Тестирање на Spring Boot + Angular апликација со примена на Selenium, Cucumber, Mockito, SpingBootTests, JMeter, RestTemlates, Input Space Partitioning и Postman

Софтверски квалитет и тестирање

Дарко Сасански 201065, Бојан Трпески 201091

Линк до GitHub репозиториумот: https://github.com/Bokica08/Swisshousing_Testing

**Апликацијата што се тестираше се наоѓа во директориумот: Swisshousing_Testing/tree/main/Homework3&4/backend

Mock tests (Junit co Mockito)

Mock тестовите, како комбинација на Junit тестови со Mockito framework, се јунит тестови што користат mock објекти за да симулираат одредени однесувања на реални објекти. Наместо да се повикува вистинската имплементација на сервисот, овие тестови симулираат одредени случаеви за да го тестираат кодот. Конкретно ние тестиравме 3 сервиси каде што симулиравме одредени однесувања на неговите методи и потврдуваме дали е запазено вистинското однесување на кодот. За HotelService, прво го дефинираме mock објектот, во нашиот случај

```
@Mock
private HotelRepository hotelRepository;
```

Кое што го инџектираме во

```
@InjectMocks
private HotelServiceImpl hotelService;
```

Дколку земеме еден од тестовите на пример тестот за зачувување на хотел

```
@Test
public void testPost() {
    Double x = 1.0;
    Double y = 1.0;
    String name = "Test Hotel";

    when(hotelRepository.save(any())).thenReturn(new Hotel());

    hotelService.post(x, y, name, "city", "street", "1",
"website.com", "12345", 5);

    verify(hotelRepository).save(any());
}
```

Најпрво дефинираме одредени атрибути што ги има хотелот, ги мокираме повиците што се случуваат во методот што го повикуваме и на крај го повикуваме методот post од hotelService. На крајот проверуваме дали навистина се извршил hotelRepository.save.

Spirng boot tests

Спирнг бут тестовите во комбинација со антоацијата @DataJpaTest овозможува поедноставено тестирање на JPA компонентите. Тој има основни вградени функцијоналности. Конкретно ние го користиме за да видиме дали ги враќа посакуваните вредности за конкретни функции, а со тоа се осигуруваме дали функцијата работи правилно.

```
@Autowired
private ReviewRepository reviewRepository;

@Autowired
private UserRepository userRepository;

@Autowired
private LocationRepository locationRepository;

private Review testReview;
private User testUser;
private Location testLocation;

@BeforeEach
public void setUp() {
    testUser = new User("testUser", "Test", "User",
"password123", Role.ROLE_USER);
    testUser = userRepository.save(testUser);

    testLocation = new Location(1.0, 1.0, "Test Location",
"Test City", "Test Street", "1A");
    testLocation = locationRepository.save(testLocation);

    testReview = new Review("test",testUser,testLocation,3);
    testReview = reviewRepository.save(testReview);
}
```

Конкретно ова е setup за тестовите на ReviewRepository, каде што со Autowire ги инџектираме сите потребни зависности што ќе ги користиме, а во методот setUp кој што е анотирам со BeforeEach даваме тест вредности за корисник, локација и review што потоа ги користиме во тестовите. За овој setup го имаме тестот:

```
@Test
public void findReviewsByLocation() {
```

```
    List<Review> reviews =
reviewRepository.findReviewsByLocation_LocationId(testLocation
.getLocationId());
    assertNotNull(reviews);
    assertEquals(1, reviews.size());
    assertEquals(testReview.getReviewId(),
reviews.get(0).getReviewId());
}
```

Кој што ги наоѓа reviews по локација. Таков метод постои во reviewRepository, па го повикуваме со тест податоците од погоре. Потоа го спроведуваме тестирањето, односно проверуваме дека reviews не е null, дека има точно еден review во reviews и дека ид-то на testReview е всушност ид-то што го има review-то во reviews, односно дека се работи за истиот review што треба да е случај.

