Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

СОФТВЕРСКИ КВАЛИТЕТ И ТЕСТИРАЊЕ

проект на тема

ТЕСТИРАЊЕ НА LARAVEL API АПЛИКАЦИЈА

Изработил: Бојан Ристов (211151)

Линк до GitHub repository: https://github.com/ristov663/Project_SKIT

13.02.2025, Скопје

Вовед

Тестирањето на софтверот претставува критичен аспект од развојниот процес, кој обезбедува сигурност, стабилност и коректност на апликацијата пред нејзиното пуштање во продукција.

Во рамките на овој проект беше развиена Laravel API апликација за резервации која потоа беше тестирана со различни техники за тестирање, вклучувајќи unit и integration тестирање со Pest, API тестирање со Postman, performance тестирање со K6 и Artillery, како и континуирано автоматизирано тестирање со GitHub Actions (техники за тестирање коишто не беа обработени во предметот Софтверски квалитет и тестирање).

Оваа документација има за цел да ги опише методите и алатките кои беа користени при тестирањето на апликацијата, како и нивната улога во осигурувањето на квалитетот на софтверот. Во истата се објаснети чекорите на тестирање, имплементацијата на различните техники и стекнатите сознанија низ процесот.

Цел на проектот

Целта на овој проект е да се обезбеди високо квалитетен и стабилен софтверски систем преку имплементација на различни техники за тестирање. Тестирањето игра клучна улога во развојниот процес, бидејќи овозможува откривање и корекција на грешки, подобрување на перформансите и зголемување на доверливоста на апликацијата. Овој проект е насочен кон применување на модерни пристапи за тестирање, вклучувајќи unit, integration, API и performance тестови, како и автоматизација на процесот на тестирање со CI/CD методологии. Со ова, се обезбедува конзистентност и стабилност на апликацијата во различни сценарија и услови на користење.

Очекувани резултати

Со примена на овие техники на тестирање, очекувани резултати од проектот се:

- Откривање и корекција на грешки во раните фази на развојот, што ќе придонесе за подобрување на стабилноста на апликацијата.
- Зголемување на сигурноста и безбедноста на софтверот преку темелна валидација на функционалностите.

- ❖ Оптимизација на перформансите и идентификација на потенцијални тесни грла со помош на load и stress тестирање.
- ❖ Автоматизирано тестирање кое ќе овозможи побрза испорака на нови функционалности со минимални дефекти.
- ❖ Подобрување на одржливоста на кодот со воведување на структурирани тест сценарија и континуирано тестирање.

Краток опис на апликацијата

Апликацијата којашто беше тестирана е Laravel API апликација која обезбедува функционалност за управување со резервации и настани. Станува збор за едноставна Laravel API апликација којашто беше развиена со активностите на предметот Имплементација на системи со слободен и отворен код.

Апликацијата е составена од два ентитети, настани и резервации. Системот овозможува корисниците да прегледуваат информации за настани и да прават резервации за одреден настан.

Во истата се имплементирани основните CRUD (Create, Read, Update, Delete) операции, овозможувајќи комуникација преку REST API ендпоинти. Додадени се и дополнителни валидациски правила за двата ентитети, како и полнење на базата на податоци со рандом податоци со уптереба на Facotries и Seeders. Апликацијата користи SQLite база на податоци. Развиените функционалности беа предмет на тестирање со различни алатки, за да се осигура нивната точност, стабилност и перформанси.

