**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

***Факультет Информационных технологий***

***Кафедра Информатики и информационных технологий***

**направление подготовки**

**09.03.02 «Информационные системы и технологии»**

**Лабораторная работа № 3**

**Дисциплина: Тестирование программного обеспечения**

**Тема: Написание первых автотестов на Python (или Java) + Selenium WebDriver**

**Выполнил: студент группы 221-371**

**Евдокимов Артемий Русланович**

**(Фамилия И.О )**

**Дата, подпись \_\_16.09.2025\_\_ *\_\_\_\_\_\_\_\_\_\_\_***

**(Дата) (Подпись)**

**Проверил: \_\_*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

**(Фамилия И.О., степень, звание)**

**Дата, подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *\_\_\_\_\_\_\_\_\_\_\_***

**(Дата) (Подпись**

**Замечания: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Москва**

**2025**

**Цель работы**

Познакомиться с концепцией нагрузочного тестирования, научиться создавать тестовые скрипты для k6 и проводить анализ производительности веб-приложений.

**Объект тестирования**

REST API, реализованное на FastAPI, со следующими эндпоинтами:

* GET / - основная страница API
* GET /status - проверка статуса API
* GET /users - получение списка пользователей
* POST /users - создание нового пользователя
* GET /users/{id} - получение пользователя по ID
* GET /heavy - эндпоинт с тяжелыми вычислениями

**Реализация**

**1. Тестовое API (api.py)**

from fastapi import FastAPI, HTTPException

import time

import random

app = FastAPI(title="Тестирование 8 лаба")

# In-memory "database" for testing

users\_db = []

posts\_db = []

@app.get("/")

async def root():

    """Основная страница API"""

    return {

        "message": "Страница тестирования 8 лаба",

        "status": "active",

        "timestamp": time.time()

    }

@app.get("/users")

async def get\_users():

    """Получить список всех пользователей"""

    # Имитация задержки базы данных

    delay = random.uniform(0.01, 0.1)

    time.sleep(delay)

    return {

        "users": users\_db,

        "count": len(users\_db),

        "delay\_seconds": delay

    }

@app.post("/users")

async def create\_user(name: str, email: str):

    """Создать нового пользователя"""

    if not name or not email:

        raise HTTPException(status\_code=400, detail="Имя или имеил написаны с ошибкой!")

    user = {

        "id": len(users\_db) + 1,

        "name": name,

        "email": email,

        "created\_at": time.time()

    }

    users\_db.append(user)

    return {"message": "Пользователь создан", "user": user}

@app.get("/users/{user\_id}")

async def get\_user(user\_id: int):

    """Получить пользователя по ID"""

    # Имитация обработки запроса

    processing\_time = random.uniform(0.05, 0.2)

    time.sleep(processing\_time)

    if user\_id < 1 or user\_id > len(users\_db):

        raise HTTPException(status\_code=404, detail="Пользователь не найден")

    return {

        "user": users\_db[user\_id - 1],

        "processing\_time": processing\_time

    }

users\_db.append(user)

    return {"message": "Пользователь создан", "user": user}

@app.get("/users/{user\_id}")

async def get\_user(user\_id: int):

    """Получить пользователя по ID"""

    # Имитация обработки запроса

    processing\_time = random.uniform(0.05, 0.2)

    time.sleep(processing\_time)

    if user\_id < 1 or user\_id > len(users\_db):

        raise HTTPException(status\_code=404, detail="Пользователь не найден")

    return {

        "user": users\_db[user\_id - 1],

        "processing\_time": processing\_time

    }

@app.get("/status")

async def status\_check():

    """Эндпоинт для проверки статуса API"""

    return {

        "status": "healthy",

        "timestamp": time.time(),

        "version": "1.0.0"

    }

@app.get("/heavy")

async def heavy\_operation():

    """Эндпоинт с тяжелой операцией для тестирования производительности"""

    # Имитация тяжелой вычислятельной операции

    computation\_time = random.uniform(0.1, 0.5)

    time.sleep(computation\_time)

