УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Вероника Маринковић

РАЗВОЈ ВОЂЕН ПОНАШАЊЕМ КРОЗ ПРИМЕНУ АЛАТА CUCUMBER

мастер рад

Београд, 2025.

Ментор:

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

Чланови комисије:

др Данијела Симић, доцент
Универзитет у Београду, Математички факултет

Иван Ристовић, асистент
Универзитет у Београду, Математички факултет

Датум одбране: октобар 2025.

Мами, тати, Филипу и Николи

Наслов мастер рада: Развој вођен понашањем кроз примену алата *Cucumber*

Резиме: Рад је посвећен анализи и примени методологије развоја вођеног понашањем. У циљу илустрације основних концепата развоја вођеног понашањем креирана је микросервисна апликација за планирање оброка коришћењем алата *Cucumber* и синтаксе *Gherkin* за дефинисање аутоматизованих тестова. Кроз ову апликацију демонстрира се како развој вођен понашањем омогућава јасну дефиницију пословних захтева, побољшава комуникацију између развојних и пословних тимова и доприноси унапређењу квалитета и поузданости софтверских система.

Кључне речи: развој вођен понашањем, аутоматско тестирање, микросервисна архитектура, алат *Cucumber*, верификација софтвера

Садржај

Глава 1

Увод

Развој вођен понашањем представља приступ развоју софтвера који наглашава рану дефиницију очекиваног понашања система кроз формалне пословне сценарије, омогућавајући јасну комуникацију између развојних, тестних и пословних тимова. Циљ овог рада је истраживање и примена ове методологије на практичном примеру микросервисне апликације за планирање оброка. Тестови су креирани у облику пословних сценарија, коришћењем алата *Cucumber* и синтаксе *Gherkin*, што омогућава проверу функционалности и усклађеност софтвера са дефинисаним пословним захтевима. Кроз ову апликацију демонстрирају се основни концепти развоја вођеног понашањем, као и његова примена у контексту микросервисне архитектуре, при чему је посебан акценат стављен на побољшање поузданости система, осигурање квалитета кода и ефикасан развојни циклус.

Глава 2 представља традиционалне методе развоја софтвера, затим развој вођен тестовима, а посебан акценат је стављен на развој вођен понашањем. Објашњени су основни концепти развоја вођеног понашањем, његове предности у односу на друге методологије, као и потенцијалне мане и ограничења у реалним развојним окружењима. Глава 3 посвећена је алату *Cucumber*. Описане су основне функционалности алата, синтакса *Gherkin* за дефинисање сценарија и начини на које *Cucumber* олакшава креирање тестова у складу са пословним захтевима. Глава 4 приказује технологије коришћене приликом имплементације апликације. Посебан акценат је стављен на микросервисну архитектуру и Јава екосистем. Глава 5 описује саму апликацију, њену поделу по микросервисима и начин комуникације између њих. Такође су описани случајеви употребе и функционалности доступне у оквиру апликације. Глава 6 анализира примену

развоја вођеног понашањем на развој апликације. Приказани су извештаји *Cucumber* тестова као и извештаји о покривености кода апликације. Рад се завршава главом 7 у којој су сумирани главни доприноси рада, закључци проучених приступа и препоруке за даља унапређења.

Глава 2

Развој вођен понашањем

Ова глава посвећена је различитим приступима у развоју софтвера, са посебним фокусом на развој вођен понашањем (енгл. *Behavior-Driven Development*, скраћено *BDD*). У почетку ће бити представљене традиционалне методе развоја софтвера које су поставиле темеље за данашње праксе, као и развој вођен тестовима (енгл. *Test-Driven Development*, скраћено *TDD*) који је унео значајан помак у развоју софтвера интеграцијом тестирања у процес израде кода. Након тога, посебна пажња биће посвећена развоју вођеном понашањем, који ставља нагласак на сарадњу између различитих актера у развојном процесу и на прецизно дефинисање понашања система кроз формализоване и јасно описане сценарије. Овај приступ доприноси унапређењу комуникације између пословних и техничких тимова и побољшању квалитета софтверског производа.

2.1 Традиционалне методе развоја софтвера

Животни циклус развоја софтвера (енгл. *Software Development Life Cycle*, односно скраћено *SDLC*) [20] представља процес који се примењује током развоја софтверских апликација и обухвата низ планираних активности и фаза. Овај процес омогућава тимовима и организацијама да систематично управљају свим аспектима развоја и адекватно их документују, што доприноси смањењу ризика и оптимизацији расположивих ресурса.

Начин на који се овај процес реализује може знатно да варира у зависности од изабране методологије развоја, што директно утиче и на положај и улогу тестирања у оквиру самог циклуса. У наставку је представљено неколико

основних модела развоја софтвера и анализирана је функција и значај процеса тестирања у сваком од њих.

2.1.1 Модел водопада

Модел водопада [35] представља један од најстаријих приступа у развоју софтвера. Као традиционални модел, он је доминирао у пројектима где су захтеви били стабилни и јасно дефинисани од почетка. Назив „водопад“ описује линеарни и секвенцијални ток фаза развоја, где свака фаза следи након претходне без повратка уназад. Овакав приступ захтева да се свака фаза потпуно заврши пре него што се пређе на следећу, што олакшава управљање пројектом, али истовремено ограничава флексибилност уношења промена током процеса. Уколико се касније уоче потребе за изменама, враћање на раније фазе је компликовано, скупо и понекад технички изазовно.

Основне фазе у моделу водопада су: анализа захтева, дизајн, имплементација, тестирање, испоручивање и одржавање. Анализа захтева подразумева дефинисање јасних и детаљних захтева пре почетка развоја. Након тога следи дизајн, у оквиру ког се планира архитектура софтвера и креирају решења на основу претходно дефинисаних захтева. Фаза имплементације обухвата кодирање и конфигурацију софтверских компоненти у складу са техничким спецификацијама. Тестирање се спроводи у циљу верификације и валидације функционалности како би се осигурало да софтвер испуњава све постављене захтеве. Испоручивање подразумева пуштање софтвера у рад у продукционом окружењу. Након тога следи одржавање, које укључује техничку подршку, корекцију грешака и периодично ажурирање софтвера у употреби.

Иако је овај модел данас мање заступљен у чистом облику, он и даље налази примену у специфичним областима као што су развој уређаја уграђених у хардвер, финансијски системи са строгом регулативом, или пројекти у којима су сви захтеви унапред познати и нису предмет промена.

Предности модела водопада огледају се у његовој једноставности и предвидљивости. Јасно дефинисане фазе олакшавају планирање и расподелу ресурса, као и управљање пројектом. Захваљујући стабилности захтева током процеса, тимови могу да се фокусирају на квалитетну реализацију дефинисаних циљева без потребе за фреквентним изменама.

Међутим, главна слабост овог модела је ограничена флексибилност. У случајевима када се током развоја јаве промене у захтевима или када је потребно

брзо реаговати на нове информације, овај модел постаје непрактичан. Враћање и модификовање већ реализованих фаза је често скуп процес, који може изазвати кашњења и повећати трошкове. Поред тога, недостатак континуиране интеракције са корисницима може резултирати софтвером који не одговара реалним потребама. Такође, унос промена у сложене системе може довести до нежељених последица и ланчане реакције грешака због међузависности компоненти.

2.1.2 *V* модел развоја софтвера

V модел [34] представља проширење класичног модела водопада, са строго дефинисаним фазама развоја и тестирања које су унапред повезане. Карактеристично за овај модел је структура у облику слова „*V*“, где лева страна представља фазе анализе, дизајна и имплементације, а десна страна фазе тестирања које одговарају свакој појединачној фази развоја. Оваква организација омогућава систематичан и дисциплинован приступ у оквиру животног циклуса развоја софтвера, јер се свака фаза развоја завршава пре почетка наредне и праћена је одговарајућом фазом верификације.

На самом почетку пројекта, у сарадњи са клијентом, врши се прикупљање и документација свих системских захтева. Након тога следи анализа захтева која омогућава детаљно разумевање пословне логике и поставља основу за даљи дизајн. У почетној фази дизајна осмишљава се укупна структура система, укључујући поделу на модуле, дефинисање интерфејса и повезивање компоненти. Затим се у фази детаљног дизајна разрађују унутрашње структуре појединачних модула, што омогућава припрему за фазу имплементације.

Након завршетка развојних активности, приступа се низу тестирања који понављају логичку путању иницијалног дизајна. Прво се спроводе тестови компоненти, којима се проверава исправност појединачних модула у складу са детаљним дизајном. Следи интеграционо тестирање, усмерено на проверу међусобне сарадње компоненти. Након тога врши се валидација целокупног система, како би се утврдило да ли крајњи производ одговара почетним захтевима и спецификацијама. На крају, обавља се прихватно тестирање у реалном окружењу, у сарадњи са клијентом, пре званичне предаје система и његовог пуштања у продукцију.

V модел је посебно погодан за пројекте у којима су захтеви јасно дефинисани, стабилни и непроменљиви. Овај приступ највише долази до изражаја

у областима где је потребно стриктно поштовање стандарда, као што су ваздухопловство, медицина или аутомобилска индустрија. Једна од највећих предности овог модела је што омогућава рано откривање грешака, јер се сценарији за тестирање дефинишу паралелно са фазама развоја. Ипак, због своје круте структуре, модел није прилагодљив променама што га чини мање погодним за динамичне пројекте у којима се захтеви често мењају.

2.1.3 Спирални модел развоја софтвера

Спирални модел [33] је модел чија је главна карактеристика итеративна природа, где се пројекат развија кроз више циклуса (спирала), а свака спирала обухвата различите фазе од којих свака почиње дефинисањем циља дизајна, а завршава се прегледом напретка од стране клијента, чиме се омогућава стална повратна информација и прилагођавање развоја.

Развојни процес у оквиру животног циклуса софтвера у спиралном моделу почиње са дефинисањем ограниченог скупа захтева, који се обрађују кроз појединачне итерације обухватајући све фазе развоја. Свака итерација састоји се из четири основне фазе. Прва је планирање, у којој се процењују трошкови, распоред и ресурси неопходни за текућу фазу рада. Након тога следи анализа ризика, током које се идентификују потенцијални проблеми и развијају стратегије за њихово смањење или избегавање, с циљем минимизирања негативног утицаја на пројекат. Трећа фаза обухвата развој софтвера и тестирање, где се прво врши кодирање, током којег се реализују функционалности описане у захтевима и дизајну система. Паралелно или непосредно након кодирања спроводе се тестови различитих нивоа како би се проверила исправност и стабилност развијеног кода. Након успешног тестирања, следи имплементација, односно постављање готовог софтвера у реално окружење или на циљну платформу, чиме корисници добијају функционални систем. Последња фаза је процена, у којој корисник врши ревизију и преглед резултата, прати се статус ризика као што су кашњења или прекорачење буџета, и на основу добијених повратних информација доноси се одлука о наставку са следећом итерацијом.

Спирални модел се препоручује за велике и сложене пројекте у којима су захтеви првобитно нејасни или подложни честим променама. Овај приступ омогућава флексибилност уношења измена у свакој итерацији, постепено унапређење софтвера и брзо откривање грешака. Посебно је погодан када је потребна израда прототипа и када је континуирана процена ризика од

суштинског значаја. Међутим, овај модел захтева високо оспособљен тим и ефикасну комуникацију са корисницима, те није погодан за мале или једноставне пројекте због сложености планирања и управљања ризицима. Уколико ризици нису адекватно идентификовани и контролисани, ефикасност развоја може значајно опасти.

2.1.4 Агилна методологија

Агилна методологија [32] представља флексибилан и итеративан приступ развоју софтвера, који омогућава брзу испоруку функционалних делова система и прилагођавање промењивим захтевима током процеса развоја. За разлику од традиционалних модела који подразумевају линеаран ток фаза, агилни приступ омогућава тимовима да у кратким временским интервалима, званим *спринтови*, испоручују функционалне и тестиране делове софтвера. Циљ овог приступа је континуирана испорука функционалности крајњем кориснику, као и стално прилагођавање решења у складу са повратним информацијама добијеним након сваке итерације.

Развојни процес у агилној методологији карактерише тесна сарадња између свих чланова тима, укључујући програмере, дизајнере, тестере и клијенте. Акценат је стављен на рану идентификацију приоритета и континуирану комуникацију са клијентом ради што бољег разумевања пословних потреба. Уместо детаљног иницијалног планирања, које је карактеристично за класичне моделе, у агилном приступу планирање се одвија динамично, током читавог трајања пројекта. Свака итерација обухвата анализу потреба, дизајн решења, развој, тестирање и преглед резултата уз активно учешће клијента. Након завршетка спринта, тим спроводи ретроспективу како би идентификовао потенцијална побољшања и унапредио наредни циклус развоја.

Агилна методологија се нарочито показала ефикасном у пројектима који се одвијају у окружењима која се брзо мењају, као и у ситуацијама где иницијални захтеви нису потпуно дефинисани или се очекују значајне измене током развоја. Њена флексибилност, брза испорука и оријентација на сарадњу доприносе бољој адаптацији софтвера стварним потребама корисника. Међутим, оваквом приступу је неопходан високо мотивисан и добро организован тим, као и стална укљученост клијента. Потенцијалне слабости агилне методологије огледају се у мањку документације, отежаној процени ресурса унапред и могућој нестабилности обима пројекта. Ипак, када се примени у одговарајућем

контексту, агилни приступ може значајно повећати ефикасност и квалитет софтверских решења.

Једна од најпознатијих агилних методологија је *Extreme Programming (XP)*, која наглашава квалитет кода, тесну сарадњу чланова тима и итеративну испоруку функционалних делова система. Управо из овог приступа развијене су праксе развој вођен тестовима и развој вођен понашањем, које конкретизују принципе агилног развоја тако што примењују континуирано тестирање кода и описивање понашања система на начин разумљив свим учесницима у развоју.