Integration tests и RestTemplates

Интеграциските тестови обезбедуваат проверка на различни делови или компоненти од кодот. Тие се покомплексни од јунит тестовите бидејќи проверуваат како две или повеќе компоненти работат меѓусебно, а не само функционалноста на една изолирана компонента. Со помош на RestTemplate ние можеме да тестираме API, односно контролер од API-то и да ги тестираме неговите повици, а со тоа да се осигураме дека се испраќаат и добиваат вистинските вредности во response-от односно request-от. RestTemplate ни овозможува да правиме HTTP повици како што се GET, POST, DELETE….

```
@Autowired
private TestRestTemplate restTemplate;

@LocalServerPort
private int port;

private String getRootUrl() {
    return "http://localhost:" + port + "/alpinehut";
}
```

Во нашитот пример дефинираме зависност до TestRestTemplate преку која ќе ги правиме повиците и дефинираме порта на која ќе слуша апликацијата. Соодветно во getRootUrl го поставуваме url-то на кое што треба да испраќаме requests и добиваме response.

```
@Test
public void testGetAlpineHutByContains() {
    String name = "Gros";
    ResponseEntity<List> response =
restTemplate.getForEntity(getRootUrl() + "/cname?name=" +
name, List.class);
    assertNotNull("Null", response.getBody());
}
```

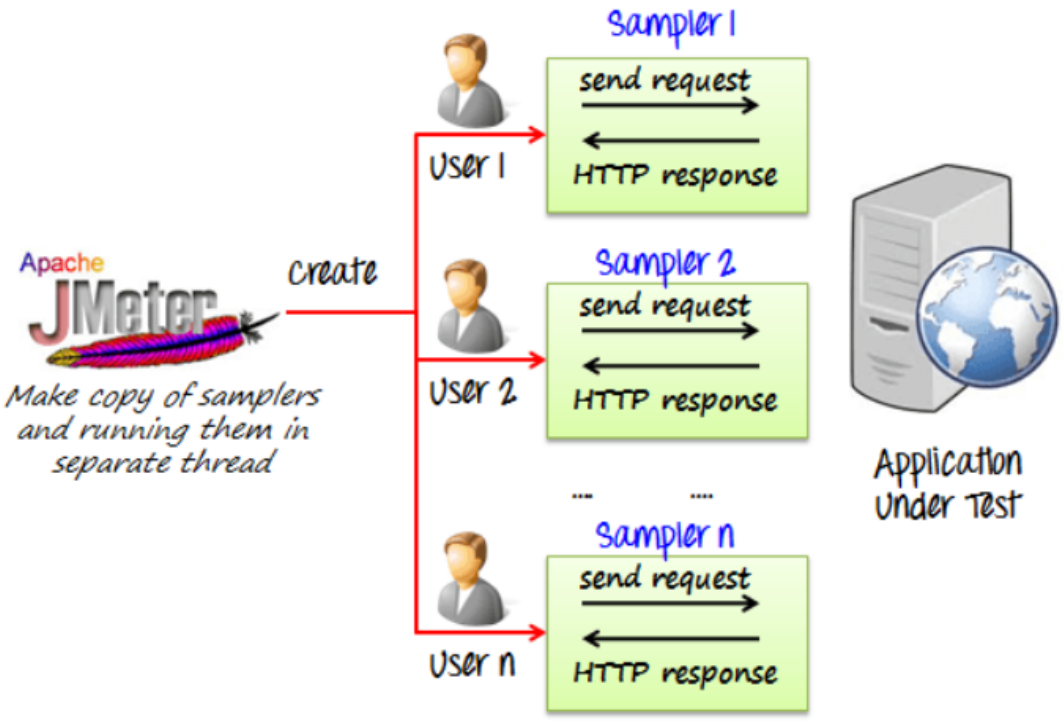
Во овој тест го тестираме повикот каде што ги земаме сите објекти кои што во своето име го содржат стрингот “Gros”. Го дефинираме тоа а потоа правиме повик кон API-от кон методот што ни ги враќа сите локации кои содржат Gros во името. Соодветно ние имаме во база неколку такви локации, па со последниот повик assertNotNull се осигураме дека body-то на response објектот не е Null, односно се вратени вредности. Со ова се осигураме дека комуникацијата на контролерот со сервисот, потоа сервисот со репозиторито и на крај репозиторито со базата функционираат меѓусебно.

JMeter Performance Testing

JMeter Performance Testing е метода за тестирање која со помош на Apache JMeter врши тестирање на перформансите на веб апликацијата. Оваа метода помага за тестирање на статичните и динамичките ресурси, откривање на максималниот број на корисници кои истовремено можат да пристапат на веб апликацијата и дополнително овозможува генерирање на графички анализи за тестирањата.

