УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Никола Димић

АУТОМАТСТКО ТЕСТИРАЊЕ МИКРОСЕРВИСНИХ АПЛИКАЦИЈА

мастер рад

Београд, 2022.

Ментор:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: септембар 2022.

Захваљујем се менторки на саветима и подршци и родитељима, сестри и Нади на помоћи и мотивацији.

Наслов мастер рада: Аутоматстко тестирање микросервисних апликација

Резиме: Микросервисна архитектура представља популарно решење за дизајнирање комплексних дистрибуираних система. Аутоматизацијом тестирања микросервисних система може се осигурати велика поузданост како појединачних сервиса тако и целокупног система. Циљ рада представља сагледавање особина микросервисне архитектуре, као и техника аутоматског тестирања које се могу применити на систем који ту архитектуру користи. У склопу рада је развијена једноставна микросервисна веб апликација *Gelos* и оквир за аутоматско тестирање те апликације на више нивоа.

Кључне речи: развој софтвера, верификација софтвера, аутоматско тестирање, микросервиси, развојни оквир *Playwright*

Садржај

Глава 1

Увод

Микросервисна архитектура постала је популаран избор за заснивање дизајна апликација услед добре скалабилности решења, мале спреге делова система и могућности брзог развоја софтвера. Интеграција постојећих микросервиса у склопу имплементације софтверских решења и брз развој додатно повећавају важност тестирања. Аутоматизација тестирања значајно убрзава процес имплементације и повећава поузданост софтверског решења, али представља велики изазов поготово у дистрибуираним системима. Циљ овог рада је опис карактеристика микросервисне архитектуре и начина на који се технике аутоматског тестирања могу применити на апликацију која користи ту архитектуру.

У оквиру рада имплементирана је микросервисна апликација *Gelos* која за циљ има претраживање и помоћ при избору филма односно књиге за читање, након чега је имплементиран систем за тестирање те апликације на више нивоа. Систем који се тестира састоји се од два микросервиса и клијентске апликације. Кроз јединичне, интеграционе и системске тестове представљене су могућности аутоматизације тестова овог система у целости.

У поглављу 2 представљене су карактеристике и особине које одликују микросервисну архитектуру. Поред тога направљен је осврт на начине интеграција сервиса, кроз обрађивање архитектуре *REST* (енг. *REpresentational State Transfer*). Поглавље се такође бави архитектуралним решењима за имплементацију појединачних микросервиса.

Поглавље 3 обрађује тему аутоматског тестирања. Дефинисањем различитих модела за поделу тестова представљени су различити начини и типови тестирања, уз осврт на примену тих типова на микросервисне апликације. У

склопу поглавља посвећена је пажња и архитектуралним решењима за систем тестирања као што је *Објектни модел странице* (енг. *Page Object Model*).

Поглавље 4 укратко даје преглед технологија употребљених приликом имплементације апликације и система за тестирање. Такође, поглавље описује начин креирања сваког од микросервиса, као и клијентске апликације. Након тога представља начин имплементације јединичних, интеграционих и системских тестова који су одговорни за поузданост апликације.

У последњем, 5. поглављу, изведен је закључак где су сумирани доприноси и начини за даља унапређења.

Глава 2

Микросервисна архитектура

Усложњавањем проблема који се решавају софтверским решењима расте и комплексност апликације која тај проблем решава. Као и код многих математичких проблема, комплексност у софтверским решењима превазилази се разбијањем домена проблема у више поддомена.

Проблем сложености софтверских решења, превазилази се техникама које теже ка што већој кохезивности кода. Кохезивност кода представља идеју груписања модула сличних функционалности заједно. Микросервисна архитектура почива на овом принципу, али је циљ комплетно раздвајање делова система уз јасно дефинисане границе.

Микросервисна архитектура представља архитектурални стил који структуира систем као скуп малих сервиса који су минимално спрегнути, оријентисани ка специфичном домену, и лаки за интеграцију и одржавање. Сваки сервис представља логичку целину која је минимално спрегнута у односу на остале делове система.

Како би се лако интегрисао, сервис дефинише јавни интерфејс преко ког може комуницирати са другим сервисима. Комуникација између сервиса најчешће се одвија путем мрежних позива, који представљају скупе операције, па се приликом пројектовања подстиче ниска спрегнутост система. [19].

2.1 Особине архитектуре

Предности и мане микросервисне архитектуре већински проистичу из особине ниске спрегнутости делова система. Како микросервисна архитектура представља вид дистрибуираних система, већински наслеђује предности и мане које одликују овај тип система.

Неке од особина архитектуре су композитност, отпорност на отказивање, лако скалирање, локализовање података, технолошка хетерогеност и могућност дистрибуирања послова. Ипак микросервисна архитектура не представља универзално софтверско решење, па је не треба користити у свим ситуацијама [19]. У наставку поглавља биће детаљније речи о поменутим особинама микросервисне архитектуре уз осврт на разлике у односу на монолитну архитектуру.

Због раздвојености делова система, детектовање грешака унутар система представља велики изазов приликом имплементације микросервисних апликација. Тема тестирања микросервисних апликација биће детаљно обрађена у поглављу 3.

Композитност

Одвајањем логичких компоненти које су одговорне за специфичан део апликације, омогућава се смањење редундантности кода и коришћење постојећих решења при имплементацији. Сличне апликације често имају заједнички подскуп функционалности. Како је путем интерфејса једноставно повезати више сервиса, већ постојећи сервиси се лако могу укомпоновати при имплементацији новог система [19].

За проблеме који су врло чести, попут ауторизације и аутентификације унутар апликације, постоји велики број постојећих микросервисних решења која се једноставно могу интегрисати у микросервисну апликацију. Уколико одређени сервис не задовољава захтеве апликације може се заменити уз минималне измене кода.

Дистрибуирање развоја и технолошка хетерогеност

Ниска спрегнутост омогућава паралелни развој делова система, али и могућност коришћења различитих технолошких и архитектуралних решења за

сваки од микросервиса. Тимови могу паралелно радити на различитим целинама система имплементирајући унапред договорен јавни интерфејс сервиса. Сваки сервис кроз свој интерфејс дефинише који ће део система бити доступан кориснику.

Променама унутар појединачних сервиса не мења се целокупан рад система, све док су интерфејси непромењени. У односу на перформансе које су захтеване, сваки микросервис може користити ону врсту базе података, радног оквира или програмског језика који најбоље одговара проблему који сервис решава. Експериментисање са новим технологијама има знатно мање последице по целокупан рад система јер се промене често изолују само на један микросервис, за разлику од монолитних система, где промене у технолошком домену подразумевају мењање целокупног система [19].

Локализовање података

Како би сервис био комплетно одговоран за специфичан посао, потребно је да буде независан и на нивоу података. Сваки микросервис користи сопствену колекцију података или податке добија од других сервиса који имају специфичну намену. Независност на нивоу података омогућава преносивост и вишеструку употребљивост али и отежава пројектовање због повећане опасности од редундантости. Ипак, постоје случајеви где ова врста редундатности није дозвољена па сервиси користе само делове дељене базе података.

Скалирање

Услед повећања броја корисника апликације потребно је повећати хардверске ресурсе додељене систему. Коришћењем монолитне архитектуре ово представља велики изазов јер је овакав тип скалирања могућ само уколико се целокупан систем скалира. С друге стране, микросервисна архитектура омогућава повећавање хардверских ресурса само за оне делове система којима је то неопходно. Скалирање се може обављати и динамички тако што ће се сервису, на основу броја захтева, доделити одговарајући ресурси [19].

Отпорност на отказивање

Отказивање делова система код монолитних апликација подразумева пад комплетне апликације. Монолитне апликације превазилазе тај проблем по-

кретањем више инстанци апликације како би се смањила вероватноћа отказа целог система. Ова метода захтева додатне ресурсе и комплексан систем који регулише апликацију у случају пада.

Насупрот томе, пад једног од делова микросервисне апликације не утиче на рад остатка система. Уколико постоји резервна инстанца сервиса који је отказао, апликација може наставити са нормалним функционисањем. Ако то није случај, апликација може наставити да ради са редукованим функционалностима.

Брзо испоручивање софтвера

Непрекидно испоручивање софтвера (енг. *continuous deployment*) један је од централних принципа агилног развоја софтвера. Испоручивање софтвера дефинисано је као низ корака које је потребно одрадити како би се нова верзија апликације оспособила за употребу. Овај процес састоји се од корака као што су дистрибуција нове верзије, инсталација, деинсталација, ажурирање и праћење верзија.

При коришћењу монолитне апликације, свака промена у коду најчешће захтева да се цела апликација испоручи. Како је испорука комплексан процес, у пракси се испоручивање софтвера врши тек када је одређен скуп промена извршен. Увећавањем скупа измена у коду, расте и шанса да ће приликом испоруке софтвера доћи до грешке. Услед великог броја измена, може бити захтевно и временски скупо наћи конкретну промену која је довела до грешке.

Микросервисна архитектура омогућава да се промене на једном микросервису испоручују независно од остатка система. Уколико се испорука не изврши исправно, налажење грешке знатно је једноставније јер је изоловано на један део система, док остали остају непромењени. Дистрибуирање нових верзија кода се извршава брже јер се пре испоруке не морају чекати остали делови система. Међутим, раздвајање испоручивања софтвера за сваки од сервиса није без последица. Како би апликација користила најновије стабилне верзије кода, потребно је имплементирати специфичан систем за испоручивање софтвера у зависности од апликације [19]. Овај систем често може бити јако комплексан па захтева детаљно планирање при дизајнирању.

2.2 Интеграција микросервиса

Начин комуникације између сервиса представља јако битан аспект пројектовања микросервисне апликације. Исправно дизајниран начин комуникације треба да омогући сервисима да кроз развој задрже своју аутономност и да се мењају независно од остатка система.

Тежећи ка једноставним и брзим начинима интеграције сервиса, микросервисна архитектура се одликује одсуством строге формализације протокола. У наставку поглавља биће речи о једном од једноставнијих, и баш из тог разлога, популарнијих начина интегрисања микросервиса — коришћењем обрасца архитектуре *REST* у комбинацији са *HTTP* протоколом.