2.2 Развој вођен тестовима

У савременом развоју софтвера, тежи се ка праксама које омогућавају већу поузданост, лакше одржавање и брже прилагођавање променама. Један од приступа који значајно доприноси овим циљевима је **развој вођен тестовима**. Овај приступ подразумева писање тестова пре саме имплементације функционалности. Основна идеја развоја вођеног тестовима јесте да се развојни процес започне формулисањем очекиваног понашања система у виду тест случајева, што омогућава програмеру да прецизније дефинише захтеве и обезбеди њихову проверу у свакој фази развоја. На овај начин, кôд се пише тек након што постоји тест који ће га одмах по писању проверити, чиме се постиже боља покривеност кода тестовима и већа сигурност у његову исправност. Развој вођен тестовима није само техничка пракса, већ и парадигма размишљања која утиче на структуру, дизајн и квалитет софтвера.

Процес развоја вођеног тестовима прати понављајући циклус познат као *Red-Green-Refactor*.

Црвена фаза (*Red*): Прво, програмер пише тест који дефинише жељену функционалност или понашање. У овој фази тест се неће успешно извршити, па се ова фаза назива „црвена“, јер означава неуспех.

Зелена фаза (*Green*): Након тога, пише се минимална количина кода неопходна да би се тест успешно извршио. Када се тест успешно изврши, фаза се назива „зелена“.

Рефакторисање (*Refactor*): На крају, кôд се рефакторише ради унапређења структуре и квалитета, уз очување исправности тако да се тест и даље успешно извршава.

2.2.1 Приступи развоју вођеном тестовима

Развој вођен тестовима може се спроводити кроз различите приступе, од којих сваки има своју стратегију и фокус у процесу тестирања и развоја. Два најчешћа приступа су „изнутра ка споља“ и „споља ка унутра“, који се разликују по начину на који се приступа тестирању и развоју различитих нивоа система.

Приступ „изнутра ка споља“

У приступу „изнутра ка споља“ (енг. *Inside Out*), развој почиње од најмањих компоненти или основних функционалности. Тестови се прво пишу за компоненте ниског нивоа, као што су класе, методе или појединачне функције. Када се тестови успешно извршавају за све основне јединице кода, развој се наставља ка спољашњим, вишим слојевима система. Овим се обезбеђује да основни делови система буду стабилни пре него што се пређе на интеграцију са осталим деловима.

Овакав приступ омогућава да се исправност основне функционалности утврди пре фокусирања на сложеније делове, чиме се гради чврста унутрашња структура и умањује ризик од грешака у каснијим фазама. Такође, олакшава се проналажење и изолација проблема на нивоу појединачних компоненти, подстиче се рад на малим и управљивим деловима система, а резултат је лакше одржив и боље тестиран кôд, заснован на стабилној основној логици.

Приступ „споља ка унутра“

Приступ „споља ка унутра“ (енг. *Outside In*) има супротан редослед од претходног и почиње од спољашњих интерфејса система, са фокусом на функционалности које су усмерене ка крајњем кориснику. На почетку се пишу тестови за компоненте вишег нивоа као што су кориснички интерфејси, API-ји или системске интеграције. Ови тестови дефинишу очекивано понашање из перспективе крајњег корисника. Када тестови падну, програмери пишу неопходан кôд како би испунили очекивања корисника.

Овакав приступ омогућава рану валидацију функционалности усмерених ка кориснику и осигурава да је систем дизајниран тако да подржи функционалност од почетка до краја. Поред тога, омогућава приоритизацију компоненти које су видљиве кориснику, као и њихових интеграција у раним фазама разво-

ја. Пружа и брзу повратну информацију о понашању система из корисничке перспективе и подстиче изградњу решења заснованих на реалним сценаријима употребе.

2.2.2 Најбоље праксе за развој вођен тестовима

Ефективна примена развоја вођеног тестовима подразумева неколико препорука и смерница које доприносе квалитету и одрживости софтверског решења. Најпре, неопходно је обезбедити јасно разумевање захтева и спецификација пре започињања писања тестова, како би се осигурало да они буду фокусирани и релевантни. Препоручује се да сваки тест буде *атомски*, односно усмерен на појединачну функционалност или понашање, што побољшава читљивост и олакшава одржавање, као и дебаговање. Почетни тестови треба да буду што једноставнији, са циљем да се избегне оптерећење комплексним сценаријима у раној фази развоја. Посебну пажњу треба посветити граничним условима и екстремним ситуацијама, јер они често откривају критичне недостатке у имплементацији. Након што се тестови успешно изврше, препоручује се рефакторисање кода како би се побољшала његова структура, при чему се не мења његово спољашње понашање.

Ефикасан развојни процес подразумева и брзу повратну петљу – тестови треба да се извршавају у кратком временском року, што омогућава брзе итерације и рано откривање грешака. Аутоматизација тестова је такође кључна, јер омогућава њихово учестало покретање, лаку интеграцију у развојни ток и обезбеђује конзистентност резултата. Пожељно је одржавати свеобухватан скуп тестова, који укључује јединичне, интеграционе и тестове прихватања, јер сваки од њих пружа различите нивое поверења у исправност система. На крају, неуспешни тестови треба да представљају полазну основу за даљи развој, будући да пружају вредне повратне информације о постојећим недостацима у имплементацији и усмеравају напоре ка побољшању квалитета софтвера.

2.2.3 Предности и мане развоја вођеног тестовима

Развој вођен тестовима доноси бројне предности у процесу израде софтвера. Једна од најзначајнијих је побољшан квалитет кода, с обзиром на то да се писањем тестова пре саме имплементације подстиче пажљивије размишљање о дизајну и функционалности, што резултира чистијим и поузданијим кодом.

Овај приступ омогућава рано откривање грешака, јер се тестови редовно покрећу у раним фазама развоја, што доводи до правовременог отклањања проблема и смањења трошкова њихове исправке. Поред тога, јединични тестови служе и као врста живе документације, пружајући јасан увид у очекивано понашање система и тиме олакшавају његово одржавање. Континуирано рефакторисање кода, праћено тестовима, чини систем лакше одрживим и проширивим, док истовремено повећава самопоуздање развојног тима у стабилност постојећих функционалности приликом увођења нових промена.

Ипак, овај приступ има и своје недостатке. Један од њих је почетно успоравање развоја, јер писање тестова пре саме имплементације може захтевати додатно време, што може представљати проблем у условима кратких рокова. Такође, уколико тестови нису адекватно написани или не покривају довољан број сценарија, може доћи до лажног осећаја сигурности у исправност система. Још једна значајна мана је и трошак одржавања тестова, јер је неопходно њихово редовно ажурирање при свакој промени функционалности, што може представљати додатно оптерећење у развојном процесу.

2.3 Дефиниција и основни концепти развоја вођеног понашањем

Развој вођен понашањем (скраћено *BDD*) [21] представља софтверску методологију која наглашава сарадњу између свих учесника у развојном процесу — програмера, тестера, бизнис аналитичара и крајњих корисника. Основна идеја *BDD-a* јесте да се захтеви за софтвером дефинишу кроз конкретна понашања система у облику јасних, разумљивих и тестабилних спецификација.

BDD надограђује и унапређује традиционалне приступе развоју вођеном тестовима, тако што ставља јачи акценат на комуникацију и заједничко разумевање између свих укључених страна. Док је развој вођен тестовима углавном усредсређен на техничку проверу кода кроз јединичне тестове, *BDD* се фокусира на понашање система из перспективе корисника или бизниса, што доприноси бољем усаглашавању развојног процеса са пословним циљевима.

Кроз *BDD*, тимови имају могућност да креирају *живу документацију* (енгл. *living documentation*) које се аутоматски ажурирају заједно са кодом и тестовима, чиме се осигурава да су спецификације увек релевантне и прецизне.

Ово омогућава континуирану валидацију и унапређење софтверског производа током читавог животног циклуса.

2.3.1 Кључни концепти и терминологија

Основни концепти у развоју вођеном понашањем обухватају неколико кључних појмова који дефинишу како се захтеви и функционалности система спецификују и тестирају, али и како се о њима међусобно комуницира. Ови концепти чине темеље *BDD* методологије, омогућавајући развој софтвера који је у складу са стварним пословним потребама и лакше постиже договор између свих учесника у развојном процесу.

Понашање (*Behavior*) Понашање представља видљиве реакције и функције система у одређеним условима. Уместо да је фокус на унутрашњој имплементацији, *BDD* инсистира на опису шта систем треба да ради из угла корисника или бизниса. Опис понашања користи се као основа за дефинисање тестова и спецификација.

Сарадња и комуникација Једна од најважнијих вредности *BDD-a* јесте подстицање тесне сарадње између програмера, тестера, пословних аналитичара и других заинтересованих страна. Заједнички језик и формати за спецификацију помажу у елиминисању неспоразума и обезбеђују да сви учесници деле исто разумевање о функционалности.

Језик сценарија *BDD* користи јасан и свима разумљив начин описивања примера и сценарија који представљају понашање система. Ови сценарији се пишу у формату који могу једнако добро разумети и технички и нетехнички учесници у развојном процесу, при чему структура обично обухвата опис почетног стања, догађаја и очекиваног исхода. Формат и конкретна синтакса нису строго дефинисани, али се често користе стандардизовани обрасци ради лакше аутоматизације тестова и доследне комуникације.

Жива документација Спецификације написане у *BDD-y* нису само документација која се једном напише и заборави. Напротив, оне се стално одржавају и извршавају у оквиру тестова, што осигурава да документација увек одража-

ва тренутно стање система. Ова жива документација постаје кључни алат у одржавању квалитета и транспарентности пројекта.

Приоритизација и спецификација захтева Када су сценарији написани, они служе као основа за развој софтвера. Тим може лако да препозна које функционалности имају највећи утицај и прво их реализује.

2.3.2 Тимски рад и сарадња у *BDD* процесу

Једна од кључних вредности *BDD-a* је побољшање сарадње и комуникације између свих учесника у развоју софтвера. Тиме се смањује ризик од неспоразума и неправилне имплементације захтева, што је чест узрок неуспеха пројеката.

BDD подстиче блиску сарадњу између три главне групе актера:

- Бизнис аналитичари и корисници — имају најбоље разумевање пословних потреба и циљева,
- Развојни тим (програмери) — претварају захтеве у функционалан кôд,
- Тестери — проверавају исправност и квалитет реализованог софтвера.

Кроз заједничке радионице, састанке и ревизије, сви учесници учествују у дефинисању сценарија понашања система. Ово обезбеђује да пословни захтеви буду прецизно и јасно изражени, а да програмери имају јасна упутства за имплементацију.

Поред тога, редовни циклуси као што су *спринт планови*, *прегледи рада* и *ретроспективе* омогућавају тимовима да размене повратне информације, идентификују проблеме и прилагоде приступ развоју. На тај начин се обезбеђује континуирана синхронизација између бизниса и технике.

BDD такође помаже у изградњи поверења међу члановима тима, јер сви раде ка заједничком циљу – испоруци софтвера који заиста решава стварне пословне проблеме и испуњава очекивања корисника. Закључно, тимски рад у *BDD-у* представља основу успешне имплементације ове методологије.

2.3.3 Предности и изазови

Развој вођен понашањем доноси више значајних предности у процесу развоја софтвера. Једна од најважнијих је јасна комуникација између тимова, јер се

користи јединствен језик и формат који омогућава боље разумевање захтева између бизнис корисника, тестера и програмера. Захтеви који се дефинишу кроз понашања омогућавају рано откривање грешака и бољу покривеност тестовима, што води ка побољшаном квалитету софтвера. Осим тога, спецификације се одржавају ажурним, јер су интегрисане са аутоматизованим тестовима, што доприноси одржавању конзистентности и представља тзв. живу документацију. Усмеравањем развоја на видљиво понашање система, овај приступ подстиче развој софтвера који боље одговара стварним потребама корисника. Додатно, употреба заједничког језика и дефинисање захтева кроз примере смањује ризик од неспоразума између различитих тимова.

Са друге стране, примена овог приступа носи и одређене изазове. Један од њих су почетни трошкови имплементације, с обзиром на то да је потребно време и труд за обучавање тимова и успостављање одговарајућих процеса. Успешна примена захтева и висока укљученост свих заинтересованих страна може представљати проблем у организацијама са слабијом унутрашњом комуникацијом. Такође, постоји ризик од превелике формализације, јер тежња ка детаљним сценаријима може успорити развој и унети непотребну сложеност. Одржавање живе документације захтева сталну дисциплину у тиму, како би спецификације остале тачне и релевантне током читавог животног циклуса софтвера.

Упркос овим изазовима, предности као што су побољшана комуникација, боље разумевање и усаглашеност захтева, као и већи квалитет софтвера, чине овај приступ веома вредним, посебно у контексту комплексних и динамичних развојних окружења.

Глава 3

Алат *Cucumber*

Cucumber [29] је алат отвореног кода који се користи за тестирање софтвера засновано на принципима развоја вођеног понашањем. Његова основна сврха је да омогући сарадњу између различитих чланова тимова у развоју софтвера кроз тестове који су написани на природном језику. Ово омогућава свим члановима тима, укључујући оне који нису технички стручњаци, да разумеју и допринесу процесу тестирања. Тестови се пишу као сценарији који описују како апликација треба да се понаша у различитим ситуацијама. Тако написани тестови лакше се мењају у поређењу са традиционалним кодираним тестовима, што их чини флексибилнијим и лакшим за одржавање.

Cucumber чита спецификације написане у тексту и потврђује да софтвер ради оно што те спецификације налажу. Спецификације се састоје од више примера или сценарија. Сваки сценарио је списак корака које *Cucumber* обрађује. *Cucumber* верификује да софтвер одговара спецификацији и генерише извештај који показује знак ✓ (успех) или знак × (неуспех) за сваки сценарио. Да би *Cucumber* разумео сценарије, они морају да прате основна правила синтаксе, која се зове *Gherkin*.

