

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

**AUTOMATIZOVANÉ TESTOVANIE (UNIT,  
INTEGRAČNÉ, END-TO-END A VÝKONNOSTNÉ TESTY)**

**DOKUMENTÁCIA K SEMESTRÁLNEMU ZADANIU PREDMETU  
ARCHITEKTÚRA SOFTVÉROVÝCH SYSTÉMOV 2022/2023**

**Bc. Filip Frank  
Bc. Viet Quoc Le  
Bc. Juraj Puzster  
Bc. Tomáš Singhofer**

# Obsah

<b>Úvod .....</b>	<b>1</b>
<b>1    Technická špecifikácia.....</b>	<b>2</b>
1.1    Informácie o kóde softvéru.....	2
1.2    Testovacie nástroje použité v ukázkach.....	2
1.3    Demo aplikácia .....	2
<b>2    Unit testy .....</b>	<b>4</b>
2.1    Teoretický úvod.....	4
2.1.1    Ciele unit testovania.....	4
2.1.2    Výhody unit testovania.....	5
2.1.3    Nevýhody unit testovania.....	5
2.1.4    Testom riadený vývoj.....	5
2.1.5    Mockovanie objektov.....	6
2.1.6    Praktiky pri unit testovaní .....	6
2.1.7    Známe nástroje pre unit testovanie.....	6
2.2    Príklad unit testovania v Jave .....	6
2.2.1    Najpoužívannejšie JUnit anotácie .....	7
2.2.2    Najpoužívannejšie metódy z triedy Assertions .....	7
2.2.3    Testovanie .....	8
<b>3    Integračné testy .....</b>	<b>11</b>
3.1    Konfigurácia .....	11
3.2    MockMvc.....	11
3.3    Testovanie.....	11
3.3.1    Metódy .....	12
3.3.2    Výstup .....	12
3.4    Testy .....	13
<b>4    End-to-end testy .....</b>	<b>14</b>

4.1	Dostupné nástroje .....	14
4.2	CypressJS.....	15
4.3	Inštalácia a spustenie .....	15
4.3.1	Test Login .....	16
4.3.2	Rozhranie .....	17
<b>5</b>	<b>Výkonnostné testy .....</b>	<b>18</b>
5.1	Teoretický úvod.....	18
5.1.1	Definícia.....	18
5.1.2	Využitie výkonnostných testov .....	18
5.1.3	Proces realizácie výkonnostných testov .....	19
5.1.4	Typy výkonnostných testov .....	19
5.1.5	Metriky sledované pri výkonnostných testoch .....	20
5.2	Príklad výkonnostného testu s využitím Apache JMeter .....	20
5.2.1	Softvér JMeter .....	20
5.2.2	Proces realizácie testu .....	20
5.2.3	Testovacie plány použité v prezentácii .....	23
5.2.4	Interpretácia výsledkov s použitím Summary Report listeneru .....	24
5.2.5	Validácia testu .....	24
	<b>Zoznam použitej literatúry .....</b>	<b>26</b>

# Úvod

Dokument slúži ako úvod do vybraných typov automatizovaného testovania softvéru riešených v prezentácii na tému Automatizované testovanie (unit, integračné, end-to-end a výkonnostné testy), ktorá bola odprezentovaná autormi v rámci semestrálneho zadania na predmete Architektúra softvérových systémov. Dokumentácia zároveň slúži ako návod pre realizáciu praktických ukážok z prezentácie, ktoré by mali čitateľovi pomôcť zahrnuté typy testovania implementovať vo svojich riešeniach.

Kapitola Technická špecifikácia obsahuje informácie o kóde demonštračného softvéru a zoznam použitých testovacích nástrojov. V ďalších častiach je dokumentácia delená do kapitol podľa typu vykonaných testov.

# 1 Technická špecifikácia

## 1.1 Informácie o kóde softvéru

Použitý programovací jazyk: Java (Backend)

Použité frameworky: Spring Boot

Kód je v čase publikácie tohto dokumentu dostupný na:

