Семинарска работа по предметот: Софтверски квалитет и тестирање

Тема: Функционално и перформансно тестирање на REST API

Милена Трајаноска 181004

Датум: 11.09.2021

Код за тестирање:

https://github.com/MilenaTrajanoska/rest_api

Тестови:

https://github.com/MilenaTrajanoska/books_rest_api_testing

Содржина:

1 Вовед во REST архитектурата

REST (Representational State Transfer) е клиент-сервер архитектурен стил кој поставува стандард на комуникација помеѓу различни сервиси на Веб. Некои од главните карактеристики на REST архитектурите се тоа што овозможуваат без-состојбеност и поделба на одговорностите помеѓу клиентот и серверот. Интеракцијата помеѓу REST системите се однесува на изведување на стандардни операции врз Веб ресурси. Ресурсите се главните компоненти на REST архитектурата и претставуваат било каков објект или документ кој што може да се складира или да се прати. Најчестите формати за размена на ресурсите се JSON и XML.

Со цел да се дефинира комуникацијата меѓу клиентот и серверот, се користи протокол кој што го разбираат и двете страни, стандардниот протокол е HTTP. Клиентот испраќа барање за читање, креирање или модифицирање на ресурси, а серверот враќа одговор на барањата на клиентот.

Едно барање од страна на клиентот потребно е да содржи: HTTP метод (GET, POST, PUT, DELETE, итн.), заглавје во кое клиентот може да наведе метаподатоци за самото барање и/или начин на автентикација со системот, патека која го идентификува ресурсот и опционално тело со податоци.

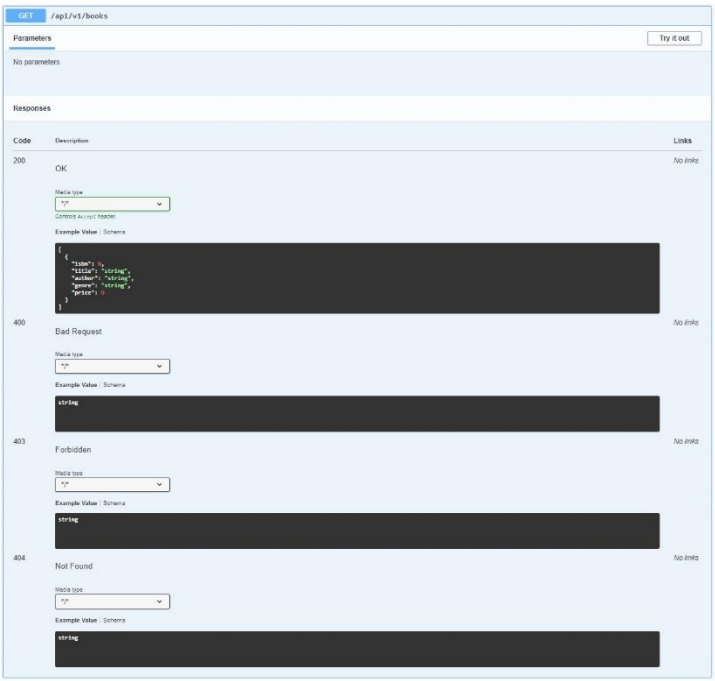
2 Опис на REST API кое се тестира

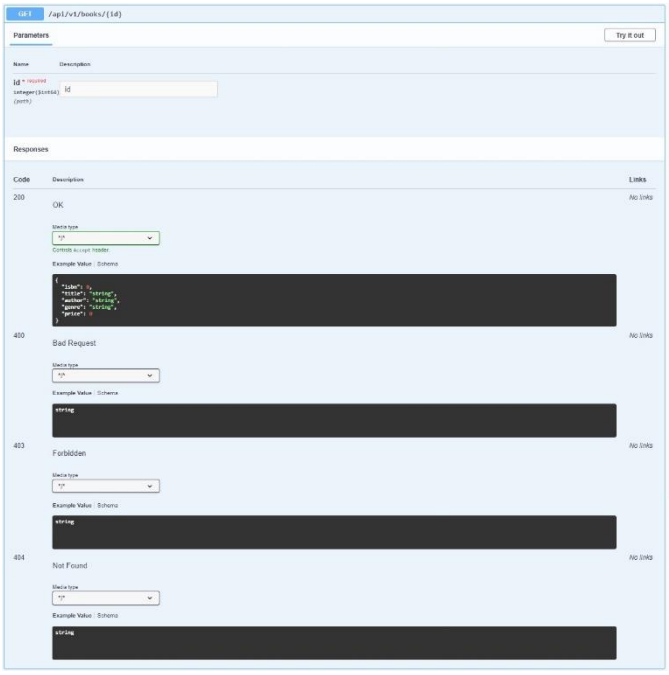
За целите на оваа семинарска работа ќе се тестира едноставно REST API кое овозможува изведување на операции врз книги. Книгите ги содржат следните атрибути: isbn – international standard book number, title – наслов на книгата, author – име на авторот на книгата, genre – жанрот на книгата и price – цена на книгата во долари. Во самата апликација се наведени ограничувања за вредностите на атрибутите: isbn – да не е null вредност, title – стринг кој не е null и има минимум 5 карактери а максимум 150, author – стринг кој не е null и има минимум 5 карактери а максимум 50, genre – стринг кој не е null и не е празен, price – double кој не е null.

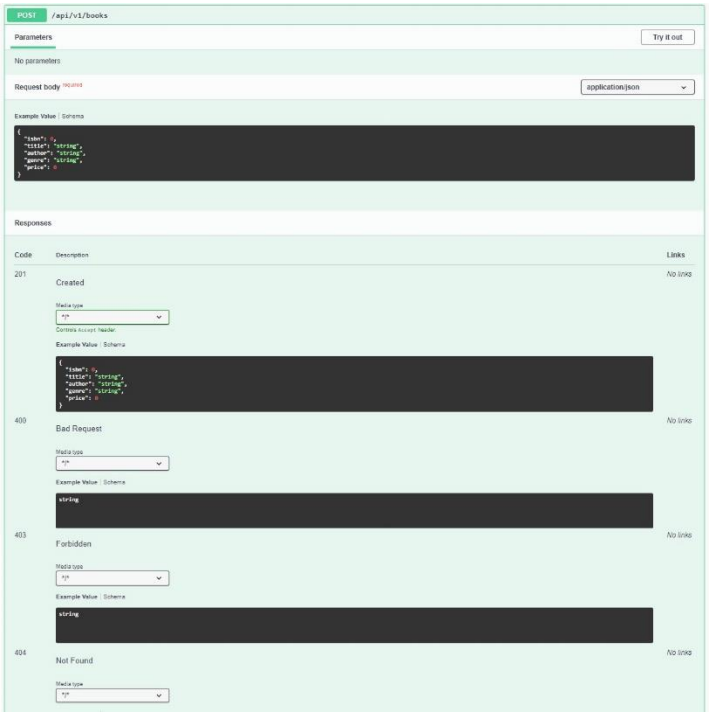
Изработено е едноставно REST API кое овозможува да се прегледаат сите книги во базата на податоци (вкупно 108 на број), да се прегледа една книга со задавање на нејзиниот isbn како идентификатор, креирање на нова книга, ажурирање на податоците на постоечка книга со задавање на нејзиниот isbn како идентификатор и бришење на книга според даден isbn. Податоците од API-то се враќаат во JSON формат. Секоја книга мора да има уникатен isbn, од ова ограничување произлегуваат и некои од тестовите наведени во следните глави.

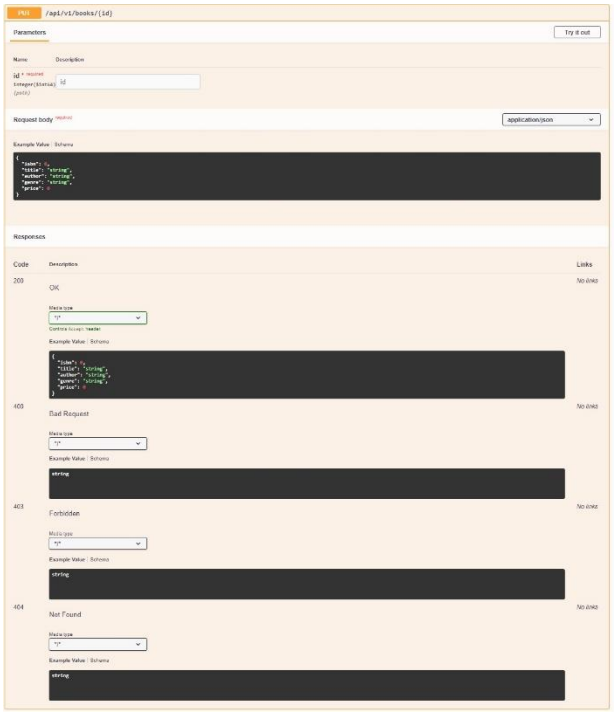
REST API-то овозможува униформен интерфејс за изведување на сите операции. Основното URI на API-то е /api/v1/books.

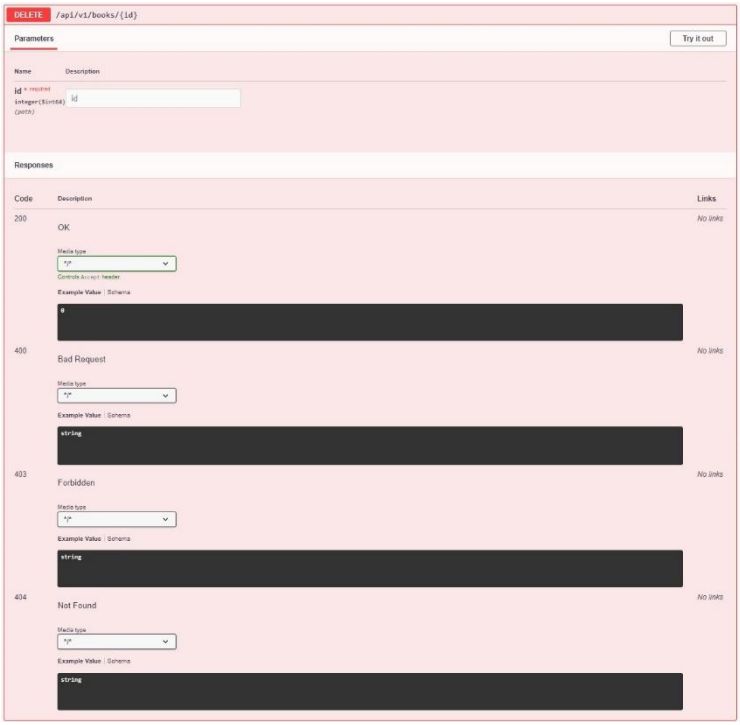
На следните слики се прикажани спецификациите на endpoint-ите кои ги нуди ова API, изгенерирани со помош на Swagger UI.