3.1 Формат *Gherkin*

Gherkin [29] је скуп граматичких правила који чини текст структурираним да би алат *Cucumber* могао да га разуме. *Gherkin* документи се чувају у посебној текстуалној датотеци са екстензијом *.feature*, која се назива *Feature* фајл. Листинг 3.1 приказује пример *Feature* фајла.

```
1 Feature: Get all recipes
```

```
2
3   @getAll
4   Scenario: Successfully retrieve all recipes when list is empty
5     Given the recipe list is empty
6     When I send a GET request to "/recipes"
7     Then I should receive a 200 response
8     And the response should contain an empty list
9
10  @getAll
11  Scenario Outline: Successfully retrieve all recipes when list
       has one recipe
12    Given the recipe list contains <number> recipe
13    When I send a GET request to "/recipes"
14    Then I should receive a 200 response
15    And the response should contain <number> recipe
16    And the response should be list of data as in JSON file "/
         responses/<fileName>"
17    Scenarios:
18      | number | fileName |
19      | 1      | get_all_recipe_size_one_response.json |
20      | 2      | get_all_recipe_size_two_response.json |
```

Листинг 3.1: Пример *Feature* фајла

Gherkin користи скуп специјалних кључних речи и већина редова у *Gherkin* документу почиње неком од ових речи. Свака кључна реч је преведена на многе говорне језике. У даљем тексту за примере биће коришћен енглески језик. Основне кључне речи су:

- кључна реч `Feature`
- кључна реч `Rule` (од верзије *Gherkin* 6)
- кључна реч `Scenario` (или `Example`)
- кључне речи `Given`, `When`, `Then`, `And`, `But` за кораке
- кључна реч `Background`
- кључна реч `Scenario Outline` (или `Scenario Template`)

- кључна реч `Scenarios` (или `Examples`)

Постоје и неке додатне кључне речи које побољшавају структуру и организацију тестова. Оне нису неопходне за основно функционисање тестова, али омогућавају већу флексибилност, читљивост и лакше управљање тестовима. Додатне кључне речи укључују:

- `"""` — документациони стрингови (енг. *doc strings*)
- | — табеле података (енг. *data tables*)
- @ — тагови (енг. *tags*)
- # — коментари (енг. *comments*)

3.1.1 Кључне речи *Feature*, *Rule* и *Scenario*

Кључне речи *Feature*, *Rule* и *Scenario* дефинишу основну структуру теста. Кључна реч *Feature* описује софтверску функцију која се тестира и групише повезане сценарије. То је прва кључна реч у *Gherkin* документу, након које следи знак :, а затим кратак текст који описује функцију. Може се додати још текста испод кључне речи *Feature* да би се додатно описала функција. Опис за *Feature* завршава се када се линија започне кључним речима *Background*, *Rule*, *Scenario* или *Scenario Outline* (или њиховим алтернативним кључним речима).

Опциона кључна реч *Rule* је део *Gherkin-a* од верзије 6. Циљ кључне речи *Rule* је да представи једно пословно правило које треба да буде имплементирано и пружи додатне информације о функцији. *Rule* се користи за груписање више сценарија која припадају овом пословном правилу. *Rule* треба да садржи једно или више сценарија која илуструју конкретно правило. Пример коришћења кључне речи *Rule* биће дат након увођења других појмова потребних за његово разумевање.

Scenario је кључна реч која се користи за конкретан пример који илуструје пословно правило. Састоји се од листе корака. Синоним за кључну реч *Scenario* је *Example*. Број корака није формално ограничен, али се препоручује три до пет корака по примеру. Превише корака може узроковати да пример изгуби своју изражајну моћ као спецификација и документација. Као целина, сценарији представљају функционалну спецификацију система.

Сценарији се састоје од три целине:

- Описивање почетног контекста (*Given* кораци)
- Описивање догађаја (*When* кораци)
- Описивање очекиваног исхода (*Then* кораци)

У свакој целини можемо имати неколико корака. Сценарио има двоструку улогу – он је истовремено спецификација и документација, али и тест, будући да представља основу за валидацију понашања система у односу на дефинисане захтеве.

3.1.2 Кључне речи *Given*, *When*, *Then*, *And* и *But*

Сваки корак у сценарију почиње са кључним речима *Given*, *When*, *Then*, *And* или *But*. *Cucumber* извршава кораке у редоследу у којем су они наведени. Када *Cucumber* покуша да изврши корак, он тражи одговарајућу дефиницију корака. Кључне речи као што су *Given*, *When*, *Then*, *And* и *But* не утичу на процес тражења дефиниције корака, што значи да се кораци који имају исти текст, без обзира на кључну реч, сматрају дупликатима. У случају да *Cucumber* нађе два корака са истим текстом, то ће бити третирано као дупликат и алат ће пријавити грешку. Кораци наведени у листингу 3.2 су примери дуплиакта.

```
1 Given there is money in my account
2 Then there is money in my account
```

Листинг 3.2: Пример корака који се сматрају дупликатима

У овом случају, оба корака имају исти текст, али се користе различите кључне речи (*Given* и *Then*). *Cucumber* ће их препознати као дупликате и пријавити грешку, јер кораци који имају исти текст треба да буду дефинисани само једном.

Given кораци се користе за описивање почетног стања система. То је обично нешто што се догодило у прошлости. Када *Cucumber* изврши *Given* корак, као што је креирање и конфигурисање објеката или додавање података у тест базу података, он ће конфигурисати систем да буде у добро дефинисаном стању. Циљ *Given* корака је да постави систем у познато стање пре него што корисник (или спољни систем) почне да интерагује са системом (у *When* корацима). Ако би се креирали случајеве употребе, *Given* кораци би били предуслови.

When кораци се користе за описивање догађаја или акције. То може бити особа која интерагује са системом, или може бити догађај који је покренут од стране другог система.

Then кораци се користе за описивање очекиваног исхода или резултата. Дефиниција корака за *Then* треба да изврши поређење стварног исхода (шта систем ради) са очекиваним исходом (шта корак каже да систем треба да уради). Исход треба да буде на неком видљивом излазу. То је нешто што излази из система (извештај, кориснички интерфејс, порука), а не понашање дубоко скривено унутар система (као што је запис у бази података). Није добра пракса имплememнтирати *Then* кораке који врше проверу над подацима у бази података.

Један тест може садржати више *Given* и *Then* корака. Листинг 3.3 приказује пример употребе више *Then* корака.

```
1  Scenario: Successfully update a recipe
2    Given the system is initialized
3    Given the recipe data in JSON file "/requests/
        update_recipe_request.json"
4    When I send a PUT request to "/recipes/Recipe1"
5    Then I should receive a 200 response
6    Then the recipe response should be like the recipe data in
        JSON file "/responses/update_recipe_response.json"
7    Then the recipe name should not change
```

Листинг 3.3: Више *Then* корака

Пример постаје читљивији заменом узастопних *Given* или *Then* корака са *And* и *But* као што је приказано у листингу 3.4. *And* се користи када се наставља корак исте врсте као претходни, на пример додатна провера након *Then* или додатни предуслов након *Given*, док се *But* користи када се жели нагласити контраст или посебан услов који додатно допуњује претходни корак, указујући на изузетак или важну спецификацију у тесту.

```
1  Scenario: Successfully update a recipe
2    Given the system is initialized
3    And the recipe data in JSON file "/requests/
        update_recipe_request.json"
4    When I send a PUT request to "/recipes/Recipe1"
```

```
5     Then I should receive a 200 response
6     And the recipe response should be like the recipe data in
          JSON file "/responses/update_recipe_response.json"
7     But the recipe name should not change
```

Листинг 3.4: Замена више *Given* корака кључним речима *And* и *But*

Пример приказан у листингу 3.5 приказује сценарио који илуструје коришћење кључне речи *Rule*.

```
1 Rule: A plan must exist for the given date and username
2     If no plan is found for the provided username or date,
3     the system should return a 404 response with an appropriate
          error message.
4
5     Scenario: No plan found for the given username
6       When I send a GET request to "/plans?username=Admin1&date
            =2025-08-28"
7       Then I should receive a 404 response
8       And the response should contain error message "Plan not
            found for date: 2025-08-28"
9
10    Scenario: No plan found for the given date
11      When I send a GET request to "/plans?username=Admin&date
            =2025-08-27"
12      Then I should receive a 404 response
13      And the response should contain error message "Plan not
            found for date: 2025-08-27"
```

Листинг 3.5: Пример са кључном речју *Rule*

3.1.3 Кључна реч *Background*

Понекад се може десити да се исти *Given* кораци понављају у свим сценаријима у једном *Feature-y*. То је показатељ да ти кораци нису кључни за описивање сценарија. Такви кораци могу се преместити у позадину, груписани

под секцију *Background*. Ова секција може садржати један или више *Given* корака, који се извршавају пре сваког сценарија.

Background се ставља пре првог *Scenario/Example* корака, на истом нивоу индентације. Пример се налази у листингу 3.6.

```
Feature: Delete a recipe

  Background:
    Given the system is initialized

  Scenario: Successfully delete a recipe
    When I send a DELETE request to "/recipes/Recipe1"
    Then I should receive a 204 response
    And the recipe "Recipe1" should be deleted from the system

  Scenario: Successfully delete a recipe by its id
    When I send a DELETE by id request
    Then I should receive a 204 response
    And the recipe "Tomato" should be deleted from the system

  Scenario: Delete a non-existing recipe
    When I send a DELETE request to "/recipes/NonExistingRecipe"
    Then I should receive a 404 response
    And the response should contain error message "Didn't find a
        recipe with name: NonExistingRecipe"

  Scenario: Delete a recipe by non existing id
    When I send a DELETE request to "/recipes?id=1"
    Then I should receive a 404 response
    And the response should contain error message "Didn't find a
        recipe with id: 1"
```

Листинг 3.6: Пример коришћења кључне речи *Backgound*

Може постојати само један *Background* део у *Feature-y*. Уколико је потребно више различитих *Background* корака за различите сценарије, треба размислити о раздвајању скупа сценарија у више *Feature-a*.

3.1.4 Кључна реч *Scenario Outline*

Копирање и лепљење сценарија да би се користиле различите вредности брзо постаје напорно и сам кôд има доста понављања. Кључна реч *Scenario Outline* може се користити за покретање истог сценарија више пута, са различитим комбинацијама вредности параметара. Кључна реч *Scenario Template* је синоним кључну реч *Scenario Outline*.

Листинг 3.7 приказује пример два сценарија у којима се понављају кораци.

```
1  Scenario: Successfully retrieve all ingredients when user has
       1 ingredient
2    Given the ingredient list contains 1 ingredient
3    When I send a GET request to "/ingredients?username=User1"
4    Then I should receive a 200 response
5    And the response should contain 1 ingredient
6    And the response should be list of data as in JSON file "/
       responses/get_all_ingredient_size_one_response.json"
7
8  Scenario: Successfully retrieve all ingredients when user has
       2 ingredient
9    Given the ingredient list contains 2 ingredient
10   When I send a GET request to "/ingredients?username=User1"
11   Then I should receive a 200 response
12   And the response should contain 2 ingredient
13   And the response should be list of data as in JSON file "/
       responses/get_all_ingredient_size_two_response.json"
```

Листинг 3.7: Пример два сценарија у којима се понављају кораци

Scenario Outline нам омогућава да ова два слична сценарија комбинујемо коришћењем шаблона са параметрима који су ограђени са $< >$, као што је приказано у листингу 3.8.

```
1  Scenario Outline: Successfully retrieve all ingredients when
       user has <number> ingredient
2    Given the ingredient list contains <number> ingredient
3    When I send a GET request to "/ingredients?username=User1"
4    Then I should receive a 200 response
5    And the response should contain <number> ingredient
```

```
6     And the response should be list of data as in JSON file "/
          responses/<fileName>"
7     Examples:
8       | number | fileName |
9       | 1      | get_all_ingredient_size_one_response.json |
10      | 2      | get_all_ingredient_size_two_response.json |
```

Листинг 3.8: Пример комбиновања два сценарија коришћешем *Scenario Outline*

Scenario Outline мора да садржи један или више секција *Scenarios* (или *Examples*). Његови кораци се тумаче као шаблон који се никада не извршава директно. Уместо тога, *Scenario Outline* се извршава једном за сваки ред у секцији *Scenarios* испод њега (не рачунајући први ред који је заглавље).

Кораци могу користити параметре ограђене са < > који упућују на заглавља у табели *Scenarios*. *Cucumber* ће заменити ове параметре са вредностима из табеле пре него што покуша да усагласи корак са дефиницијом корака. Параметри се могу користити и у описима *Scenario Outline-a*.

3.1.5 Кључне речи *Doc Strings* и *Data Tables*

У неким случајевима је потребно проследити више података кораку него што може да стане у један ред. За ову сврху *Gherkin* има *Doc Strings* и *Data Tables*.

Doc Strings су корисни за прослеђивање већег дела текста дефиницији корака. Текст треба да буде одвојен делимитерима који се састоје од три двострука наводника """. Уместо три двострука наводника, *Doc Strings* такође подржава знаке ``` као делимитер. Листинг 3.9 приказује пример коришћења *Doc Strings*.

```
1  Scenario: Successfully create a new ingredient
2     Given the ingredient data:
3     """
4     {
5       "name": "Avocado",
6       "calories": 160,
7       "category": "Fruit",
8       "username": "TestUser"
9     }
```

```
    """
    When I send a POST request to "/ingredients"
    Then I should receive a 201 response
    And the ingredient response should be:
    ```
    {
      "id": 1,
      "name": "Avocado",
      "calories": 160,
      "category": "Fruit",
      "username": "TestUser"
    }
    ```
```

Листинг 3.9: Пример коришћења *Doc Strings*

У дефиницији корака, нема потребе експлицитно тражити садржај *Doc Strings*-а и усаглашавати га са образцем, јер ће он аутоматски бити прослеђен као последњи аргумент.