Соодветните ендпоинти за апликацијата се следниве:

```
GET|HEAD        api/events ..........
POST            api/events ..........
GET|HEAD        api/events/{event} ..........
PUT|PATCH       api/events/{event} ..........
DELETE          api/events/{event} ..........
POST            api/reservations ..........
PUT             api/reservations/{reservation}/cancel ..........
PUT             api/reservations/{reservation}/confirm ..........
```

Техники за тестирање на апликацијата

Во рамките на проектот беа применети неколку техники за тестирање за да се обезбеди квалитетот и стабилноста на софтверот. Користените техники вклучуваат:

- **Unit и integration тестирање** – Со помош на Pest, беа тестирани индивидуални компоненти, како и интеграцијата помеѓу различни модули на апликацијата за да се осигура нивната точност и коректно функционирање.
- **API тестирање** – Тестирање на REST API ендпоинтите со Postman и Newman CLI за да се потврди точноста на одговорите и валидноста на обработените податоци.
- **Performance тестирање** – Користење на K6 и Artillery за да се анализираат перформансите на апликацијата под различни оптоварувања.
- **Автоматизација на тестирањето** – Имплементација на GitHub Actions за континуирано тестирање и автоматизирана валидација на кодот при секоја промена.

Pest тестирање

Pest е PHP тестирачки Framework базиран на PHPUnit, кој овозможува едноставен и експресивен начин на пишување на unit и integration тестови. Тестирањето со Pest беше фокусирано на:

- Тестирање на индивидуални компоненти од апликацијата (unit тестирање)
- Тестирање на поврзаноста меѓу различни делови од системот (integration тестирање)
- Валидација на однесувањето на API ендпоинтите
- Проверка на враќаните HTTP статуси и JSON структурата на одговорите

Во продолжение ќе бидат прикажани неколку Pest тестови коишто се дел од Pest тестирањето на оваа апликација.

```php
// Тестира дали постои табелата "events" и дали ги содржи очекуваните полиња
it('verifies events table structure', function () {

    expect(Schema::hasTable('events'))->toBeTrue();

    $expectedColumns = [
        'id', 'name', 'slug', 'description', 'location',
        'event_date', 'capacity', 'created_at', 'updated_at',
    ];

    foreach ($expectedColumns as $column) {
        expect(Schema::hasColumn('events', $column))->toBeTrue();
    }
});
```

```php
// Тестира преглед на конкретен настан со неговите резервации
it('shows a single event with its reservations', function () {
    $event = Event::factory()->create([
        'name' => 'Seniors Meetup',
        'location' => 'Skopje, Macedonia',
        'event_date' => now()->addDays(30)->toDateString(),
        'capacity' => 500,
    ]);

    $reservations = Reservation::factory()->count(2)->create([
        'event_id' => $event->id,
        'user_name' => 'John Doe',
        'ticket_quantity' => 5,
    ]);

    $response = $this->getJson("/api/events/{$event->id}");

    $response->assertStatus(200);

    $response->assertJson([
        'data' => [
            'name' => 'Seniors Meetup',
            'location' => 'Skopje, Macedonia',
            'capacity' => 500,
            'reservations' => $reservations->map(function ($reservation) {
                return [
                    'id' => $reservation->id,
                    'user_name' => $reservation->user_name,
                    'ticket_quantity' => $reservation->ticket_quantity,
                ];
            })->toArray(),
        ],
    ]);
});
```

```php
// Тестира успешно креирање на нов настан
it('allows creating a valid event', function () {

    $eventData = [
        'name' => 'Seniors Meetup',
        'description' => 'Seniors meetup for engineers.',
        'location' => 'Skopje, Macedonia',
        'event_date' => now()->addDays(30)->toDateString(),
        'capacity' => 50,
    ];
    $response = $this->postJson('/api/events', $eventData);
    $response->assertStatus(201);
    $this->assertDatabaseHas('events', $eventData);
});
```

```php
// Тестира дали API враќа грешка при креирање на настан без задолжителните
полиња (capacity and slug)
it('rejects event creation without required fields', function () {

    $response = $this->postJson('/api/events', []);

    $response->assertStatus(422)->assertJsonValidationErrors([
        'name', 'description', 'location', 'event_date'
    ]);
});
```