Достапните endpoints се прикажани и во Табела 1. Достапни endpoints и поддржани HTTP методи. Сите барања кои се со HTTP метод различен од GET потребно е да бидат авторизирани. Во конкретниот случај се користи основна авторизација (Authorization: basic) со корисничко име user и лозинка user123*. Сите GET барања се достапни незаштитени.

Статус кодовите кои ги враќаат endpoint-ите може да бидат: 200 Ok (успешно извршена операција) за GET, PUT и DELETE барањата, 201 Created за POST барањата доколку успешно се креира нова книга во базата, 400 Bad Request доколку се прати барање со погрешни параметри, било да е во телото на барањето, заглавјата и сл. Потоа 403 Forbidden доколку се изврши забранета опција, во конкретниот случај тоа е креирање на книга со isbn кој веќе постои во базата, при што се враќа порака "Book with isbn: {isbn} already exists". Следен статус код кој може да се врати е 404 Not Found, доколку се побара книга со isbn кој не постои во базата при GET, PUT и DELETE методите, при што пораката која се враќа е "Book with isbn: {isbn} could not be found". Доколку се испушти заглавјето за авторизација или се пратат невалидни параметри се враќа статус код 401 со порака "Unauthorized".

Во сите успешни случаи се враќа одговор во JSON формат кој содржи барана/креирана/модифицирана книга/книги, или isbn број на избришана книга.

Кодот за рест API-то е достапен на: https://github.com/MilenaTrajanoska/rest_api

За да може истиот успешно да се изврши, потребно е да биде поврзан со PostgreSQL база на податоци во која е креирана база со име book_db. За да ги добиете истите податоци кои ги користи системот можете да ја извршите скриптата за креирање на табелата book и внесување на податоците достапна на:

https://github.com/MilenaTrajanoska/rest_api/blob/main/books_rest_api/src/main/resources/scripts/books_data_insert.sql

Првиот дел од семинарската работа се фокусира на функционално тестирање на REST API-то, со цел да се одреди дали функционира правилно. За таа цел се користи рамката за тестирање на REST APIs во Java – RestAssured [2] заедно со JUnit 5 [3] за целосно автоматизирање на тестирањето.

Вториот дел од семинарската работа се однесува на тестирање на перформансите на горенаведеното REST API за книги. Поконкретно, со користење на алатката Apache JMeter [6] се изведува load, spike, endurance и stress тестирање на основниот endpoint кој ги враќа сите книги и endpoint-от за детали на специфична книга, бидејќи се очекува ова да е најчестата операција која ќе ја изведуваат голем број на корисници истовремено. Дополнително, со цел да се одреди робусноста на системот доколку се запишува големо количество на нови книги одеднаш, изведено е и load тестирање на endpoint-от за креирање на нова книга во базата на податоци.

3 Функционално тестирање со RestAssured и JUnit 5

3.1 Опис на можностите на RestAssured

RestAssured претставува Java библиотека која нуди можности за пишување на моќни и одржливи тестови за REST APIs. Овозможува лесно за користење fluent API со кое на разбирлив начин може да се специфицираат тестовите. Еден пример за испраќање на GET барање до одредено API:

```
given().
    when().
        get("http://localhost:8080/api/v1/books").
    then().
        assertThat().
        statusCode(200).
    and().
        contentType(ContentType.JSON)
```

Со овој пример се праќа GET барање до основното URI /api/v1/books и откако ќе се добие одговорот од серверот се проверува дали статус кодот на одговорот е 200 (OK) и дали заглавјето Content-Type има вредност application/json, односно дали се вратени податоци во JSON формат. Доколку горенаведените барања се исполнети, тестот ќе помине успешно.

Дополнително, за покомплексни барања, RestAssured овозможува да се креира спецификација на барањата со помош на класата RequestSpecification која дозволува да се поставуваат соодветните заглавја, тело на барањето доколку е потребно, и подоцна да се испрати барањето со специфицирање на соодветен HTTP метод.

За утврдување на вредностите кои се добиени како резултат од испратеното барање, освен методот assertThat() од fluent API-то на RestAssured, може да се искористи и друга рамка за тестирање. Во оваа семинарска работа, за автоматизација на процесот на тестирање и изведување на тестовите покрај RestAssured ќе се користи и рамката JUnit 5.

3.2 Дефинирање на функциски тест барања и критериуми за покривање

Начинот на дефинирање на тест барања за REST APIs се разликува од стандардниот начин за дефинирање на барања за unit testing. Поради огромниот број на различни комбинации кои можат да се специфицираат при испраќање на HTTP барања: различни заглавја, дополнителни параметри во барањето, податоци во телото на барањето во најразлични формати, менување на HTTP методи и слично, потребно е да се направи соодветна тест стратегија која ќе овозможи да се тестираат најважните комбинации за одредено API. Тест стратегијата која се користи во оваа семинарска работа се базира на [1].

Дефинирање на тест барања за endpoints

Табела 1. Достапни endpoints и поддржани HTTP методи

Endpoint	HTTP Method
/api/v1/books	GET, POST
/api/v1/books/{id}	GET, PUT, DELETE

Во Табела 1. Достапни endpoints и поддржани HTTP методи се претставени endpoints кои ги овозможува REST API-то како и HTTP методите кои ги поддржуваат истите.

Како минимални барања кои треба да бидат исполнети за тестирање на endpoints се следните: тестирање на секој endpoint за колекција, тестирање на барем еден endpoint за индивидуален ресурс од секој тип на ресурс. Како негативен тест, може да се тестира однесување на апликацијата при пристап до endpoint кој не постои.

Потребно е да се изведат тестови кои ќе ги покријат овие барања за да се смета критериумот за endpoints како задоволен.

Дефинирање на тест барања за HTTP методи

За секој од наведените endpoints во Табела 1. Достапни endpoints и поддржани HTTP методи се прикажани подржаните HTTP методи. Повторно, тестовите за овој тип на барања може

да се поделат на позитивни и негативни. Позитивните тестови опфаќаат тестирање на сите поддржани методи на секој endpoint за колекција, и на барем еден endpoint за индивидуален ресурс од секој тип на ресурс. Како негативни тестови може да се изведат тестирање на неподдржан метод на секој endpoint за колекција, и на барем еден endpoint за индивидуален ресурс од секој тип на ресурс.

Дефинирање на тест барања за заглавја на HTTP барањето

Заглавјата на едно HTTP барање содржат податоци кои ни даваат дополнителни инфромации за податоците во телото на барањето (content type, charset итн.), автентикација и авторизација со системот и слично.

Повторно може да се дефинираат позитивни и негативни тест барања кои произлегуваат од заглавјата. Позитивните барања се однесуваат на верифицирање на секој endpoint за колекција, и на барем еден endpoint за индивидуален ресурс од секој тип на ресурс со сите неопходни и валидни заглавја. Најпрво се верифицираат сите неопходни заглавја, па подоцна може да се додадат и опционалните (доколку постојат). Негативните барања се однесуваат на верификација на однесување на системот доколку недостасуваат неопходните заглавја (тестирање кога недостасуваат еден по еден), да се верифицира однесувањето доколку се даде погрешно дефинирано неопходно заглавје и да се верифицира однесувањето во случај на неподдржано заглавје.

Дефинирање на тест барања за податоци во телото на HTTP барањето (POST и PUT)

Позитивните барања се состојат од тестирање на endpoint-от со испраќање на тело на барањето со валидни информации. Негативните тестови се состојат од испраќање на барања кои во телото содржат полиња со невалидна вредност (дупликати на уникатни вредности, надвор од дефинирани граници), полиња со погрешен тип на податок, празни полиња или null, недостаток на полиња, додавање на редундантни полиња.

Дополнително, доколу се работи за DELETE барање, како негативен тест може да се земе бришење на непостоечки ресурс.

Тестирање на одговор од серверот

Со цел успешно да се изведе тестирањето, потребно е да извршиме валидација на одговорот на барањата од страна на серверот. Одговорите содржат статус код, заглавја и опционално тело. Потребно е да се осигураме дека вратениот статус код одговара на очекуваното сценарио, на пример доколку испратиме барање за креирање на книга на страна на серверот очекуваме одговор со статус код 201 Created, доколку пак побараме непостоечка книга во базата, би очекувале одговор од серверот со статус код 404 Not Found.

Покрај статус кодот, добро е да се верифицира дека заглавјата кои ги очекуваме во одговорот од серверот се присутни. На пример, доколку побараме постоечка книга, очекуваме дека серверот ќе врати одговор во кој ќе ја има книгата во JSON формат. За таа

цел би тестирале дали Content-Type заглавјето на одговорт од серверот ја има вредноста application/json.

Доколку серверот враќа ресурс како одговор на барањето на клиентот, потребно би било да верифицираме дека сите очекувани полиња се присутни во ресурсот и ги имаат соодветните очекувани вредности и податочни типови.

За мапирање на ресурсите во тест апликацијата се користат POJO објекти во кои се мапираат JSON објектите испратени од серверот. Соодветната POJO класа е BookDTO. За испраќање на барања со погрешен тип на полиња или полиња кои недостасуваат се користат Jackson објекти.

Дефинирање на множество на комбинации од барања

Бидејќи горенаведените комбинации од барања се во голем број, потребна покомпактна дефиниција на барањата. За таа цел, некои од барањата кои може да се задоволат истовремено се комбинирани во едно подетално барање. Компактното множество на барања кое произлезе од овој процес е дадено во Табела 2. Множество на тест барања.