Data Tables су корисне за прослеђивање листе вредности у дефиницију корака. Листинг 3.10 приказује пример коришћења *Data Tables*.

```
Scenario: Successfully create a new ingredient
  Given the following ingredient data:
    | name      | calories | category | username  |
    | Avocado   | 160      | Fruit    | TestUser  |
  When I send a POST request to "/ingredients"
  Then I should receive a 201 response
```

Листинг 3.10: Пример коришћења *Data Tables*

Нови ред у ћелији табеле може се написати као `\n`. Знак `|` као део ћелије записује се као `\|`. Знак `\` записује се као `\\`.

3.1.6 Тагови

Тагови су кључне речи које се користе да би се тестови груписали на основу одређених критеријума. Могу се користити да се тестови организују

по категоријама као што су типови тестова (нпр. `@smoke`, `@regression`) или по функционалним областима (нпр. `@user-account`, `@payment`). Тагови се постављају пре сценарија и могу бити коришћени за филтрирање тестова приликом извршавања. Пример коришћења тагова може се видети у листингу 3.11.

```
Feature: Get a recipe by name

  Background:
    Given the system is initialized

  @getByName
  Scenario: Successfully retrieve a recipe by name
    When I send a GET request to "/recipes/Recipe1"
    Then I should receive a 200 response
    And the recipe response should be like the recipe data in
        JSON file "/responses/get_recipe_by_name_response.json"

  @error
  Scenario: Get a non-existing recipe
    When I send a GET request to "/recipes/NonExistingRecipe"
    Then I should receive a 404 response
    And the response should contain error message "Nije
        pronadjen recept sa imenom: NonExistingRecipe"
```

Листинг 3.11: Пример комбиновања тестова по категоријама

Тагови су корисни јер омогућавају:

- Лако филтрирање тестова током извршавања (нпр. покретање само тестова који имају одређени таг).
- Организовање тестова по групама, што чини њихово управљање лакшим.
- Бржи приступ специфичним тестовима, као што су тестови за регресију или функционални тестови.

3.1.7 Коментари и говорни језици

Коментари су дозвољени само на почетку новог реда, било где у *Feature* датотеци. Започињу са нула или више размака, након чега следи знак **хеш** (#) и текст који желите да додате. Блок коментари тренутно нису подржани у *Gherkin-y*. За индентацију се могу користити размаци или табови, али препоручени ниво индентације је два размака.

Језик који се изабере за *Gherkin* треба да буде исти као језик који корисници и стручњаци користе када разговарају о домену. Треба избегавати превођење између два језика.

Због тога је *Gherkin* преведен на више од 70 језика, међу којима је и српски језик. Сценарији се могу писати на ћирилици (табела 3.1) и латиници (табела 3.2). Пример сценарија на српском је дат у листингу 3.12.

Кључна реч	**Еквивалент**
Feature	Функционалност, Могућност, Особина
Background	Контекст, Основа, Позадина
Rule	Правило
Scenario	Сценарио, Пример
Scenario Outline	Структура сценарија, Скица, Концепт
Scenarios	Примери, Сценарији
Given	За дато, За дате, За дати
When	Када, Кад
Then	Онда
And	И
But	Али

Табела 3.1: Ћирилица (sr-Cyrl)

Кључна реч	**Еквивалент**
Feature	Funkcionalnost, Mogućnost, Mogucnost, Osobina
Background	Kontekst, Osnova, Pozadina
Rule	Pravilo
Scenario	Scenario, Primer
S*cenario Outline*	Struktura scenarija, Skica, Koncept
Scenarios	Primeri, Scenariji

Кључна реч	**Еквивалент**
Given	Za dato, Za date, Za dati
When	Kada, Kad
Then	Onda
And	I
But	Ali

Табела 3.2: Латиница (sr-Latn)

```
# language: sr-Latn
Funkcionalnost: Obrisati recept

Pozadina:
  Za dato da recept postoji

Scenario: Brisanje recepta sa datim imenom
  Kada pošaljem DELETE zahtev na "/recipes/Recipe1"
  Onda treba da dobijem odgovor sa statusom 204
  I recept "Recipe1" treba da bude obrisan iz sistema
```

Листинг 3.12: Пример *Gherkin* сценарија написаног српском латиницом

Заглавље *# language:* у првом реду *Feature* фајла говори *Cucumber-y* који језик да користи. На пример, *# language: sr-Latn* означава српску латиницу. Ако се изостави ово заглавље, *Cucumber* ће подразумевано користити енглески (en). Неке *Cucumber* имплементације такође омогућавају да се подразумевани језик подеси у конфигурацији, тако да није потребно стављати *# language* заглавље у сваки фајл.

3.2 Дефиниције и резултати корака

Дефиниција корака [29] је метода са изразом који је повезан са једним или више *Gherkin* корака. Она представља везу између текста корака у *Feature* фајлу и реалне логике која се извршава у тестном коду. За имплементацију дефиниција корака могу се користити различити програмски језици који имају подршку за *Cucumber*, као што су Јава, *JavaScript*, *Ruby* или *Python*. У овом

раду тестови су реализовани у Јава програмском језику због интеграције са Јава *Spring Boot* апликацијом. Сваки *Gherkin* корак повезује се са одговарајућом Јава методом у класи са дефиницијама корака, у којој се пише логика извршења теста, као што је слање *HTTP* захтева или провера одговора апликације.

Изрази одређују како се текст у кораку упарује са одговарајућим методама у тестном кôду. Постоје два главна типа израза који се користе у овом контексту, регуларни изрази (енгл. *regular expressions*) и *Cucumber* изрази (енгл. *Cucumber expressions*). Регуларни изрази су низови карактера који чине образац за претрагу. Користе се за упаривање образаца унутар стрингова. На пример, регуларни израз `/\d+/` одговара једној или више цифара (0–9) унутар стринга.

Рецимо да у *Gherkin* сценарију имамо корак приказан у листингу 3.13.

```
1    Then I should receive a 200 response
```

Листинг 3.13: Пример *Then* корака

Уместо писања појединачних дефиниција корака за сваки могући улаз (нпр. `,,Then I should receive a 200 response''`, `,,Then I should receive a 404 response''`), могуће је користити регуларне изразе за креирање општије дефиниције корака која обухвата бројеве или друге динамичне делове. Листинг 3.14 приказује пример дефиниције корака са регуларним изразом.

```
1  @Then("I should receive a (\\d+) response")
2  public void iShouldReceiveAResponse(int statusCode) {
3    assertEquals(HttpStatus.valueOf(statusCode), response.
       getStatusCode());
4  }
```

Листинг 3.14: Пример са регуларним изразом у Cucumber-у

У овом примеру:

- `\d+` у дефиницији корака значи „упари једну или више цифара”. У низу карактера написаном у Јави, коса црта (\) има посебно значење – Јава је тумачи као почетак контролне или *escape* секвенце (нпр. \n за нови ред или \t за табулатор). Да би се коса црта интерпретирала дословно као део текста, мора се ставити још једна коса црта испред ње, па се

регуларни израз \d+ пише као \\d+. Затворене заграде `()` хватају ове цифре како би могле бити прослеђене као параметри метода.

- `I should receive a (\d+) response` ће одговарати фрази као што је „I should receive a 200 response", а број 200 ће бити прослеђен као аргумент `statusCode`.

Cucumber изрази [28] су специјално дизајнирани за *Cucumber* и пружају једноставнији и читљивији начин дефинисања корака. Пружају јасну и концизну синтаксу која је блиска природном језику. Смањују могућност прављења грешака које могу настати услед сложеније синтаксе регуларних израза. Они користе параметре као што су `{int}`, `{string}`, итд., који омогућавају да се делимично фиксни делови корака комбинују са променљивим вредностима, што повећава флексибилност и поновну употребљивост тестова. У табели 3.3 су приказани неки од уобичајених параметризованих типова у *Cucumber*-у.

Табела 3.3: Уобичајени параметризовани типови у Cucumber-у

Параметар	**Опис**
{int}	Одговара целобројним вредностима, као што су 71 или -19.
{float}	Одговара децималним бројевима, као што су 3.6, .8 или -9.2. Конвертује се у 32-битни *float* ако платформа то подржава.
{word}	Одговара једној речи без размака, на пример „banana" (али не и „banana split").
{string}	Одговара текстовима унутар једноструких или двоструких наводника, као што су „banana split" или 'banana split'.
{} анонимни	Одговара било којем тексту (еквивалентно са /.*/ у регуларним изразима).
{bigdecimal}	Прихвата исте вредности као {float}, али конвертује резултат у *BigDecimal* ако платформа то подржава.
{double}	Прихвата исте вредности као {float}, али конвертује резултат у 64-битни *float* ако платформа то подржава.

Параметар	**Опис**
{biginteger}	Прихвата исте вредности као {int}, али конвертује резултат у тип *BigInteger* ако платформа то подржава.
{byte}	Прихвата исте вредности као {int}, али конвертује резултат у 8-битни целобројни тип ако платформа то подржава.
{short}	Прихвата исте вредности као {int}, али конвертује резултат у 16-битни целобројни тип ако платформа то подржава.
{long}	Прихвата исте вредности као {int}, али конвертује резултат у 64-битни целобројни тип ако платформа то подржава.

У *Cucumber-y*, сваки корак може имати један од следећих резултата:

Успех Када *Cucumber* пронађе одговарајућу дефиницију корака, извршиће је. Ако метод у дефиницији корака не изазове грешку, корак ће бити обележен као успешан (зелен).

Недефинисано Ако *Cucumber* не може да пронађе одговарајућу дефиницију корака, корак ће бити обележен као недефинисан (жут), а сви наредни кораци у сценарију ће бити прескочени.

На чекању Када метода дефиниције корака експлицитно позове функцију `pending`, корак ће бити обележен као „на чекању“ (жуто), што указује да корак још увек није имплементиран и да је потребан даљи рад од стране програмера.

Неуспех Ако метод дефиниције корака изазове грешку током извршавања, корак ће бити обележен као неуспешан (црвен).

Прескочен Кораци који следе након недефинисаних, на чекању или неуспешних корака неће бити извршени, чак и ако постоји одговарајућа дефиниција корака. Ови кораци су обележени као прескочени (тиркизно плава).

Нејасно Ако више различитих дефиниција одговара истом кораку, *Cucumber* неће бити у могућности да одлучи коју дефиницију треба да примени. У том случају, корак ће бити обележен као нејасан, што указује на потребу за разрешењем конфликта у дефиницијама. Понашање у овом случају зависи од тога који се програмски језик користи за писање дефиниција корака. У програмском језику Јава генерише се `AmbiguousStepDefinitionsException` изузетак. За овај случај у документацији није назначена одговарајућа боја.

Ови статуси помажу у разумевању извршења тестова и идентификовању области које захтевају пажњу.

3.3 Позивне тачке

Позивне тачке (енг. *hooks*) [29] су блокови кода који се могу извршавати у различитим фазама *Cucumber*-овог циклуса извршења. Обично се користе за подешавање и чишћење окружења пре и после сваког сценарија. Место где је позивна тачка дефинисана не утиче на сценарије или кораке за које ће бити извршена.

Постоје два начина писања позивних тачака у Јави, јер оба метода пружају флексибилност у зависности од потреба програмера, као и одређених стилова кодирања:

- Коришћење анотација: Овај начин је класичан и популаран у старијим верзијама. Овакав приступ нуди већи степен контроле, јер програмер мора експлицитно да дефинише методе и анотације које ће се користити. Такође, методи написани са анотацијама су лакше разумљиви и прилагодљиви за тимове који користе стандардне методе у тестирању.

- Коришћење ламбда израза: Ламбда изрази су уведени са Јава 8 и представљају лакши и краћи начин дефинисања функција. Када се користе за дефинисање позивних тачака, они омогућавају краћи и чистији кôд, који је често лакши за читање и имплементацију, посебно у модерним пројектима који користе новије верзије Јаве.

Постоје различите врсте позивних тачака. У наставку текста биће детаљније представљене њихове карактеристике.

3.3.1 Позивне тачке сценарија

Позивне тачке *Before* се извршавају пре првог корака сваког сценарија и служе за припрему окружења или постављање почетних услова. У листингу 3.15 приказан је пример позивне тачке *Before* написане у Јави коришћењем анотације, где метода прима објекат *Scenario* који представља текући сценарио, омогућавајући приступ информацијама о њему.

```
1 @Before
2 public void doSomethingBefore(Scenario scenario){
3     // Do something before each scenario
4 }
```

Листинг 3.15: Пример позивне тачке *Before*

Позивне тачке се могу писати и коришћењем ламбда израза. Пример је приказан у листингу 3.16, где број 10 као први аргумент одређује редослед извршавања позивне тачке у случају када постоји више позивних тачака. Позивна тачка са нижим редоследом ће се извршити пре позивне тачке са већим редоследом. У овом примеру ламбда не прихвата аргумент *Scenario*, јер припремне радње које се изводе у овој позивној тачки не захтевају информације о конкретном сценарију. То су обично припрема окружења, ресетовање података или покретање сервиса.

```
1 Before(10, () -> {
2     // Do something before each scenario
3 });
```

Листинг 3.16: Пример позивне тачке *Before* написане коришћењем ламбда израза

Уколико је потребно приступити сценарију унутар позивне тачке написане коришћењем ламбди то би изгледало као у листингу 3.17.

```
1 Before(10, (Scenario scenario) -> {
2     // Do something with scenario
3 });
```

Листинг 3.17: Пример позивне тачке *Before* са Scenario аргументом

Редослед извршавања позивних тачака може се одредити и у традиционалном облику са анотацијом ‘@Before‘, коришћењем атрибута `order`, као што је приказано у листингу 3.18.

```
@Before(order = 10)
public void doSomething(){
    // Do something before each scenario
}
```

Листинг 3.18: Пример позивне тачке са дефинисаним редоследом

Све што се догађа у позивној тачки *Before* је невидљиво људима који читају само *Feature-е*. Требало би размотрити коришћење *Background* корака као јасније алтернативе, посебно ако подешавање треба да буде читљиво за не-техничке особе. Позивну тачку *Before* треба користити само за логику ниског нивоа као што је покретање претраживача или брисање података из базе података.