<https://github.com/j-pusztter/testovanie>

## 1.2 Testovacie nástroje použité v ukážkach

Unit testy: JUnit

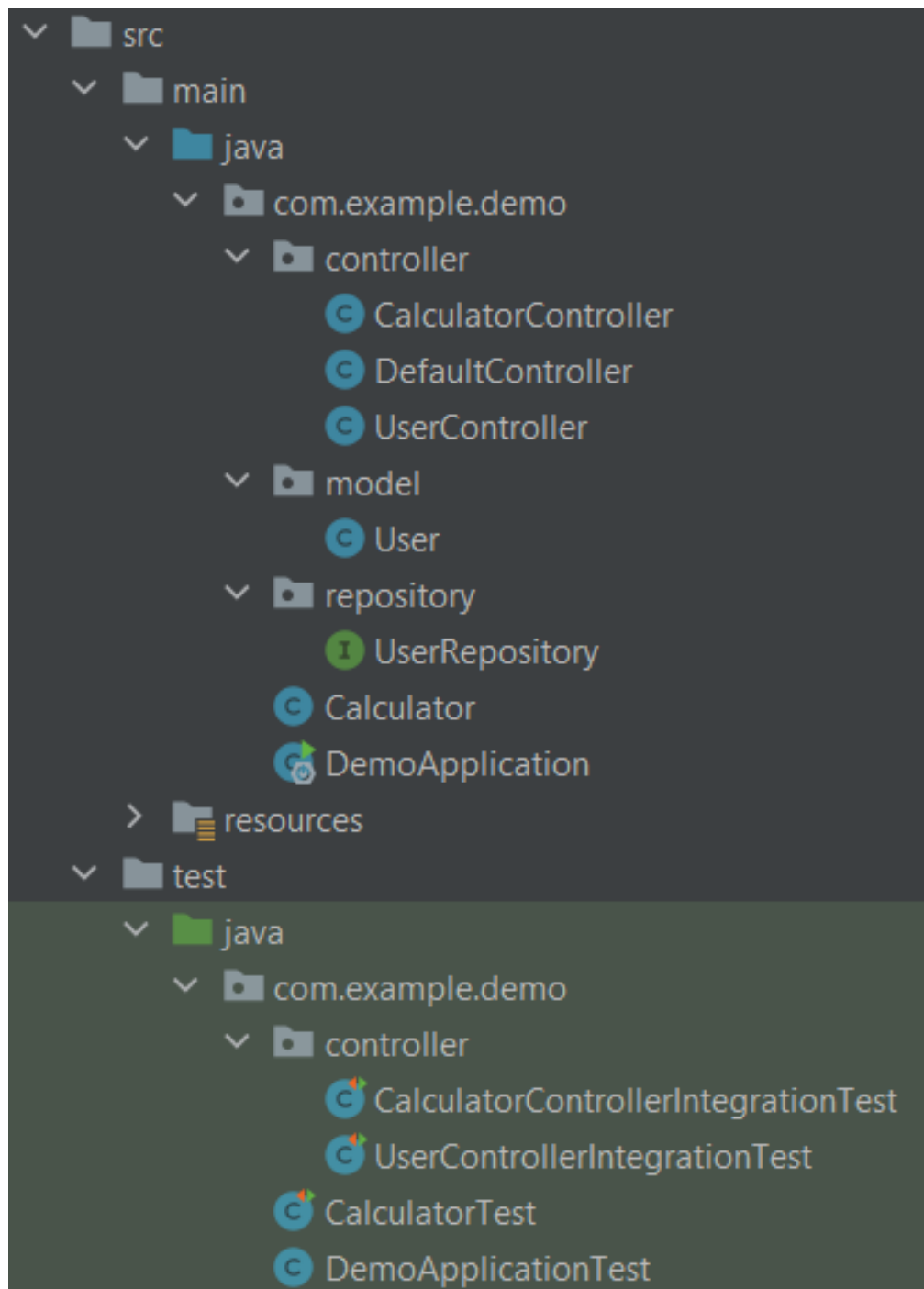
Integračné testy: Spring

End-to-end testy: CypressJS

Výkonnostné testy: JMeter

## 1.3 Demo aplikácia

Jednotlivé typy testov boli vykonané na originálnej demo aplikácii. Nasledujúci obrázok zobrazuje štruktúru kódu aplikácie.



## 2 Unit testy

### 2.1 Teoretický úvod

Unit testovanie je typ testovania softvéru, pri ktorom sa testujú jednotlivé jednotky alebo komponenty. Účelom je overiť, či každá jednotka softvérového kódu funguje podľa očakávania. Unit testovanie sa vykonáva počas vývoja (počas fázy kódovania) aplikácie vývojármi. Unit testy izolujú časť kódu a overujú jeho správnosť. Jednotkami softvéru považujeme individuálne funkcie, metódy, procedúry, moduly alebo objekty. Testovanie softvéru začína ešte pred dokončením aplikácie. Týmto spôsobom sú chyby odhalené skôr, ako sa stratia v kódach.

Unit testovanie je prvou vrstvou celého testovacieho procesu, ktorým musí softvér prejsť pred jeho spustením a vydaním. Toto predbežné testovanie často vykonáva tím vývojárov alebo softvérový inžinier, ktorý napísal kód softvéru. Vyššia úroveň povedomia o zložitosti programu zvyšuje šance na vykonanie dôkladnej práce. Inžinieri zabezpečenia kvality sú tiež vyškolení na vykonávanie unit testov. Tieto testy je možné vykonávať manuálne, avšak zvyčajne sú automatizované. To zaisťuje že časti softvéru spĺňajú očakávania.

Unit testy tvoria základ, na ktorom sú postavené všetky ostatné testy. Ich presnosť a dôkladnosť je významným faktorom softvérového vývoja. Ovplyvňujú, ako dobre sa dajú vykonať ostatné testy a tiež výkon softvéru ako celku.

Unit testovanie je súčasťou Test Driven Development (TDD), metodiky, ktorá využíva opakované testovanie na vytváranie kvalitných produktov. Každá jednotka musí byť nezávislá od akéhokoľvek externého faktora alebo kódu, aby tester mohli jasne interpretovať výsledky.

Vývojári softvéru sa niekedy snažia ušetriť čas pri minimálnom unit testovaní. Toto vedie k opačnému efektu. Nevhodné unit testovanie má za následky vysoké náklady na opravu defektov v neskorších fázach vývoja. Ak sa unit testovanie vykoná na začiatku vývoja, potom to v konečnom dôsledku šetrí čas a peniaze.

#### 2.1.1 Ciele unit testovania

- Overenie presnosti častí kódu
- Dosiahnutie samostatných a nezávislých častí kódu

- Identifikovanie chýb na začiatku vývoja softvéru
- Zvýšenie porozumenia kódu programátorom
- Jednoduchšie vykonávanie zmien v kóde
- Opakované používanie kódu

### **2.1.2 Výhody unit testovania**

- Vývojári, ktorí chcú zistiť, aké funkcie poskytuje jednotka a ako ju používať, sa môžu pozrieť na unit testy, aby získali jej základné pochopenie
- Unit testy umožňujú programátorovi neskôr zrefaktorovať kód a uistiť sa, že modul stále funguje správne (regresné testovanie). Postup spočíva v napísaní testovacích prípadov pre všetky funkcie a metódy, aby bolo možné kedykoľvek, keď zmena spôsobí poruchu túto poruchu rýchlo identifikovať a opraviť.
- Vzhľadom na modulárny charakter unit testov, môžeme testovať časti projektu bez toho, aby sme čakali na dokončenie ostatných častí.

### **2.1.3 Nevýhody unit testovania**

- Nedá sa očakávať, že unit testovanie zachytí každú chybu v programe. Nie je možné vyhodnotiť všetky cesty vykonávania programu ani v tých najtriviálnejších programoch
- Unit testovanie sa zo svojej podstaty zameriava na jednotky kódu. Preto nedokáže zachytiť chyby integrácie alebo rozsiahle systémové chyby.

### **2.1.4 Testom riadený vývoj**

Unit testovanie v TDD (test driven development) zahŕňa rozsiahle používanie testovacích frameworkov. Framework pre unit testy sa používa pre vytvorenie automatizovaných unit testov. Tieto frameworky nie sú jedinečné pre TDD, ale sú preň nevyhnutné.

Testom riadený vývoj obnáša:

- Testy sú napísané pred kódom
- Testy sa ťažko spoliehajú na frameworky
- Všetky triedy softvéru sú testované
- Je možná rýchla a jednoduchá integrácia



### **2.1.5 Mockovanie objektov**

Unit testovanie sa spolieha na vytváranie mockovaných objektov na testovanie častí kódu, ktoré ešte nie sú súčasťou kompletnej aplikácie. Mockované objekty dopĺňajú chýbajúce časti programu. Môžeme mať napríklad funkciu, ktorá potrebuje premenné alebo objekty, ktoré ešte nie sú vytvorené. Pri testovaní sa budú vytvárať vo forme mockovaných objektov vytvorených výlučne na účely testovania vykonaného v danej časti kódu.