Образац архитектуре *REST* и *HTTP* протокол

Образац архитектуре *REST* (енг. *REpresentational State Transfer*) представља скуп принципа и ограничења помоћу којих системи могу комуницирати. Принципи архитектуре који омогућавају лаку интеграцију микросервиса су:

Клијент - сервер — архитектура подразумева раздвајање клијентског и серверског дела система. Све док је интерфејс стандардизован, клијентски и серверски део система се могу независно развијати. Раздвајање система на микросервисе представља уопштење овог принципа.

Без стања — архитектура подразумева да сваки клијентски захтев ка серверу мора садржати све потребне информације без обзира на тренутно стање сервера. Сервер не чува информације о тренутној сесији између клијента и сервера. Уопштено на микросервисну архитектуру, ниједан од сервиса не чува информације о комуникацији са клијентом или другим сервисом.

Ресурси — архитектура као централну аутономну јединицу поставља ресурс. Ресурси представљају податке којима сервис располаже. Начин на који сервис интерно рукује подацима сакривен је од корисника. Репрезентација у којој се подаци шаљу кориснику зависи од клијентског захтева и стога се репрезентација ресурса може мењати.

Стандардизација интерфејса, односно порука које се шаљу између сервиса, захтева добро дефинисан протокол. При имплементацији архитектуре *REST* користи се протокол *HTTP*.

HTTP (енг. *HyperText Transfer Protocol*) протокол користи скуп метода помоћу којих је могуће операсати подацима, као што су *GET*, *POST*, *PUT* и *DELETE*. Уместо креирања посебних функција за баратање ресурсима, могу се користити постојеће методе које протокол већ пружа. Како је ресурс централна јединица архитектуре, свака од ових метода има јасно значење:

- *GET* — дохвата ресурс,
- *POST* — креира ресурс,
- *PUT* — ажурира ресурс,
- *DELETE* — брише ресурс.

Помоћу дефинисаних метода шаљу се захтеви ка серверу или сервису. Протокол дефинише и формат одговора на послате захтеве. У сваком одговору налази се информација о статусу извршене операције у форми статусног кода. Сваки од кодова има семантичко значење, па тако код *200* значи да је операција успешно завршена, док код *404* обавештава корисника да ресурс није пронађен. Осим статусних кодова, одговор може садржати податке као што су репрезентације ресурса, путање до тих ресурса или метаподатке[1] попут броја ресурса којима се може приступити.

[1]Метаподаци представљају информације о подацима (нпр. број или врста података)

2.3 Дизајн микросервиса

Микросервисна архитектура омогућава коришћење различитих архитектуралних решења приликом имплементације микросервиса. Архитектурални стил сваког микросервиса мора да подржава раздвајање одговорности од остатка апликације. Један од образаца архитектуре коришћених за пројектовање микросервиса јесте архитектура Презентација-Апстракција-Контролер (енг. *Presentation-Abstraction-Controller, PAC*) о којој ће бити речи у наставку поглавља.

Образац архитектуре Презентација-Апстракција-Контролер

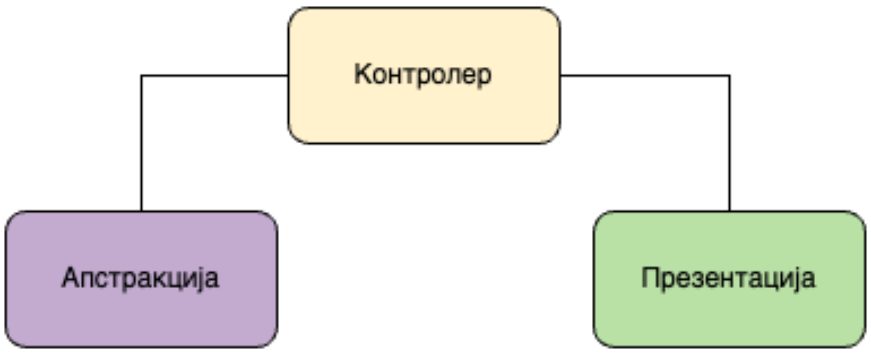
Образац архитектуре Презентација-Апстракција-Контролер заснива се на подели система на три целине, које имају различите функције.

Презентација представља део система одговоран за приказ података.

Апстракција представља део система одговоран за структуру података.

Контролер представља део система одговоран за комуникацију између презентационог и слоја апстракције, као и имплементацију пословне логике система.

Овај образац представља хијерархијску архитектуру у којој слој апстракције и презентациони слој комуницирају искључиво преко контролера. Начин повезивања структурних компоненти архитектуре приказан је на слици 2.1.



Слика 2.1: Образац архитектуре Презентација-Апстракција-Контролер

У контексту микросервиса спољашњи слојеви архитектуре, односно презентациони слој и слој апстракције, су веома танки. Контролер имплементира комплетну пословну логику самог сервиса, односно управља подацима. Традиционално, када је у питању примена архитектуре *PAC* на веб апликације, презентација представља приказ података у корисничком окружењу. Међутим, при имплементацији микросервиса, презентација представља јавни интерфејс, тј. оно што је изложено кориснику сервиса на коришћење. Раздвојеност делова омогућава паралелан рад на деловима сервиса, као и лакше измене унутар целина, попут замене базе података.

Глава 3

Аутоматско тестирање софтвера

Тестирање софтвера представља процес провере софтвера са циљем проналажења грешака и провере квалитета. Процес тестирања проверава да ли су испуњени претходно дефинисани функционални и нефункционални захтеви [7].

Тестирање софтвера може потврдити постојање грешака у систему, али не може потврдити њихово одсуство. Самим тим, битно је тестирати што је више и чешће могуће како би поузданост система била већа. Ипак, није могуће све тестирати како би сваки могући случај употребе био покривен, услед експоненцијалног броја улазних вредности које треба проверити. Како би се доступни ресурси искористили на најбољи могући начин, треба бирати шта и када се тестира. Аутоматско тестирање представља технику која значајно убзава процес верификације софтвера и самим тим омогућава да се већи број функционалности тестира.

Препорука је почети са тестирањем што раније у процесу развоја софтвера. Проналажење грешака у систему у каснијим фазама развоја има огроман утицај на квалитет софтвера јер се те грешке по правилу тешко отклањају. Континуалним тестирањем, смањује се опасност од појављивања грешака у касним фазама развоја. Процес аутоматизације значајно олакшава константне провере система и омогућава поновљив начин за тестирање развијених функционалности.

3.1 Категоризација и начини тестирања

Аутоматско тестирање представља технику тестирања која користи специјалне алате како би извршила проверу исправног извршавања теста. Насупрот томе, мануелно тестирање извршава особа не користећи ову врсту алата. Поставља се питање које тестове је могуће и како аутоматизовати.

Како би се исправно приступило проблему аутоматизације потребно је разумети различите категорије тестова као и циљеве тестирања које они имају. Категоризација тестова дефинисана је квадратом тестирања Брајана Марика приказаног на слици 3.1 [16]. У доњем делу квадрата налазе се тестови који су окренути технологијама, односно помажу програмерима при имплементацији система. Аналогно, у горњем делу квадрата налазе се тестови који су окренути бизнис логици система, односно проверавају да ли систем задовољава захтеве корисника [19]. Посматрајмо сваку од категорија појединачно.

Тестови прихватљивости (енг. *acceptance tests*) [1] — проверавају да ли је имплементирана функционалност одговара дефинисаним захтевима. Како ова врста тестова има јасно дефинисане захтеве односно улазне вредности и очекиване излазе за те вредности, погодни су за аутоматизацију. Представљају подршку процесу имплементације.

Истраживачки тестови (енг. *exploratory tests*) — проверавају да ли је могуће покварити неки део система коришћењем апликације. Како би овај тип тестирања био ефективан потребно је деструктивно приступити извршавању теста. Ово се најбоље постиже коришћењем интуиције приликом тестирања, па се овај вид тестова најчешће извршава мануелно.

Јединични тестови (енг. *unit tests*) — проверавају да ли имплементирана функционалност исправно функционише. Услед малог скупа провера које је потребно извршити, ови тестови се углавном аутоматизују и представљају најосновнији али и најбржи вид аутоматских тестова.

Тестови перформанси и безбедности (енг. *performance and security tests*) — проверавају да ли систем испуњава очекиване перформансе, као што

[1] Тестови прихватљивости разликују се од корисничких тестова прихватљивости (енг. *user acceptance tests*), који представљају врсту мануелног тестирања које извршавају реални корисници апликације. Како је циљ ове врсте тестова кориснички дојам о функционалностима апликације, ова врста тестова не може се аутоматизовати па није обрађена у склопу овог рада.

је време првог учитавања странице. Тестови безбедности проверавају у којој мери је систем заштићен од малициозних напада. Аутоматизовањем тестова перформанси проверава се у којој је мери систем оптимизован.

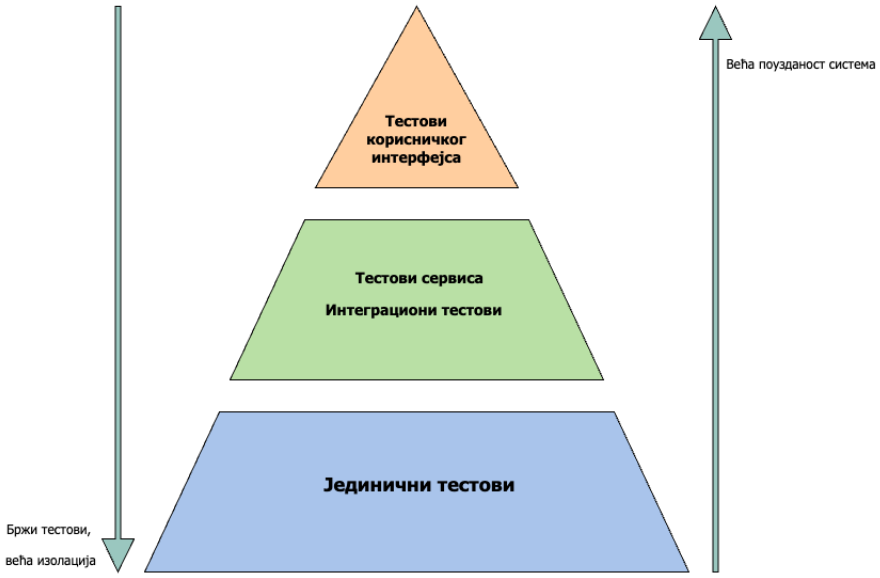
Слика 3.1: Квадрат тестирања Брајана Марика

Основни услов за аутоматизацију теста јесте детерминистичко понашање функционалности која се тестира. У пракси није могуће, а ни пожељно све аутоматизовати. Мануелно тестирање има предности за одређене типове тестирања али већински се тежи ка што већем нивоу аутоматизације. Како се систем мења тако је потребно мењати и аутоматске тестове који га тестирају. Приликом додавања нових функционалности битно је паралелно додавати нове аутоматске тестове, како би поузданост система остала иста. Време извршавања теста као и број тестова варирају у зависности од типа теста. У наставку биће приказани типови аутоматских тестова као и препоручена расподела количине тестова у односу на тип.