Табела 2. Множество на тест барања

Идентификатор	Опис
FTR1	Да се испрати валидно GET барање до /api/v1/books со сите неопходни заглавја (барањето нема потреба од заглавја)
FTR2	Да се испрати валидно GET барање до /api/v1/books/{id} со сите неопходни заглавја (барањето нема потреба од заглавја)
FTR3	Да се испрати GET барање до /api/v1/books/{id} со сите неопходни заглавја за параметар id кој е невалиден (барањето нема потреба од заглавја)
FTR4	Да се испрати POST барање до /api/v1/books со валидни податоци за книга во кое недостасуваат неопходни заглавја, едно по едно
FTR5	Да се испрати валидно POST барање до /api/v1/books со валидни податоци за книга со сите неопходни заглавја
FTR6	Да се изпрати POST барање до /api/v1/books со невалидни податоци за книга, но со сите неопходни заглавја
FTR7	Да се испрати PUT барање до /api/v1/books/{id} со валидни податоци за книга во кое недостасуваат неопходни заглавја, едно по едно
FTR8	Да се испрати PUT барање до /api/v1/books/{id} со валидни податоци за книга со сите неопходни заглавја, за книга која постои во базата

FTR9	Да се испрати PUT барање до /api/v1/books/{id} со невалидни податоци за книга со сите неопходни заглавја, за книга која постои во базата
FTR10	Да се испрати PUT барање до /api/v1/books/{id} со валидни податоци за книга со сите неопходни заглавја, за невалиден параметар id
FTR11	Да се испрати DELETE барање до /api/v1/books/{id} за книга која постои во базата, но без заглавјето за авторизација
FTR12	Да се испрати DELETE барање до /api/v1/books/{id} за книга која постои во базата, со сите неопходни заглавја
FTR13	Да се испрати DELETE барање до /api/v1/books/{id} за невалиден параметар id, со сите неопходни заглавја
FTR14	Да се направи барање на /api/v1/books/ со неподдржан HTTP метод
FTR15	Да се направи барање на /api/v1/books/{id} со неподдржан HTTP метод
FTR16	Да се направи барање на /api/v1/books/ со неподдржано заглавје
FTR17	Да се направи барање на /api/v1/books/{id} со неподдржано заглавје
FTR18	Да се тестира резултатот од паралелно повикување на endpoints кои поддржуваат POST, PUT и DELETE методи
FTR19	Сите барања кон REST API-то не треба да се извршуваат подолго од 2 секунди

3.3 Дефинирање на тест случаи и тестови

Според горенаведените тест барања, следниот чекор е да се дефинираат соодветни тест случаи кои ќе ги задоволат барањата. Голем дел од барањата се задоволени со по еден тест случај, но за некои од нив (кои бараат испраќање на невалидни податоци) потребни се повеќе тест случаи за целосно да бидат задоволени, на пример испраќање на null или празни полиња, додавање редундантни атрибути, отстранување на неопходни атрибути, гранични вредности на некои од атрибутите и слично. Направени се вкупно 32 тестови кои ги задоволуваат горенаведените барања. Не сите тестови се успешно извршени, што ни сугерира проблем во некои аспекти од функционирањето на системот.

Тест случаите и извршените тестови се документирани во табелата која почнува на следната страна. Колоната TC го означува идентификаторот на тест случајот кој се извршува, колоната

TR го означува идентификаторот на тест барањето за кое се однесува тест случајот, во колоната Title е даден опис на тоа што треба да прави тестот. Овие колони се означени со црна боја на позадината.

Колоните со сина боја на позадината се однесуваат на спецификациите на HTTP барањето кое се испраќа до серверот во самиот тест. Request URI ја означува патеката до ресурсот кој го пристапуваме, HTTP Method означува кој HTTP метод се користи за извршување на барањето. Во Request Headers колоната потребно е да се наведат сите заглавја кои ги специфицираме на барањето кое се испраќа до серверот (освен заглавјето за автентикација кое има посебна колона), во Request Body колоната специфицираме кое е телото на барањето, доколку испраќаме податоци во самото барање, и на крајот колоната Authentication потребно е да го содржи типот на автентикација со системот, доколку истата е неопходна.

Со зелена боја се означени колоните кои ги претставуваат очекуваните карактеристики на одговорот испратен од серверот. Response статус колоната го означува статус кодот на одговорот, Response Body го означува телото на одговорот, односно податоците кои се пратени од серверот, и колоната Response Headers означува кои заглавја се присутни во одговорот од серверот. Сите овие колони се користат за изведување на тестовите. Наведените податоци во нив ги означуваат очекуваните вредности, не вистинските кои се добиени во одговорот.

Последната колона Pass/Fail означува дали барањето се извршило успешно или не, и доколку е неуспешно е наведена причината.

Имплементацијата на тестовите е дадена на линкот:
https://github.com/MilenaTrajanoska/books_rest_api_testing/blob/main/src/test/java/BooksRestApiFunctionalTests.java

TC	TR	Title	Request URI	HTTP Method	Request Headers	Request Body	Authentication	Response Status	Response Body	Response Headers	Pass / Fail
TC1	FTR1	Should return all of the books saved in the database	/api/v1/books	GET	/	/	/	200	List of 108 objects from BookDTO class	Content-Type: application /json	Pass no errors
TC2	FTR2	Should return the book with isbn equal to 8781234567894 from the database	/api/v1/books/ 8781234567894	GET	/	/	/	200	{"isbn": 8781234567894, "title": "The Dark Highlander", "author": "Karen Marie Moning", "genre": "Romance", "price": 6.99}	Content-Type: application /json	Pass no errors
TC3	FTR3	Should return 404 error when with error message for book with isbn 500 which is not in the database	/api/v1/books/ 500	GET	/	/	/	404	Book with isbn: 500 could not be found	Content-Type: text/plain; charset= UTF-8	Pass no errors
TC4	FTR3	Should return 400 bad request error when requesting book with null isbn	/api/v1/books/ null	GET	/	/	/	400	/	Content-Type: application /json	Pass no errors
TC5	FTR4	Should not create a book unauthorized	/api/v1/books	POST	Content-Type: application /json	{"isbn": 123, "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	/	401	Unauthorized	Content-Type: application /json	Pass no errors
TC6	FTR4	Should not create a book when missing content type header	/api/v1/books	POST	/	{"isbn": 123, "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username : password"	415	Content type 'text/plain; charset=ISO-8859-1' not supported	Content-Type: application /json	Pass no errors
TC7	FTR5	Should create a valid book in the database	/api/v1/books	POST	Content-Type: application /json	{"isbn": 7781234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username: password"	201	{"isbn": 7781234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	Content-Type: application /json	Pass no errors
TC8	FTR6	Should not create a book with a null isbn	/api/v1/books	POST	Content-Type: application /json	{"isbn": null , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username: password"	400	/	Content-Type: application /json	Pass no errors
TC9	FTR6	Should not create a book with an empty author name	/api/v1/books	POST	Content-Type: application /json	{"isbn": 7781234567891 , "title": "Hamlet", "author": "", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username: password"	400	/	Content-Type: application /json	Pass no errors
TC10	FTR6	Should not create a book with a title shorter than 20 letters	/api/v1/books	POST	Content-Type: application /json	{"isbn": 7781234567891 , "title": "Abc", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username: password"	400	/	Content-Type: application /json	Pass no errors
TC11	FTR6	Should not create a book with a negative value for price	/api/v1/books	POST	Content-Type: application /json	{"isbn": 2 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": -1.0}	{"Authorization": "username: password"	201	{"isbn": 2 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": -1.0}	Content-Type: application /json	Failed should not have created the book
TC12	FTR6	Should not create a book with an isbn that already exists	/api/v1/books	POST	Content-Type: application /json	{"isbn": 8781234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 13.45}	{"Authorization": "username: password"	403	Book with isbn: 8781234567891 already exists	Content-Type: text/plain; charset= UTF-8	Pass no errors
TC13	FTR6	Should not create a book with wrong data type for author name	/api/v1/books	POST	Content-Type: application /json	{"isbn": 55 , "title": "Bird Box", "author": 123455, "genre": "Horror", "price": 3.5}	{"Authorization": "username: password"	201	{"isbn": 55 , "title": "Bird Box", "author": 123455, "genre": "Horror", "price": 3.5}	Content-Type: application /json	Failed should not have returned 201, book is not created in the db

TC14	FTR6	Should not create a book when missing field for price	/api/v1/books	POST	Content-Type: application /json	{“isbn”: 55 , "title": "Title", "author": “Author”, "genre": "Horror"}	{“Authorization”: “username: password”	400	/	Content-Type: application /json	Pass no errors
TC15	FTR6	Should ignore redundant fields when creating book	/api/v1/books	POST	Content-Type: application /json	{“isbn”: 55 , "title": "Title", "author": “Author”, "genre": "Horror", “price”: 23.3, “random”: “Random field”}	{“Authorization”: “username: password”	201	{“isbn”: 55 , "title": "Title", "author": “Author”, "genre": "Horror", “price”: 23.3}	Content-Type: application /json	Pass no errors
TC16	FTR7	Should not update a book from an unauthorized user	/api/v1/books/18	PUT	Content-Type: application /json	{“isbn”: 18 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	/	401	Unauthorized	Content-Type: application /json	Pass no errors
TC17	FTR7	Should not update book with missing content type header	/api/v1/books/18	PUT	/	{“isbn”: 18 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	{“Authorization”: “username: password”	415	Content type 'text/plain; charset=ISO-8859-1' not supported	Content-Type: application /json	Pass no errors
TC18	FTR8	Should update existing book with valid information	/api/v1/books/9781234567108	PUT	Content-Type: application /json	{“isbn”: 9781234567108 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	{“Authorization”: “username: password”	200	{“isbn”: 7781234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	Content-Type: application /json	Pass no errors
TC19	FTR9	Should not update existing book with new information and null isbn	/api/v1/books/7781234567891	PUT	Content-Type: application /json	{“isbn”: null , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	{“Authorization”: “username: password”	400	/	Content-Type: application /json	Pass no errors
TC20	FTR10	Should not update or create new book when updating book with isbn that does not exist	/api/v1/books/6681234567891	PUT	Content-Type: application /json	{“isbn”: 6681234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	{“Authorization”: “username: password”	404	Book with isbn: 6681234567891 could not be found	Content-Type: text/plain; charset= UTF-8	Pass no errors
TC21	FTR10	Should return bad request error when trying to update book with null isbn in url	/api/v1/books/null	PUT	Content-Type: application /json	{“isbn”: 6681234567891 , "title": "Hamlet", "author": "William Shakespeare", "genre": "Tragedy", "price": 20.15 }	{“Authorization”: “username: password”	400	/	Content-Type: application /json	Pass no errors
TC22	FTR11	Should not delete a book unauthorized	/api/v1/books/7781234567891	DELETE	/	/	/	401	Unauthorized	Content-Type: application /json	Pass no errors
TC23	FTR12	Should delete existing book from database	/api/v1/books/7781234567891	DELETE	/	/	{“Authorization”: “username: password”	200	Book deleted successfully	Content-Type: application /json	Pass no errors
TC24	FTR13	Should not delete book that doesn’t exist	/api/v1/books/600	DELETE	/	/	{“Authorization”: “username: password”	404	Book with isbn: 600 could not be found	Content-Type: text/plain; charset= UTF-8	Pass no errors
TC25	FTR13	Should return bad request error when deleting book with null isbn	/api/v1/books/null	DELETE	/	/	{“Authorization”: “username: password”	400	/	Content-Type: application /json	Pass no errors
TC26	FTR14	Should return method not allowed error when accessing endpoint with unsupported method	/api/v1/books	DELETE	/	/	/	405	/	Content-Type: application /json	Pass no errors
TC27	FTR15	Should return method not allowed error when accessing endpoint with unsupported method	/api/v1/books/7781234567891	POST	/	/	/	405	/	Content-Type: application /json	Pass no errors