    # Имитация вычислений

    result = sum(i \* i for i in range(10000))

    return {

        "message": "Завершены тяжелые вычисления",

        "computation\_time": computation\_time,

        "result": result

    }

if \_\_name\_\_ == "\_\_main\_\_":

    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)

**2. Скрипты нагрузочного тестирования**

**Основной тестовый скрипт (load\_test.js):**

import http from 'k6/http';

import { check, sleep } from 'k6';

import { Rate, Trend, Counter } from 'k6/metrics';

const errorRate = new Rate('errors');

const responseTimeTrend = new Trend('response\_time');

const requestsCounter = new Counter('total\_requests');

export const options = {

  stages: [

    // Первый сценарий: 10 виртуальных пользователей в течение 30 секунд

    { duration: '30s', target: 10 },

    // Второй сценарий: 50 виртуальных пользователей в течение 10 секунд

    { duration: '10s', target: 50 },

    // Фаза снижения нагрузки

    { duration: '5s', target: 0 },

  ],

  thresholds: {

    http\_req\_duration: ['p(95)<500'], // 95% запросов должны быть быстрее 500ms

    errors: ['rate<0.1'],

  },

};

// Базовый URL API

const BASE\_URL = 'http://localhost:8000';

export default function () {

  // Список эндпоинтов для тестирования

  const endpoints = [

    '/',

    '/status',

    '/users',

    '/heavy'

  ];

// Выбираем случайный эндпоинт

  const endpoint = endpoints[Math.floor(Math.random() \* endpoints.length)];

  const url = `${BASE\_URL}${endpoint}`;

  // Параметры запроса

  const params = {

    headers: {

      'Content-Type': 'application/json',

      'User-Agent': 'k6-load-test',

    },

    tags: {

      endpoint: endpoint,

    },

  };

  // Отправляем GET-запрос

  const response = http.get(url, params);

  // Считаем кастомные метрики

  requestsCounter.add(1);

  responseTimeTrend.add(response.timings.duration);

  errorRate.add(response.status >= 400);

  // Проверяем ответ

  const checkResult = check(response, {

    'status is 200': (r) => r.status === 200,

    'response time < 1000ms': (r) => r.timings.duration < 1000,

    'has valid JSON': (r) => {

      try {

        JSON.parse(r.body);

        return true;

      } catch (e) {

        return false;

      }

    },

  });

  // Добавляем случайную задержку между запросами (0.5-2 секунды)

  sleep(Math.random() \* 1.5 + 0.5);

}

// Функция setup выполняется один раз перед тестом

export function setup() {

  console.log('Starting load test...');

  console.log('Target URL:', BASE\_URL);

  // Создаем несколько тестовых пользователей

  const responses = [];

  for (let i = 1; i <= 5; i++) {

    const url = `${BASE\_URL}/users`;

    const payload = JSON.stringify({

      name: `Test User ${i}`,

      email: `user${i}@test.com`

    });

    const params = {

      headers: {

        'Content-Type': 'application/json',

      },

    };

    const response = http.post(url, payload, params);

    responses.push(response);

  }

  return { setup\_complete: true };

}

// Функция teardown выполняется после завершения теста

export function teardown(data) {

  console.log('Load test completed');

  console.log('Setup data:', data);

}

**Тест 10 виртуальных пользователей в течение 30 секунд (test\_10vus\_30s.js):**

import http from 'k6/http';

import { check, sleep } from 'k6';

export const options = {

  stages: [

    { duration: '5s', target: 10 },  // Плавный рост до 10 пользователей

    { duration: '30s', target: 10 }, // Удержание 10 пользователей

    { duration: '5s', target: 0 },   // Плавное снижение

  ],

};

export default function () {

  const response = http.get('http://localhost:8000/status');

  check(response, {

    'status is 200': (r) => r.status === 200,

    'response time < 500ms': (r) => r.timings.duration < 500,

  });

  sleep(1);