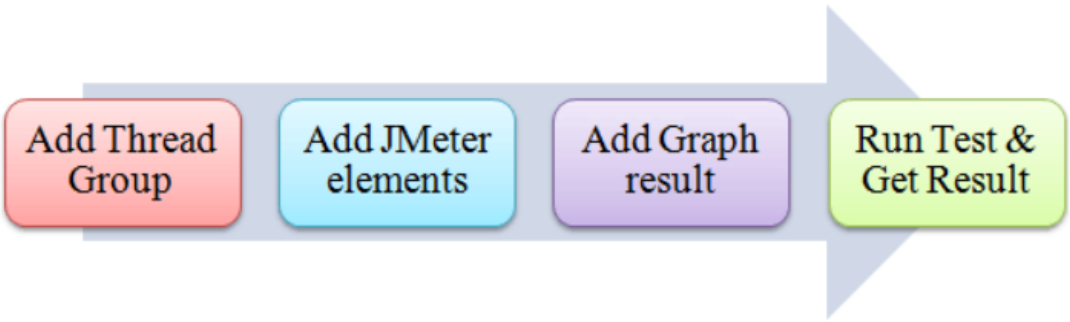
Оваа метода вклучува load и stress тестирање на веб апликацијата.

На следната илустрација е прикажан процесот на тестирање каде што се симулираат истовремени барања испратени кон апликацијата од различни корисници истовремено со цел да се откријат слабостите во перформансите.



Ние направивме тестирање на два endpoints од нашата backend апликација и тоа на endpointot /alpinehut каде што корисниците добиваат листа од сите достапни Alpine Huts и на endpoint /hotel/cname?name= кој што овозможува пребарување на хотелите по нивното име.

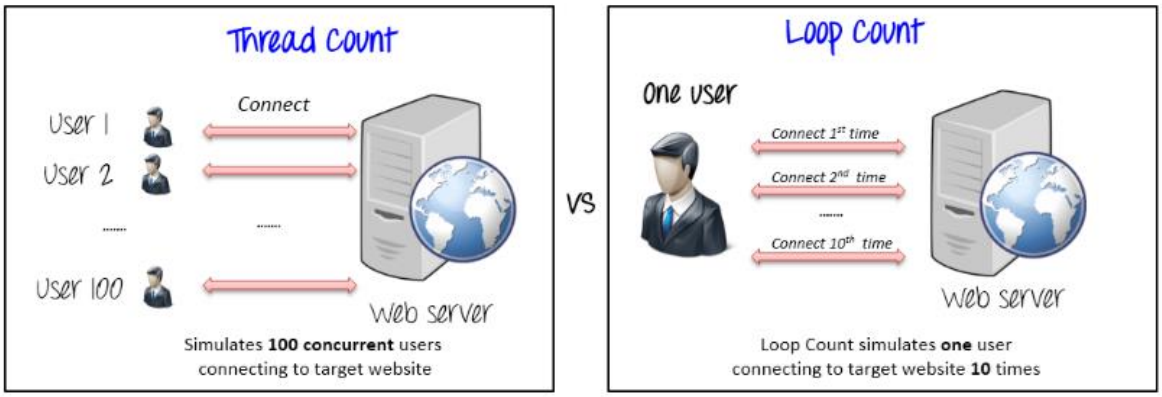
Во продолжение подетално ќе го објасниме тестирањето на вториот endpoint.



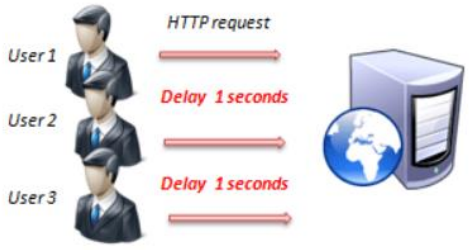
Како прв чекор што треба да се преземе со цел тестирање на наведениот endpoint е да се креира План за тестирање во кој што подоцна ќе ги додаваме останатите компоненти од тестот.

По креирањето на Планот за тестирање креираме Thread Group за тој план каде што наведуваме број на нишки(корисници), број на повторување и вкупното време за чекање помеѓу нишките.