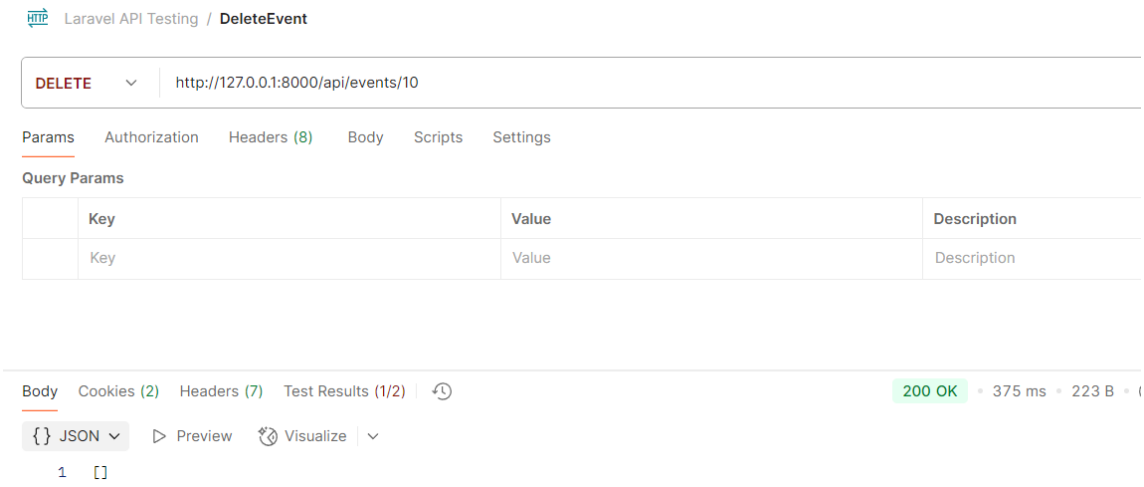
```php
// Тестира неуспешно додавање на резервација ако настанот не постои
it('fails to create a reservation if event_id does not exist', function () {

    $this->assertDatabaseMissing('events', ['id' => 9999]);

    $data = [
        'event_id' => 9999,
        'user_name' => 'John Doe',
        'ticket_quantity' => 5,
    ];

    $response = $this->postJson('/api/reservations', $data);

    $response->assertStatus(422)
        ->assertJsonValidationErrors(['event_id']);

    $this->assertDatabaseMissing('reservations', [
        'user_name' => 'John Doe',
        'ticket_quantity' => 5,
    ]);
});
```

Тестирање со Postman

Postman е популарна алатка за тестирање на API апликации, која овозможува дефинирање на барања, испраќање на параметри и анализирање на одговорите. Тестирањето со Postman вклучуваше:

- Тестирање на REST API ендпоинтите со различни типови на HTTP барања (GET, POST, PUT, DELETE)
- Автоматизирање на тест сценарија преку Postman тест скрипти
- Проверка на response body, headers и статус кодови

Во продолжение ќе бидат прикажани неколку тестови извршени во Postman.



Laravel API Testing / **GetAllEvents**

GET http://127.0.0.1:8000/api/events

Params Authorization Headers (8) Body Scripts Settings

Query Params

	Key	Value	Description
	Key	Value	Description

Body Cookies (2) Headers (7) Test Results (1/2) 200 OK 456 ms 2.88 KB

JSON Preview Visualize


```
{
    "data": [
        {
            "name": "Katharina Crooks",
            "slug": "katharina-crooks",
            "location": "Krisville",
            "event_date": "2011-12-09",
            "capacity": 148
        },
        {
            "name": "Delilah Dietrich",
            "slug": "delilah-dietrich",
            "location": "Kemmerbury",
            "event_date": "1970-05-05",
            "capacity": 198
        },
```


Laravel API Testing / **GetEvent**

GET http://127.0.0.1:8000/api/events/8

Params Authorization Headers (8) Body Scripts Settings

Query Params

	Key	Value	Description
	Key	Value	Description

Body Cookies (2) Headers (7) Test Results (1/2) 200 OK 372 ms 370 B

JSON Preview Visualize


```
{
    "data": {
        "name": "Dr. Judge Feeney I",
        "slug": "dr-judge-feeney-i",
        "event_date": "2012-10-08",
        "capacity": 241,
        "location": "East Dortha",
        "reservations": []
    }
}
```