### **2.1.6 Praktiky pri unit testovaní**

- Návrh vhodných názvov testov
- Vytvorenie jednoduchých testov
- Vytvorenie deterministických testov
- V teste sa venujeme jedinému prípadu použitia
- Zameranie sa na maximálne pokrytie testov
- Návrh testov aby boli čo najrýchlejšie
- Minimalizácia závislostí testov
- Automatizácia testov

### **2.1.7 Známe nástroje pre unit testovanie**

- JUnit – testovací framework pre programovací jazyk Java
- NUnit – testovací framework pre všetky .Net jazyky
- DBUnit – rozšírenie JUnit, pre veľké databázov riadené projekty
- HTMLUnit – nástroj na testovanie, používa sa na testovanie web aplikácií
- PHPUnit – testovací framework pre programovací jazyk PHP
- SimpleTest – testovací framework pre programovací jazyk PHP
- Embunit – testovací nástroj pre programovací jazyk C a C++

## **2.2 Príklad unit testovania v Jave**

Pre ukážku unit testovania v jave použijem knižnicu JUnit 5. Je to najpoužívanější java framework pre unit testovanie. Pravidlá spúšťania testov definujeme pomocou anotácií a samotné testy vyhodnocujeme pomocou funkcií z triedy Assertions.

### 2.2.1 Najpoužívanéjšie JUnit anotácie

- `@Test` – Táto anotácia označuje, že metóda je testovacia metóda.
- `@ParametrizedTest` - Parametrizované testy umožňujú spustiť test viackrát s rôznymi argumentami. Okrem toho je nutné deklarovať aspoň jeden zdroj, ktorý bude poskytovať argumenty pre každé volanie testu a potom tieto argumenty použiť v testovacej metóde.
- `@ValueSource` – Anotácia špecifikuje zdroj argumentov pre parametrizované testy.
- `@RepeatedTest` - Zopakujte test špecifikovaním celkového počtu požadovaných opakovaní.
- `@DisplayName` - Testovacie triedy a testovacie metódy môžu deklarovať vlastné názvy, ktoré budú zobrazované test runnermi a v test reportoch.
- `@BeforeEach` - Označuje, že anotovaná metóda by sa mala vykonať pred každou testovacou metódou.
- `@AfterEach` - Označuje, že anotovaná metóda by sa mala vykonať po každej testovacej metóde.
- `@BeforeAll` - Táto anotácia označuje metódu, ktorá sa vykoná raz pred všetkými testami
- `@AfterAll` - Táto anotácia označuje metódu, ktorá sa vykoná raz po všetkých testoch
- `@Tag` - Túto anotáciu môžeme použiť na deklarovanie tagov pre testy či už na úrovni triedy alebo metódy. Testy potom môžeme pomocou nich filtrovať.
- `@Disabled` - Anotácia sa používa na zakázanie alebo preskočenie testov na úrovni triedy alebo metódy. Pri deklarácii na úrovni triedy sa preskočia všetky metódy `@test`. Keď použijeme anotáciu na úrovni metódy, preskočí sa iba anotovaná metóda.

### 2.2.2 Najpoužívanéjšie metódy z triedy Assertions

- `assertTrue` – Overí podmienku či je pravdivá
- `assertFalse` - Opačná metóda k `assertTrue`
- `assertNull` – Overí či hodnota je null
- `assertNotNull` – Opačná metóda k `assertNull`
- `assertEquals` – Overí, či dva parametre sú rovnaké
- `assertNotEquals` – Opačná metóda k `assertEquals`

- `assertSame` – Overí, či hodnoty dvoch parametrov referencujú rovnaký objekt
- `assertNotSame` – Opačná metóda k `assertSame`
- `assertLinesMatch` – Overí, či dva listy stringov sú rovnaké
- `assertArrayEquals` – Overí, či dve polia sú rovnaké
- `assertIterableEquals` – Overí, či dva iterable sú rovnaké
- `assertThrows` – Overí, či funkcia na vstupe vyhodí správnu výnimku
- `assertDoesNotThrow` – Opačná metóda k `assertThrows`
- `assertTimeout` – Overí, či vstupná funkcia skončí pred špecifikovaným timeoutom
- `assertAll` – Táto funkcia umožňuje vytvorenie skupinových asertions, kde sa spustí každá assertion a ich zlyhania sú hlásené spoločne.
- `fail` – táto metóda spôsobí zlyhanie testu

### 2.2.3 Testovanie

V triede `Calculator` sa nachádzajú statické metódy, ktoré počítajú prvočísla. Niektoré metódy majú chybnú implementáciu, pre ukážku funkcionality testov. Nasledujúce obrázky obsahujú vytvorené ukážkové testy a ich výsledky.

```

class CalculatorTest {

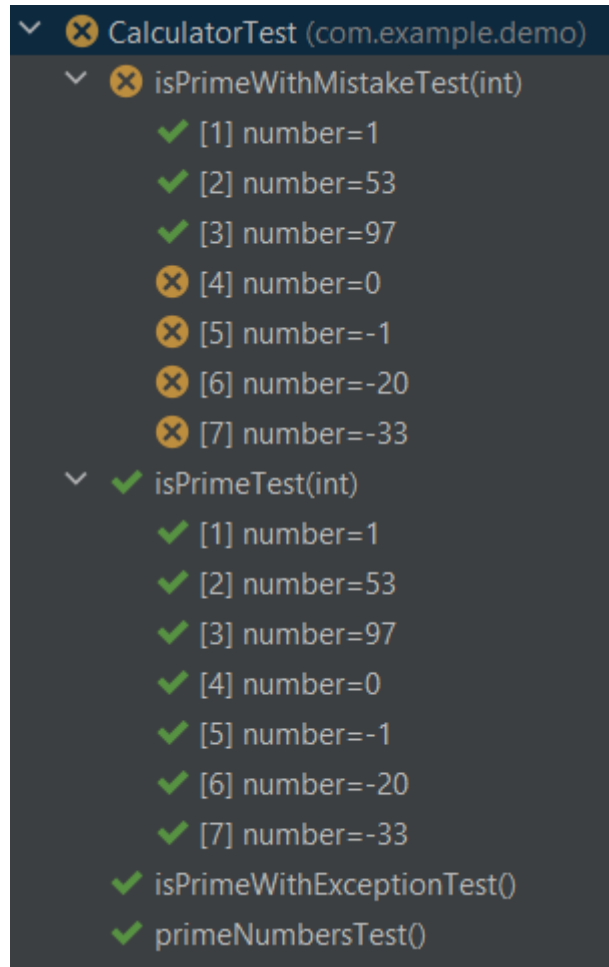
    @Test
    void primeNumbersTest() {
        List<Integer> result = primeNumbers(10);
        assertIterableEquals(Arrays.asList(1,3,5,7), result);
        result = primeNumbers(-10);
        assertIterableEquals(List.of(), result);
        result = primeNumbers(0);
        assertIterableEquals(List.of(), result);
        result = primeNumbers(1);
        assertIterableEquals(Arrays.asList(1), result);
    }

    @ParameterizedTest
    @ValueSource(ints = { 1, 53, 97, 0, -1, -20, -33})
    void isPrimeTest(int number) {
        if(number > 0)
            assertTrue(isPrime(number));
        else
            assertFalse(isPrime(number));
    }

    @ParameterizedTest
    @ValueSource(ints = { 1, 53, 97, 0, -1, -20, -33})
    void isPrimeWithMistakeTest(int number) {
        if(number > 0)
            assertTrue(isPrimeWithMistake(number));
        else
            assertFalse(isPrimeWithMistake(number));
    }

    @Test
    void isPrimeWithExceptionTest() {
        assertThrows(IllegalArgumentException.class, () -> {isPrimeWithException(0)});
        assertThrows(IllegalArgumentException.class, () -> {isPrimeWithException(-2)});
        assertDoesNotThrow(() -> {isPrimeWithException(1)});
        assertDoesNotThrow(() -> {isPrimeWithException(6)});
    }
}