Бројот на нишки го означува бројот на корисници кои истовремено ќе ги пребаруваат хотелите, додека бројот на повторување означува колку пати секој од корисниците ќе го повтори request-от.



Вкупното време за чекање пак означува колку вкупно време ќе се чека помеѓу барањата на различните корисници. На пример ако имаме 100 корисници и 100 секунди вкупно време за чекање, ќе се чека по 1 секунда пред да почне следниот корисник.



Следно ги дефинираме деталите за HTTP барањата со тоа што наведуваме име на серверот(во случајов имаме локален сервер), број на порта и патека во случајов /hotel/cname?name=${SearchTerm}.

Додаваме и csv датотека од каде што секој од корисниците ќе го чита query параметарот.

На крај додаваме и компоненти за анализа на резултатите како на пример Results Three(детална листа од сите барања) и Graph Results(за графичка анализа на перформансите)

Кај графичкиот приказ на тестот имаме репрезентација на неколку параметри прикажани со различна боја.

Црно: Вкупниот број на тековни испратени примероци.

Сино: Тековниот просек на сите испратени примероци.

Црвено: Тековната стандардна девијација.

Зелено: Throughput rate која го претставува бројот на барања во минута со кои се справува серверот.

За да ги анализираме перформансите на веб-серверот што се тестира, треба да се фокусираме на 2 параметри:

- Пропусната моќ
- Девијација

Пропусната моќ е најважниот параметар. Ја претставува способноста на серверот да се справи со тежок товар. Колку е поголема пропусната моќ, толку подобри се перформансите на серверот.

Девијацијата пак индицира на отстапување од просекот и колку е помало тоа отстапување толку е подобро.

Cucumber with Selenium

Cucumber рамката во Selenium е open source рамка за тестирање која подржува Behavior Driven Development за автоматско тестирање на веб апликации. Тестовите првин се пишуваат во форма на едноставно сценарио кое го опишува очекуваното однесување на системот од перспектива на корисникот.

BDD (Behavior Driven Development) пристапот овозможува на developers, QAs и останатите нетехнички тимови да соработуваат активно.

Сценаријата се пишуваат со користење на Domain Specific Language како што е на пример Gherkin. Секое сценарио се состои од кратки фрази напишани најчесто на англиски кои имаат за цел да го опишат однесувањето на апликацијата и исходот од нејзините акции. Cucumber, како софтверска алатка пак овозможува автоматизација на извршувањето на овие сценарија.

Ние со помош на Cucumber ги истестиравме функционалностите поврзани со најава на корисници и креирање, пребарување и едитирање на достапните хотели. Во продолжение ќе објасниме подетално околу тестот за додавање на нови хотели во системот.

Cucumber BDD се состои од три главни делови: Feature File, Step Definitions и Test Runner File.

Feature датотеката во себе содржи објаснување, сценаријата и податоците за една функционалност како што е во случајов додавање на нов хотел. Овие функционалности поедноставно може да се наречат Feature. Секој од овие Features потребно е да бидат истестирани со помош на Selenium интегриран со Cucumber.

Cucumber тестовите се напишани во овие датотеки кои ја имаат екстензијата .feature.

```
Feature: Hotel Management

  Background:
    Given I am logged in as "JDoe" with password "Password@123"
    And I am on the home page and redirected to add hotel page

  Scenario: Add a Hotel
    When I add a hotel with the following details:
      | X    | Y     | Name        | City      | Street       | House Number | Description        | Image Pat
      | 5    | 5     | HotelName   | CityName  | StreetName   | 123          | Hotel Description  | path/to/i
    Then I should be redirected to the hotels page
    When I select "Name" as the search attribute
    And I enter "HotelName" in the search field
    Then I should see "HotelName" in the search results
```

Во овој feature file е објаснета функционалноста за менаџмент за хотели, т.е. за додавање на нови хотели.

Првин го имаме делот Background кој што ги дефинира чекорите кои што мораат да преземат пред да се изврши сценариото односно ги дефинира чекорите за најава на корисникот со однапред дефинирани параметри.

Во делот за сценарио детално се објаснети чекорите за додавање на нов хотел, наведени се податоците за хотелот што треба да се додаде, а кои исто така треба да бидат внесени во формата на страницата за додавање на нов хотел.