Позивна тачка *After*

Позивне тачке *After* се извршавају после последњег корака сваког сценарија, без обзира на то да ли је резултат корака био неуспешан, недефинисан, на чекању или прескочен. У листингу 3.19 је приказан пример позивне тачке *After* написане у програмском језику Јава, коришћењем анотације.

```
@After
public void doSomethingAfter(Scenario scenario){
    // Do something after each scenario
}
```

Листинг 3.19: Пример позивне тачке *After*

Написана коришћењем ламбда израза у програмском језику Јава позивна тачка *After* изгледа као у листингу 3.20.

```
After((Scenario scenario) -> {
    // Do something after each scenario
});
```

Листинг 3.20: Пример позивне тачке *After* написане коришћењем ламбда израза

3.3.2 Позивне тачке корака

Позивне тачке корака се извршавају пре и после сваког корака. Ове позивне тачке имају семантику *„invoke around”*, што значи да ако се изврши позивна тачка *BeforeStep*, онда ће се извршити и позивна тачка *AfterStep*, без обзира на резултат корака. Ако корак није прошао, следећи корак и његове позивне тачке ће бити прескочени.

Позивна тачка *BeforeStep*

Позивна тачка *BeforeStep* извршава се пре сваког корака. У листингу 3.21 је приказан пример позивне тачке *BeforeStep* написане у програмском језику Јава, коришћењем анотације.

```
1 @BeforeStep
2 public void doSomethingBeforeStep(Scenario scenario){
3     // Do something before each step
4 }
```

Листинг 3.21: Пример позивне тачке *BeforeStep*

Написана коришћењем ламбда израза у програмском језику Јава, позивна тачка *BeforeStep* изгледа као у листингу 3.22.

```
1 BeforeStep((Scenario scenario) -> {
2     // Do something before each step
3 });
```

Листинг 3.22: Пример позивне тачке *BeforeStep* написане коришћењем ламбда израза

Позивна тачка *AfterStep*

Позивна тачка *AfterStep* извршава се након сваког корака. У листингу 3.23 је приказан пример позивне тачке *AfterStep* написане у програмском језику Јава, коришћењем анотације.

```
1 @AfterStep
2 public void doSomethingAfterStep(Scenario scenario){
3     // Do something after each step
4 }
```

Листинг 3.23: Пример позивне тачке *AfterStep*

Написана коришћењем ламбда израза у програмском језику Јава, позивна тачка *AfterStep* изгледа као у листингу 3.24.

```
1 AfterStep((Scenario scenario) -> {
2     // Do something after each step
3 });
```

Листинг 3.24: Пример позивне тачке *AfterStep* написане коришћењем ламбда израза

3.3.3 Условне позивне тачке

Позивне тачке могу се условљавати таговима дефинисаним унутар сценарија. На тај начин, њихово извршавање ограничава се искључиво на сценарије који задовољавају задате услове, чиме се обезбеђује контрола над током тестирања.

У листингу 3.25 приказан је пример условне позивне тачке који ће се извршити након сваког сценарија који има таг `@browser`.

```
1 @After("@browser")
2 public void doSomethingAfter(Scenario scenario){
3     // Do something based on condition
4 }
```

Листинг 3.25: Пример позивне тачке која ће се извршити након тага `@browser`

Исти пример написан помоћу ламбди изгледао би као у листингу 3.26.

```
After("@browser", (Scenario scenario) -> {
    // Do something based on condition
});
```

Листинг 3.26: Пример позивне тачке која ће се извршити након тага `@browser`

3.3.4 Глобалне позивне тачке

Глобалне позивне тачке се извршавају једном пре него што се изврши било који сценарио или након што су сви сценарији извршени. Означавају се са *BeforeAll* и *AfterAll*.

Позивна тачка *BeforeAll* се извршава једном пре него што се изврши било који сценарио. Ова позивна тачка је корисна за постављање глобалних ресурса или иницијализацију окружења које је потребно пре свих тестова.

Позивна тачка *AfterAll* се извршава након што су сви сценарији завршени. Ова позивна тачка је корисна за обављање општих операција чишћења или ослобађања ресурса који су били у употреби током тестирања. У листингу 3.27 приказан је пример позивних тачака *BeforeAll* и *AfterAll*.

```
@BeforeAll
public static void setUp() {
    // This method is executed before all scenarios
    // Set up global resources
}

@AfterAll
public static void tearDown() {
    // This method is executed after all scenarios
    // Clean up global resources
}
```

Листинг 3.27: Пример глобалних позивних тачака

Глава 4

Технологије коришћене у развоју апликације

У овој глави представљене су кључне технологије и концепти који су коришћени у развоју апликације. Посебна пажња посвећена је микросервисној архитектури, која представља савремени приступ развоју сложених софтверских система кроз поделу на мале, независне и лако управљиве сервисе. Након тога, описан је развој у оквиру Јава екосистема, који представља стабилно и широко прихваћено окружење за развој софтвера. Представљени су програмски језик Јава, развојни оквир *Spring Boot*, као и библиотека *React* за изградњу корисничког интерфејса, који заједно омогућавају модеран и ефикасан развој апликација. На крају, анализиран је процес изградње и управљања пројектом помоћу алата *Maven*.

4.1 Микросервисна архитектура

Микросервисна архитектура [14] представља парадигму развоја софтверских система у којој се сложени апликативни системи граде као скуп малих и независних сервиса који међусобно сарађују ради испуњавања пословних захтева. Сваки од ових сервиса енкапсулира одређену пословну функцију, поседује свој модел података, логику и могућности складиштења, а развијају се, тестирају, испоручују и скалирају засебно.

Овај приступ подразумева декомпозицију система по јасно дефинисаним пословним потребама, као што су корисничка регистрација, управљање производима или обрада плаћања, чиме се омогућава независан развој и постављање

сервиса. Поред тога, микросервиси често користе различите технологије и програмске језике у складу са специфичним захтевима сваког сервиса, а сваки сервис поседује своју изоловану базу података како би се одржала слаба повезаност и изолација података. Комуникација између сервиса обично се реализује преко добро дефинисаних јавних интерфејса.

У односу на традиционалну монолитну архитектуру, где су кориснички интерфејс, пословна логика и приступ бази података интегрисани у једну целовиту апликацију, микросервисни приступ омогућава већу флексибилност тимова који могу независно радити на појединачним сервисима. Промене у једном делу система не захтевају компилацију или постављање нове верзије целог пројекта, што убрзава развојни циклус и смањује ризик од увођења грешака.

Микросервисна архитектура доноси бројне предности, укључујући модуларност и лаку скалабилност система, јер је могуће појединачно скалирати само оне делове који су под највећим оптерећењем. Омогућава паралелни развој више тимова, што убрзава испоруку нових решења и повећава флексибилност у избору технологија, јер сваки сервис може користити најприкладнији језик и алате. Уз то, систем је отпорнији на грешке јер квар једног сервиса не доводи нужно до отказа целог система.

Ипак, овај приступ носи и значајне изазове. Управљање великим бројем микросервиса захтева напредну инфраструктуру и оркестрацију, што повећава сложеност у одржавању система. Одржавање конзистентности података у окружењу у ком сваки сервис има своју базу података представља изазов који захтева пажљиво пројектовање дистрибуираних трансакција и механизама сагласности. Поред тога, интензивна мрежна комуникација између сервиса може довести до додатних трошкова у погледу мрежних ресурса и безбедности.

Микросервисни приступ посебно је користан за велике апликације које обухватају различите домене и захтевају рад великих тимова, као и за системе код којих је неопходна честа и континуирана испорука нових функционалности, као и висока доступност и скалабилност. Уколико се ради о мањим тимовима или мање комплексним апликацијама, употреба микросервисне архитектуре може бити непотребно оптерећење и додатна сложеност која није оправдана.

4.2 Развој у оквиру Јавиног екосистема

Развојна архитектура апликације реализована је у оквиру Јавиног екосистема, који представља једно од најстабилнијих и најраспрострањенијих окружења за развој софтвера. Јавин екосистем обухвата велики број алата, оквира и библиотека, који подржавају целокупан процес развоја, од писања кода, преко тестирања, до изградње и постављања апликације. Комбинација технологија као што су Јава, *Spring Boot*, *React* и *Maven* омогућила је брз, модуларан и лако одржив развој.

4.2.1 Програмски језик Јава

Јава [16] је објектно оријентисани програмски језик високог нивоа, развијен са циљем да буде преносив, безбедан и једноставан за употребу. Овај језик је дизајниран тако да омогући програмерима да напишу кôд једном, а да се он може покретати на различитим хардверским и софтверским платформама без потребе за поновним компајлирањем. Овај концепт је познат под називом „Напиши једном, извршавај било где“ (енг. *„Write Once, Run Anywhere“*).

Јава се најчешће компајлира у универзални бајткод, који се затим извршава на специјализованој виртуелној машини – *Java Virtual Machine* (*JVM*). На тај начин, апликације написане у Јави могу да се извршавају на било којој платформи на којој постоји имплементација *JVM-a*, што значајно повећава преносивост и флексибилност развоја софтвера.

Синтакса Јаве је инспирисана језицима *C* и *C++*, али је поједностављена како би се избегле честе грешке које се јављају код тих језика, као што је руковање меморијом на ниском нивоу. Јава такође пружа динамичке могућности, попут рефлексије и модификације кода током извршавања, што јој омогућава велику флексибилност у развоју. Због свог дизајна и концепата, Јава је постала један од најпопуларнијих програмских језика у свету.

Принципи програмског језика Јава

Када је Јава дизајнирана, њени творци су имали пет основних принципа које је језик морао да испуни како би био успешан и користан у различитим областима развоја софтвера. Ови принципи су омогућили да Јава постане један од најпопуларнијих и најпоузданијих програмских језика, погодан за широк

спектар апликација, од малих мобилних апликација до великих пословних решења.

Једноставност, објектна оријентисаност и препознатљивост: Јава је дизајнирана тако да буде лако разумљива и једноставна за употребу. Синтакса је инспирисана језицима *C* и *C++*, али су уклоњене сложеније и технички захтевне операције над меморијом које могу изазвати грешке. Објектно оријентисани приступ омогућава јасну структуру програма и подстиче поновну употребу кода, што је основа савременог развоја софтвера. Једноставност се, поред јасне синтаксе, огледа и у аутоматском управљању меморијом, односно систему за сакупљање неупотребљених објеката (енг. *garbage collection*), чиме се елиминише потреба за ручним управљањем меморијом и значајно смањује вероватноћа настанка грешака.

Робустност и безбедност: Јава има развијен систем за управљање меморијом и руковање изузецима, што омогућава отпорност програма на грешке и спречавање рушења апликација. Јава виртуелна машина даје додатни слој безбедности контролом приступа ресурсима и изолацијом извршавања кода, штитећи систем од злонамерног софтвера. Робустност Јаве додатно се осигурава строгом провером типова током процеса компајлирања, што омогућава откривање потенцијалних проблема пре самог извршавања програма.

Независност од архитектуре и преносивост: Јава је дизајнирана као архитектурно неутралан језик, што значи да написани кôд може да се извршава на различитим платформама без потребе за модификацијом или поновним компајлирањем. Ова преносивост остварује се преко бајткода и Јава виртуелне машине, што је једна од кључних карактеристика Јаве.

Високе перформансе: Иако је Јава језик са виртуелном машином, употреба технологија као што су *Just-In-Time (JIT)* компајлери омогућава да се кôд током извршавања оптимизује и претвара у нативни машински кôд, што значајно унапређује перформансе апликација. Овај приступ је омогућио Јави да се користи и у захтевним окружењима као што су сервери и пословни системи.

Интерпретираност, вишенитно програмирање и динамичност: Јава обезбеђује могућност динамичког учитавања класа, што значи да се делови кода могу учитавати и извршавати у реалном времену, без потребе за поновним покретањем целе апликације. Такође, Јава има уграђену подршку за рад са више нити, што омогућава паралелно извршавање задатака и боље коришћење ресурса система.

Развојни оквир *Spring*

Spring [22] представља свеобухватан оквир отвореног кода намењен развоју апликација заснованих на програмском језику Јава, који је прилагодљив различитим платформама за извршавање. Оквир обезбеђује богату инфраструктурну подршку и поједностављује развој софтвера тако што преузима на себе сложене техничке аспекте инсталационе инфраструктуре. На тај начин омогућава програмерима да се фокусирају на развој пословне логике, без потребе да се баве детаљима ниског нивоа имплементације.

Кључне функционалности оквира *Spring* обухватају неколико важних концепата који доприносе квалитетнијем, одрживом и флексиблијем развоју софтвера. Једна од најзначајнијих карактеристика је убризгавање зависности (енг. *Dependency Injection, DI*) [2], које омогућава слабију повезаност компоненти и тиме олакшава тестирање и одржавање кода. Контрола животног циклуса објеката (енг. *Inversion of Control*) управља креирањем, конфигурацијом и животним циклусом објеката у оквиру апликације, обезбеђујући јасну организацију и структуру система. Аспектно оријентисано програмирање (енг. *Aspect-Oriented Programming, AOP*) [1] доприноси модуларизацији попречних аспеката као што су безбедност, логовање и управљање трансакцијама, чиме се одвајају овакве функционалности од пословне логике.

Што се тиче развоја веб апликација, *Spring* пружа подршку кроз модуле као што су *Spring MVC* [24] за класичне веб апликације, као и *Spring WebFlux* [26] за реактивни приступ развоју веб сервиса. Поред тога, оквир обезбеђује интеграцију са бројним системима и технологијама, укључујући рад са базама података, кеширање, безбедност кроз *Spring Security* [25], микросервисну архитектуру и друге кључне компоненте које омогућавају изградu савремених, скалабилних апликација. Управљање трансакцијама је такође поједностављено и конзистентно, што доприноси стабилности и поузданости система.