```



## 3 Integračné testy

Cieľom integračných testov je overenie, či časti softvéru spolu fungujú. Tento typ testovania sa využíva hlavne pri väčších projektoch. Integračnými testami môžeme testovať napríklad endpointy aplikácie a porovnať očakávaný výstup.

### 3.1 Konfigurácia

`@SpringBootTest` – potrebný pre spustenie celej aplikácie, vytvorí `ApplicationContext` potrebný pre načítanie tried pomocou dependancy injection

`@AutoConfigureMockMvc` – Automatické nakonfigurovanie `MockMvc`

```
@SpringBootTest
@AutoConfigureMockMvc
```

### 3.2 MockMvc

Hlavný vstupný bod pre komunikáciu s aplikáciou. Vďaka `MockMvc` môžeme volať requesty bez toho aby sa musel spúšťať server.

```
@Autowired
private MockMvc mvc;
```

### 3.3 Testovanie

Endpointy voláme pomocou metódy `perform` v ktorej môžeme volať rôzne metódy a následne môžeme zadať čo očakávame, aký status by mal endpoint vrátiť, aké hlavičky by mal vrátiť, aké dáta má vrátiť a ďalšie parametre.

### 3.3.1 Metódy

Metódy voláme pomocou triedy `MockMvcRequestBuilder` ktorý obsahuje statické metódy ako `get` a `post` na ktoré je možné následne zadať aký typ dát posielame a vložiť dané dáta.

GET

```
mvc.perform(get( urlTemplate: "/user/list"))
```

POST

```
mvc.perform(post( urlTemplate: "/user/create")
    .contentType(MediaType.APPLICATION_JSON)
    .content(mapper.writeValueAsString(newUser)))
```

### 3.3.2 Výstup

Pomocou triedy `MockMvcResultHandler` a jej statickej metódy `print()` test zobrazí všetky podrobnosti o requeste. Zároveň je možné porovnávať výsledky s očakávaným výsledkom pomocou triedy `MockMvcResultMatchers` a jej metód:

- `status()` – status odpovede
- `content()`
  - formu odpovede (typ dát, kódovanie...)
  - samotnú odpoveď - porovnanie základného stringu (`string()`)
- `header()` – hlavičku odpovede
- `jsonPath()` – porovnanie konkrétnej hodnoty v json-e

## 3.4 Testy

```
@Test
public void listUsers() throws Exception {
    mvc.perform(get( uriTemplate: "/user/list"))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath( expression: "$").isArray()) // result je pole
        .andExpect(jsonPath( expression: "$[5]").doesNotExist()) // v poli je len 5 položiek
        .andExpect(jsonPath( expression: "$[0].username").value( expectedValue: "a")); // prvý používateľ má username 'a'
};
```

```
@Test
public void createUser() throws Exception {
    User newUser = new User( id: 5, username: "f", password: "f"); // Vytvorenie nového používateľa
    ObjectMapper mapper = new ObjectMapper();
    mvc.perform(post( uriTemplate: "/user/create")
        .contentType(MediaType.APPLICATION_JSON) // Nastavenie typu odosielaných dát
        .content(mapper.writeValueAsString(newUser))) // Vloženie dát do requestu
        .andDo(print())
        .andExpect(status().isOk()) // Očakávaný status
        .andExpect(header().string( name: "Age", value: "1")) // Očakávaná hlavička
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON)) // Očakávaný typ dát
        .andExpect(jsonPath( expression: "$.username").value( expectedValue: "f"));
};
```



## 4 End-to-end testy

Softvérové systémy sú dnes zložité a prepojené mnohými podsystémami. Ak niektorý z podsystémov zlyhá, môže dôjsť k výpadku celého softvérového systému. Toto je veľké riziko a dá sa mu vyhnúť end-to-end testovaním. End-to-end testovanie je technika, ktorá testuje celý softvérový produkt od začiatku do konca, aby sa zabezpečilo, že tok aplikácie sa bude správať podľa očakávania. Hlavným účelom end-to-end testovania, je testovanie skúsenosti koncového používateľa simuláciou scenára skutočného používateľa a overením testovaného systému a jeho komponentov z hľadiska integrácie a integrity údajov

End-to-end testovanie je multi-disciplinárna činnosť, ktorá zahŕňa vývojárov, testerov, manažérov a používateľov. - Vývojári profitujú z toho, že väčšinu testovania a zabezpečovania kvality môžu presunúť na tím QA (quality assurance), čím uvoľnia vývojárom, ktorý následne môže implementovať a pridávať ďalšie funkcie do aplikácie. Pre testerov je jednoduchšie písať E2E testy, pretože sú založené na správaní používateľa, ktoré boli možné pozorovať počas testovania používateľnosti. E2E testovanie zjednodušuje zachytávanie problémov pred uvoľnením softvéru koncovým používateľom Identifikáciou dôležitosti pracovného toku pre používateľov v reálnom svete pomáha manažérom, ktoré úlohy uprednostniť vo vývoji.