TC28	FTR16	Should not accept json response when accept header is text/html	/api/v1/books	GET	Accept: text/html	/	/	406	Not acceptable	Content-Type: text / html; charset= UTF-8	Pass no errors
TC29	FTR17	Should not accept json response when accept header is xml	/api/v1/books/8781234567891	GET	Accept: application/xml	/	/	406	Not acceptable	Content-Length: 0	Pass no errors
TC30	FTR18	Should create resource only once when executing POST request in parallel	/api/v1/books	POST	Content-Type: application /json	{“isbn”: 2 , "title": "Bird Box", "author": "Josh Malerman", "genre": "Horror", "price": 3.5 }	{“Authorization”: “username: password”	One request 201 - created, six requests – 403 frobidden	/	/	Failed with 4 Threads trying to save the same book int the database, 500 internals server error
TC31	FTR18	Should keep last resource update when executing in parallel a PUT request	/api/v1/books/8781234567892	PUT	Content-Type: application /json	Last update: {isbn: 8781234567892, title: 'The Martian', author: 'Andy Weir', genre: 'Sci-Fi', price: 12.2}	{“Authorization”: “username: password”	200 OK for each thread	{isbn: 8781234567892, title: 'The Martian', author: 'Andy Weir', genre: 'Sci-Fi', price: 12.2}	Content-Type: application /json	Pass no errors
TC32	FTR18	Should delete resource only once when executing in parallel	/api/v1/books/5	DELETE	/	/	{“Authorization”: “username: password”	200 OK once, 404 not found 6 times	/	/	Failed, got internal server error 500, two times instead of 404 not found

Забелешка: кога се креира ресурс со веќе постоечки isbn паралелно, истиот ресурс не се зачувува во базата бидејќи се бришат сите ресурси кои го нарушуваат ограничувањето за уникатен isbn број. Потребно е да се реши овој проблем. Апликацијата не е безбедна за конкурентно користење. Потребно е да се искористат механизми за справување со паралелното извршување. **Поради непредвидливиот начин на распоредување на нишките на страна на серверот, можно е при извршување некој од тестовите TC30 или TC32 да помине успешно, но сепак при повеќекратно извршување на истите тестови се забележува дека системот има проблем со справување со паралелно извршување.**

Дополнително, бидејќи истиот BookDTO модел се користи од страна на апликацијата и во POST и во PUT барањата, тестовите за PUT не прават детална проверка на сите атрибути бидејќи истите се проверени во POST тестовите. Направен е само еден тест за невалидност со цел да се провери случајот да биде заборавено вршењето на валидација во PUT методот. Повторувањето на останатите тестови би било редундантно.

3.4 Извештај од функциското тестирање

Главните проблеми кои беа откриени во системот се однесуваат на валидација на испратените податоци при креирање / ажурирање на ресурси, како и паралелното извршување на некои операции врз ресурсите (POST, PUT и DELETE).

Еден од тестовите кој не требаше да резултира со креирање на нова книга, беше тестот каде што за цена на книгата беше наведен негативен број (-1.0). Оттука, може да се заклучи дека системот нема соодветна валидација на ограничувачките вредности за цената на книгата.

Следниот тест кој беше неуспешен беше креирање на книга каде името на авторот е погрешен податочен тип, односно наместо низа од знаци беше испратен бројот 123455. Ова резултираше со успешно креирање на нова книга, што не требаше да се случи. За таа цел, покрај валидација на тоа дали е празно или null името на авторот (и другите атрибути) потреба е и соодветна валидација на регуларни изрази која нема да дозволи броеви во името на авторот. Истото важи и за насловот на книгата и за жанрот.

Последните два теста кои не се извршија успешно беа поврзани со паралелно извршување на операции врз книгите. Првата операција беше креирање на истата книга во базата истовремено од 7 нишки. Очекуваниот резултат беше книгата да се креира успешно првиот пат и потоа серверот да врати соодветна порака со статус код 403 Forbidden за секој наредeн обид на креирање на книга со дупликат isbn. Но случјаот беше што книгата беше креирана повеќе пати, па самото тоа резултираше со серверска грешка (статус код 500) бидејќи беше извршена невалидна опреација во самата база на податоци.

Вториот тест за паралелизам се однесуваше на ажурирање на истиот ресурс истовремено од 7 паралелни нишки. Во тој процес, се очекуваше при самото ажурирање на податоците да се зачува последната промена, односно да не бидат измешани различни податоци за различни книги. Овој тест помина успешно, што значи дека методот за ажурирање на книги работи добро и при конкурентно извршување.

Последниот тест кој што не беше успешен беше тестот за паралелно бришење на ист ресурс. Книгата со isbn 5 се брише истовремено од 7 нишки. Очекуваниот резултат беше едно успешно бришење со статус код 200 од страна на серверот, и 6 одговори со статус код 404 – книгата не е пронајден. Но, добиениот резултат беше покрај горенаведните статус кодови и статус кодот 500, односно серверска грешка, што значи дека системот се обидува да избрише книга која не постои.

Од горенаведените проблеми, потребно е да се посвети повеќе внимание при валидација на податоците и паралелното извршување на операции врз самите книги. Во Табела 3. Откриени проблеми и препораки се наведени сумарно откриените проблеми како и некои препораки за разрешување на истите.

Табела 3. Откриени проблеми и препораки

Тест случај	Проблем	Предлог
TC11	Успешно се креира книга со негативна вредност за цена	Да се постави валидација на рангот на вредностите за цената во моделот за книги

TC13	Успешно се креира книга со цел број наместо низа од знаци за име на автор/ наслов на книга/жанр	Да се постави валидација со регуларен израз за името на авторот, насловот на книгата, жанрот и isbn бројот кој ќе го ограничи фроматот на можни вредности за атрибутите
TC30	Повеќекратно креирање на книга во базата со ист isbn при паралелно извршување	Да се воведе заклучување при извршување на операциите: проверка дали ресурсот постои во база и неговото креирање.
TC32	Повеќекратно бришење на иста/непостоечка книга при паралелно извршивање	Да се воведе заклучување на операциите: проверка дали книгата постои во базата, бришење на книгата која постои во базата

4 Перформансно тестирање со Apache JMeter

4.1 Опис на Apache JMeter

Apache JMeter претставува алатка која е дизајнирана да овозможи тестирање на функционирањето на одредена апликација при оптоварување (load testing), и мерење на перформансите. Иницијално е создадена за тестирање на апликации и сервиси на Веб, но се користи и за други примени.

Apache JMeter може да се користи за тестирање на перформансите и на статички и динамички ресурси, се користи за симулирање на тешко оптоварување на Веб серверот, на група од сервери или на мрежата за да се анализираат перформансите на системот под различни оптоварувања.

Алатката овозможува тестирање на оптоварување и перформанси на различни апликации, сервери и со различни протоколи вклучувајќи: Web (HTTP/HTTPS), SOAP / REST, FTP, JDBC, LDAP, MOMs, SMTP, POP3, IMAP, TCP, команди на шел скрипти, Java објекти итн.

Дополнително, овозможува и multi-threading односно истовремено семплирање од многу нишни, и истовремено извршување на повеќе групи на нишки (thread groups).

Дефинирање на поими кои се користат

За да се креира тест во Apache JMeter потребна ни е спецификација на тестот која во случајот се нарекува тест план (**Test Plan**). Еден Test Plan може да има повеќе различни елементи. Наједноставните елементи на тест планот се групи на нишки и семплери. Самиот тест план поддржува опција за зачувување на одговорот од серверот во датотека, доколку е вклучена опцијата Functional Testing, но истата ги влошува перформансите и нема да се користи при изведување на перформансните тестови во оваа семинарска работа.

Thread group – групите на нишки се основните елементи на секој тест план. Сите семплери и контролери мора да се наоѓаат под некоја група на нишки. Овој елемент го контролира бројот на нишки кои се користат за извршување на самиот тест. Покрај бројот на нишки може да се постави и **ramp up / ramp down** период и број на повторувања за извршување на тестот. Секоја нишка која се креира е независна од другите, се креираат повеќе нишки истовремено за да симулираат конкурентни корисници.

Ramp up / ramp down периодот му кажува на JMeter за колку време од почетокот на тестот треба да го распореди наведениот број на нишки на рамномерен начин, односно за колку време да ги запре на крајот од тестот, соодветно.

Дополнително, за нишките може да се конфигурира и **duration** – колку време се извршува секоја група на нишки **startup delay** – времето потребно да се почека пред да се стартува групата.

Controllers – во JMeter потојат два типа на контролери: **Samplers** и **Logical Controllers** кои го водат извршувањето на тестот. Семплерите му кажуваат на JMeter како да го испрати барањето до серверот (пр. HTTP Request Sampler за HTTP барање). Логичките контролери овозможуваат да се направи одлука кога да се прати барањето до серверот.

Listeners - овозможуваат пристап до информациите кои ги собира JMeter за време на извршување на тестовите. Овие елементи даваат различни погледи на податоците кои може да се зачуваат во датотеки на диск за анализа подоцна. Пример за listeners се: **View Results Tree listener** кој овозможува да се видат деталите од барањата и одговорите од серверот и може да прикаже XML или HTML приказ на резултатите, **View Results in Table** прикажува табуларен приказ на податоците за барањата и одговорите како латентност, време на конекција, статус код, број на испратени / примени бајти итн.