3.2 Типови и опсег аутоматских тестова

Појавом агилних методологија развоја софтвера, повећала се потреба за аутоматским тестирањем софтвера. Потребно је тестирати сваку испоручену

итерацију кода на стабилан и поновљив начин, у кратком времену. Модел *пирамиде тестирања* (енг. *test pyramid*) дефинише скуп препорука како аутоматски тестирати апликацију. На основу опсега који тестирају, модел дефинише четири врсте тестова, као и расподелу броја тих тестова у односу на врсту [19]. Детаљан опис модела налази се на слици 3.2.



Слика 3.2: Модел *пирамиде тестирања*

Како су јединични тестови најмањег опсега, логично су и најбржи за извршавање, па је препорука максимално тестирати јединице кода. Аутоматско тестирање интеграција[2] делова система представља начин да поузданост система као целине буде већа, док је цена извршавања и даље мала. Самим тим, тежи се да број ових тестова буде што већи. Тестови корисничког интерфејса представљају скупе операције јер захтевају рад целокупног система, па је приликом одлучивања шта аутоматизовати важно одредити приоритете.

Опсег тестова у контексту микросервисне архитектуре

Имплементација микросервисне апликације представља компликован процес у ком на различитим деловима апликација често раде другачији тимови. Да би целокупна апликација била поуздана, потребно је тестирати све њене

[2]Појмови тестирање интеграција и интеграционо тестирање (енг. *integration testing*) се могу користити као синоними.

делове у изолацији, као и интеграције тих делова. Коначно, потребно је тестирати апликацију на начин како је корисник користи, односно целокупан систем истовремено.

Микросервисна архитектура, због своје грануларности, омогућава аутоматско тестирање различитих делова апликације у потпуној изолацији, што при коришћењу монолитне архитектуре није могуће. Ово омогућава бржи развој апликације, већу поузданост али и могућност да се модел пирамиде у потпуности примени на систем.

3.3 Тестирање јединица кода

Јединично тестирање представља проверу исправности појединачних логичких целина у изолацији. Циљ тестова је верификовати да се свака јединица кода понаша у складу са дефинисаним захтевима. Од свих врста тестова, ови тестови су најбржи и најчешће одстрањују велики број грешака.

Тестови јединица кода се могу писати одмах након, али и пре имплементације одређене функционалности, уколико се користи методологија развоја вођеног тестовима (енг. *Test Driven Development — TDD*). Одговорност за имплементацију јединичних тестова најчешће преузимају програмери који су радили на самој функционалности, јер је за писање тестова често потребно техничко, али и доменско знање. Познавање граница изоловане функционалности омогућава тестирање јединице за различите улазне параметре.

Како би се тестови јединица кода исправно аутоматизовали, потребно је комплетно одстранити зависност сваке јединице од остатка система приликом тестирања. Реални подаци од којих зависи функционалност јединице, замењују се „лажним" подацима (енг. *mock data*) који су претходно припремљени. Коришћење овог система има више предности. Одстрањивањем зависности омогућава се тестирање комплексних компоненти апликације које зависе од много улазних параметара, али се и омогућава независно развијање функционалности. Уколико је формат улазних зависности унапред познат, јединица система може несметано имплементирати нове фунцкионалности користећи предефинисане податке.

Јединични тестови су неизоставан део непрекидне интеграције софтвера (енг. *continuous integration*) на основу које се након сваке нове измене покреће скуп аутоматских тестова. Пролазећи кроз итерације испоруке софтвера,

јединичним тестовима се осигурава да ни једна нова измена није условила пад неке од постојећих јединица система.

3.4 Тестирање интеграција и сервиса

Поузданост да логичке целине исправно раде у изолацији доприноси укупној поузданости система, али не гарантује да ће интеграција тих целина исправно функционисати. Тестови интеграције проверавају комуникацију између више јединица кода са циљем да пронађу грешке у интерфејсима интегрисаних компоненти.

Аутоматизација интеграционих тестова омогућава брзу проверу комуникације између измењених компоненти у процесу развоја софтвера. Како ова врста тестова покрива веће логичке целине, извршавање тестова је спорије у односу на јединичне тестове, па је и број тестова мањи.

При коришћењу микросервисне архитектуре, више јединица кода интегрише се у један сервис. Ова интеграција тестира се тестовима сервиса, где је циљ проверити исправност јавног интерфејса који сервис пружа.

Као што је претходно наведено, микросервисна архитектура подразумева велику грануларност система и самим тим захтева јасан начин комуникације између свих делова система. Како се комуникација одвија између више сервиса, повећава се значај интеграционих тестова. Тестови појединачних сервиса, као и тестови интеграција између различитих сервиса, представљају додатни ниво поузданости који се при коришћењу монолитне архитектуре не омогућава. Пронађена грешка тачно указује која интеграција тј. који интерфејс унутар система не даје очекиване резултате, што значајно олакшава процес елиминације грешке.

3.5 Системско тестирање

Системско тестирање представља најкомплекснију врсту тестирања јер посматра систем као униформну целину. Тестови пролазе кроз све делове апликације па се због тога често називају и тестови с краја на крај (енг. *end to end — e2e*). Пре самог тестирања целине потребно је тестирати јединице кода, као и интеграције тих јединица. Упркос великој поузданости у квалитет софтвера коју пружају интеграциони и јединични тестови, коришћењем свих

делова система истовремено могу се пронаћи проблеми који другачије нису могли бити уочени.

Посматрањем система из угла крајњег корисника добија се реална слика о томе како се систем као целина понаша. Системски тестови се често извршавају мануелно јер захтевају корисничко развојно окружење и низ комплексних акција које описују случај употребе система. Међутим, алатима који симулирају корисничко понашање могуће је аутоматизовати процес системског тестирања. Услед комплексности ових тестова, време извршавања је знатно веће од осталих типова тестова па их због тога није практично имати превише. Приликом писања тестова треба осигурати да су изабрани случајеви употребе репрезентативни, односно да осликавају реалну употребу апликације. Слично као и код јединичних тестова, дефинисање случајева употребе који ће се аутоматизовати може се одвијати после, али и пре имплементације функционалности.

Системски тестови имају значајну улогу у методологији агилног развоја софтвера. Покретањем аутоматских системских тестова пре сваке испоруке обезбеђује се сигурност у то да сви покривени случајеви употребе и даље раде. Уколико то није случај, најчешће се генерише извештај који садржи слику или скуп корака који описује како је до грешке дошло.

Како се систем пројектован коришћењем микросервисне архитектуре састоји од много делова који се често не развијају истовремено, системски тестови представљају врло важан корак у развоју апликације. Како би системски тест био исправно извршен, велики број независних делова мора синхроно да функционише. У пракси се дешава да тест падне због разлога који нису нужно везани за функционалност која је тестирана, нпр. због привременог кашњења одговора сервиса. Тестови који некад пријављују грешку, а некад не, без детерминистичког правила, називају се нестабилни тестови (енг. *flaky tests*). Приликом паралелног извршавања већег броја сличних тестова често долази до нестабилности тестова, услед проблема који се јављају приликом коришћења дељених ресурса.

Нестабилни тестови представљају велики проблем и циљ их је у што већој мери елиминисати. Информација да нестабилан тест пријављује грешку у систему, не доприноси поузданости, јер су разлози због којег тест пада невезани за тестирану функционалност. Уколико се у скупу тестова који се извршавају налази много нестабилних тестова, губи се поверење у валидност резултата

тестова. Овај феномен назива се нормализација одступања.

Технике тестирања корисничког интерфејса

Како би се симулирало корисничко понашање при извршавању тестова користе се алати попут развојних оквира *Playwright* [17] и *Selenium* [29]. Могуће је аутоматизовати системске тестове у различитим системима али у склопу овог рада фокус ће бити на веб апликацијама.

Покретање претраживача у ком се извршавају тестови регулише коришћени развојни оквир, о чему ће бити више речи у поглављу 4.1. Развојни оквир пружа функције за дохватање елемената са странице помоћу кориснички дефинисаних селектора. Селектори представљају шаблоне помоћу којих се одређени елемент може лоцирати на страници апликације, односно унутар *DOM* стабла које претраживач приказује. Специфични развојни оквири подржавају различите типове селектора. Неки од њих су:

- *CSS* селектори — лоцирају елементе странице на основу скупа правила која се користе за стилизовање *HTML* елемената,
- *xpath* селектори — лоцирају елементе странице на основу путање која дефинише пут кроз *XML* односно *HTML* страну,
- текстуални селектори — лоцирају елемент на основу текста који се налази у склопу елемента.

Након што је елемент на страници пронађен, случај употребе апликације извршава се помоћу функција које дефинише изабрани алат. Коришћењем библиотека за верификацију проверава се да ли је случај употребе испунио предефинсане захтеве.

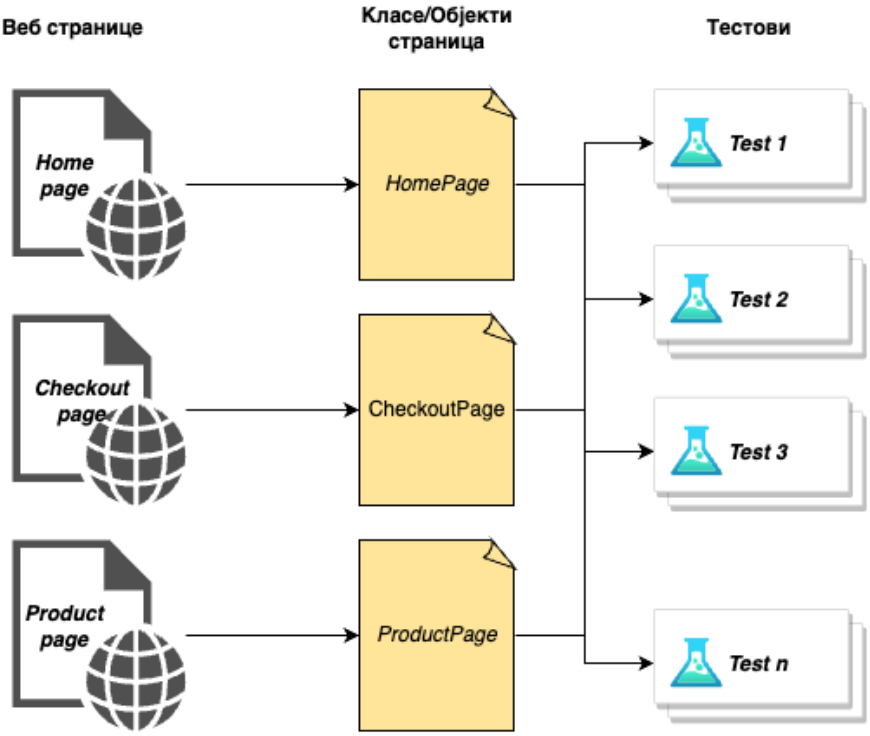
Многи случајеви употребе имају преклапајуће скупове корака. Како би се елиминисала редундатност при писању системских тестова, при пројектовању оквира за тестирање користи се објектни модел странице.