### 4.1 Dostupné nástroje

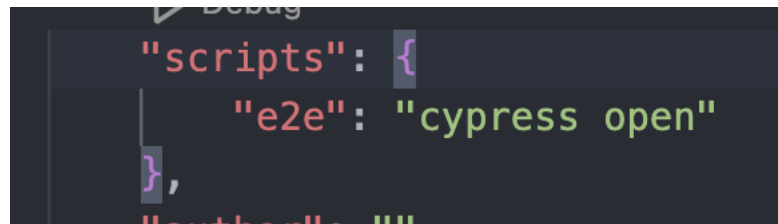
- Autify
- testRigor
- QA Wolf
- SmartBear
- Cypress

## 4.2 CypressJS

Je to JavaScriptová knižnica na testovanie UI front endov moderných webstránok. Cypress umožňuje písať rôzny typy testov ( E2E testy, test komponentov, integračné testy a Unit testy), vie otestovať všetko čo beží v prehliadači.

## 4.3 Inštalácia a spustenie

Na inštaláciu knižnice CypressJS je potrebné mať nainštalovaný NodeJS. Po nainštalovaní, vytvoríme priečinok, do ktorého stiahneme CypressJS. Predtým ako začneme sťahovať Cypress, musíme priečinok inicializovať príkazom „npm init“, preklikáme otázky a následným príkazom stiahneme Cypress „*npm install cypress --save-dev*“. V priečinku sa nám vygeneroval nový priečinok „cypress“ v tomto priečinku budeme písať príkazy na testovanie. Cypress treba spustiť. Na spustenie CypressJS potrebujeme pridať do súboru „package.json“ medzi „scripts“ jeden riadok vid’ na obrázku.



```
"scripts": {  
  "e2e": "cypress open"  
},  
"author": ""
```

Teraz môžeme Cypress spustiť konzolovým príkazom „npm run e2e“. Zobrazí sa nám uvítacie okno Cypress-u. Máme tu voľbu medzi E2E testovaniu a test komponentov. Následne je možné si vybrať v akom prehliadači vykonávať testovanie.

### 4.3.1 Test Login

Kód testu pre prihlásenie môžeme vidieť na obrázku.

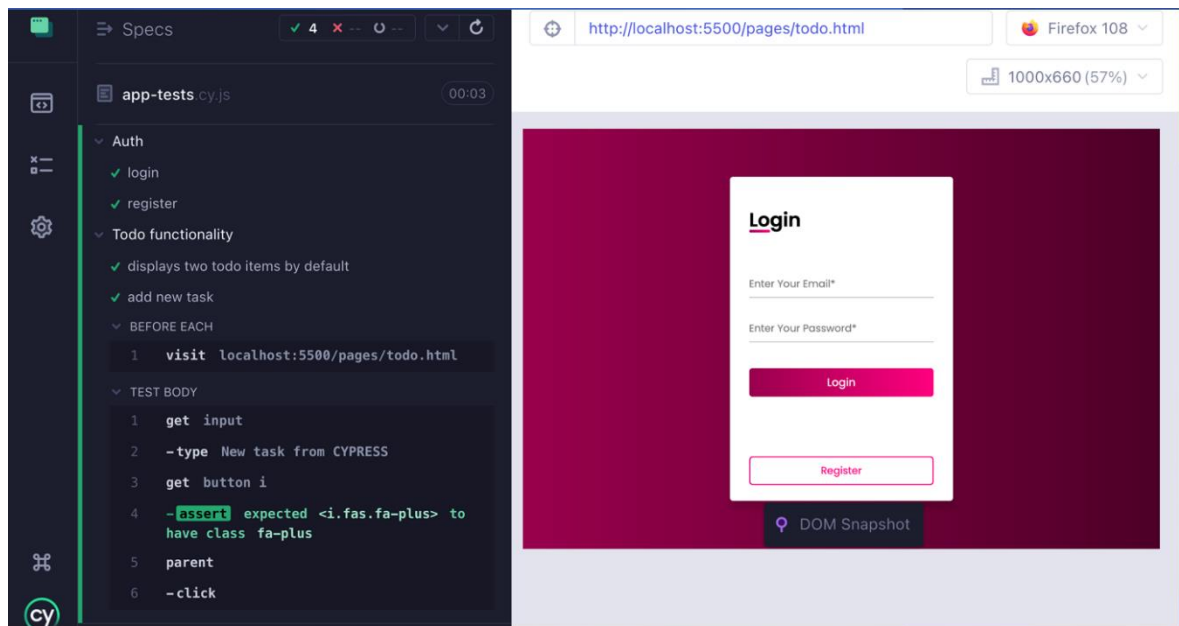
```
it("login", () => {  
  // Verify page  
  cy.visit("localhost:5500");  
  cy.get("title").should("contain", "Login");  
  
  // Type in input  
  cy.get("#email").type("example@gmail.com");  
  cy.get("#password").type("password123");  
  
  // Submit Login  
  cy.get("button").should("contain", "Login").click({ timeout:  
    1000 });  
});
```

It je Individuálny test, prijíma dva parametre, reťazec názov testu, callback funkcia, ktorá sa ma vykonať. Kroky, ktoré sme robili :

1. Navštívime stránku, ktorú chceme otestovať
2. Získame názov stránky v elemente <title> a pozrieme sa či obsahuje reťazec „Login“
3. Nájdeme vstupné polia pre email a password
4. Vyplníme polia hodnotami
5. Odošleme žiadosť o prihlásenie

### 4.3.2 Rozhranie

Cypress rozhranie vyzerá nasledovne. Na ľavej strane vidíme testy, ktoré si vieme rozkliknúť, po rozkliknutí sa zobrazí postupnosť vykonávanie príkazov. Keď po tých príkazoch prechádzame myškou, tak na pravej strane, uvidíme snímky priebehu ako sa vykonávajú príkazy.



## **5 Výkonnostné testy**

### **5.1 Teoretický úvod**

#### **5.1.1 Definícia**

Výkonnostné testovanie predstavuje proces identifikácie a eliminácie tých častí softvéru, ktoré ho limitujú najmä v týchto kategóriach:

- Rýchlosť (schopnosť aplikácie plynulo bežať)
- Responzivita
- Stabilita (schopnosť aplikácie odolávať zmenám v miere záťaže)
- Spoľahlivosť (schopnosť aplikácie chovať sa podľa očakávania)
- Škálovateľnosť (schopnosť aplikácie udržiavať správny chod pri zvýšení počtu používateľov)
- Využitie výpočtových zdrojov

#### **5.1.2 Využitie výkonnostných testov**

Výkonnostné testy sa používajú takmer výhradne pre softvér s klient-server architektúrou.