Тестирање со K6 и Artillery

K6

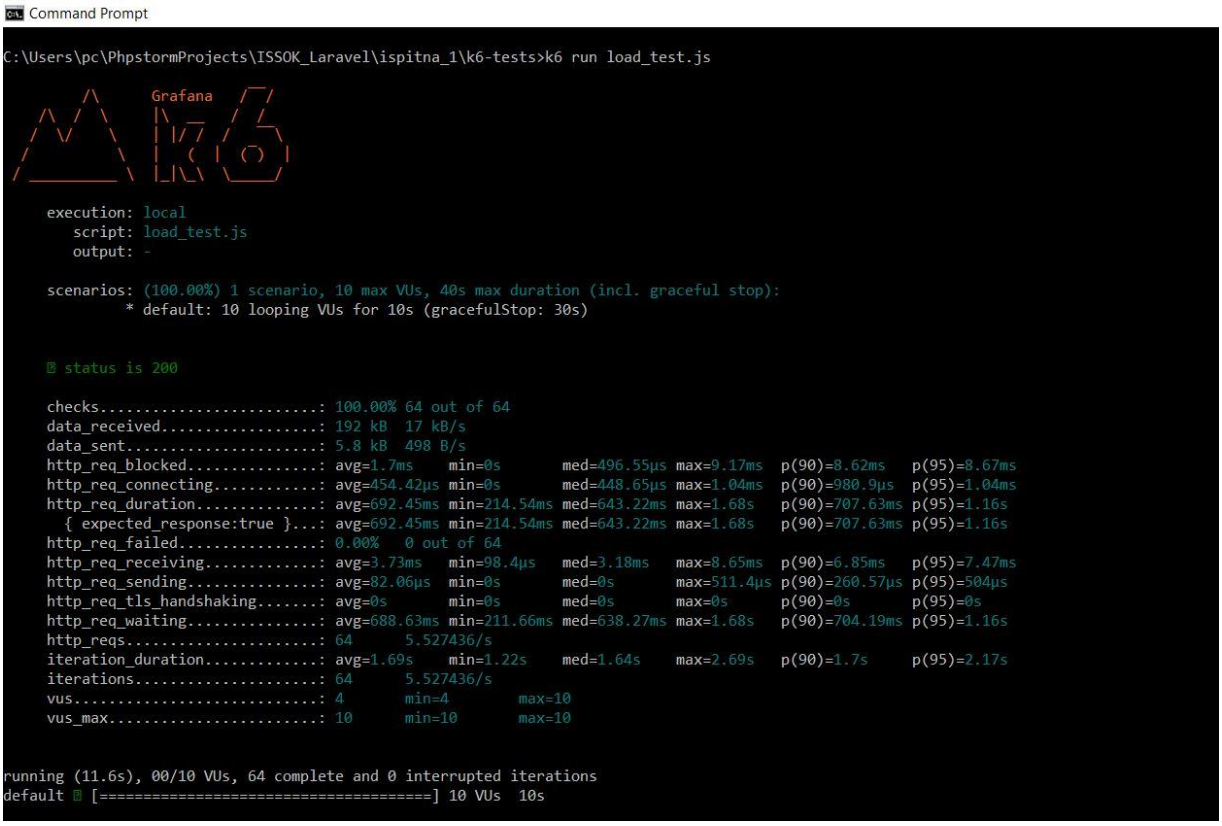
K6 е алатка за load тестирање која овозможува симулирање на повеќе корисници кои истовремено користат апликацијата. Тестовите со K6 вклучуваат:

- **Smoke тест** – Брз тест за основна проверка
- **Load тест** – Тестира како API-то се однесува под нормално оптоварување
- **Stress тест** – Проверува колку корисници може да поднесе API-то
- **Spike тест** – Тестира како API-то реагира на ненадејни нагли оптоварувања
- **Soak тест** – Долготрајно оптоварување за да открие мемориски протекувања

Во продолжение ќе бидат прикажани неколку од K6 тестовите, како и резултатите од нивните извршувања.