Објектни модел странице

Објектни модел странице користи постулате објектно-оријентисаних методологија, конкретно, особине интерфејса и енкапсулације како би раздвојио шта се тестира од начина тестирања са циљем смањења комплексности. За

сваку страницу апликације или неки смислен функционалан део, прави се класа која представља интерфејс за управљање том страницом. Класа садржи дефиниције селектора који се налазе на датој страници, као и методе помоћу којих се барата самом страницом. На овај начин се све информације о страници апликације налазе на једном месту, па су измене при ажурирању тестова минималне.

Тестови користе дефинисане класе као интерфејсе за извршавање корисничких акција и јасно осликавају случај употребе. Детаљи имплементације сваког корака случаја употребе су сакривени, па је структура теста концизна и јасна. Графички приказ примера имплементације овог архитектуралног модела може се видети на слици 3.3. Пример представља начин моделовања сваке од три странице апликације као и како се те странице користе унутар тестова. Као што је приказано, тестови често користе више различитих интерфејса, јер случај употребе који тестирају обухвата више страница апликације.



Слика 3.3: Објектни модел странице

Глава 4

Имплементација и тестирање апликације *Gelos*

Практични део рада представља имплементацију микросервисне апликације, као и система за тестирање те апликације. Циљ апликације је претраживање и помоћ при избору филма односно књиге за читање. Посебна пажња при изради апликације посвећена је системима за тестирање апликације како на јединичном, тако и на интеграционом и системском нивоу. Архитектурално решење за сваки део овог система је обрађено у склопу овог поглавља. Изворни кôд апликације и система за тестирање може се видети на *GitHub* страници пројекта [5].

4.1 Кратак преглед коришћених технологија

За израду микросервиса коришћен је развојни оквир *Express.js* [25] писан за извршно окружење *Node.js* [8]. За складиштење података коришћен је систем за управљање базом података *SQLite* [23] у комбинацији са алатом за објектно-релационо мапирање *Sequelize* [4].

За израду клијентске апликације коришћена је библиотека *React* [22]. За стилизовање корисничког интерфејса коришћен развојни оквир *Tailwind* [14].

За израду оквира за интеграционо и функционално тестирање система коришћен је развојни оквир *Playwright* [17] док је за генерисање лажних података за тестирање коришћена библиотека *MSW* (енг. *Mock Service Worker*) [32]. У склопу израде компонентних и јединичних тестова коришћена је би-

блиотека *React Testing Library* [6] у комбинацији са развојним оквиром *Jest* [27].

Окружење *Node.js*

Node.js представља извршно окружење који омогућава извршавање *JavaScript* кода ван оквира претраживача односно на самом серверу. Могућност да се и клијентска и серверска страна апликације пишу истим програмским језиком и самим тим убрза процес развоја апликације, допринела је великој популарности језика. Ипак, главну особину овог развојног оквира представља могућност да извршава неблокирајући, асинхрон кôд и тако елиминише потребу за чекањем [31].

У склопу овог пројекта коришћен је и *npm* (енг. *node package manager*) који омогућава једноставну контролу инсталација и верзија коришћених библиотека [21]. Како се пројекат састоји из више независних целина коришћена је „*monorepo*" стратегија за контролу верзија. Ова стратегија омогућава развијање више независних пројеката унутар истог репозиторијума, као и независно покретање сваког од делова система [20].

Развојни оквир *Express.js*

Инспирисан развојним оквиром *Sinatra* [18], *Express.js* се одликује својом једноставношћу, уз могућност да кроз различите библиотеке испуни специфичне захтеве апликације. *Express.js* пружа минималан скуп функционалности које омогућавају једноставну обраду *HTTP* захтева, рутирање и интеграцију са системом за управљање базама података [25].

Како *Express.js* представља веома ниску апстракцију *HTTP* протокола, има врло добре перформансе, па је због те особине и своје једноставности постао стандардан алат за креирање веб сервера и програмских интерфејса апликације унутар окружења *Node.js* [3].

Систем за управљање базама података *SQLite* и библиотека *Sequelize*

SQLite је систем за управљање базама података написан у програмском језику *C*. Дизајниран је тако да буде стабилан, брз и компатибилан са разли-

читим платформама. Насупрот другим *SQL* системима за управљање базама података, нема засебан серверски процес већ комплетну базу података чува унутар једне датотеке [23, 30]

Објектно-релационо мапирање представља технику која омогућава да се подацима из базе података приступа користећи парадигму објектно оријентисаног програмирања. Ова техника омогућава да се подацима приступа кроз објекте односно методе тих објеката па се компликовани *SQL*-упити апстрахују у интуитивне позиве функција. У потпуности се елиминише потреба за употребом језика *SQL* па то додатно убрзава процес развоја софтвера [15]. *Sequelize* је библиотека за објектно релационо мапирање која је заснована на технологији *Node.js* и подржава велики број система за управљање базама података попут *Postgres, MySQL, MariaDB, Microsft SQL Server* и *SQLite* [4].

Библиотека *React*

Библиотека *React* представља једну од најпопуларнијих *JavaScript* библиотека за израду корисничког интерфејса. Почива на идеји да се свака апликација састоји од скупа компоненти које представљају енкапсулиране целине. Свака компонента представља вид шаблона па се може употребити више пута са различитим параметрима што смањује редундантност написаног кода. Како су компоненте јединичне целине оне у себи садрже и приказ саме компоненте у виду *HTML* елемената и *CSS* правила, као и начин на који се те компоненте мењају, у виду *JavaScript* кода [22].

React прави виртуелно *DOM* (енг. *Document Object Model*) стабло које ажурира право *DOM* стабло само за компоненте које су промењене при некој акцији на корисничком интерфејсу. Ово омогућава да се промене дешавају брже и без ажурирања целе веб странице што омогућава креирање такозваних једностраничних апликација [22].

Сама библиотека имплементира само основне функционалности потребне за креирање корисничког интерфејса док се за функционалности попут рутирања морају инсталирати додатне библиотеке. Једна од таквих је библиотека *React router* која је коришћена у склопу овог пројекта [24].

Развојни оквир *Tailwind*

Стилизовање модерних веб страница које се приказују другачије у односу на величину екрана уређаја представља велики изазов у процесу развоја веб апликације. Појава *CSS* развојних оквира који садрже већ стилизоване шаблоне омогућила је једноставно имплементирање модерног корисничког интерфејса. Коришћењем већ постојећих шаблона губи се могућност за прављењем јединственог корисничког интерфејса па се и поред развојних оквира као што је *Bootstrap* користе и стандардна *CSS* правила [26].

Tailwind представља *CSS* развојни оквир који решава проблем на мање рестриктиван начин, задржава брзину и једноставност развоја коју пружају горе поменути развојни оквири, а дозвољава флексибилност коју пружа коришћење *CSS* правила. То се постиже коришћењем предефинисаних класа које у себи садрже велики број функционалности које омогућавају кориснику да лако имплементира комплексне промене на корисничком интерфејсу [14].

Развојни оквир *Playwright*

Playwright је развојни оквир направљен за аутоматизацију системских тестова од стране *Microsoft* тима. Оквир подржава више програмских језика као што су *Java*, *Python*, *C#* и *JavaScript* [17].

Сваки претраживач садржи погонски део који је одговоран за трансформисање *HTML* текста у веб страницу која се приказује на самом уређају. *Playwright* користи протокол *DevTools* који омогућава директну комуникацију са погонским делом претраживача у циљу извршавања тестова у претраживачу [9]. Ова особина чини оквир знатно бржим и стабилнијим од конкуретних технологија које користе протокол *WebDriver*. [29, 2]. Оквир подржава извршавање на прегледачима као што су *Chrome, Firefox* и *Safari* , односно њиховим погонским деловима *Chromium*, *Firefox* и *Webkit* респективно[17, 28, 1].

Једна од главних особина оквира је аутоматско чекање. Пре сваке акције над елементом на веб страници, проверава се да ли је елемент видљив и активан па се тек онда наставља са извршавањем теста. Ово елиминише потребу за имплицитним чекањима у аутоматским тестовима. Имплицитна чекања, односно заустављање теста на предефинисан број секунди како би се одређен елемент учитао, представљају чест узрок нестабилних тестова. Аутоматско чекање такође смањује потребу за експлицитним чекањима, која заустављају

тест док одређен услов није испуњен, јер се већина стандардно коришћених услова аутоматски проверава пред интеракцију са елементом. Смањен број експлицитних чекања у тесту доприноси томе да тест буде концизан и јасан.

Конкурентно извршавање тестова је подржано и једноставно се имплементира услед асинхроне природе оквира *Node.js*. Оквир не подржава тестирање на реалним мобилним уређајима али подржава емулацију претраживача за мобилне телефоне.

Уз алате за тестирање корисничког интерфејса, *Playwright* такође пружа алате за визуелно тестирање, компонентно тестирање, и тестирање програмских интерфејса апликације [17]. Неки од тих алата биће приказани у даљем тексту.