На крају, оквир *Spring* нуди опсежну подршку за различите нивое тестира-

ња што има кључну улогу у обезбеђивању високог квалитета и одрживости развијених софтверских решења.

Преглед развојног оквира *Spring Boot*

Spring Boot [23] представља проширење развојног оквира *Spring*, осмишљено са циљем поједностављења и убрзавања развоја Јава апликација базираних на овом оквиру. За разлику од традиционалних *Spring* пројеката, који захтевају опсежну конфигурацију, *Spring Boot* елиминише велики део шаблонске конфигурације и долази са унапред дефинисаним подешавањима и смерницама које олакшавају конфигурацију и развој. На тај начин пружа унапред дефинисане почетне зависности које обједињују најчешће коришћене библиотеке и аутоматски подешавају различите аспекте апликације у зависности од тога које су библиотеке присутне у окружењу за извршавање.

Поред тога, оквир обезбеђује уграђене веб сервере као што су *Tomcat* [30] или *Jetty* [5] , што омогућава покретање апликација као самосталних Јава програма без потребе за додатним спољним серверима. Конфигурација апликације се најчешће врши коришћењем анотација и подразумеваних подешавања, чиме се смањује потреба за ручним подешавањима и употребом спољашњих конфигурационих фајлова.

Важно је нагласити да *Spring Boot* није замена за *Spring Framework*, већ надоградња која олакшава примену његових функционалности, омогућавајући брзо покретање и лакшу имплементацију нових апликација уз минималан напор. На тај начин, *Spring Boot* унапређује развојни процес пружајући структурираније и ефикаснији приступ креирању продукцијски спремних *Spring* апликација са знатно смањеним конфигурационим оптерећењем.

4.2.2 Библиотека *React*

React [19] је библиотека за развој корисничког интерфејса заснована на компонентном моделу који омогућава изградњу интерфејса као скупа независних и поново употребљивих целина. Компоненте у *React*-у могу бити дефинисане као класе или функције. Резултат ових компоненти је *JSX* или *TSX*, што је синтаксно проширење *JavaScript* или *TypeScript* језика које омогућава писање елемената сличних *HTML*-у унутар *JavaScript* или *TypeScript* кода. Овај приступ омогућава програмерима боље логичко повезивање структуре

корисничког интерфејса са његовом функционалношћу, што доприноси већој читљивости и бољој организацији кода.

Кључну улогу у перформансама *React*-а има виртуелни *DOM*, који представља унутрашњу репрезентацију интерфејса у меморији. При промени стања компонената, *React* ажурира само оне делове виртуелног *DOM*-а који су се променили, а затим те промене ефикасно преноси на стварни *DOM*. Оваква оптимизација значајно унапређује брзину и реактивност апликација. Управљање стањем представља један од централних концепата, где свака компонента може имати своје локално стање које се мења током извршавања апликације.

Податке у *React*-у карактерише ток у једном смеру, што значи да се они крећу од родитељских ка дечјим компонентама кроз својства (енг. *props*) — статичке вредности које се преносе као атрибути. Овај начин преноса података значајно олакшава праћење променљивих и отклањање грешака током развоја. Животни циклус компоненти је још један важан аспект *React*-а. Он обухвата фазе креирања, ажурирања и уништавања компоненти, уз подршку метода као што су `componentDidMount` и *React* хукова попут `useEffect`, који омогућавају управљање тим фазама на прецизан и контролисан начин.

Још једна важна предност *React*-а је постојање велике и активне заједнице, која креира бројне библиотеке, алате и водиче, чиме се убрзава развој и олакшава решавање техничких проблема. Поред тога, *React* пружа могућност серверског рендеровања путем оквира као што је *Next.js* [15], што доприноси бољој оптимизацији за претраживаче и краћем времену учитавања веб-страница. Такође, библиотека је дизајнирана тако да се лако интегрише са различитим технологијама у позадини, као што су *REST API*-ји, *GraphQL* и други системи за управљање стањем.

React у микросервисној архитектури

У контексту микросервисне архитектуре, *React* се обично користи за изградњу фронтенд дела апликације, који функционише као засебна јединица и комуницира са микросервисним *backend*-ом путем *REST* или *GraphQL* интерфејса. Захваљујући својој модуларној структури, *React* омогућава изолован развој и независно одржавање корисничког интерфејса, што је у складу са принципима раздвајања одговорности и флексибилности који су карактеристични за микросервисне системе. Такав приступ омогућава и лакше тестирање, унапређивање и скалирање интерфејса без потребе за изменама у самој пословној

логици апликације.

4.2.3 Процес изградње коришћењем алата *Maven*

Maven [13] је моћан алат за аутоматизацију процеса изградње, управљање зависностима и животним циклусом софтверских пројеката, нарочито у Јава екосистему. Омогућава поједностављење и стандардизацију компилације, тестирања и постављања апликација.

Основне карактеристике алата *Maven* укључују декларативни приступ конфигурацији, при чему се уместо писања сложених скрипти за изградњу користи датотека `pom.xml` (*Project Object Model*) у којој се на декларативан начин наводе подаци о пројекту, зависностима, софтверским додацима (енг. *plugin*) и задацима. Овакав приступ омогућава већу прегледност и једноставније одржавање. Управљање зависностима је још једна важна карактеристика, јер *Maven* аутоматски преузима и управља зависностима из централизованих репозиторијума као што је *Maven Central*, чиме се елиминише потреба за ручним додавањем библиотека и избегавају се конфликти верзија.

Животни циклус пројекта је стандардизован и обухвата унапред дефинисане фазе као што су `compile`, `test`, `package`, `install` и `deploy`, што доприноси доследности у раду на различитим пројектима. *Maven* је такође проширив захваљујући великом броју софтверских додатака који омогућавају извођење различитих задатака као што су генерисање документације, анализа кода, тестирање и интеграција са алатима за континуирану интеграцију. Поред тога, *Maven* пружа подршку за мултимодуларне пројекте, што омогућава бољу организацију и поновну употребу компоненти у већим системима.

Међу предностима коришћења *Maven*-а издвајају се уштеда времена при конфигурацији и изградњи пројеката, стандардизован начин рада који је лако разумљив и применљив, једноставно управљање верзијама зависности, као и подршка за алате за континуирану интеграцију као што су *Jenkins* [4] и *Travis CI* [31]. Централизована конфигурација додатно омогућава једноставан пренос пројеката између различитих тимова и окружења.

Са друге стране, неки од недостатака алата *Maven* укључују спорије преузимање великих зависности или извођење сложених софтверских додатака, као и релативно стрму криву учења због великог броја концепата и *XML* синтаксе. Рад са сложеним мултимодуларним пројектима може захтевати додатно искуство и разумевање интерне структуре алата *Maven*.

Глава 5

Апликација за планирање оброка *Plan&Prep*

У овој глави представљена је апликација *Plan&Prep* [9]. Апликација је осмишљена као студија случаја која практично илуструје примену развоја вођеног понашањем у развоју софтвера коришћењем алата *Cucumber*. Управо кроз ову имплементацију приказује се како се апстрактни концепти описани у претходним погављима могу успешно пренети у реалан развојни процес. У наредним поглављима биће детаљно описана архитектура система као и намена апликације и случајеви употребе.

5.1 Архитектура система

Апликација је конципирана као систем који се састоји из два слоја – серверског и клијентског. Серверски део реализован је кроз групу независних микросервиса, од којих сваки има јасно дефинисан интерфејс за комуникацију са клијентском апликацијом. Клијентски слој користи ове интерфејсе како би приступио функционалностима и подацима које пружају поједини микросервиси.

Сваки микросервис управља сопственом базом података, чиме се обезбеђује изолација података и лако одржавање целокупног система. Размена информација између микросервиса, као и комуникација са клијентским делом, одвија се преко *HTTP* протокола. У наредним деловима овог поглавља биће детаљније објашњени конкретни имплементациони аспекти, укључујући структуру микросервиса као и начин организовања база података.

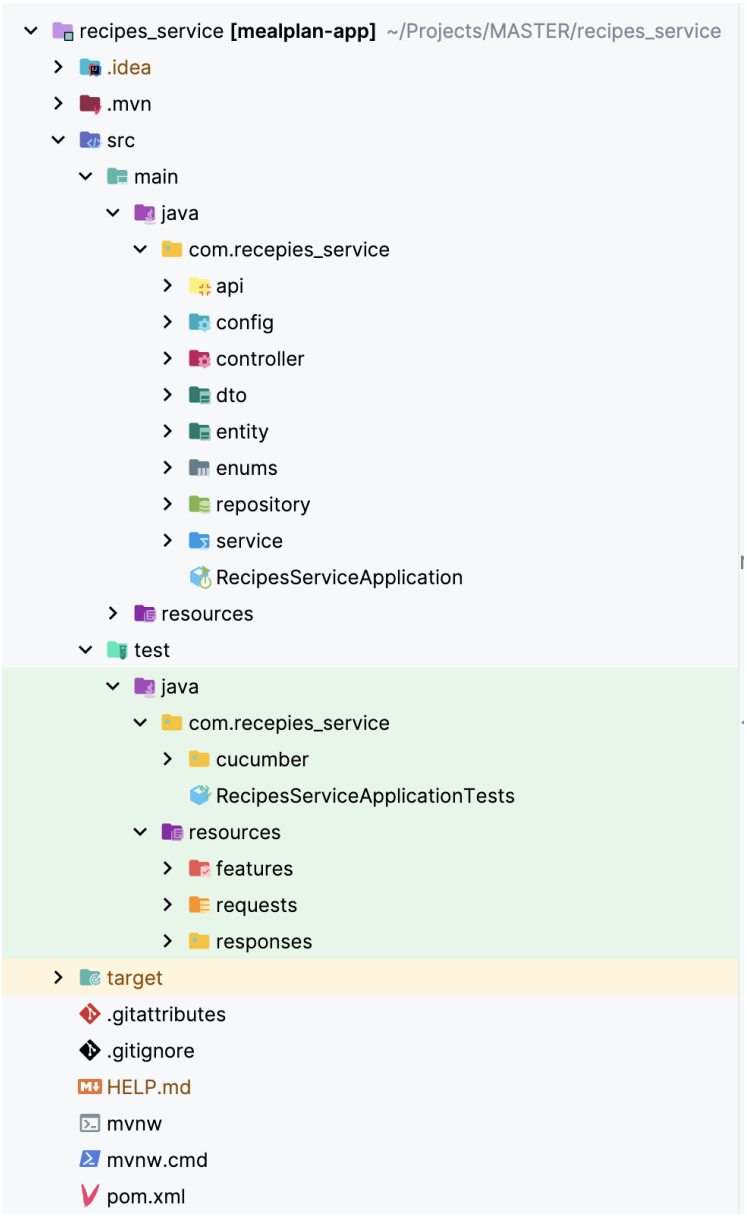
5.1.1 Дизајн микросервиса

Серверски део апликације састоји се од четири независна микросервиса: *Users*, *Ingredients*, *Recipes* и *Plans*. Сви микросервиси су организовани по тролојној архитектури која обухвата слојеве контролера, сервиса и репозиторијума и садрже ентитете, који представљају табеле у бази података, објекте који служе за пренос података између слојева апликације (енгл. *Data Transfer Object*, скраћено *DTO*) и корисничког интерфејса и енуме за дефинисање фиксних скупова вредности, као што су категорије састојака и рецепата. У апликацији је коришћена *PostgreSQL* [17] база података јер је погодна за рад са микросервисним апликацијама, омогућава једноставно одржавање и управљање подацима и лако се интегрише са различитим алатима.

Поред овога, сваки микросервис садржи јасно дефинисан *API* интерфејс, који омогућава клијентском делу апликације или другим микросервисима да комуницирају са њим на стандардан начин преко *HTTP* захтева. За лакше коришћење и преглед функционалности, сваки API је документован помоћу алата *Swagger* [27]. *Swagger* је алат за документацију и тестирање *REST API-ja*. Он омогућава да сваки јавни интерфејс микросервиса буде јасно описан – укључујући податке о улазним и повратним вредностима, као и о могућим грешкама. *Swagger* генерише интерактиван веб интерфејс где се могу видети сви доступни *API* позиви, њихови параметри и пример одговора. Позиви се могу и директно тестирати. Ово представља додатну документацију апликације која знатно олакшава развој, тестирање и коришћење *API-ja*.

У сваком микросервису апликације дефинисан је основни ниво безбедности. Конфигурисан је *CORS* филтер који омогућава контролисан проток података између различитих сервиса и фронтенда. Поред тога, конфигурација безбедности омогућава приступ искључиво ресурсима који су неопходни за функционисање апликације, док је остали приступ подразумевано ограничен. На овај начин обезбеђен је стабилан основ за контролисан приступ ресурсима и постављена је база за даље унапређење безбедности без утицаја на тренутну функционалност сервиса.

На слици 5.1 представљено је стабло микросервиса *Recipes* које одговара и свим осталим микросервисима. Микросервиси садрже и тестове развијене помоћу алата *Cucumber*, што омогућава верификацију функционалности кроз развој вођен понашањем и осигурава да апликација исправно реагује на све дефинисане сценарије. Ови тестови се налазе у директоријуму *test*.



Слика 5.1: Приказ стабла микросервиса

Сервис *Users*

Сервис *Users* [12], односно сервис корисника, задужен је за управљање подацима о корисницима система. У оквиру базе података постоји једна табела у којој се чувају сви релевантни подаци о кориснику, као што су корисничко име, лозинка, године старости, висина, тренутна и циљна телесна тежина, као и информације о полу и нивоу физичке активности. На основу ових вредности израчунава се дневна калоријска потреба сваког корисника, која се такође

чува у бази.