Najčastejšími problémami tohto typu softvéru, ktoré výkonnostné testy riešia, sú:

- Pomalé načítavanie
- Dlhá doba odpovede na požiadavku
- Zlá škálovateľnosť
- Bottlenecky (slabé články aplikácie a hardvéru, na ktorom je aplikácia nasadená)

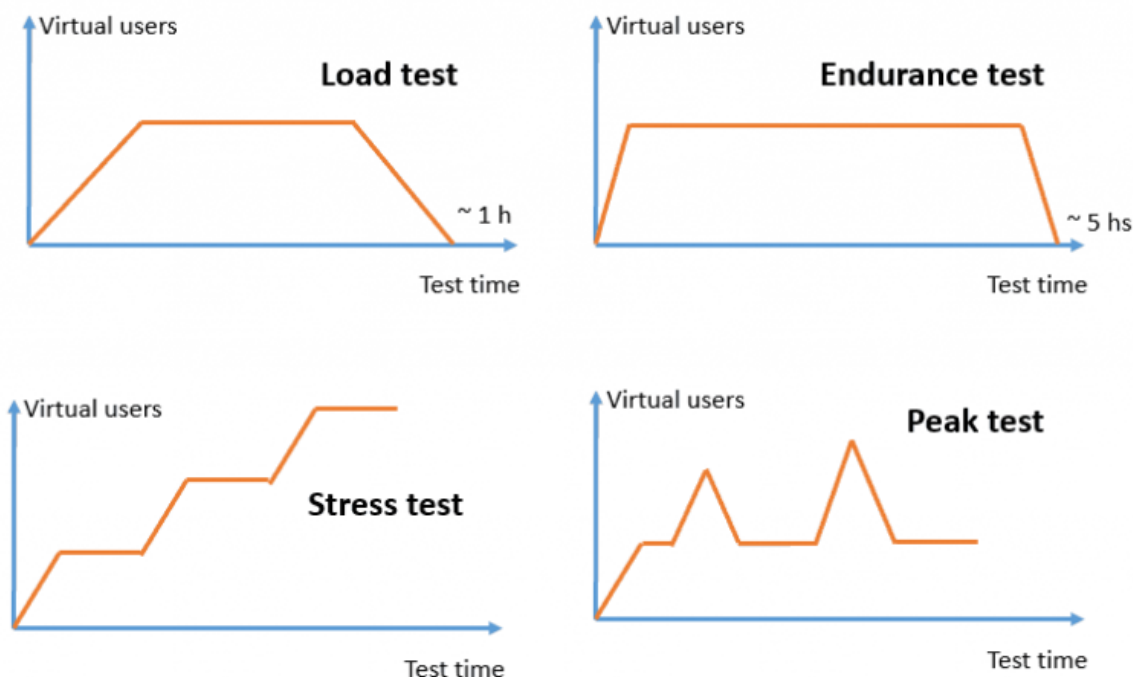
### 5.1.3 Proces realizácie výkonnostných testov



Obrázok 1: Schéma procesu realizácie výkonnostných testov. Prevzaté z <https://www.guru99.com/performance-testing.html>

### 5.1.4 Typy výkonnostných testov

- Load testing – testovanie očakávanej záťaže
- Stress testing – testovanie hraničnej záťaže
- Spike testing – testovanie náhleho zvýšenia záťaže
- Endurance testing – testovanie dlhodobej záťaže
- Volume testing – testovanie správneho chodu aplikácie pri zaťažení databázy veľkým dátovým objemom
- Scalability testing – testovanie možností zvýšenia akceptovateľnej záťaže



Obrázok 2: Porovnanie typov výkonnostných testov pomocou grafu závislosti počtu virtuálnych používateľov (threads) na testovacom čase. Obrázok prevzatý z <https://abstracta.us/blog/performance-testing/types-performance-tests/>.

### **5.1.5 Metriky sledované pri výkonnostných testoch**

Voľba sledovanej metriky pri výkonnostných testoch závisí od cieľov testovania – nefunkcionálnych požiadaviek. Medzi najčastejšie sledované metriky patria:

- Metriky odpovedí (miera chybovosti, priemerný a najvyšší čas odpovede)
- Objemové metriky (počet konkurentných používateľov, požiadaviek za sekundu, priepustnosť – throughput)
- Metriky výpočtových zdrojov (využitie CPU, RAM, disku)

## **5.2 Príklad výkonnostného testu s využitím Apache JMeter**

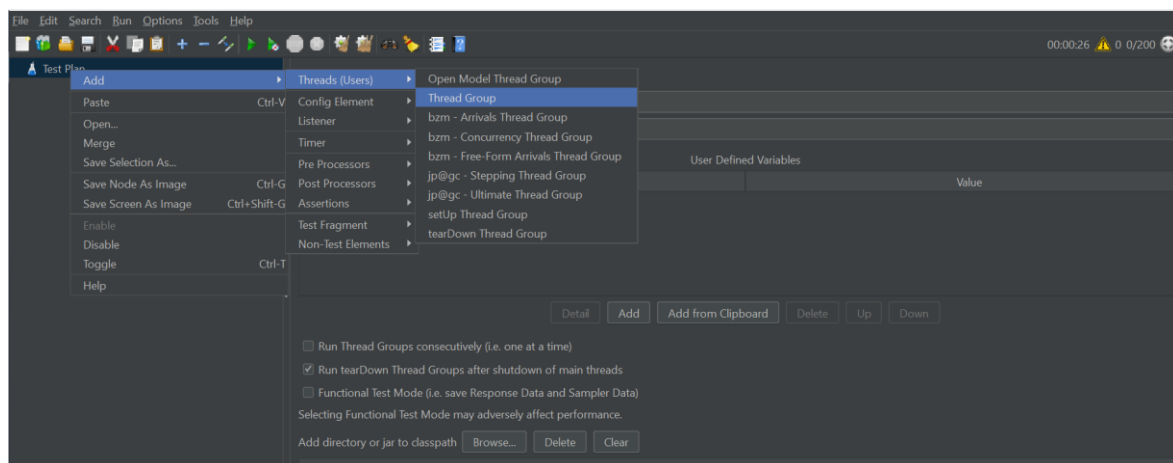
### **5.2.1 Softvér JMeter**

JMeter je open-source nástroj pre výkonnostné testovanie najmä webových klient-server aplikácií, ktorý je dostupný v čase publikácie tohto dokumentu na stiahnutie na adrese [https://jmeter.apache.org/download\\_jmeter.cgi](https://jmeter.apache.org/download_jmeter.cgi).