Библиотека за тестирање *React* апликација и библиотека *Jest*

Тестирање појединачних јединица и компоненти система који користи библиотеку *React* може се постићи коришћењем библиотеке за тестирање *React* апликација (енг. *React testing library*) [6]. Библиотека омогућава тестирање појединачних чворова *DOM* стабла из угла корисника. Свака *React* компонента се претвара у *DOM* чвор и помоћу *JavaScript* библиотеке за тестирање — *Jest*, проверавају се очекиване вредности унутар чвора. Ова библиотека пружа сигурност да се подаци приказују на очекиван начин без обзира на друге делове система, као и да различите јединице функционишу унутар компоненти којима припадају [6, 27].

Библиотека *MSW*

Како би тестови били поуздани и подаци који се користе унутар тестова морају бити поуздани. Такође потребно је тестирати клијентски део апликације независно од серверског дела.

MSW (енг. *Mock Service Worker*) представља библиотеку која омогућава да се *API* позиви који су упућени ка серверској страни апликације пресретну и уместо правих, клијентској апликацији врате предефинисани „лажни" одговори. Ово омогућава како тестирање клијентског дела апликације у изолацији, тако и могућност независног развоја клијентског и серверског дела апликације [32]. Да би одговарајући *API* позиви били пресретнути потребно

је дефинисати тачке јавног интерфејса и предефинисане податке који се шаљу клијентској страни апликације.

4.2 Архитектура и дизајн апликације

Развијена апликација подељена је у две целине, на серверски и клијентски део. Серверски део апликације организован је у међусобно незавнисне микросервисе. Сваки микросервис има јасно дефинисан јавни интерфејс помоћу ког клијентска страна апликације приступа ресурсима и функционалностима које микросервис пружа. Подаци које микросервиси користе организовани су тако да сваки микросервис има своју базу података. Комуникација између микросервиса и клијентског дела апликације извршава се коришћењем *HTTP* протокола. У наставку ове секције биће детаљније приказани имплементациони детаљи као и архитектурална решења за сваки од делова апликације.

Микросервиси

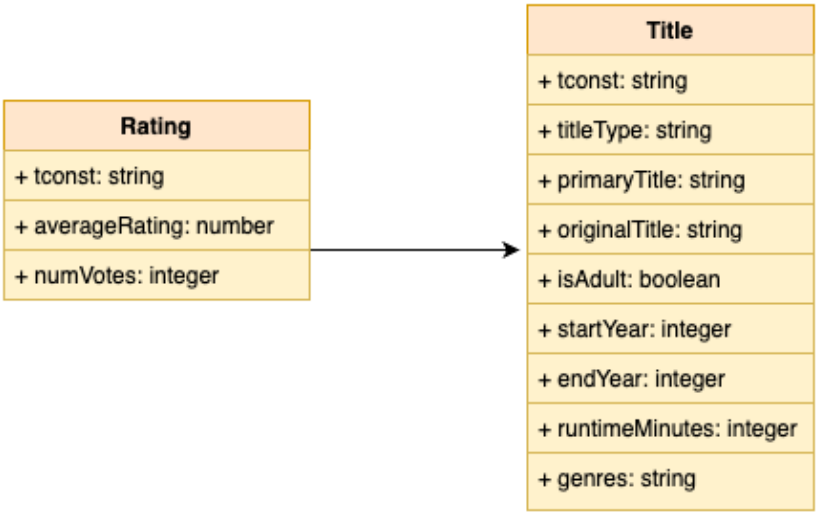
Серверски део апликације састоји се од два независна микросервиса *Movies* и *Books*. Архитектура коришћена при имплементацији оба микросервиса је *Презентација-Апстракција-Контролер* па је структура пројекта подељена у одговарајуће целине:

1. Покретање и јавни интерфејс микросервиса (у кореном директоријуму):
   - Датотека *index.js* — покреће претходно дефинисан сервер на одговарајућем порту.
   - Датотека *server.js* — дефинише јавни интерфејс микросервиса.
2. Рутери (у директоријуму *routers*) — повезују добијене *HTTP* захтеве са компонентама које су одговорне за обраду тих захтева.
3. Контролери (у директоријуму *controllers*) — имплементирају бизнис логику микросервиса тако што обрађују захтеве и делегирају операције доменским моделима.
4. Описи и методе за иницијализацију модела (у директоријуму *models*) — дефинишу поља модела и имплементирају методе које учитавају и парсирају податке из базе података.

5. Конфигурација система за управљање базом података (у директоријуму *services*) — конфигурише објектно-релационо мапирање и путању до фајла у ком се налази база података.

Сервис *Movies*

Сервис *Movies*, односно сервис филмова, је сервис задужен за управљање подацима о филмовима и оценама за те филмове. Доменски модел овог сервиса налази се на слици 4.1. Ентитет *Title* садржи податке о филму као што су име, година снимања, број минута и жанр филма, док ентитет *Rating* садржи информације о просечној оцени као и о броју корисника који су дали оцену за филм на сајту *IMDB* [11]. Како је скуп података коришћен у овом раду преузет са поменутог сајта, структура базе података прилагођена је доступним подацима.



Слика 4.1: Доменски модел сервиса *Movies*

Сервис пружа јавни интерфејс помоћу ког је могуће добити информације о свим филмовима, о специфичном филму на основу његовог идентификатора као и добити листу филмова на основу упита. Такође, могуће је изменити оцену филма као и обрисати нежељени филм из базе података. Одговори које интерфејс пружа представљени су у формату *JSON* (енг. *JavaScript Object Notation*). Детаљнији опис интерфејса приказан је у табели 4.1.

Табела 4.1: Опис јавног интерфејса сервиса *Movies*

Опис	*Параметри*	*Одговор при успешној обради*
GET api/v1/version		
Дохватање актуелне верзије интерфејса		*version* — тренутна верзија интерфејса
GET api/v1/movies		
Дохватање информација о свим доступним филмовима	*page* — број странице резултата (опционо) *size* — број филмова на страници (опционо)	*data* — објекат који садржи листу филмова са оценама *meta* — објекат који садржи информације о доступним подацима
GET api/v1/movies/:id		
Дохватање информација о специфичном филму	*id* — идентификатор филма (обавезно)	*data* — објекат који садржи информације о специфичном филму
PUT api/v1/movies/:id		
Измена информација о оцени филма	*id* — идентификатор филма (обавезно) Тело захтева (*json*): *rating* — нова оцена филма (обавезно)	*data* — објекат који садржи информације о специфичном филму са измењеном оценом
DELETE api/v1/movies/:id		
Брисање филма из базе података	*id* — идентификатор филма (обавезно)	објекат који садржи информације о броју обрисаних филмова
GET api/v1/movies/search		
Претрага филмова на основу имена филма	*page* — број странице резултата (опционо) *size* — број филмова на страници (опционо) *query* — кључна реч за претрагу (обавезно)	*data* — објекат који садржи листу филмова *meta* — објекат који садржи информације о доступним подацима
GET api/v1/ratings		
Дохватање свих оцена филмова	*page* — број странице резултата (опционо) *size* — број филмова на страници (опционо)	*data* — објекат који садржи листу о оцена филмова са њиховим насловом *meta* — објекат који садржи информације о доступним подацима

Сервис *Books*

Сервис *Books*, односно сервис књига, је сервис задужен за управљање подацима о популарним књигама. Сервис омогућава претрагу популарних наслова, као и добијање информација о аутору, оцени наслова и других информација о књигама. Доменски модел овог сервиса налази се на слици 4.2. Сервис пружа програмски дефинисан интерфејс који омогућава излиставање целокупне листе књига које се налазе у бази података, као и претрагу по насловима. Интерфејс такође омогућава измену оцене доступне књиге као и креирање нове или брисање постојеће књиге из базе података.

Сервис књига има врло сличан јавни интерфејс као и сервис филмова, али због разлика у доменима и количини података са којима располажу, овај сервис у склопу исте табеле садржи информације о књигама и информације о оценама тих књига. Одговори које интерфејс пружа представљени су у формату *JSON*. Подаци коришћени у склопу овог рада преузети су са сајта *Goodreads* [10]. Детаљан опис функционалности које микросервис пружа може се видети у табели 4.2.

Слика 4.2: Доменски модел сервиса *Books*

Табела 4.2: Опис јавног интерфејса сервиса *Books*

Опис	*Параметри*	*Одговор при успешној обради*
GET api/v1/version		
Дохватање актуелне верзије интерфејса		*version* — тренутна верзија интерфејса
GET api/v1/books/		
Дохватање информација о свим доступним књигама	*page* — број странице резултата (опционо) *size* — број књига на страници (опционо)	*data* — објекат који садржи листу књига *meta* — објекат који садржи информације о доступним подацима
PUT api/v1/books/:id		
Измена инфомација о оцени књиге	*id* — идентификатор књиге (обавезно) Тело захтева: *rating* — нова оцена књиге	информације о књизи са измењеном оценом
DELETE api/v1/books/:id		
Брисање информација о књизи из базе података	id — идентификатор књиге (обавезно)	број обрисаних књига из базе података
POST api/v1/books/:id		
Креирање информација о новој књизи	*id* — идентификатор књиге (обавезно) Тело захтева: *title* — наслов књиге (обавезно) *authors* — аутори књиге (обавезно) *isbn* — међународни стандардни број књиге (обавезно поље) *publisher* — издавачка кућа (обавезно)	информације о новој књизи
GET api/v1/books/search		
Претрага књига на основу наслова књиге	*page* — број странице резултата (опционо) *size* — број књига на страници (опционо) *query* — кључна реч за претрагу (обавезно)	*data* — објекат који садржи листу књига *meta* — објекат који саджи мета податке о листи

Клијентска апликација

У склопу пројекта направљена је клијентска апликација која користи фукционалности које имплементирани микросервиси пружају. За имплементацију коришћена је библиотека *React* као и помоћна библиотека *React router* [24]. Апликација је подељена на компоненте које представљају изоловане целине система и у себи садрже логику и визуелни приказ дела апликације.