}

**Тест 50 виртуальных пользователей в течение 10 секунд (test\_50vus\_10s.js):**

import http from 'k6/http';

import { check, sleep } from 'k6';

export const options = {

  stages: [

    { duration: '5s', target: 50 },  // Быстрый рост до 50 пользователей

    { duration: '10s', target: 50 }, // Удержание 50 пользователей

    { duration: '3s', target: 0 },   // Быстрое снижение

  ],

  thresholds: {

    http\_req\_duration: ['p(95)<800'],

    http\_req\_failed: ['rate<0.15'],

  },

};

export default function () {

  const endpoints = ['/', '/status', '/users', '/heavy'];

  const endpoint = endpoints[Math.floor(Math.random() \* endpoints.length)];

  const url = `http://localhost:8000${endpoint}`;

  const response = http.get(url);

  check(response, {

    'status is 200': (r) => r.status === 200,

  });

  sleep(0.5);

}

**Результаты автоматического тестирования**

**Тест 1: Основной нагрузочный тест (load\_test.js)**

**Параметры теста:**

* Продолжительность: 46.1 секунды
* Максимальное количество VUs: 50
* Сценарий: 3 этапа (10 VUs/30s → 50 VUs/10s → 0 VUs/5s)

**Результаты:**

Проверки (checks):

- Всего проверок: 1230

- Успешных: 99.34% (1222)

- Неудачных: 0.65% (8)

Детали проверок:

✓ status is 200: 100% успеха

✗ response time < 1000ms: 98% успеха (8 провалов из 410 итераций)

✓ has valid JSON: 100% успеха

Производительность:

- HTTP запросы: 415 (8.99907 запросов/секунду)

- Неудачные запросы: 1.20% (5 из 415)

- Время ответа:

\* Среднее: 270.19ms

\* Медиана: 133.74ms

\* Максимум: 7.1s

\* 95-й процентиль: 804.9ms

Пороги (thresholds):

✓ errors: rate<0.1 - выполнено (0.00%)

✗ http\_req\_duration: p(95)<500 - НЕ ВЫПОЛНЕНО (p(95)=804.9ms)

**Тест 2: 10 виртуальных пользователей в течение 30 секунд (test\_10vus\_30s.js)**

**Параметры теста:**

* Продолжительность: 40.1 секунды
* Максимальное количество VUs: 10
* Сценарий: 3 этапа

**Результаты:**

Проверки (checks):

- Всего проверок: 706

- Успешных: 99.57% (703)

- Неудачных: 0.42% (3)

Детали проверок:

✓ status is 200: 100% успеха

✗ response time < 500ms: 99% успеха (3 провала из 353 итераций)

Производительность:

- HTTP запросы: 353 (8.806405 запросов/секунду)

- Неудачные запросы: 0%

- Время ответа:

\* Среднее: 10.13ms

\* Медиана: 1.05ms

\* Максимум: 1.22s

\* 95-й процентиль: 1.87ms

**Тест 3: 50 виртуальных пользователей в течение 10 секунд (test\_50vus\_10s.js)**

**Параметры теста:**

* Продолжительность: 18.7 секунды
* Максимальное количество VUs: 50
* Сценарий: 3 этапа

**Результаты:**

Проверки (checks):

- Всего проверок: 669

- Успешных: 100% (669)

- Неудачных: 0%

Детали проверок:

✓ status is 200: 100% успеха

Производительность:

- HTTP запросы: 669 (35.72487 запросов/секунду)

- Неудачные запросы: 0%

- Время ответа:

\* Среднее: 593.36ms

\* Медиана: 469.31ms

\* Максимум: 2.44s

\* 95-й процентиль: 1.64s

Пороги (thresholds):

✓ http\_req\_failed: rate<0.15 - выполнено (0.00%)

✗ http\_req\_duration: p(95)<800 - НЕ ВЫПОЛНЕНО (p(95)=1.64s)