Pre spustenie programu je nutné mať nainštalované Java JDK verzie 8 a vyššie. V operačnom systéme Windows sa program spúšťa otvorením súboru jmeter.bat.

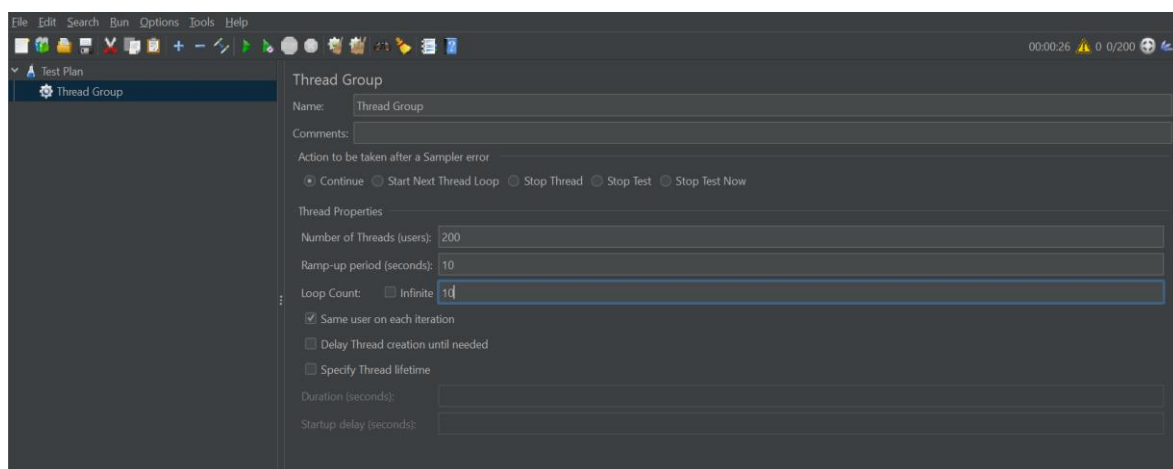
### **5.2.2 Proces realizácie testu**

1. Vytvorenie “Thread Group” (skupina virtuálnych používateľov, pre ktorých bude ďalej špecifikovaná interakcia so softvérom, ktorú chceme otestovať).



## 2. Nastavenie parametrov Thread Group

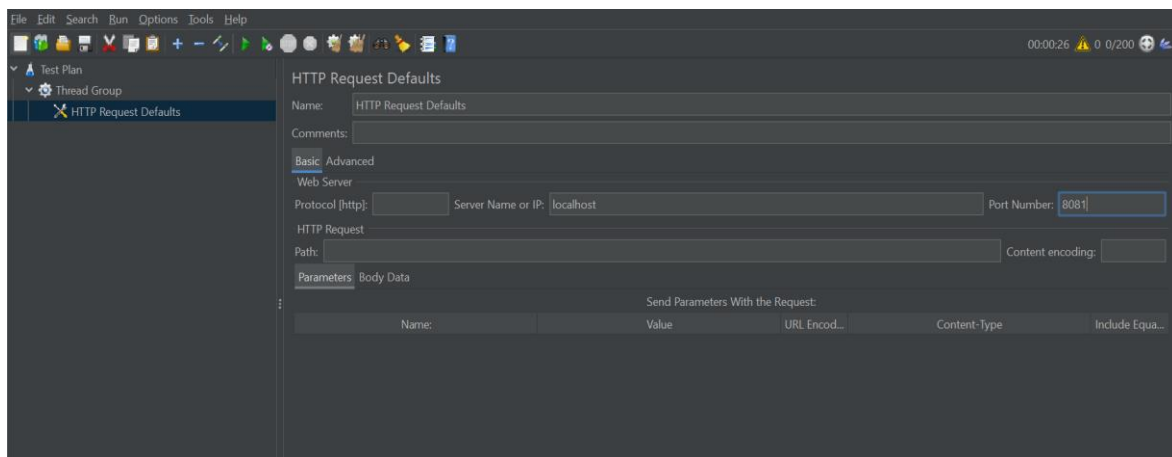
- Počet vlákien (virtuálnych používateľov)
- Interval pre postupné zvyšovanie záťaže (počet používateľov sa rozpočíta na tento interval, záťaž je postupná)
- Počet cyklov



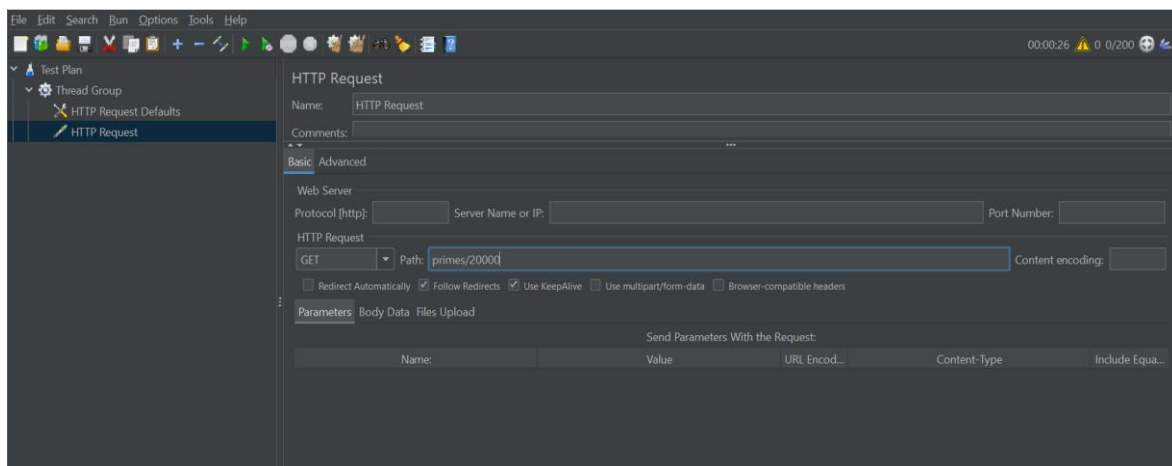
## 3. Pridanie JMeter elementov

- Config element – HTTP Request Defaults – špecifikuje základné informácie o endpointoch aplikácie (doména, port)