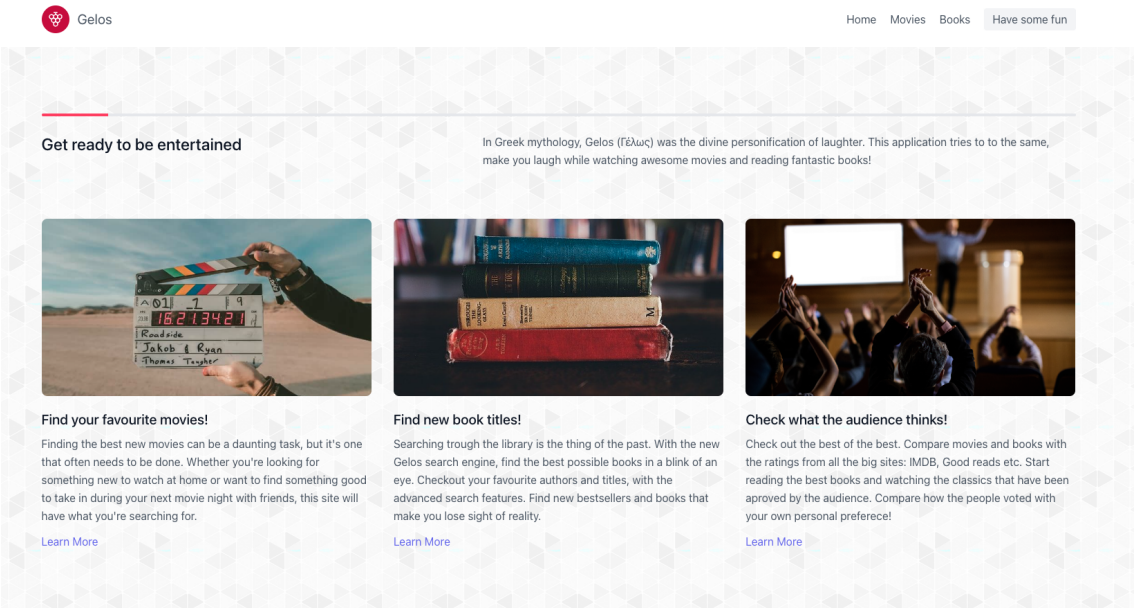
Централни део хијерархијске структуре апликације имплементиран је у компоненти *App*. Она сваку компоненту приказује унутар компоненте *Layout* која дефинише распоред компоненти на страници, и изнад и испод сваке странице приказује компоненту *Header* и *Footer* респективно. *Header* имплементира заглавље које садржи мени за навигацију, док *Footer* представља подножје странице. Коришћењем библиотеке *React router*, компонента *App* приказује одговарајућу страницу у зависности од *URL* путање. Уколико је путања неисправна биће приказана компонента *Page404* која корисника обавештава о неисправно унешеној путањи.

Странице интегришу више компоненти и имплементирају комуникацију са серверским делом апликације. Страница *Homepage* представља полазну тачку клијентске апликације и приказује више презентационих картица које користе податке дефинисане у *contents* директоријуму. Навигацијом кроз презентационе картице или кроз горњи мени за навигацију корисник се преусмерава на неку од страница.

Страница *MoviesPage* омогућава кориснику презентацију као и претрагу оцењених филмова. Компонента *MoviesPage* користи јавни интерфејс *Movies* сервиса и у складу са корисничким уносом података приказује листу филмова односно компоненти *Movie*. У склопу ове странице имплементирано је и приказивање вишестраничних резултата упита кроз компоненту *Pagination*. Аналогно, страница *BooksPage* приказује листу књига и омогућава претрагу књига по њиховом наслову користећи јавни интерфејс *Books* сервиса. Обе компоненте користе компоненту *SearchBar* која регулише унос наслова за претрагу. Празан унос у поље за претрагу кориснику враћа иницијалан скуп података.

Структура директоријума клијентске апликације осликава функционалност сваке од компоненти апликације. Компоненте које представљају странице апликације налазе се у директоријуму *pages* док се компоненте које представљају делове страница налазе у директоријуму *components*. Искључ-

чиво презентационе компоненте попут *Loading* или *Icons* компоненти налазе се у директоријуму *ui*. Поред конфигурационих датотека у склопу апликације се налазе и систем за тестирање апликације као и систем за пресретање *API* позива о којима ће бити речи у наредном поглављу. Графички приказ клијентске апликације приказан је на слици 4.3.



Слика 4.3: Клијентска апликација *Gelos*

4.3 Аутоматско тестирање апликације

Систем за тестирање апликације имплементиран је на више нивоа, па су у слопу пројекта дефинисани јединични, интеграциони и системски тестови. Свака врста тестова доприноси поузданости целокупне апликације и омогућава лако налажење проблема при развоју нових или изменама постојећих функционалности.

Јединичним тестовима проверена је исправност сваке функционалне компоненте клијентске апликације у изолацији, уз помоћ система за пресретање *API* позива. Странице апликације као и мање компоненте које чине те странице покривене су тестовима па је укупно имплементирано четрдесет пет јединичних тестова.

Интеграција клијентске апликације и микросервиса проверава се тестовима дефинисаним у директоријуму *integration*. Тестови су груписани у два скупа у зависности од интеграције коју тестирају, а свака група се састоји од осам различитих тестова, па је укупно имплементирано шеснаест тестова.

Како је потребно тестирати апликацију на начин како је корисник користи, имплементирани су тестови који тестирају целокупан систем. Тестови су дефинисани у директоријуму *e2e* и покривају најчешће случајеве употребе. Имплементирано је двадесет два теста који се извршавају на три различита претраживача. Тестови су подељени у пет група у зависности од функционалности коју проверавају.

У наставку ове секције биће детаљно приказани имплементациони детаљи аутоматских тестова на различитим нивоима, док ће на крају поглавља бити речи о временским оквирима извршавања тестова.

Јединично тестирање апликације

Како јединично тестирање апликације представља проверу исправности појединачних логичких целина у изолацији потребно је дефинисати те целине и окружење у којем их је могуће тестирати. Свака од *React* компоненти може се посматрати као изолован део клијентске апликације. Тестирање ових компоненти у изолацији могуће је само уз одстрањивање зависности од остатка система. То се постиже коришћењем библиотеке за тестирање *React* апликација и библиотеке *MSW*.

Тестирање сваке компоненте састоји се од генерисања *DOM* стабла за ту компоненту, и провере да ли се компонента приказује и мења у складу са предефинисаним очекивањима. Тестови су ради прегледности названи идентично као и компоненте које тестирају уз наставак *.test*.

У наставку је представљен кôд који тестира компоненту *SearchBar* односно поље за претрагу филмова или књига. Након генерисања *DOM* стабла компоненте (линија 5), проверава се да ли се поље за унос и дугме за претрагу налазе унутар стабла (линије 9 и 10). После уноса текста у поље проверава се да ли је вредност исправно унета у поље (линија 13). На овај начин се тестира функционалност поља за унос без обзира на резултате претраге или друге компоненте које се користе у комбинацији са *SearchBar* компонентом.

```
1  import { fireEvent, render, screen } from "@testing-library/react";
2  import SearchBar from "../components/SearchBar.jsx";
3
4  test("Search bar unit test", async () => {
5    render(<SearchBar></SearchBar>);
6    let searchField = screen.getByPlaceholderText("Search by title");
7    let searchButton = screen.getByRole('button')
8
9    expect(searchField).toBeInTheDocument();
10   expect(searchButton).toBeInTheDocument();
11
12   fireEvent.change(searchField, { target: { value: "Some random query"
         } });
13   expect(searchField.value).toBe("Some random query");
14 });
15 }
```

Пример кода 4.1: Јединично тестирање *SearchBar* компоненте

Аналогно за сваку од јединичних компоненти које има смисла тестирати попут компоненти које су одговорне за приказивање појединачне књиге (*Book*) односно филма (*Movie*), покреће се тест који проверава исправно генерисање *DOM* стабла за предефинисане податаке. Покретањем ових тестова смањује се могућност неисправног приказа компоненти како за исправне тако и за неисправне уносе.

Како су микросервиси независне целине у односу на клијентску апликацију, потребно их је детаљно тестирати. Јединичним тестирањем самих микросервиса олакшава се детектовање грешака унутар система, што у дистрибуираним системима може бити велики изазов. Како се сва пословна логика сервиса налази у контролеру, јединични тестови покривају све функционалности контролера за велики број улазних параметара.

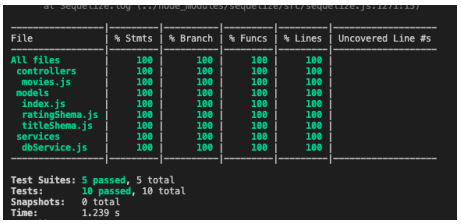
У наставку је приказан кôд који тестира контролер за измену оцене књиге у случају неисправног улаза. Након креирања захтева без обавезног идентификатора књиге (линија 2), који се прослеђује контролеру (линија 9), проверава се да ли контролер враћа одговарајући статусни кôд за грешку (линија 11).

Оба микросервиса детаљно су покривена јединичним тестовима. Након извршавања јединичних тестова приказује се проценат покривености линија

кода који је приказан на слици 4.4.

```
1  test("Edit handler - invalid request", async () => {
2      const req = httpMocks.createRequest({
3        method: "PUT",
4        body: {
5          rating: 4.6,
6        },
7      });
8      const res = httpMocks.createResponse();
9      await BooksController.editRating(req, res);
10     let resStatus = res._getStatusCode();
11     expect(resStatus).toBe(500);
```

Пример кода 4.2: Јединично тестирање контролера за измену оцене књиге — неисправан улаз



Слика 4.4: Извештај о покривености микросервиса јединичним тестовима

„Лажни” подаци

Неке од *React* компоненти при покретању користе више мањих компоненти као и податке које пружају микросервиси. Циљ је тестирати овакве компоненте независно од *backend* система.

Приликом покретања јединичних тестова заустављају се *API* позиви упућени ка микросервисима и враћају се „лажни” одговори који се користе при проверама унутар самог теста. Коришћењем овог система, странице *BookPage* и *MoviePage* се тестирају у изолацији без обзира на то да ли микросервиси исправно функционишу јер користе предефинисан скуп података.

Уколико приликом покретања клијентског дела апликације промењљива окружења *MOCK_DATA* има позитивну вредност, сви позиви ка микросервисима биће пресретнути од стране поменутог система и користиће се подаци

дефинисани у датотеци *mockData.json*. У склопу директоријума *mocks* налазе се следеће датотеке:

- *server.js* — иницијализује систем за пресретање позива у тестовима,
- *handlers.js* — дефинише који „лажни" подаци ће се вратити за сваку од дефинисаних путања,
- *mockData.json* — скуп „лажних" података,
- *browser.js* — иницијализује систем за пресретање позива при покретању клијентске апликације.

Тестирање интеграција апликације

Интеграција између микросервиса и клијентске апликације одвија се коришћењем *HTTP* протокола. Помоћу библиотеке *Playwright* имплементирани су тестови који шаљу *API* позиве ка јавним интерфејсима микросервиса и проверавају да ли су добијени резултати у складу са очекивањима клијентског дела апликације. У зависности од прослеђених параметара тестирају се различити очекивани резултати.