Load_test.js:

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
    vus: 10, // број на корисници (Virtual Users)
    duration: '10s', // времетраење на тестот
};

export default function () {
    let res = http.get('http://localhost:8000/api/events');
    check(res, {
        'status is 200': (r) => r.status === 200,
    });
    sleep(1); // пауза од 1 секунда помеѓу барањата
}
```



Load Test - Успешен

Сите барања се успешни (status 200)

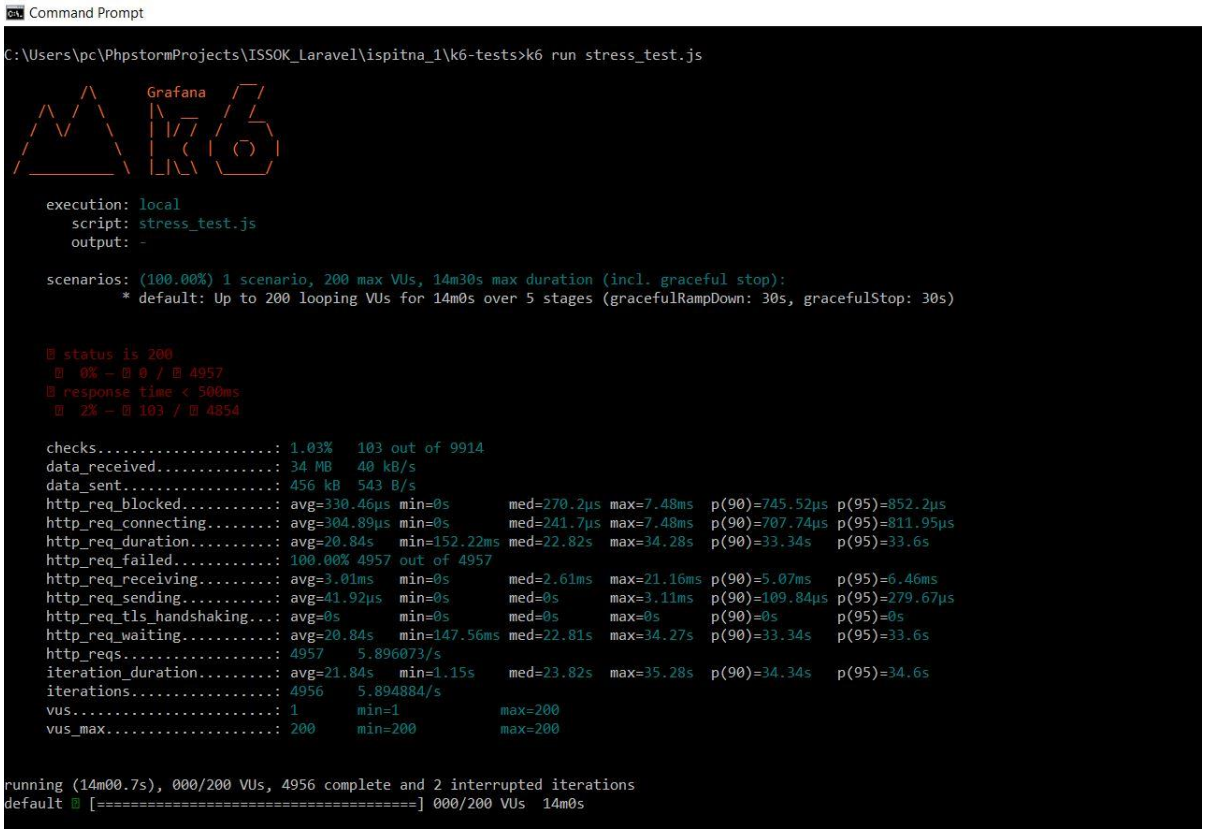
- 64 HTTP барања, сите добиле успешен одговор 200 OK
- Просечно време на одговор: 692.45ms, што е малку високо, но стабилно
- Ниту едно барање не пропаднало (http_req_failed: 0.00%)
- Максимално време на одговор: 1.68s, што значи дека некои барања биле побавни, но генерално API-то функционира стабилно
- Проточност: 5.52 requests per second (RPS), што значи дека серверот успешно обработува околу 5-6 барања во секунда

Заклучок

Тестот покажува дека API-то работи стабилно при помало оптоварување (до 10 истовремени корисници). Може да се подобри со оптимизација на базата или кеширање.

Stress_test.js:

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
    stages: [
        { duration: '2m', target: 50 },  // Постепено зголемување на 50 VUs
за 2 минути
        { duration: '3m', target: 100 }, // Одржување на 100 VUs за 3 минути
        { duration: '2m', target: 200 }, // Зголемување на 200 VUs за 2
минути
        { duration: '5m', target: 200 }, // Одржување на 200 VUs за 5 минути
        { duration: '2m', target: 0 },   // Намалување на оптоварувањето
    ],
};

export default function () {
    let res = http.get('http://localhost:8000/api/events');
    check(res, {
        'status is 200': (r) => r.status === 200,
        'response time < 500ms': (r) => r.timings.duration < 500,
    });
    sleep(1);
}
```