Configuration elements – овие елементи работат заедно со семплерите и можат да ги модифицираат. Пример за ваков елемент е **CSV Data Set Config** со кој што може да се читаат податоци од .csv датотека и истите да се користат при испраќање на барања до серверот во телото на барањето, во самата патека и слично.

Покрај горенаведените елементи постојат и многу други. Важни се и предпроцесорите и постпроцесорите, кои се елементи кои вршат некакви акции пред и после испраќање на барањата до серверот, соодветно. Дополнително, JMeter овозможува дефинирање на

функции и променливи, како и користење на веќе готови функции на пример Random, counter и слично.

Повеќе детали околу сите елементи и нивните можности може да се најдат во официјалната документација на Apache JMeter [7].

4.2 Дефинирање на тест барања

При дефинирање на барањата за тестирање на перформанси потребно е да се направи разграничување на некои од поимите кои ќе се користат. Конкретно, во оваа семинарска работа ќе се изведат тестови за анализирање на оптоварувањето на системот (load testing), стрес тестирање (stress testing), тестирање на скокови (spike testing), и тестирање на издржливоста на ситемот (endurance testing).

Тестирање на оптоварувањето (load testing)

Тестирањето на оптовареноста на системот претставува нефункциски тест во кој што перформансите на системот се тестираат со специфицирано очекувано оптоварување, односно се тестира како функционира системот при очекуван број на корисници кои го употребуваат истовремено.

Целта на ваквото тестирање е да се одредат одредени тесни грла во дизајнот и да се осигура стабилноста на системот пред неговото испорачување.

Со ваков тип на тестирање најчесто ги идентификуваме: максималниот капацитет со кој што апликацијата успешно се справува, дали моменталната инфраструктура е соодветна за извршување на апликацијата, дали е одржливо функционирањето на апликацијата во однос на максималното оптоварување и бројот на истовремени корисници што апликацијата може да ги поддржи.

Тестирањето на оптоварувањето може да ги идентификува проблемите со следните аспекти:

- Време на одговор за секоја трансакција
- Перформанси на системските компоненти при различно оптоварување
- Перформанси на базата на податоци при различни оптоварувања
- Мрежното доцнење меѓу клиентот и серверот
- Дизјнот на софтверот
- Конфигурацијата на серверот
- Ограничувања на хардверот (CPU, RAM ...)

Стрес тестирање (stress testing)

Стрес тестирањето претставува нефункциско тестирање на системот при кое се обидуваме на серверот да му дадеме екстремно оптоварување и да одредиме при која точка на оптоварување престанува да функционира правилно.

Во некои ситуации, како на пример промоции и намалени цени на книги, би очекувале многу поголем број од вообичаените корисници истовремено да пристапуваат кон системот. За таа цел потребно ни е да изведеме стрес тестирање и да видиме до која точка може системот да се справи со корисниците. Доколку системот не ги задоволува нашите потреби тогаш би било неопходно да се зголеми скалабилноста на системот за да може да го задоволи огромниот број на корисници.

Тестирање на скокови (spike testing)

Тестирањето со скокови е тип на перформансно тестирање на системот кое се крористи при тестирање на софтвер со екстремно зголемување и намалување на оптоварувањето во повеќе циклуси.

Целта на ваквиот тип на тестирање е да се одреди однесувањето на софтверот при динамично зголемување и намалување на оптоварувањето како и времето потребно системот да се опорави по ваквите екстремни осцилации. Тестирањето се прави за да се откријат слабите страни на софтверот.

Метриките кои се потребни да се анализираат при ваков тип на тестирање се бројот на испади на системот, времето на одговор од серверот до клиентот, бројот на истовремени корисници и слично.

Тестирање на издржливост (endurance testing)

Тестирањето на издржливоста се спроведува со цел да се одреди дали системот може да се справи со очекуваното процесирачко оптоварување во подолг временски период. При ваквиот тип на тестирање важно ни е да ја мониториираме потрошувачката на меморија од страна на системот со цел да забележиме дали ќе наидеме на memory leak.

Дополнително, потребно е да се провери дали после подолг временски период од големо оптоварување, деградирњето на перформансите на системот е во прифатливи рамки.

Дефинирање на множество на нефункциски барања

При дефинирање на нефункциските барања, покрај техничкиот аспект, потребно е да се земе предвид и бизнис аспектот на самиот систем. Во конкретниот случај, API-то би се користело за манипулација на книги од база на податоци. Се очекува апликацијата да ја користат голем број на корисници дневно кои ќе ги разгледуваат самите книги.

Дополнително како важна операција во конкретното сценарио се смета и запишувањето на големо количество на податоци одеднаш, односно при пристигнување на нови книги, би сакале истите да ги зачуваме во самата база на податоци.

Најважни за корисниците се операциите со преглед на сите книги и на поединечни книги, додека пак за администраторите на системот најважна би била операцијата внесување на нови книги во големо количество одеднаш.

За тие цели, важно ни е да обезбедиме теситрање кое ќе потврди дека системот добро се справува со овие операции при очекуван број на корисници, но и дека брзо се опоравува при големи скокови во бројот на активни корисници како и долги временски периоди на користење.

Од горе-наведеното образложение произлегува множеството на нефукциски барања кое е наведено во Табела 4. Множество на нефункциски барања.

Табела 4. Множество на нефункциски барања

Идентификатор	Опис	Endpoint	Метод	Очекувани метрики	Тип на тест
NFTR1	Да сет тестира очекувано оптоварување од 100 истовремени корисници. При што се очекува дека постепено ќе се зголемува бројот на корисници додека не достигне 100 и потоа останува ова оптоварување подолго време.	http://localhost:8080/api/v1/books	GET	Elapsed time per request <= 2000 ms Latency <= 1500ms Connect time <= 200ms Fail rate < 5%	Load testing
NFTR2	Да се тестира со екстремно оптоварување од 500 корисници истовремено, каде постепено се зголемува бројот на корисници и потоа останува одреден период. Потребно е да се процени	http://localhost:8080/api/v1/books	GET	Elapsed time per request <= 10000 ms Latency <= 6500 ms Connect time <= 200ms Fail rate < 10%	Stress testing

	издржливоста на системот во таков случај.				
NFTR3	Да се тестира динамично зголемување и намалување на бројот на корисници во краток временски период со цел да се одреди робусноста на системот и времето на опоравување од екстремни скокови.	http://localhost:8080/api/v1/books	GET	Elapsed time per request <= 1500 ms Latency <= 1000 ms Connect time <= 500ms Fail rate < 5%	Spike testing
NFTR4	Да се тестира истрајноста на за долги временски периоди на извршување со очекувано оптоварување од 100 корисници.	http://localhost:8080/api/v1/books	GET	Elapsed time per request <= 4000 ms Latency <= 3500 ms Connect time <= 700ms Fail rate < 5%	Endurance testing
NFTR5	Да се оптоварување од 500 истовремени нишки кои креираат валидни ресурси на серверот	http://localhost:8080/api/v1/books	POST	Elapsed time per request <= 5000 ms Connect time <= 500ms Latency <= 4500 ms Connect time <= 600ms Fail rate < 5%	Load testing

Проблем може да настане при нефункциското тестирање бидејќи немаме конкретно дефинирани тест цели за да одредиме кога некој тест е успешен а кога е неуспешен. Поради тоа потребно е да се дефинираат некои референтни метрики според кои ќе се водиме при извршување на тестовите.

Еден пример за тоа е времето на одговор од серверот. Не постои строго дефинирано време кое се смета за прифатливо во сите случаи, туку зависи од контекстот во кој се извршува апликацијата. Но, голем дел од корисниците почнуваат да го губат вниманието кога ќе чекаат одговор од системот подолго од неколку секунди. Оттука, треба да се дефинира ранг на вредности кои се прифатливо времетраење за одговор на барањето од страна на системот. Истото важи за сите други метрики.

Во оваа семинарска работа ќе се разгледуваат следните референтни метрики: изминато време (elapsed time) од испраќање на барањето се до добивањето на последниот одговор од серверот, латентност (latency) изминатото време од испраќање на барањето се дод добивање на првиот одговор од страна на системот, време за конекција (connect time) вклучувајќи го и SSL ракувањето, и ратата на неуспешни барања no_failed_requests / no_all_requests. Очекуваните метрики кои треба да бидат исполнети за секое барање се дадени во колоната „Очекувани метрики" од Табела 4. Множество на нефункциски барања.

4.3 Дефинирање на тест случаи и тестови

За да се задоволат претходно наведените нефункциски барања, потребно е да се креираат тест случаи во форма на кориснички сценарија кои ќе ги опфатат очекуваните акции од страна на корисниците.

Целта е да ги покриеме барањата и соодветно да ги измериме посакуваните метрики и да споредиме дали добиените резултати се задоволителни или има проблем со системот кој треба да се реши пред да се пушти во продукција.

Во Табела 5. Тест случаи за нефункциските тест барања се дефинирани тест случаи кои одговараат на претходно наведените барања.

Колоната NFTR го претставува индикаторот на нефункциското барање на кое се однесува тестот, колоната TC го означува индикаторот на самиот тест, во колоната „Тест случај" е наведен опис на случајот / сценариото кое треба да се имплементира во Apache JMeter, колоната Endpoint го означува endpoint-от кон кој се праќаат барањата и колоната „Метод" го означува HTTP методот од барањето до серверот.