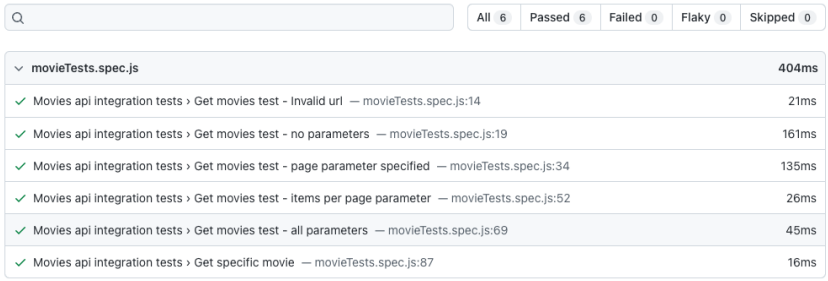
У наставку је представљен један тест који тестира интеграцију између клијентске апликације и сервиса *Movies*, тако што проверава да ли приликом узимања скупа филмова без прослеђених параметара клијент добија податке у очекиваном формату. Након слања *API* позива сервису (линија 2), проверава се статус самог одговора (линија 3) и очекиване вредности за мета параметре (линија 5). Затим се проверавају типови добијених вредности за сваки од филмова, као и да ли су подаци исправни (линија 10) , односно у очекиваном опсегу вредности (линије 11 и 12). Овим проверама се значајно повећава сигурност да ће подаци који су добијени од стране микросервиса бити исправно приказани на клијентској страни апликације.

```
1  test("Get movies test - no parameters", async ({ request }) => {
2      const response = await request.get(`${baseUrl}/api/v1/movies`);
3      expect(response.status()).toBe(200);
4      const responseBody = JSON.parse(await response.text());
5      verifyMetaValues(responseBody.meta, defaultPage,
           defaultItemsPerPage);
6
7      let movies = responseBody.data.movies;
8      expect(movies.length).toBe(defaultItemsPerPage);
9      movies.forEach((movie) => {
10       verifyMovieTypes(movie);
11       verifyMovieLimits(movie);
12       verifyMovieRatingLimits(movie.Rating);
13     });
14   });
```

Пример кода 4.3: Тест интеграције сервиса *Movies* и клијентске апликације

Тестови су груписани у складу са интеграцијом коју тестирају, па се тестови који проверавају конекцију између клијентске апликације и сервиса *Movies* односно *Books* налазе у датотеци *moviesTests* и *booksTests* респективно. Сваки скуп тестова се може независно покретати, што је корисно у случају да је потребно тестирати утицај промена на само једну интеграцију. Директоријум *helpers* садржи помоћне функције за верификацију типова и вредности података који се тестирају.

Након извршавања тестова интеграције генерише се веб страница са извештајем који приказује колико је тестова успешно или неуспешно извршено. За сваки од тестова могуће је видети на ком је кораку пао, па се јасно може утврдити место у систему у ком се налази грешка. Пример извештаја након извршавања интеграционих тестова може се видети на слици 4.5.



Слика 4.5: Извештај након извршавања интеграционих тестова за сервис *Movies*

Тестови система

Комплексност микросервисне архитектуре лежи у чињеници да се систем састоји од много независних компоненти које треба кохезивно да функционишу. Системско тестирање микросервисне апликације подразумева проверу да ли целокупан систем испуњава претходно дефинисане захтеве.

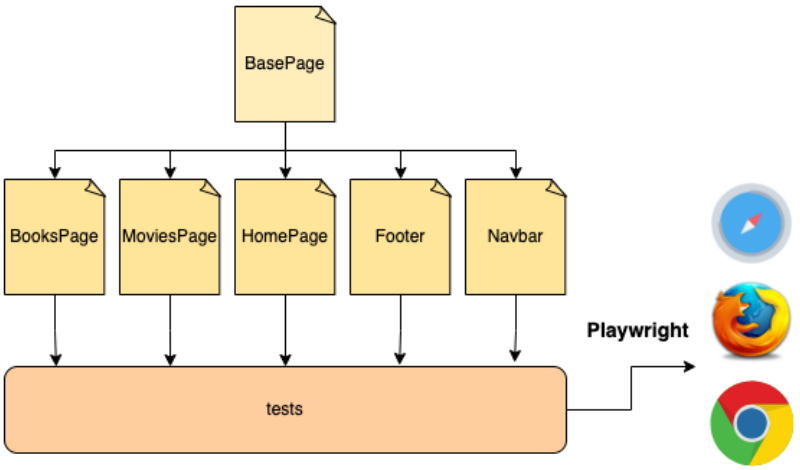
Развојни оквир *Playwright* омогућава аутоматско покретање претраживача и тестирање апликације из угла корисника. На овај начин могуће је тестирати различите случајеве употребе система на брз и поновљив начин. Случајеви употребе који нису међусобно зависни могу се извршавати паралелно и на тај начин додатно убрзати процес системског тестирања, које важи за најспорију врсту тестирања.

Архитектурално решење коришћено за имплементацију системских тестова је модел странице као објекта. Класе које дефинишу елементе и методе на свакој од страница налазе се у директоријуму *pages*.

Страница *BasePage* представља базну класу за све друге странице и садржи опис елемената који се налазе на свакој страници апликације као и методе које управљају тим елементима. Свака од страница у конструктору садржи селекторе који описују елементе странице апликације. Странице *BookPage*, *MoviePage* и *HomePage* имплементирају интерфејсе за управљање главним страницама апликације, док су странице *Navbar* и *Footer* одговорне за управљање доњим и горњим менијем апликације респективно.

Тестови се налазе у директоријуму *e2e* и користећи интерфејс који пружају описане странице апликације тестирају функционалности система. Група тестова имплементирана је у једној датотеци. Тестови из исте групе извршавају се секвенцијално док се тестови из различитих група извршавају паралелно.

Максималан број паралелних процеса као и друге конфигурационе променљиве дефинисане су у датотеци *playwright.config*. Детаљан приказ архитектуре коришћене за имплементацију налази се на слици 4.6.



Слика 4.6: Архитектурално решење системских тестова

У наставку је представљен тест који проверава случај употребе претраге књига где је очекиван резултат претраге цела страница резултата. Коришћењем дефинисаних интерфејса страница проверава се очекиван број резултата на страници (линија 6) као и укупан очекиван број резултата (линија 7).

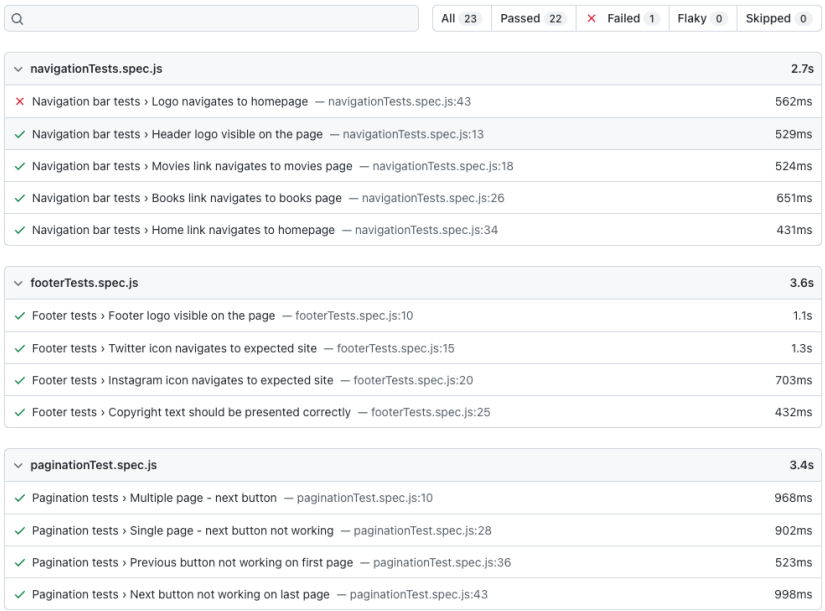
```
1  test("Valid search query - full page result", async ({ page }) => {
2      const booksPage = new BooksPage(page);
3      let numOfMovies = await booksPage.getNumOfBooksOnPage();
4      let totalNumOfResults = await booksPage.getTotalNumOfBooks();
5      await booksPage.search("Harry");
6      await booksPage.verifyNumOfBooksOnPageEquals(numOfMovies);
7      await booksPage.verifyTotalNumOfBooksLessThen(totalNumOfResults);
8   });
```

Пример кода 4.4: Тест случаја употребе — претрага књига

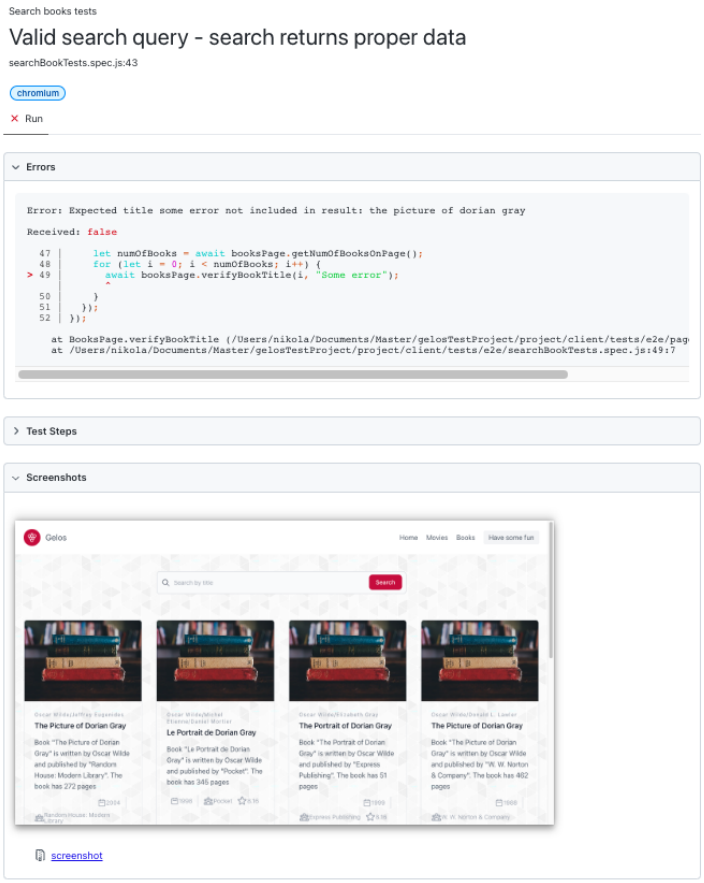
Системским тестовима покривени су најчешћи случајеви употребе апликације из угла корисника. Тестови су подељени у следеће групе:

- *footerTests* — група тестова за проверу функционалности заглавља странице,
- *navigationTests* — група тестова за проверу функционалности горњег навигационог менија,
- *paginationTest* — група тестова за проверу функционалности страничења на свим страницама које приказују више података,
- *searchBookTests* — група тестова за проверу функционалности претраге књига,
- *searchMovieTests* — група тестова за проверу функционалности претраге филмова.