Сервис обезбеђује интерфејс кроз који је могуће креирати новог корисника у систему и извршити његову пријаву. При креирању налога врши се провера да ли корисничко име већ постоји у систему, а лозинка се чува као хеш, који представља сигурну верзију оригиналне лозинке. Такође на основу унетих података рачуна се дневни калоријски унос. У случају пријаве проверава се исправност унетих података, након чега сервис враћа информације о кориснику и вредност дневних калорија која му припада.

Сервис *Ingredients*

Сервис *Ingredients* [8] омогућава корисницима да евидентирају и управљају својим састојцима који се користе приликом креирања рецепата и планова исхране. Сви подаци у оквиру овог сервиса везани су за конкретног корисника, тако да је омогућено да сваки корисник има увид искључиво у сопствене састојке. Идентификација се врши преко корисничког имена (енгл. *username*, параметар `username`), које служи као основни параметар при сваком дохватању података. У бази података овог сервиса налази се једна табела која чува све информације о састојцима.

Основне функционалности сервиса обухватају креирање новог састојка, при чему корисник може да дода састојак у своју личну базу уз унос основних информација попут назива и нутритивних вредности. Сервис такође омогућава дохватање целокупне листе састојака за одређеног корисника на основу његовог корисничког имена. Поред тога, могућа је претрага по називу састојка и корисничком имену како би се издвојио конкретан састојак, као и дохватање више састојака у једном захтеву на основу њихових јединствених идентификатора.

Сервис *Recipes*

Сервис *Recipes* [11] омогућава корисницима креирање, измену и управљање рецептима, који користе састојке евидентиране у *Ingredients* сервису. Сви рецепти су везани за конкретног корисника, тако да сваки корисник има увид искључиво у своје рецепте. Идентификација се врши преко корисничког имена (`username`), које служи као основни параметар при дохватању података. У бази података овог сервиса налазе се две табеле: `recipes`, која чува основне

информације о рецепту, и `recipe_ingredients`, која успоставља везу између рецепата и састојака.

Основне функционалности сервиса обухватају креирање новог рецепта, при чему корисник уноси основне информације о рецепту и повезује га са једним или више састојака. Поред тога, корисник може да мења постојеће рецепте или да их обрише. Приликом креирања или измене рецепта, сервис аутоматски израчунава број калорија по порцији на основу састојака. Сервис такође омогућава дохватање целокупне листе рецепата за одређеног корисника на основу његовог корисничког имена.

Сервис *Plans*

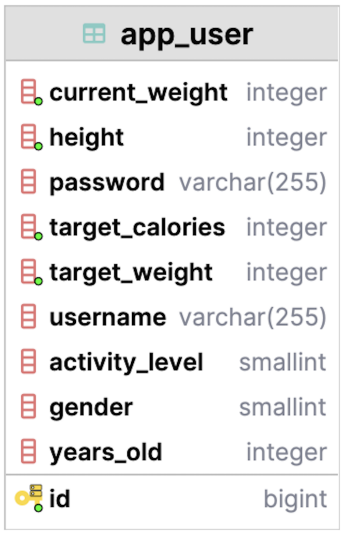
Сервис *Plans* [10] омогућава корисницима креирање и управљање плановима исхране за одређени датум. Сви планови су повезани са конкретним корисником преко његовог корисничког имена (`username`), што осигурава да сваки корисник има увид само у своје планове. У бази података налази се једна табела која чува податке о плановима, док додатна табела успоставља везу између плана и рецепата из сервиса *Recipes*. Ови рецепти садрже информације потребне за аутоматско израчунавање укупног броја калорија потрошених по плану.

Основне функционалности сервиса обухватају креирање новог плана и измену постојећег плана за одређени датум, као и дохватање планова на основу корисничког имена и датума. Приликом креирања или измене плана, сервис користи податке о рецептама да израчуна укупни број калорија који је предвиђен планом. Овај приступ омогућава корисницима да једноставно планирају исхрану и прате број планираних калорија.

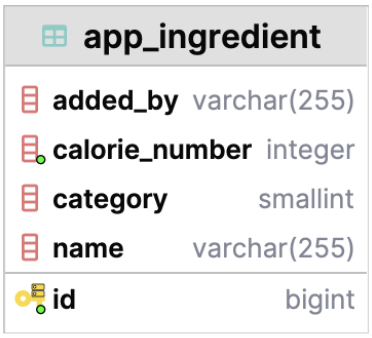
Дијаграми табела из базе података приказани су на сликама 5.2, 5.3, 5.4 и 5.5.

5.1.2 Комуникација између сервиса

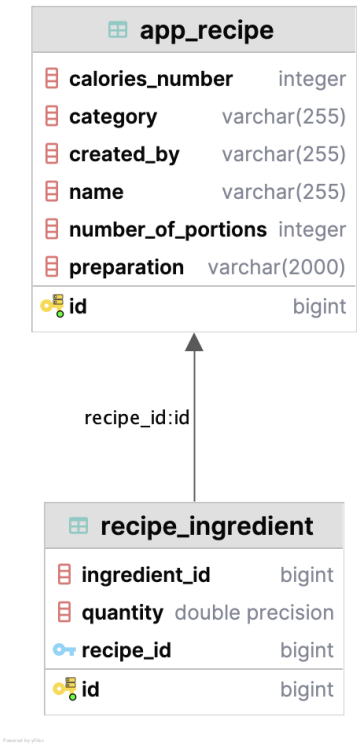
За међусобну комуникацију микросервиса у апликацији користи се компонента *RestTemplate* из развојног оквира *Spring*. Ова компонента омогућава слање синхроних *HTTP* захтева ка другим *REST* сервисима. *RestTemplate* је једноставан за коришћење, интегрисан са *Spring-ом* и погодан за пренос података између сервиса без потребе за сложенијим клијентским библиотекама.



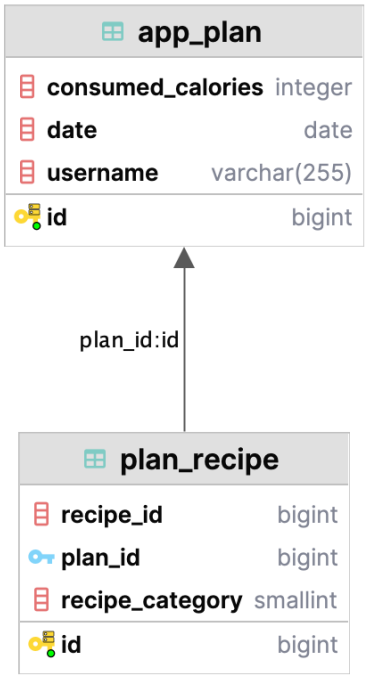
Слика 5.2: Корисници



Слика 5.3: Састојци



Слика 5.4: Рецепти



Слика 5.5: Планови

Овде се користи јер омогућава брзу и директну размену информација између сервиса који управљају рецептима, састојцима и плановима.

За конкретну имплементацију комуникације креирани су *IngredientClient* и

RecipeClient. Ови клијенти представљају специјализоване класе чија је сврха да инкапсулирају логику комуникације са другим микросервисом преко *HTTP* захтева. На тај начин се обезбеђује апстракција и поједностављује употреба спољних сервиса. *IngredientClient* омогућава сервису за рецепте да приступа приступним тачкама сервиса за састојке и добије податке о састојцима по њиховим идентификаторима или по имену и корисничком налогу. Слично, *RecipeClient* омогућава сервису за планове да позива сервис за рецепте и израчуна укупан број калорија једног или више рецепата у плану.

Као што је приказано на претходним дијаграмима, у базама података постоје додатне табеле које чувају везу између рецепата и састојака, као и рецепата и планова. Ове табеле омогућавају прецизно праћење калорија и правилну размену података између сервиса преко *REST* позива.

5.1.3 Клијентска апликација

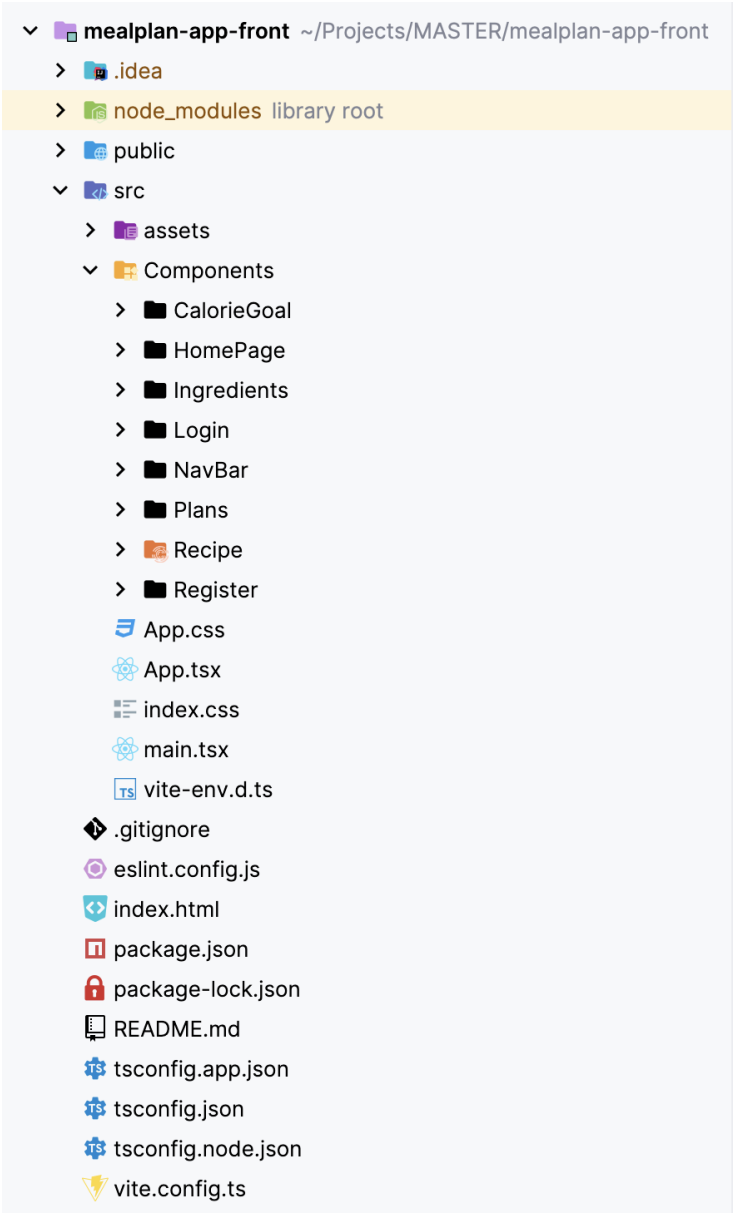
Клијентска апликација [7] је развијена коришћењем библиотеке *React* и организована је кроз више компоненти које представљају различите целине апликације. Свака компонента садржи одговарајућу логику и визуелни приказ дела апликације, а стилови су реализовани помоћу *CSS*-а.

Компоненте омогућавају јасну поделу функционалности и лаку одрживост апликације, док *React Router* обезбеђује приказ одговарајуће странице у зависности од *URL* путање. Апликација обухвата следеће компоненте:

- *HomePage* — почетна страница на којој се налазе информације о апликацији
- *Login* — приказ форме за пријаву корисника на систем.
- *Register* — приказ форме за регистрацију нових корисника.
- *CalorieGoal* — приказ корисниковог циљаног дневног уноса калорија.
- *IngredientsPage* — приказ листе унетих састојака и могућност додавања нових.
- *RecipesPage* — приказ листе унетих рецепата, могућност додавања нових и промене или брисања већ постојећих

- *PlansPage* — приказ или измена планова исхране за одабрани датум уз праћење броја испланираних калорија
- *NavBar* — навигација између страница са плановима, састојцима и рецептима

Структура директоријума клијентске апликације приказана је на слици 5.6. Графички приказ корисничког интерфејса и примери интеракције корисника са апликацијом биће представљени у наредном поглављу.



Слика 5.6: Стабло директоријума клијентске апликације

5.2 Намена апликације и случајеви употребе

Апликација *Plan&Prep* осмишљена је као подршка корисницима у организовању свакодневне исхране. Њена основна сврха јесте да омогући евидентирање састојака, креирање рецепата и израду планова исхране у складу са индивидуалним калоријским потребама. На основу података које корисник уноси приликом регистрације, систем аутоматски израчунава његов циљани дневни калоријски унос, што омогућава персонализовано планирање. Апликација је организована тако да кроз навигациони мени омогућава једноставан приступ свим главним функционалностима: састојцима, рецептима и плановима исхране. У наставку су приказани случајеви употребе апликације.