**Сравнительный анализ результатов**

**Пропускная способность (Throughput):**

* **10 VUs**: 8.81 запросов/секунду
* **50 VUs**: 35.72 запросов/секунду
* **Смешанная нагрузка**: 8.99 запросов/секунду

**Время ответа (Response Time):**

| Тест | Среднее | Медиана | p95 | Максимум |
| --- | --- | --- | --- | --- |
| 10 VUs | 10.13ms | 1.05ms | 1.87ms | 1.22s |
| 50 VUs | 593.36ms | 469.31ms | 1.64s | 2.44s |
| Смешанный | 270.19ms | 133.74ms | 804.9ms | 7.1s |

**Надежность (Reliability):**

* **Процент успешных запросов**: 98.8%-100%
* **Процент успешных проверок**: 99.34%-100%

**Ручное тестирование:**

**1. Проверка пропускной способности (Throughput Testing)**

**Цель:** Определить максимальное количество запросов в секунду  
**Результаты:** Система показывает линейный рост пропускной способности от 8.81 RPS при 10 VUs до 35.72 RPS при 50 VUs  
**Вывод:** Пропускная способность масштабируется с увеличением нагрузки

**2. Проверка устойчивости (Endurance Testing)**

**Цель:** Проверить стабильность при длительной нагрузке  
**Результаты:** При 40-секундном тесте с 10 VUs система сохраняет стабильное время ответа (медиана 1.05ms)  
**Вывод:** Система демонстрирует хорошую устойчивость при умеренной нагрузке

**3. Тестирование пиковой нагрузки (Spike Testing)**

**Цель:** Проверить поведение при резком увеличении нагрузки  
**Результаты:** При резком росте до 50 VUs время ответа увеличивается в 58 раз (с 10.13ms до 593.36ms)  
**Вывод:** Система справляется с пиковой нагрузкой, но с значительной деградацией производительности

**4. Стресс-тестирование (Stress Testing)**

**Цель:** Определить пределы системы  
**Результаты:** При 50 VUs 95-й процентиль времени ответа достигает 1.64 секунды  
**Вывод:** Система достигает предела производительности при 50 одновременных пользователях

**5. Тестирование надежности (Reliability Testing)**

**Цель:** Проверить стабильность работы при стандартной нагрузке  
**Результаты:** 98.8%-100% успешных запросов во всех тестах  
**Вывод:** Система демонстрирует высокую надежность при различных сценариях нагрузки

**Выявленные проблемы**

**Критические проблемы:**

1. **Превышение времени ответа** - все тесты показали превышение пороговых значений p95
2. **Значительная деградация производительности** при увеличении нагрузки с 10 до 50 VUs

**Проблемы средней важности:**

1. **Неравномерное распределение времени ответа** - большой разброс между медианой и максимальными значениями
2. **Наличие выбросов** - максимальное время ответа достигает 7.1 секунды

**Положительные аспекты:**

1. **Высокая надежность** - минимальное количество ошибок (0-1.2%)
2. **Хорошая масштабируемость пропускной способности** - рост в 4 раза при увеличении нагрузки в 5 раз

**Выводы**

**Достижения:**

1. Успешно проведено нагрузочное тестирование с различными сценариями нагрузки
2. Получены объективные метрики производительности API
3. Выявлены реальные ограничения системы
4. Определены точки деградации производительности

**Рекомендации и варианты оптимизации:**

1. **Оптимизировать "тяжелые" эндпоинты** - эндпоинт /heavy значительно влияет на общую производительность
2. **Внедрить кэширование** для уменьшения времени ответа повторяющихся запросов
3. **Рассмотреть асинхронную обработку** для длительных операций
4. **Настроить мониторинг** 95-го и 99-го процентилей времени ответа

**Общий вывод:**

Тестовое API демонстрирует хорошую надежность, но требует оптимизации для обеспечения стабильного времени ответа под нагрузкой. Нагрузочное тестирование с k6 позволило объективно оценить производительность системы и выявить направления для улучшения.