Табела 5. Тест случаи за нефункциските тест барања

NFTR	TC	Тест случај	Endpoint	Метод
NFTR1	NFTC1	Endpoint-ите за GET барањата потребно е да можат да опслужат очекувано оптоварување од 100 корисници истовремено, со ramp up период од 25 корисници на 30 секунди додека не се достигне потребното оптоварување, потоа да се држи оптоварувањето 5 минути, и да се намалува бројот на корисници со ramp down период од 25 корисници на 30 секунди	http://localhost:8080/api/v1/books http://localhost:8080/api/v1/books/{id}	GET
NFTR2	NFTC2	Endpoint-ите за GET барањата потребно е да можат да опслужат екстремно оптоварување од 500 корисници истовремено, со ramp up период од 50 корисници на 100 секунди додека не се достигне потребното оптоварување, потоа да се држи оптоварувањето 5 минути, и да се намалува бројот на корисници со ramp down период од 50 корисници на 100 секунди	http://localhost:8080/api/v1/books http://localhost:8080/api/v1/books/{id}	GET
NFTR3	NFTC3	Да се тестираат endpoint-ите за GET барањата со 5 скока од по 20 корисници, со startup и shutdown времиња од 60 секунди, првите три скока да се држат активни 90 секунди, четвртата со 180 и последната 60 секунди, со иницијални доцнења од 60, 300, 600, 1000 и 1600 секунди соодветно	http://localhost:8080/api/v1/books http://localhost:8080/api/v1/books/{id}	GET
NFTR4	NFTC4	Да се креира оптоварување од 100 истовремени корисници на endpoint-ите за GET барањата со ramp up период од 5 секунди за секои 20 корисници, и да се држи оптоварувањето 15 минути	http://localhost:8080/api/v1/books http://localhost:8080/api/v1/books/{id}	GET
NFTR5	NFTC5	Да се тестира оптоварувањето на endpoint-от за креирање на книги со 500 истовремени запишувања на нови книги	http://localhost:8080/api/v1/books	POST

Забелешка: Во пракса, endurance тестирањето треба да се одвива подолг период (некогаш и со денови). Во конкретната семинрска работа, оптоварувањето се чува само 15 минути

поради ограниченост на ресурси. Исто така стрес тестирањето би требало да се одвива со зголемување на бројот на корисници се до одредена точка на пад на системот, но во овој случај истото не можеше да се изведе поради ограничувањето дека тестовите и апликацијата се извршуваа на иста машина.

Сите тестови се изведуваат на локална машина каде што е поставен и Веб серверот, базата на податоци и самата апликација која се извршува. Од тие причини некои од тестовите би имале подобри перформанси доколку се тестира сервер кој што е специфично конфигуриран да ги поддржи горе-наведените оптоварувања и се наоѓа во посебна околина од машината на која се изведува тестирањето. Дополнително, овде не може да се забележи мрежна латентност бидејќи и апликацијата и тестовите се извршуваат на иста машина, во реалноста времето на извршување на апликацијата би било побрзо но мрежната латентност би била поголема.

Имплементирање на тестовите во Apache JMeter

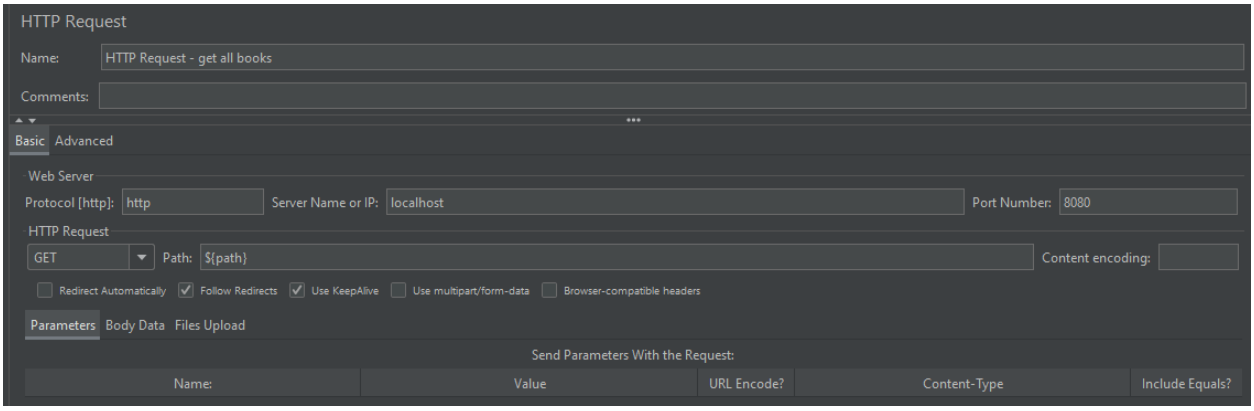
За креирање на тестовите, беа креирани два тест плана: едниот се однесуваше на тестирање на GET методите за сите книги и за поединечна книга, а вторитот тест план се однесуваше на endpoint-от за креирање на нова книга.

Тест план: PerformanceTestingGetBooks

Во првиот тест план се имплементирани тест случаите од NFTC1 до NFTC4. За нивната имплементација се искористени четири посебни групи на нишки (thread groups).

Првата група се состои од HTTP Request Sampler во кој што се специфицира HTTP барањето до серверот. Тука важно е да се специфицира протоколот кој се користи, во овој случај е HTTP, потоа името на серверот или IP адресата, портата на која слуша серверот, HTTP методот кој се користи за барањето и патеката до ресурсот. Сите овие спецификации може да се видат на Слика 1. Дефинирање на HTTP Request Sampler.

Важно е да се забечежи дека за полето Path во овој случај е искористена променлива чие што име е path. Со помош на ${path} се пристапува вредноста на променливата path.

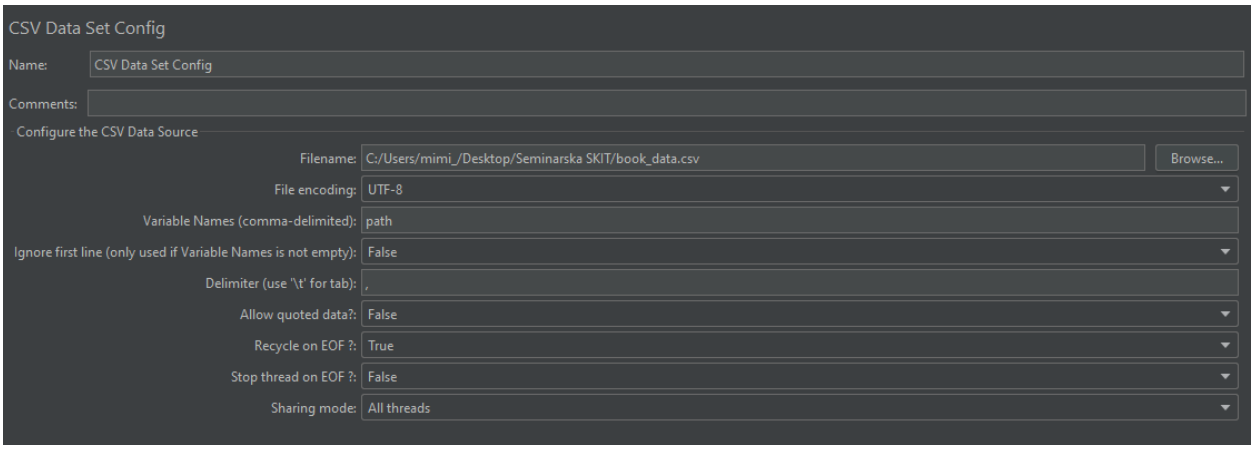


Слика 1. Дефинирање на HTTP Request Sampler

Оваа променлива содржи URI адреса која во конкретниот случај се зема од .csv датотека со помош на CSV Data Set Config елементот. Целта е да може истото барање да се изврши на различни ресурси со цел да се види како системот се справува кога голем број на корисници би барале различни книги истовремено.

Самата датотека содржи само една колона во која секоја редица содржи URI на пример: /api/v1/books, /api/v1/books/1, /api/v1/books/80000 итн. Истата е достапна на: https://github.com/MilenaTrajanoska/books_rest_api_testing/tree/main/src/main/resources/performance_testing/book_data.csv

Конфигурацискиот елемент е прикажан на Слика 2. Конфигурација на податоци од датотека. Потребно е да се постави патеката до датотеката во полето Filename, типот на енкодинг на датотеката и најважно името на променливата на која соодветствуваат вредностите во датотеката. Во овој случај тоа е променливата path која е истата променлива која се користи во HTTP Sampler-от.



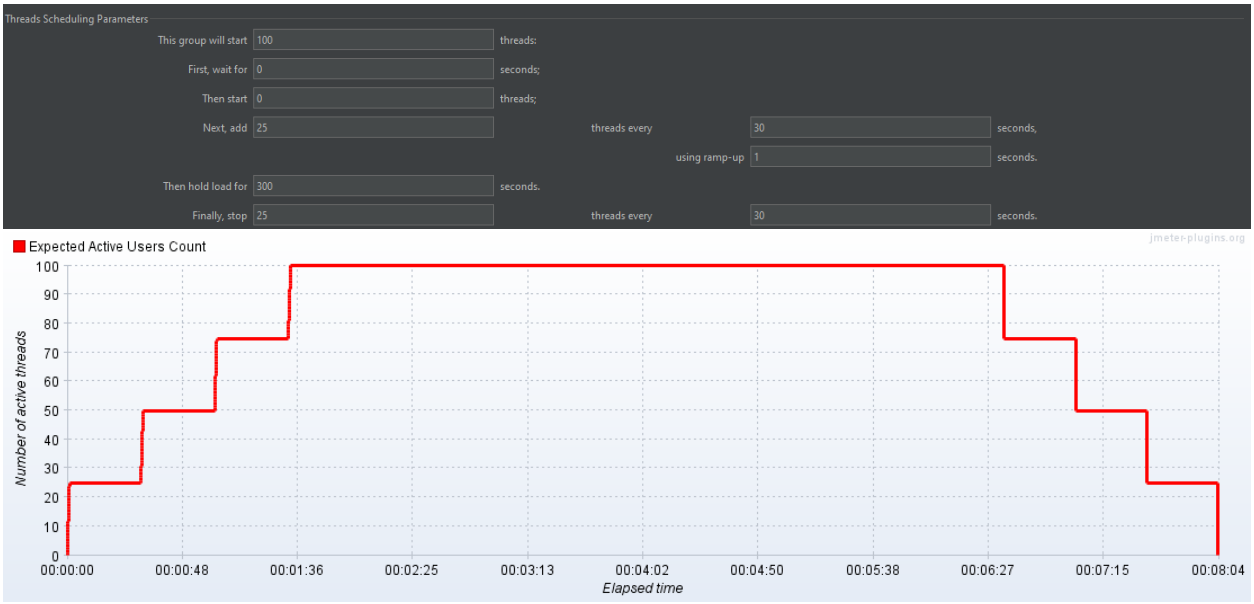
Слика 2. Конфигурација на податоци од датотека

Со специфицирање на вредностите Recycle on EOF да биде True и Stop thread on EOF да биде False, се овозможува кога ќе се прочитаат сите податоци од датотеката, истиот процес да се повтори одново, со што истото URI обезбедуваме да се повика повеќе пати.