Дополнително потоа се дефинирани и чекорите за пребарување на хотелите според нивното име. Како вредност на полето за пребарување се дава името на веќе внесениот нов хотел и во листата што се враќа како резултат се проверува дали е присутен новододаденият хотел.

Откако беше дефиниран Feature датотеката за додавање на нов хотел потребно беше и да се дефинира и кодот за истото сценарио.

За да може да се пресликаат исказите од Feature датотеката во Java код се користат Step Definitions. Step Definitions дополнително можат да користат и Java и Selenium команди за Java функциите кои се напишани со цел да се измапира Feature датотеката во код.

```
@When("I add a hotel with the following details:")
public void iAddAHotelWithTheFollowingDetails(DataTable dataTable) throws InterruptedException {

    List<Map<String, String>> hotelData = dataTable.asMaps(String.class, String.class);

    WebElement xInput = driver.findElement(By.name("x"));
    WebElement yInput = driver.findElement(By.name("y"));
    WebElement nameInput = driver.findElement(By.name("name"));
    WebElement cityInput = driver.findElement(By.name("city"));
    WebElement streetInput = driver.findElement(By.name("street"));
    WebElement houseNumberInput = driver.findElement(By.name("houseNumber"));
    WebElement descriptionInput = driver.findElement(By.name("description"));
    WebElement imagePathInput = driver.findElement(By.name("imagePath"));
    WebElement websiteInput = driver.findElement(By.name("website"));
    WebElement phoneNumberInput = driver.findElement(By.name("phone"));
    WebElement starsInput = driver.findElement(By.name("stars"));

    xInput.sendKeys(hotelData.get(0).get("X"));
    yInput.sendKeys(hotelData.get(0).get("Y"));
```

```
DarkoSasanski
@When("I select {string} as the search attribute")
public void iSelectAsTheSearchAttribute(String searchAttribute) {
    WebElement searchAttributeDropdown = driver.findElement(By.name("hotels"));

    Select select = new Select(searchAttributeDropdown);

    select.selectByVisibleText(searchAttribute);
}

DarkoSasanski
@And("I enter {string} in the search field")
public void iEnterInTheSearchField(String hotelName) {
    WebElement searchInput = driver.findElement(By.name("filter"));
    searchInput.sendKeys(hotelName);
}
```

На крај за да може да се извршат овие тестови потребно е да се дефинира Test Runner File, кој претставува JUnit Test Runner Class која што ги содржи потребните метаподатоци за извршување на тестовите, како на пример локацијата на Feature датокетите.

Оваа класа е анотирана со @RunWith анотацијата од Junit и со @CucumberOptions анотацијата која служи за дефинирање на локацијата на Step Definitions и Feature датотеките.

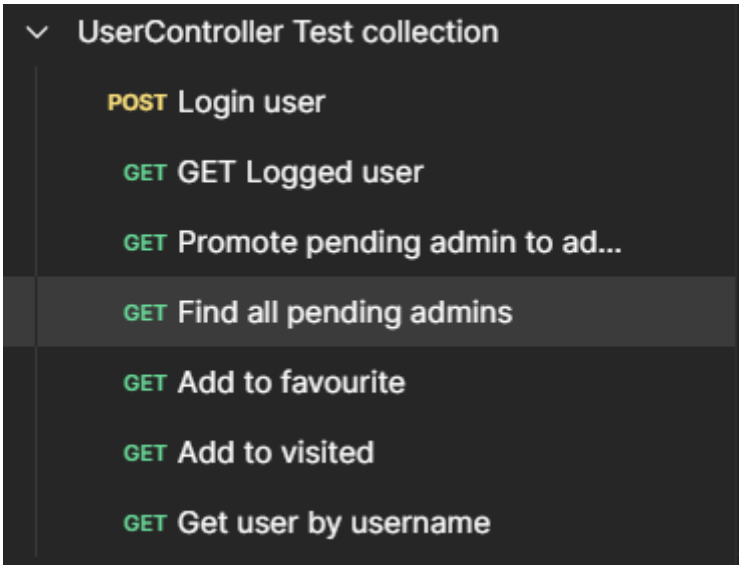
```
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/java/mk/ukim/finki/tech_prototype/cucumber/featureFiles", // Location of your feat
        glue = "mk.ukim.finki.tech_prototype.cucumber.stepDefinitions", // Package where your step definitions
        plugin = {"pretty", "html:target/cucumber-reports"} // Optional: Define output format and location
)
public class TestRunner {
}
```