Stress Test - Неуспешен

Сите барања не успеале (http_req_failed: 100%)

- 4,957 барања испратени – сите пропаднале
- Просечно време на одговор: 20.84s, што е критично бавно
- Само 1.03% од барањата имале време на одговор под 500ms
- Серверот очигледно не може да се справи со оптоварување од 200 виртуелни корисници.
- Најдолгото време на одговор е 34.28s, што е огромно.

Artillery

Artillery е уште една load тестирачка алатка, која нуди можност за дефинирање на сценарија за тестирање. Тестирањето со Artillery вклучуваше:

- **Load Test** – Проверува како API-то се однесува под нормално оптоварување.
- **Stress Test** – Испитува колку барања може да издржи API-то пред да почне да дава грешки.
- **Spike Test** – Испраќа ненадеен, краткотраен скок на барања за да видиме дали системот брзо се опоравува.
- **Soak Test** – Испраќа барања подолг временски период за да тестира стабилност.Овие тестови помогнаа да се откријат потенцијални тесни грла во системот и да се направат потребните оптимизации за подобрување на перформансите.

Во продолжение ќе бидат прикажани неколку од K6 тестовите, како и резултатите од нивните извршувања.

Load_test.yml:

```
config:
  target: "http://localhost:8000/api/events"
  phases:
    - duration: 60  # Тестот трае 60 секунди
      arrivalRate: 10  # 10 корисници во секунда
scenarios:
  - flow:
      - get:
          url: "/api/posts"
```

Stress_test.yml:

```yaml
config:
  target: "http://localhost:8000/api/events"
  phases:
    - duration: 60
      arrivalRate: 20  # Почнува со 20 корисници во секунда
    - duration: 60
      arrivalRate: 50  # Потоа расте на 50 корисници во секунда
    - duration: 60
      arrivalRate: 100  # На крај, 100 корисници во секунда
scenarios:
  - flow:
      - get:
          url: "/api/posts"
```

