využít MSTest a JavaScript je možné testovat nástrojem Jest. Tyto nástroje lze využít pro jednotkové testy, ale i testy integrační.

Pro webovou aplikaci napsanou v jakémkoli uvedeném jazyce lze testovat webové rozhraní pomocí nástroje Selenium, které lze využít i pro akceptační testy.
Tato bakalářská práce by mohla sloužit pro pochopení základních principů testování a pomoci tak při budoucím vývoji této aplikace, ale i dalším vývojářům, kteří vytvářejí nebo budou vyvíjet nějakou aplikaci a chtěli by testování začlenit do vývoje, ale nemají ještě potřebné zkušenosti nebo váhají, jestli se testování vůbec vyplatí.

7 Seznam použité literatury

[1] BECK, Kent. Programování řízené testy. Praha: Grada, 2004. Moderní programování. ISBN 80-247-0901-5

[2] KRIPNER, Matěj. Unit testy v Javě a JUnit. [online]. 6.7.2015 [cit. 20.3.2020]. Dostupné z: https://www.itnetwork.cz/java/testovani/java-unit-testy-v-junit

[3] PATEL, Pankaj. System Testing. [online]. 15.4.2017 [cit. 24.3.2020]. Dostupné z: https://www.geeksforgeeks.org/system-testing/.

[4] MEHTA, Bhumika. What is System Testing in Software Testing?. [online]. 11.3.2020 [cit. 25.3.2020]. Dostupné z: https://www.softwaretestinghelp.com/system-testing/.

[5] SETTER, Matthew. User Acceptance Testing – How To Do It Right!. [online]. 24.8.2018 [cit. 29.3.2020]. Dostupné z: https://usersnap.com/blog/user-acceptance-testing-right/.

[6] BERGMANN, Sebastian. PHPUnit Manual. [online]. 20.3.2020 [cit. 30.3.2020]. Dostupné z: https://phpunit.readthedocs.io/en/9.1/index.html

[7] HLAVA, Tomáš. Fáze a úrovně provádění testů. Testování softwaru. [online]. 21.8.2011[cit. 30.1.2020]. Dostupné z: http://testovanisoftwaru.cz/category/metodika-testovani/druhy-typy-a-kategorie-testu/.

[8] KAJZAR, Dušan. C) Specifikace požadavků na systém. Projektování informačních systémů II. [online]. [cit. 1.2.2020]. Dostupné z: http://zdenek2.euweb.cz/doc3/prois7c.html

[9] GRUDL, David. Nette Tester. Průvodce. [online]. 14.1.2020 [cit. 28.3.2020] Dostupné z: https://tester.nette.org/cs/guide.

[10] ASTELS, D., 2003. Test Driven development: A Practical Guide. Prentice Hall. ISBN 978-0-13-101649-1

[11] JANZEN, D. and SAIEDIAN, H. "Does Test-Driven Development Really Improve Software Design Quality?," in IEEE Software, 2008.

[12] MASSOL, V. and HUSTED, T. JUnit in Action. Mannig 2003. ISBN 978-1-930110-99-1

[13] MACHEK, Z. PHPUnit Essentials. Packt 2014 ISBN 978-1-78328-343-9

[14] LEUNG, H. K. N. and WHITE, L., "A study of integration testing and software regression at the integration level," Proceedings. Conference on Software Maintenance 1990, San Diego, CA, USA, 1990 ISBN 0-8186-2091-9

[15] BRIAND, L., LABICHE, Y. A UML-Based Approach to System Testing. [online] 12.9.2002 [cit. 25.3.2020]. Dostupné z: https://doi.org/10.1007/s10270-002-0004-8

[16] HSIA, P., KUNG, D. and SELL, C. Software requirements and acceptance testing. *Annals of Software Engineering*, [online] 1997 [cit. 30.3.2020]. Dostupné z: https://doi.org/10.1023/A:1018938021528

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/19 .

Studijní program: Aplikovaná informatika
Forma:
Prezenční Obor/komb.: Aplikovaná informatika (ai3-p)

1 Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Habr Ondřej	Raichlové 855, Náchod	I1700079

TÉMA ČESKY: Vývoj webových aplikací řízený testy

TÉMA ANGLICKY: Test driven development of web applications

VEDOUCÍ PRÁCE: Mgr. Daniela Ponce Ph.D.

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je porovnat metodu vývoje řízeného testy s běžným postupem a zhodnotit tuto metodu praktickým vývojem webové aplikace.

SEZNAM DOPORUČENÉ LITERATURY:

Beck, K. (2000): Test-Driven Development by Example
Astels, D. (2003): Test-driven Development: A Practical Guide
Langr, J (2015): Pragmatic Unit Testing in Java 8 with JUnit
Freeman, S. a Nat P. (2010): Growing object-oriented software, guided by tests.

Podpis studenta: ..... **Datum:** .....

2 Podpis vedoucího práce: ..... **Datum:** .....