Input Space Partitioning

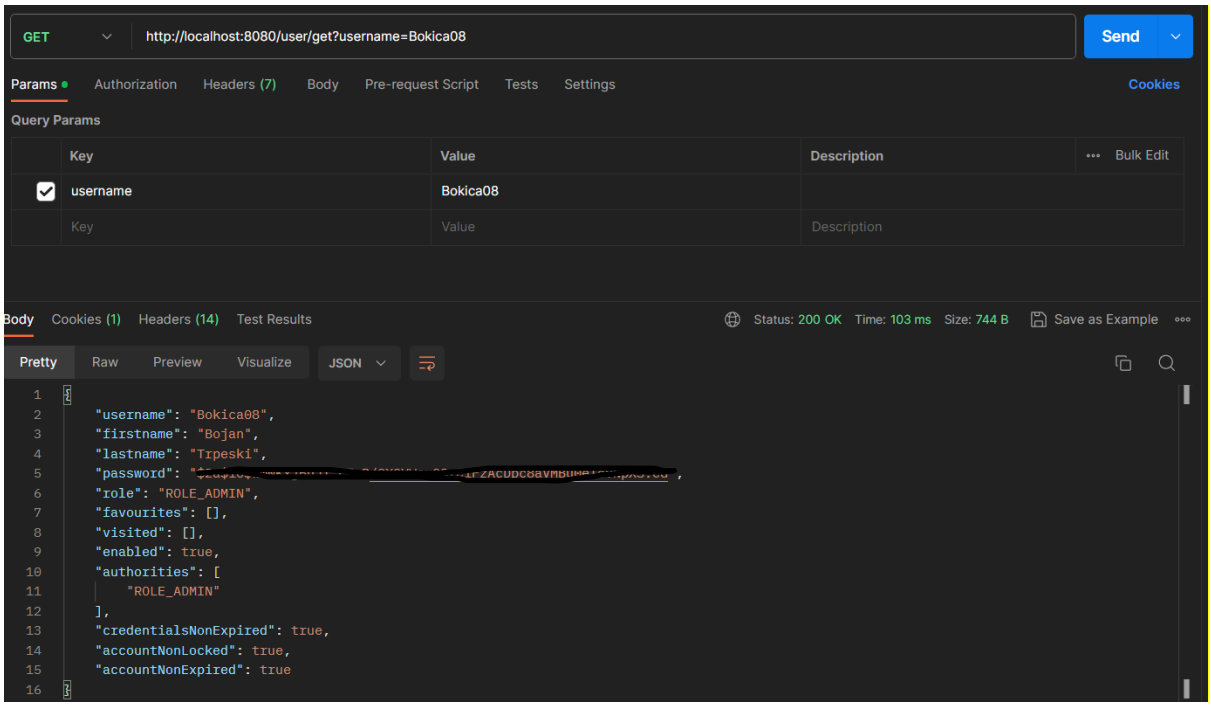
Во посебна датотека, која што исто така е дел од репозиториумот, напишавме детално објаснување за тестирањето на UserService сервисот користејќи ја техниката Input Space Partitioning. Дополнително напишавме и Junit тестови за тестните случаи кои произлегоа од Input Space Partitioning. Исто така тука користевме и Mockito, со цел тестирање на сервисот независно од репозиториумите.

Postman Collections

Алатката Postman се користи за тестирање и развој на API. Со нејзе можеме да тестираме разни видови на HTTP повици (GET, POST, PUT…) се со цел да се осигураме дека тие работат правилно и ги враќаат односно запишуваат посакуваните податоци. Една од можностите што ја нуди Postman е пишувањето на колекции. Тоа не е ништо друго туку група на повици кон одредено API, се со цел подобра организација, документација, споделување на повиците, како и зачувување на тест информациите што сме ги внеле за нив се со цел да се осигураме дека ги добиваме посакуваните одговори. Тие ни зачувуваат време, па при промени во API-то, ние зачуваниот повик со зачуваните информации можеме да го повикаме и да се увериме дека API-то работи како претходно, односно ни ги враќа точните информации.



Нашата колекција за UserController



Пример од GET повик да ни го врати корисникот со даден username (Bokica08) и body-то на response-от со информациите за тој корисник