На крајот, додаден е View Results in Table Listener со помош на кој резултатите од извршување на барањата се зачувуваат во датотека на фајл системот за да можат да се анализираат подоцна.

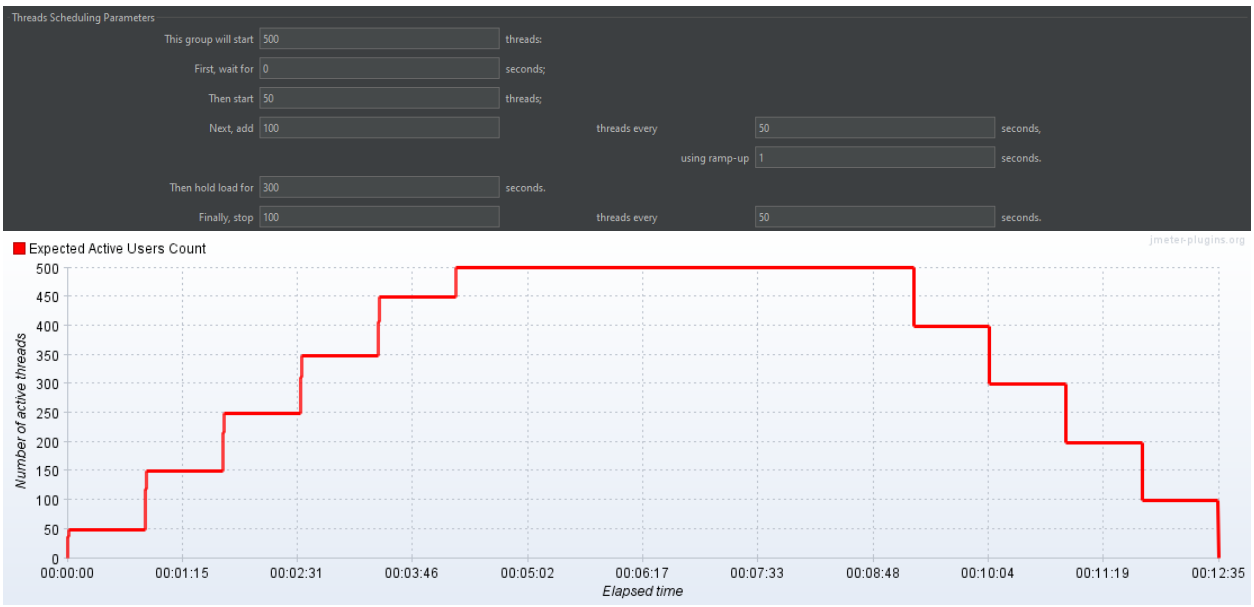
Овие конфигурации се еднакви за секоја од четирите групи на нишки. Тоа што се разликува е типот на thread group и конфигурацијата на секоја од четирите групи.

Првата thread group група е дефинирана како stepping thread group која ќе активира 100 нишки со 25 нишки на секои 30 секунди со ramp up од 1 секунда додека не достигне 100. Потоа се држат активни 100те нишки 300 секунди (5 минути) и на крајот се намалуваат по 25 нишки на секои 30 секунди. Оваа спецификација одговара на првиот тест случај NFTC1, графичкиот приказ може да се види на Слика 3. Креирање на load тест.



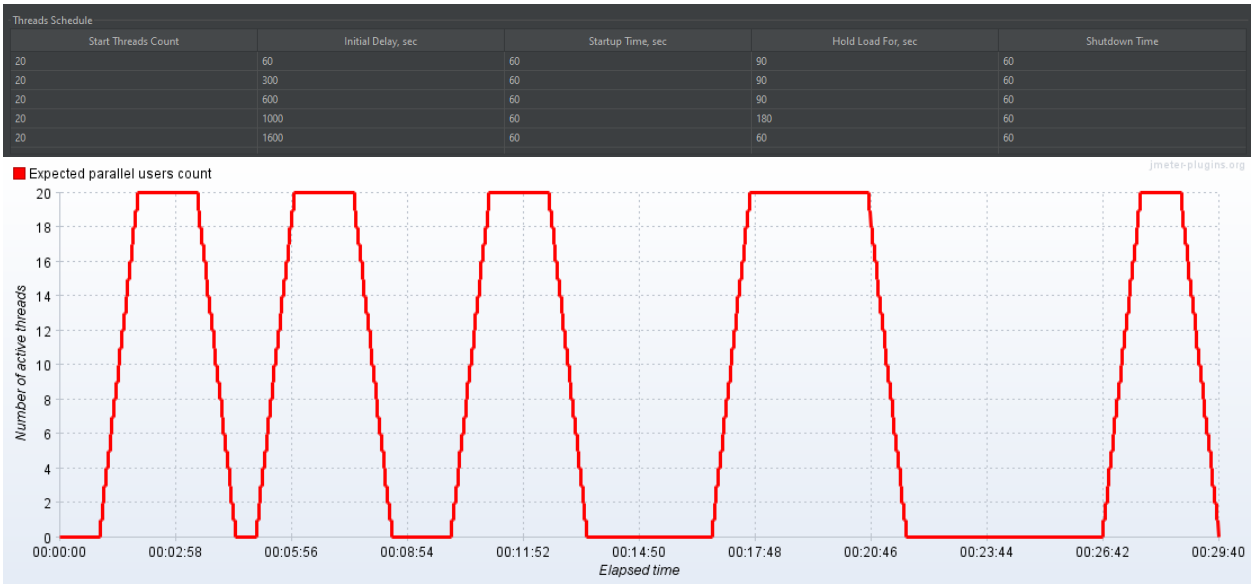
Слика 3. Креирање на load тест

Втората thread group група е инстанца од Stepping thread group која што го дефинира тестот кој се однесува на тест случајот NFTC2 за stress тестирање. Се поставува екстремно оптоварување од 500 корисници истовремено, со ramp up период од 50 корисници на 100 секунди додека не се достигне потребното оптоварување, потоа да се држи оптоварувањето 300 секунди (5 минути), и да се намалува бројот на корисници со ramp down период од 50 корисници на 100 секунди, за брзо зголемување на бројот на активни корисници. Графички приказ е даден на Слика 4. Дефинирање на stress тест.



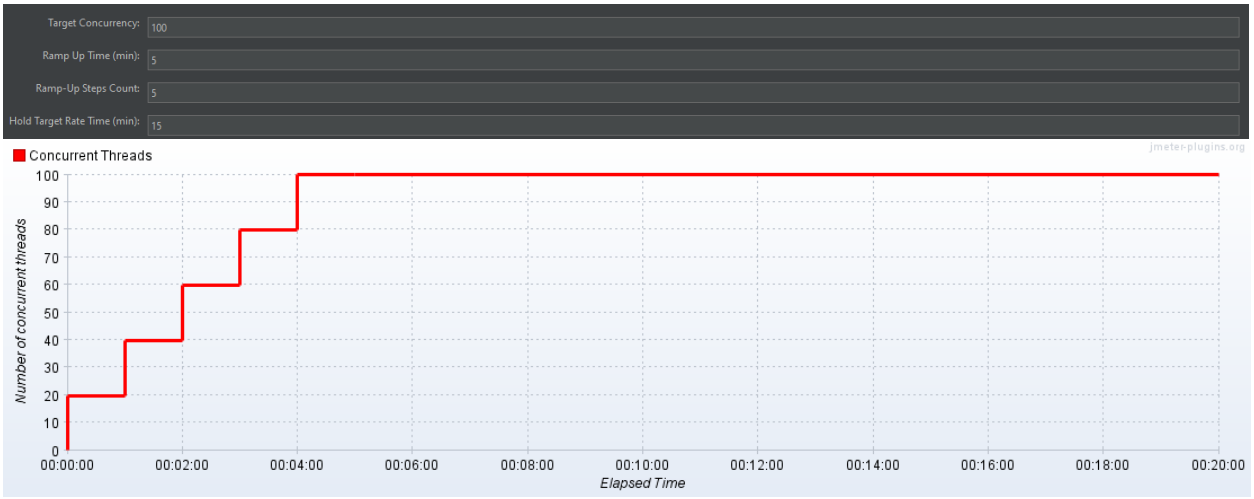
Слика 4. Дефинирање на stress тест

Третата thread group група е инстанца од Ultimate thread group чија што спецификација е дефинирана за изведување spike тест за тест случајот NFTC3. Специфицирани се 5 скока, секој со по 20 нишки, со startup и shutdown времиња од 60 секунди, првите три скока да се држат активни 90 секунди, четвртата со 180 и последната 60 секунди, со иницијални доцнења од 60, 300, 600, 1000 и 1600 секунди соодветно. Графички приказ на тестот е даден на Слика 5. Дефинирање на spike тест.



Слика 5. Дефинирање на spike тест

Четвртата thread group група се однесува на тест случајот NFTC4 за дефинирање на endurance тест. Креира оптоварување од 100 истовремени корисници на endpoint-ите со ramp up период од 5 секунди во 5 чекори (20 корисници на 5 секунди), и го држи оптоварувањето 15 минути, со цел да се испита долготрајно оптоварување на системот со очекуван број на корисници. Графички приказ на тестот е даден на Слика 6. Дефинирање на endurance тест.



Слика 6. Дефинирање на endurance тест

При извршување на самиот тест план потребно е да се постави опцијата Run Thread Groups consecutively со цел да се избегне извршување на групите на нишки истовремено и да создадат дополнително оптоварување на системот.

Тест план: PerformanceTestingCreateBook

Овој тест план се однесува на тестирање на тест случајот NFTC5, односно load тестирање на endpoint-от за креирање на нова книга. Содржи една thread group група која што е обична група. Се дефинира оптоварување од 500 корисници кои истовремено запишуваат нови книги во базата. Ramp up периодот е 0 секунди, односно сите 500 нишки истовремено запишуваат валидни книги во базата на податоци. Секоја нишка запишува само еднаш, односно на крајот потребно е да бидат успешно запишани 500 нови книги.