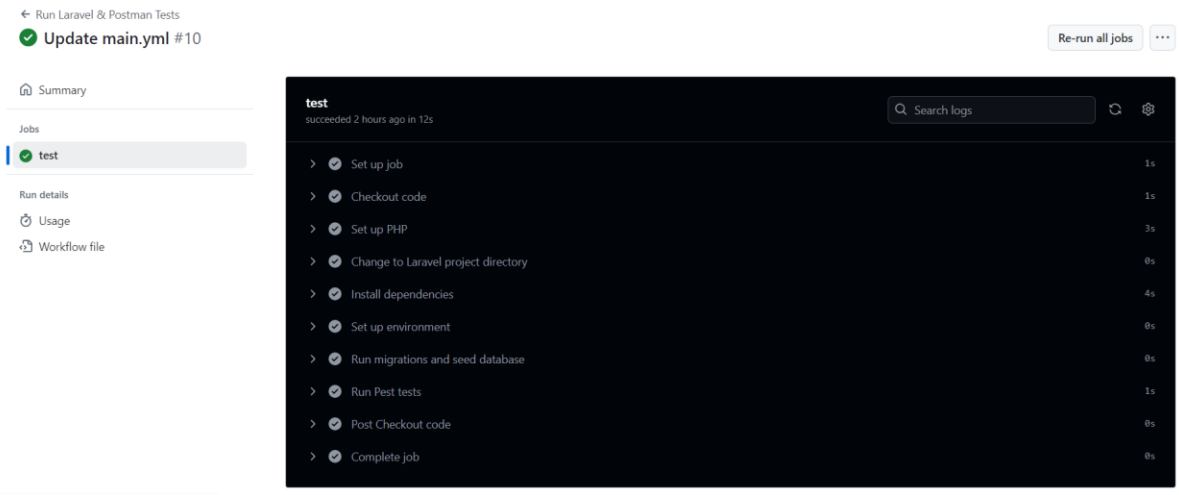
Карактеристика	k6	Artillery
Јазик	JavaScript (скриптирање во `.js` датотеки)	YAML и JavaScript
Сценарии	Подетални и програмски контролирани	Полесни за читање и пишување (YAML)
Резултати	Многу детални во CLI	Поедноставни, но поддржуваат JSON и HTML извештаи
Скалабилност	Оптимизиран за големи тестови	Исто така добар, но повеќе за API тестирање
Едноставност	Посложен за нови корисници	Полесен за почетници

Автоматизација на тестирањето

За делумна автоматизација на тестирањете искористив **GitHub Actions**, со кои се овозможува автоматско извршување на тестови при секој push или pull request на апликацијата на GitHub. Во GitHub Actions workflow-от беа интегрирани следниве чекори:

- Инсталација на зависности со Composer
- Поставување на .env фајлот и генерирање на application key
- Извршување на миграции и полнење на базата со тест податоци
- Извршување на Pest тестовите

Ова автоматизирање овозможува континуирана проверка на квалитетот на кодот и рано откривање на грешки.



Ci-pest-tests.yml:

```yaml
name: Run Laravel Pest Tests

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: 8.2
          extensions: sqlite, pdo_sqlite
          tools: composer

      - name: Install dependencies
        run: |
          composer install --prefer-dist --no-progress

      - name: Set up environment
        run: |
          cp .env.example .env
          php artisan key:generate

      - name: Run migrations and seed database
        run: |
          php artisan migrate --seed --env=testing

      - name: Run Pest tests
        run: |
          vendor/bin/pest tests/Feature --colors=always
```

Заклучок

Тестирањето на софтверот е од суштинско значење за осигурување на неговата стабилност, безбедност и перформанси. Во овој проект беше искористен структурен пристап кон тестирањето со комбинација од unit, integration, API и load тестови.

Со Pest беа валидирани основните функционалности на апликацијата, со Postman беше обезбедена коректноста на API ендпоинтите, а со K6 и Artillery беа тестирани перформансите под различни услови на оптоварување. Дополнително, со GitHub Actions се овозможи автоматизирано континуирано тестирање, со што се подобри квалитетот на кодот и се намали ризикот од неочекувани проблеми во продукција.

Овој систематски пристап покажа дека комбинацијата од различни алатки и техники овозможува комплетно покривање на различните аспекти на тестирањето, со што се зголемува доверливоста на софтверското решение. Оваа методологија може да се искористи и во идни проекти за постигнување на висок степен на квалитет во развојот на софтверски апликации.

Користена литература

1. Материјали од курсот Софтверски квалитет и тестирање
2. https://pestphp.com
3. https://www.postman.com
4. https://www.artillery.io
5. https://grafana.com/docs/k6/latest/
6. https://github.com/features/actions
7. https://github.com