Након извршавања групе системских тестова генерише се извештај у форми веб странице где је приказан проценат успешности тестова. Приказ извештаја након једног извршавања скупа тестова приказан је на слици 4.7. Навигацијом кроз извештај могуће је видети који случај употребе и из ког разлога није успешно извршен. Извештај такође садржи и снимак екрана у оном тренутку када је тест неуспешно завршен што се може видети на слици 4.8.



Слика 4.7: Пример извештаја приликом извршавања скупа системских тестова



Слика 4.8: Пример извештаја за неуспешно завршен тест

Временски оквири извршавања тестова

Табела 4.3 приказује временске оквире извршавања тестова за сваку од врста тестова. Друга колона представља укупан број извршених тестова одређеног типа. Укупно време за паралелно извршавање свих тестова одређеног типа представљено је у трећој колони док четврта колона садржи информације о времену извршавања једног теста. Системски тестови су подељени на три дела, у зависности који претраживач користе при покретању тестова.

На основу табеле видимо да јединични тестови представљају најбрже и најстабилније тестове. Како не користе мрежне позиве, максимално и минимално време извршавања тестова се не разликује много. Услед непостојања зависности између тестова, тестови се извршавају паралелно, па су разлике између трајања извршавања једног теста и целог скупа тестова занемарљиве.

Табела 4.3: Временски оквири извршавања тестова

Врста тестова	**Број тестова**	**Паралелно извршавање целог скупа**	**Извршавање једног теста**
Јединични тестови - клијентска апликација	45	0.8-1 s	0.8 s
Јединични тестови - микросервис *Movies*	10	1 s	0.4 s
Јединични тестови - микросервис *Books*	10	1 s	0.4 s
Интеграциони тестови	16	0.4-2 s	0.4 s
Системски тестови — *Chrome*	23	7-9 s	1 s
Системски тестови — *Firefox*	23	7-8 s	1 s
Системски тестови — *Safari*	23	7-11 s	1 s

Интеграциони тестови користе мрежне позиве па брзина извршавања може варирати у односу на брзину одговора одговарајућег сервиса. Ипак, у идеалним условима, ова врста тестова се може извршавати подједнако брзо као и јединични тестови, што се може приметити у приказаној табели.

Системски тестови представљају најспорију и најкомплекснију врсту тестова. Како сваки тест покреће претраживач при извршавању, време извршавања може зависити од врсте претраживача. Иако су разлике у приказаном примеру минималне, за велики број тестова оне могу бити пресудне при одлуци за који претраживач треба оптимизовати апликацију. Како су ови тестови скупи за извршавање треба бити опрезан у одлуци које случајеве употребе аутоматизовати. Развојни оквир *Playwright* омогућава паралелно извршавање системских тестова па се овај скуп може проширити уколико се повећа процесорска моћ, одосно број нити при покретању тестова.

Глава 5

Закључак

Овај рад обрадио је основне принципе имплементације и аутоматског тестирања микросервисне апликације. Пример имплементације микросервиса показан је кроз конкретну имлементацију два независна микросервиса. Опис коришћења микросервисне архитектуре употпуњен је кроз имплементацију клијентске апликације која користи оба сервиса. Кроз конкретне примере, представљен је начин тестирања овакве апликације. Имплементиран је систем који тестира апликацију на више нивоа па су тако дизајнирани системи за јединично, интеграционо и системско тестирање.

Рад је кроз опис микросервисне архитектуре обрадио специфичности архитектуре као и начине на који се те специфичности могу искористити при тестирању. Анализом различитих врста тестова и њиховом имплементацијом, рад је приказао како могућности аутоматског тестирања као и важност тестирања на сваком од поменутих нивоа.

Кроз рад је представљено структуирање тестова на скалабилан начин, као и брзина извршавања сваке од врсте тестова. На примеру јединичних тестова приказана је важност „лажних" података, док је кроз имплементацију система за системско тестирање апликације, представљена архитектура Објектни модел странице. Обрађене су све особине ових технологија и техника као и њихова важност при провери квалитета микросервисних апликација.

Могућа су разна унапређења апликације, како на нивоу проширења функционалности, тако и додавањем нових врста тестова. Важно унапређење представља имплементација сервиса задуженог за креирање и одржавање корисничких налога који би комуницирао са оба постојећа сервиса. Поред тога, систем за аутоматско тестирање апликације може се унапредити како додат-

ним тестовима тако и тестовима перформанси. Интеграцијом модерних алата за генерисање извештаја попут алата *Allure repoter* [12], прегледност резултата тестова може бити на још већем нивоу. Коначно, аутоматизација самог развоја апликације кроз систем за непрекиднуну испоруку *Jenkins* [13], може допринети већој поузданости система и знатно бржем развоју.

Библиографија

[1] Apple. Webkit, 2021. on-line at: https://webkit.org/.

[2] Pablo Calvo. WebDriver vs Chrome DevTools (Speed Test), 2021. on-line at: https://dev.to/pjcalvo/webdriver-vs-chrome-devtools-speed-test-441h.

[3] MDN Contributors. Express/Node introduction, 2022. on-line at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

[4] Sequelize Contributors. Sequelize.js. on-line at: https://sequelize.org/docs/v6/.

[5] Nikola Dimić. Gelos test project, 2022. on-line at: https://github.com/d1mic/gelosTestProject.

[6] Kent C. Dodds. React testing library introduction, 2022. on-line at: https://testing-library.com/docs/react-testing-library/intro/.

[7] International Software Testing Qualifications Board. Foundation. Certified Tester Foundation Level (CTFL) Sylabus. on-line at: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf.

[8] OpenJS Foundation. About Node.js. on-line at: https://nodejs.org/en/about/.

[9] Ganesh Hegde. Playwright Framework Tutorial: Learn Basics and Setup, 2022. on-line at: https://www.browserstack.com/guide/playwright-tutorial.

[10] Goodreads Inc. Goodreads, 2022. on-line at: `https://www.goodreads.com/api`.

[11] Imdb Inc. IMDB, 2022. on-line at: `https://www.imdb.com/interfaces/`.

[12] WebDriver IO. Allure, 2022. on-line at: `https://webdriver.io/docs/allure-reporter/`.

[13] Kohsuke Kawaguchi. Jenkins, 2022. on-line at: `https://www.jenkins.io/`.

[14] Tailwind Labs. Tailwind - Utility-First Fundamentals, 2022. on-line at: `https://tailwindcss.com/docs/utility-first`.

[15] Mia Liang. Object - Relational Mapping, 2021. on-line at: `https://www.altexsoft.com/blog/object-relational-mapping/`.

[16] Jenet Gregory Lisa Crispin. *Agile Testing: A Practical Guide for Testers and Agile Teams*. O'Reilly Media, Inc., 2009.

[17] Microsoft. Playwright - Getting started, 2022. on-line at: `https://playwright.dev/`.

[18] Blake Mizerany. About Sinatra. on-line at: `http://sinatrarb.com/about.html/`.

[19] Sam Newman. *Building Microservices, First edition*. O'Reilly Media, Inc., 2015.

[20] Alexander Noel. Guide to monorepos. on-line at: `https://www.toptal.com/front-end/guide-to-monorepos`.

[21] Node package manager team. About npm. on-line at: `https://docs.npmjs.com/about-npm`.

[22] Meta Platforms. React - A JavaScript library for building user interfaces, 2022. on-line at: `https://reactjs.org/`.

[23] SQLite project. About SQLite. on-line at: `https://www.sqlite.org/about.html`.

[24] Remix. React router - main concepts, 2022. on-line at: `https://reactrouter.com/docs/en/v6/getting-started/concepts`.

[25] IBM StrongLoop. Express.js. on-line at: `https://expressjs.com/`.

[26] Bootstrap team. Bootstrap, 2022. on-line at: `https://getbootstrap.com/`.

[27] Facebook Open Source team. Jest, 2022. on-line at: `https://jestjs.io/`.

[28] Google Chrome team. Chromium, 2021. on-line at: `https://www.chromium.org/Home/`.

[29] Garima Tiwari. Playwright vs Selenium: A Comparison, 2021. on-line at: `https://www.browserstack.com/guide/playwright-vs-selenium`.

[30] SQLite tutorial team. What Is SQLite, 2022. on-line at: `https://www.sqlitetutorial.net/what-is-sqlite/`.

[31] W3Schools. Node.js intro. on-line at: `https://www.w3schools.com/nodejs/nodejs_intro.asp`.

[32] Artem Zakharchenko. Mock Service Worker - API mocking of the next generation, 2022. on-line at: `https://mswjs.io/docs/`.

Биографија аутора

Никола Димић рођен је 21.11.1995. у Београду, где је са одличним успехом завршио основну школу и природни смер у Шестој београдској гимназији. Смер Информатика на Математичком факултету Универзитета у Београду уписао је 2015. године, а завршио у септембру 2019. године са просечном оценом 8.8. Након завршених основних студија, уписао је мастер студије информатике на истом факултету.

У октобру 2015. године запошљава се у компанији *Ninety Apples* где ради на мануелном и аутоматском тестирању веб апликација за куповину. Кроз организацију *Toptal*, у октобру 2018. године почиње да ради за компанију *Deckers Brands*, где ради на развијању оквира за аутоматско тестирање и као део тима осваја награду за најбољу нову имплементацију на конференцији *SauceCon* 2019. године. У априлу 2022. године почиње да ради као консултант за аутоматско тестирање у компанији *Ezderm*. Тренутно ради на развијању оквира за аутоматско тестирање система за дерматолошке прегледе и медицинска осигурања, као и на усавршавању знања у областима тестирања и развоја софтвера.