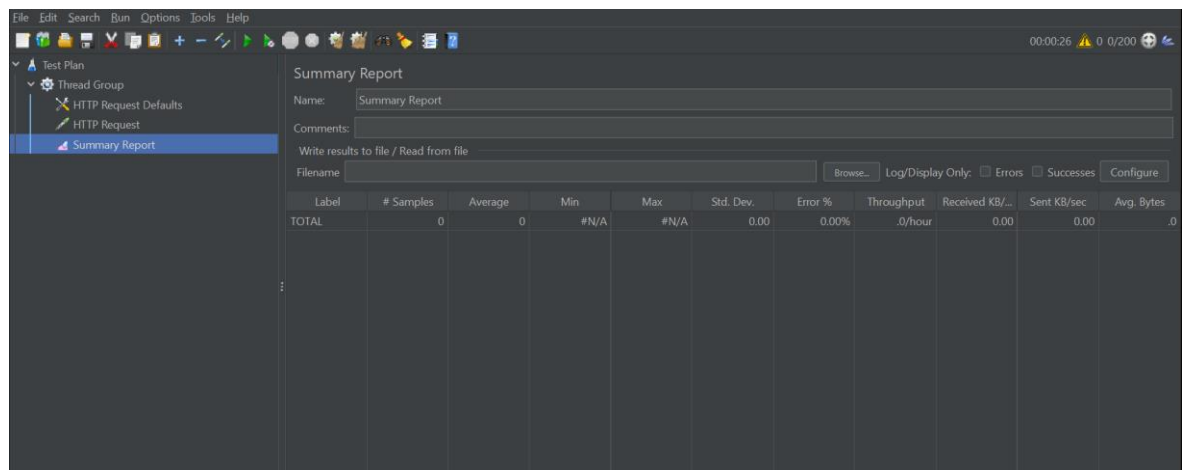


b. Sampler – HTTP Request – špecifikuje testovanú interakciu používateľa s aplikáciou.



c. Listener

i. Summary Report – listener pre sledovanie priebehu testovania s real-time zobrazením metrík.



ii. View Results in Table – listener použitý pre validáciu testu  
(na obrázku zobrazený po zbehnutí testu)

The screenshot shows the JMeter View Results in Table window. The left sidebar has 'View Results in Table' selected. The main area displays a table with columns: Sample #, Thread Name, Label, Sample Time..., Status, Bytes, Sent Bytes, Latency, and Connect Time... The table contains 20 rows of test results, all with a status of 'Success' (green checkmark). The window title is 'View Results in Table' and the name is 'View Results in Table'.

Sample #	Thread Name	Label	Sample Time...	Status	Bytes	Sent Bytes	Latency	Connect Time...
1981	Thread Grou...	HTTP Request	143	Success	12329	128	143	0
1982	Thread Grou...	HTTP Request	178	Success	12329	128	178	0
1983	Thread Grou...	HTTP Request	205	Success	12329	128	205	0
1984	Thread Grou...	HTTP Request	151	Success	12329	128	151	0
1985	Thread Grou...	HTTP Request	205	Success	12329	128	205	0
1986	Thread Grou...	HTTP Request	155	Success	12329	128	155	0
1987	Thread Grou...	HTTP Request	122	Success	12329	128	121	0
1988	Thread Grou...	HTTP Request	134	Success	12329	128	134	0
1989	Thread Grou...	HTTP Request	133	Success	12329	128	133	0
1990	Thread Grou...	HTTP Request	87	Success	12329	128	87	0
1991	Thread Grou...	HTTP Request	87	Success	12329	128	87	0
1992	Thread Grou...	HTTP Request	86	Success	12329	128	86	0
1993	Thread Grou...	HTTP Request	95	Success	12329	128	95	0
1994	Thread Grou...	HTTP Request	89	Success	12329	128	89	0
1995	Thread Grou...	HTTP Request	83	Success	12329	128	83	0
1996	Thread Grou...	HTTP Request	82	Success	12329	128	82	0
1997	Thread Grou...	HTTP Request	89	Success	12329	128	89	0
1998	Thread Grou...	HTTP Request	90	Success	12329	128	90	0
1999	Thread Grou...	HTTP Request	79	Success	12329	128	78	0
2000	Thread Grou...	HTTP Request	79	Success	12329	128	78	0

4. Spustenie testovania

5. Vyhodnotenie výsledkov

### 5.2.3 Testovacie plány použité v prezentácii

Testovacie plány sú dostupné v priloženom kóde na ceste  
demo\src\main\resources\JMeterPlans v dvoch súboroch :

Primes\_load\_200.jmx, Primes\_load\_5000.jmx. Líšia sa v počte používateľov v rámci thread group. Pre oba platí táto špecifikácia:

### Thread Group

- Počet virtuálnych používateľov – 200 (resp. 5000)
- Interval postupného zaťaženia – 10 sekúnd
- Počet opakovaní – 10

### Testovaný endpoint

- Typ - GET
- Umiestnenie - localhost:XXXX/primes/{n}
- Odpoveď je zoznam prvočísel menších ako n.

## 5.2.4 Interpretácia výsledkov s použitím Summary Report listeneru



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
HTTP Reque...	26611	1486	0	29096	5262.49	90.55%	700.3/sec	2426.28	8.27	3547.6
TOTAL	26611	1486	0	29096	5262.49	90.55%	700.3/sec	2426.28	8.27	3547.6

Sledovanou metrikou je miera chybovosti (Error %), ktorá uvádza pomer nesprávnych odpovedí na požiadavku k celkovému počtu odpovedí. V prípade ukážkového testu nenulová hodnota znamená neschopnosť správneho behu aplikácie v prípade špecifikovaného počtu konkurentných používateľov (v našom prípade 5000).

## 5.2.5 Validácia testu

Správnosť testu možno overiť pomocou View Results in Table listeneru, ak počet zachytených požiadaviek zodpovedá počtu používateľov vynásobenému počtom cyklov po normálnom priebehu testovania (bez predčasného

ukončenia), a zároveň správnosť odpovedí zodpovedá údaju Error %, test možno považovať za správny.

# Zoznam použitej literatúry

<https://www.guru99.com/unit-testing-guide.html>

<https://www.spiceworks.com/tech/devops/articles/what-is-unit-testing/>

<https://devqa.io/junit-5-annotations/>

<https://www.appsdeveloperblog.com/an-overview-of-junit5-assertions-with-examples/>

<https://abstracta.us/blog/performance-testing/types-performance-tests/>

<https://www.guru99.com/performance-testing.html>

<https://jmeter.apache.org/>