Слика 7. Дефинирање на load тест за запишување книги

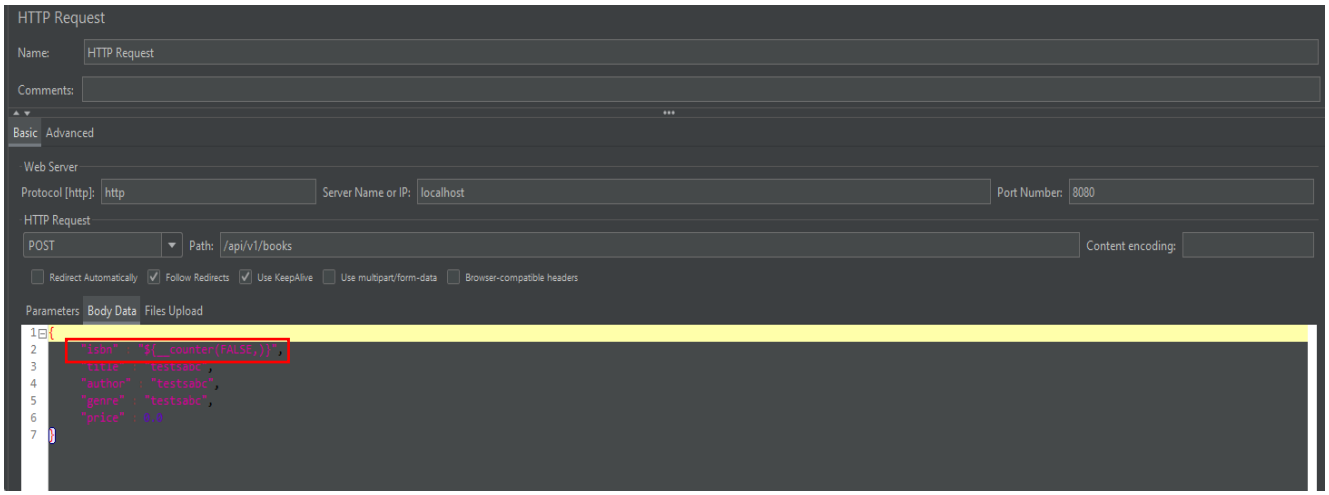
Дефиницијата на тестот е дадена на Слика 7. Дефинирање на load тест за запишување книги.

Во овој случај, потребно е да се дефинираат и заглавја за Content-Type и авторизација на корисниците со системот. За таа цел, покрај основниот HTTP Request Sampler потребно е да се додадт конфигурациски елементи, HTTP Header Manager и HTTP Authorization Manager. Во Header Manager елементот потребно е да се дефинира заглавјето со Name: Content-Type Value: application/json. Authorization Manager елементот потребно е да содржи ред во кој ќе се дефинира основната URL адреса http://localhost:8080, корисничкото име user, лозинката user123* и механизмот на автентикација кој во случајот е BASIC.

Откако ќе се дефинираат конфигурациите на барањето, треба во HTTP Request Sampler-от да се дефинира и самото барање. Во овој случај барањето се разликува од претходно бидејќи го користи HTTP Post методот на патеката /api/v1/books наместо GET како претходно кое користеше патека прочитана од датотека.

Дополнително, потребно е да се додаде валидно тело со податоци на книга кое ќе се испрати и запише на серверот. За таа цел во Body Data делот е дефиниран JSON објект кој содржи isbn, title, author, genre и price атрибути. Важно е тоа што сите атрибути освен isbn се фиксни, бидејќи има ограничување на вредностите на isbn да бидат уникатни. За да се овозможи тоа во овој случај се користи counter функцијата од JMeter која генерира последователни боеви почнувајќи од 1. Со наведување на вредноста FALSE во функцијата кажуваме дека за секоја нишка треба да се креира нов број од counter функцијата наместо да се користи истиот број за сите нишки во рамки на thread групата.

Деталите може да се видат на Слика 8. Дефинирање на POST барање за креирање на книга, дефинирањето на уникатен isbn е означено со црвен правоаголник.



Слика 8. Дефинирање на POST барање за креирање на книга

Нефункциските тестови се достапни во:
https://github.com/MilenaTrajanoska/books_rest_api_testing/tree/main/src/test/performance_testing/

За да може истите успешно да се извршат потребно е да ги промените патеките до .csv ресурсите и патеките за View Results in Table listenter да одговараат на вашите локални.

4.4 Извештај од перформансното тестирање

Тестовите беа извршени на истата машина на која се наоѓаше Веб серверот, базата и апликацијата. Поради тоа се очекува некои од измерените метрики да бидат подобри кога системот би се префрлил на Веб сервер кој е посебна машина, но тука дополнително во латентноста би ни фигурирала и јачината на Интернет врската. Резултатите се дадени во Табела 6. Резултати од нефункциските тестови.

Табела 6. Резултати од нефункциските тестови

NFTC	Max elapsed time (ms)	Max latency (ms)	Max connect time (ms)	Fail rate (%)
NFTC1	14932ms	14932ms	194ms	0.258445 %
NFTC2	41542ms	41542ms	6150ms	3.36%
NFTC3	882ms	882ms	94ms	0%
NFTC4	32496ms	32496ms	592ms	0%
NTFC5	25731ms	25731ms	663ms	0%

Според претходно наведените очекувани метрики можеме да забележиме дека максималмното изминато време (elapsed time) за NFTC1 го надминува очекуваното од 2000 ms, истото важи и за латентноста која надминува 1500ms, од друга страна, времето за конекција е во очекуваните рамки, додека пак ратата на неуспех на API-то е многу ниска (0.26 проценти).

За NFTC2 изминатото време и латентноста значително го надминуваат очекуваното изминато време од 1000ms и латентноста од 6500ms. Ваквите големи отстапувања се неприфатливи во овој случај, што значи дека ќе мора да се направат подобри конфигурации на серверот или да се оптимизира базата на податоци. Времето за конекција е исто многу високо, а рата на грешка е значителмно помала од 10% што значи е прилично прифатлива.

Во однос на NTFC3, сите метрики се задоволителни. Ратата на неуспешни барања е 0% што значи дека сите барања поминале успешно. Бидејќи сите метрики се подобри од очекувањето можеме да заклучиме дека системот добро се справува со нагли скокови во однос на бројот на истовремени корисници.

Според метриките од четвртиот тест, можеме да забележиме дека со очекувано оптоварување од 100 корисници, доколку го држиме активно оптоварувањето подолг временски период, системот почнува да деградира во однос на перформансите. Повеќе од

двојно подолга латентност има во случајот на endurance тестирање со 100 корисници отколку во случајот на load тестирањето со 100 кориснци. Иако ратата на неуспешни барања е 0%, сепак двојно поголемата латентност и изминато време за одговор се неприфатливи во однос на поставените метрики. Максималното време за конекција е во рамките на очекуваното.

На крајот за последниот тест на креирање нови книги, повторно метриките за латентност и изминато време се значително полоши од очекуваните, односно не ги задоволуваат перформансните барања. Максималното време на конекција е малку полошо од очекуваното, но во рамки на прифатливо отстапување од помалку од 100ms. Ратата на неуспех е 0% што значи дека сите барања за креирање на нова книга се извршиле успешно.

Од сите мерења, можеме да заклучиме дека системот се справува во најголем случај без падови со сите испратени барања. Времињата на чекање за сите тестирани операции, и покрај тоа што успешно се изведени барањата, се најчесто неприфатливи за корисниците и потребно е да се направат промени во однос на хардверската конфигурација на серверот. Дополнително, потребно е да се оптимизираат операциите со базата на податоци, или самата база да се префрли на одделен сервер за да може системот да биде помалку оптоварен. Апликацијата треба да се оптимизира и за конкурентно користење, на пример да се воведат опции за кеширање на барања, или веќе побарани податоци од базата.

Резултатите од перформансните тестови се достапни во:

https://github.com/MilenaTrajanoska/books_rest_api_testing/tree/main/src/main/resources/performance_testing

Endurance и spike резултатите од тестовите се во .zip архива бидејќи зафаќаат повеќе од граничните 100MB.

5 Заклучок

Целите на оваа семинарска работа беа да се изведе функционално и перформансно тестирање на REST API за изведување на основни операции со книги.

Од направената анализа, можеме да заклучиме дека во најголем дел од функционалностите на API-то работеа коректно, освен во случајот на паралелен пристап и валидации со регуларни изрази на самите податоци од моделот.

Во однос на перформансното тестирање мора да се земе предвид ограничувањето дека системот кој се тестираше се извршуваше на Веб сервер на истата машина со системот за управување со базата на податоци и со системот за тестирање. Оттука, може да се претпостави дека дел од измерените перформанси се деградирани, но исто така намалена е мрежната латентност поради истите причини. И покрај тоа што реалните перформанси се можеби подобри од тоа што е измерено, сепак има неприфатливи отстапувања во однос на

латентноста и времето на одговор што мора да се адресираат со промени во самиот систем. Можни промени се вовдеување на кеширање на одговорите од серверот или податоците земени од базата и прилагодување за паралелно извршување на операциите од повеќе корисници.

Алатките кои беа користени за тестирање на системот се RestAssured библиотеката во Java со JUnit5 за комплетно автоматизирање на процесот на функциско тестирање. Изведените функциски тестови ја доведуваат базата на податоци до првичната форма откако ќе се извршат сите. За перформансното тестирање се користеше алатката ApacheJMeter. При изведување на последниот тест за креирање на нови книги, во базата на податоци има дополнителни 500 книги освен оригиналните, кои имаат isbn вредности од 1 до 500, и треба да се избришат од базата по завршување на тестот.

6 Референци

1. https://testconf.ru/wp-content/uploads/2018/04/6-H1-19-%D0%98%D0%B2%D0%B0%D0%BD-%D0%9A%D0%B0%D1%82%D1%83%D0%BD%D0%BE%D0%B2-Test-Design-and-Automation-for-REST-API_TestConf-min.pdf
2. https://rest-assured.io/
3. https://junit.org/junit5/
4. https://blog.testproject.io/2021/07/28/rest-api-automation-from-scratch/
5. https://www.toolsqa.com/rest-assured-tutorial/
6. https://jmeter.apache.org/
7. https://jmeter.apache.org/usermanual/test_plan.html
8. https://www.blazemeter.com/blog/rest-api-testing-how-to-do-it-right
9. https://chamikakasun.medium.com/rest-api-load-testing-with-apache-jmeter-a4d25ea2b7b6
10. https://spring.io/guides/tutorials/rest/
11. Код за REST API: https://github.com/MilenaTrajanoska/rest_api
12. Код од тестовите: https://github.com/MilenaTrajanoska/books_rest_api_testing