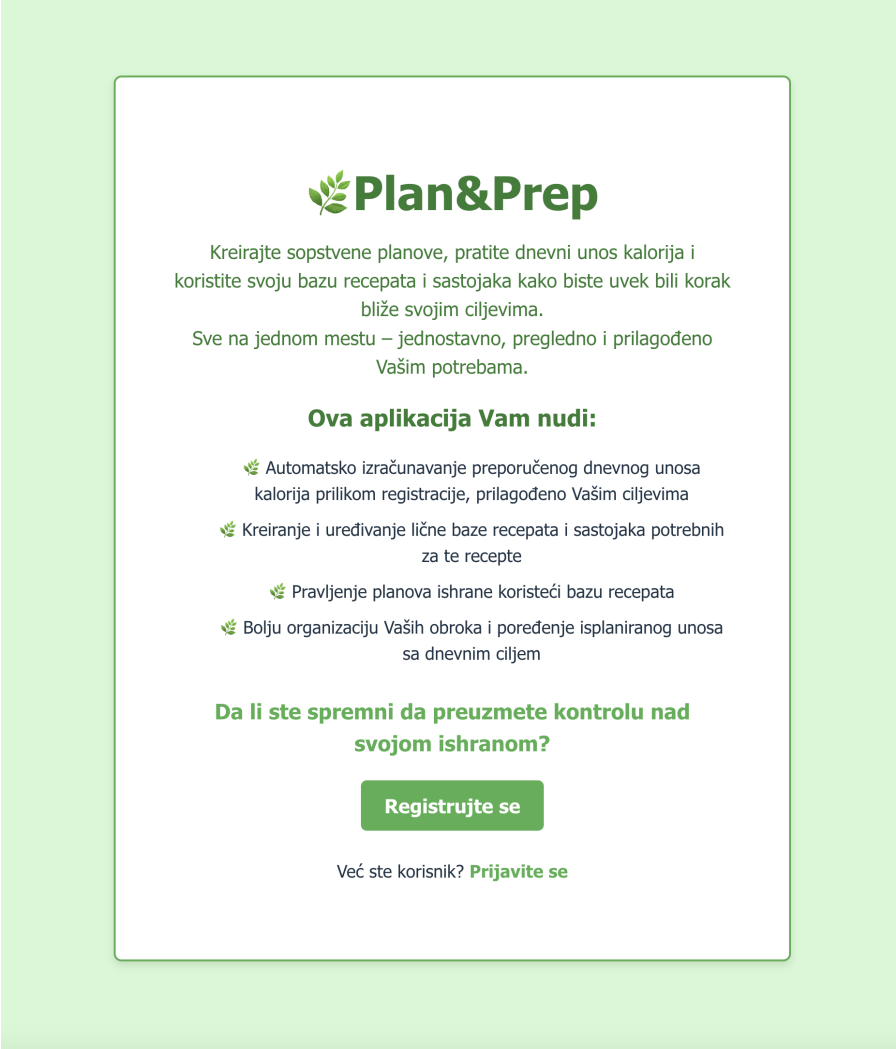
5.2.1 Регистрација и пријављивање

Почетна страна апликације нуди будућим корисницима основне информације о томе чему је апликација намењена и зашто им може бити корисна. Изглед ове странице приказан је на слици 5.7. Корисник има опцију да се региструје, или пријави уколико је већ корисник апликације.

Слика 5.8 приказује екран за регистровање корисника. Приликом регистрације уносе се основни параметри — корисничко име, лозинка, висина, године, тренутна тежина, циљана тежина, ниво физичке активности и пол. Корисничко име је јединствено у бази корисника и уколико корисник унесе име које у њој већ постоји добиће поруку о грешци као на слици 5.9.

Коришћењем података који су унети приликом регистрације израчунава се циљани дневни унос калорија корисника. На слици 5.10 приказано је како изгледа информација о препорученом дневном уносу корисника, која се приказује након успешне регистрације. За потребе израчунавања примењује се Мифлин–Сент Џеорова формула (енг. *Mifflin-St Jeor equation*), која се у литератури сматра једном од најпоузданијих метода за одређивање базалног метаболизма. Базални метаболизам представља количину енергије коју организам троши у стању мировања ради одржавања основних животних функција као што су дисање, рад срца и регулација телесне температуре. Облик формуле за мушкарце приказан је у једначини 5.1, док је облик формуле за жене приказан у једначини 5.2:

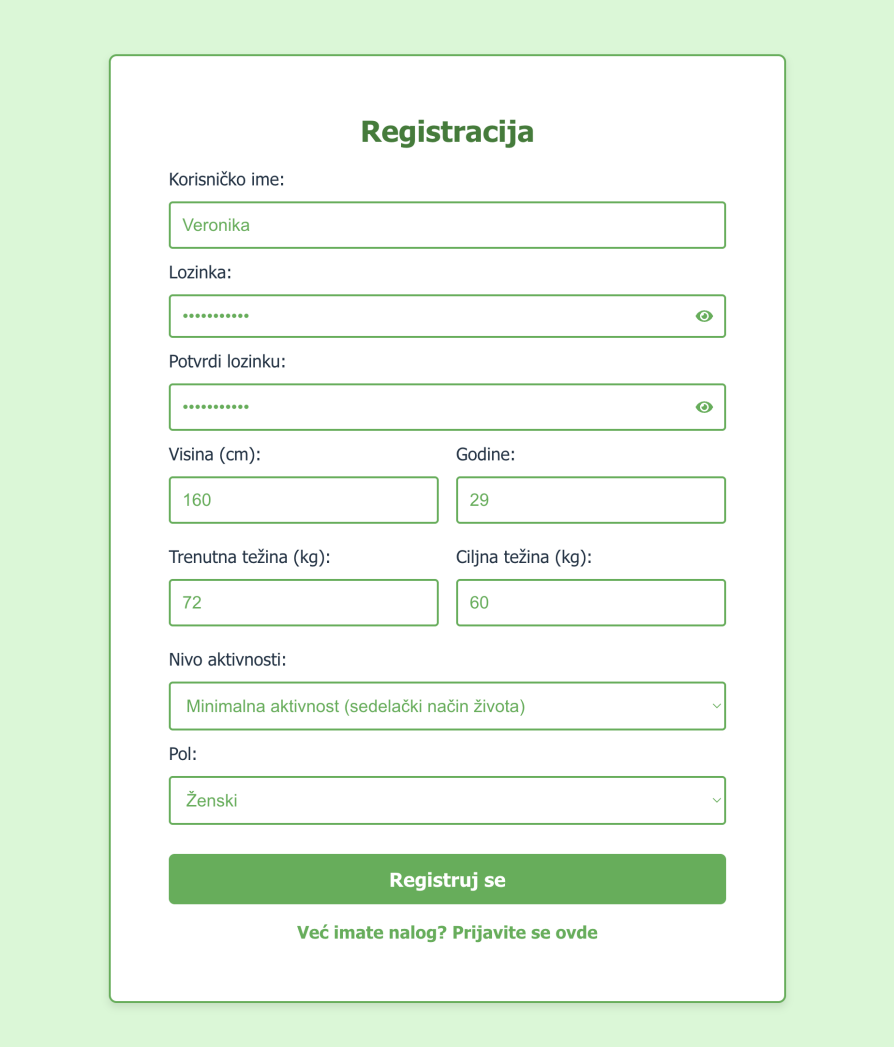
$$BMR_{\text{мушкарац}} = 10 \times \text{тежина (kg)} + 6.25 \times \text{висина (cm)} - 5 \times \text{године} + 5 \quad (5.1)$$



Слика 5.7: Почетна страна

$$BMR_{\text{жена}} = 10 \times \text{тежина (kg)} + 6.25 \times \text{висина (cm)} - 5 \times \text{године} - 161 \quad (5.2)$$

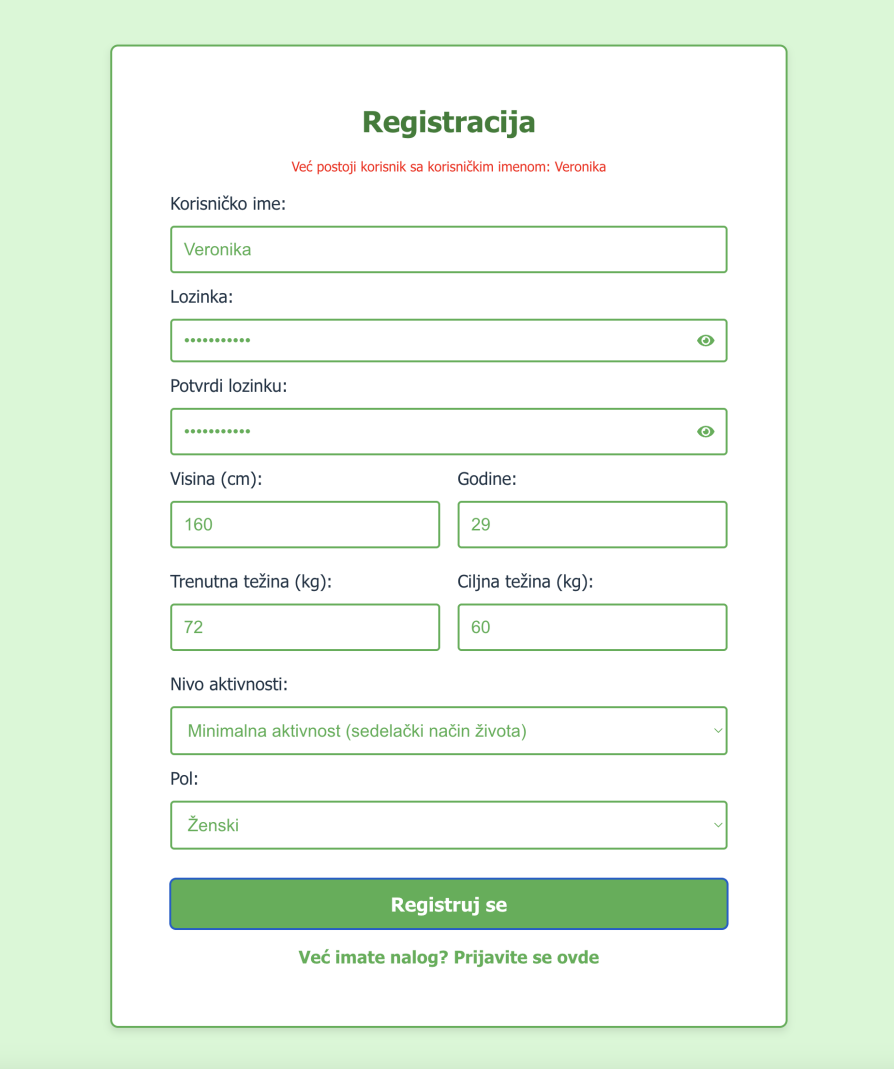
Након израчунавања базалног метаболизма, добијена вредност се множи одговарајућим коефицијентом физичке активности како би се добила процена укупне дневне потрошње енергије. Поред тога, укупни број калорија додатно се прилагођава у складу са циљем који корисник жели да постигне. Уколико је циљ смањење телесне тежине, од добијене вредности се одузима одређени калоријски дефицит (вредност се умањује за 20%), док се у случају повећања телесне масе додаје калоријски суфицит (вредност се повећава 15%). Уколико корисник жели да одржи постојећу телесну тежину, задржава се



Слика 5.8: Екран за регистрацију

израчуната вредност без додатних корекција. Укључивањем Мифлин–Сент Џеорове формуле у функционалности апликације обезбеђује се научно утемељен и персонализован приступ планирању исхране, који узима у обзир не само физиолошке карактеристике већ и индивидуалне циљеве корисника.

На слици 5.11 је приказан екран за пријављивање где корисник уноси корисничко име и лозинку. Предуслов за овај случај употребе је да корисник већ има налог у апликацији. Уколико лозинка није исправна или је унето погрешно корисничко име приказаће се порука о грешци као на сликама 5.12 и 5.13.

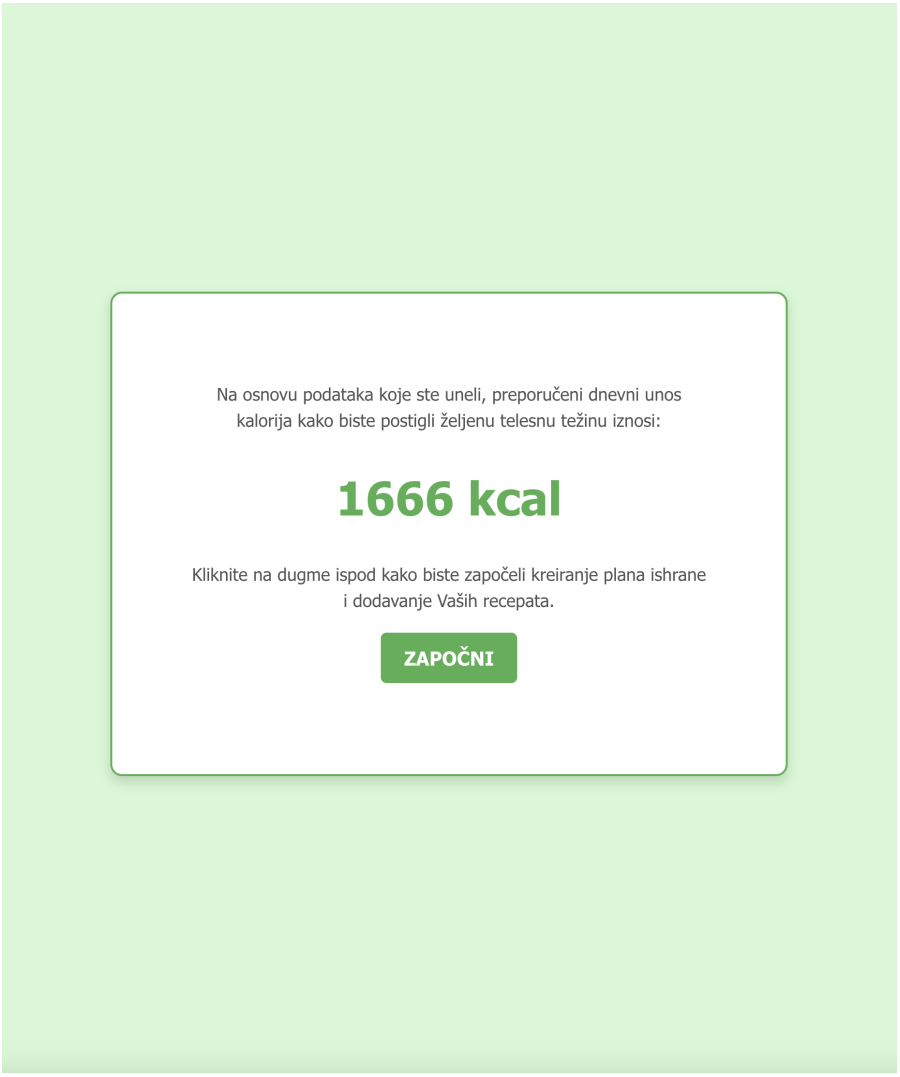


Слика 5.9: Екран за регистрацију - корисничко име које већ постоји

5.2.2 Навигација кроз апликацију и одјављивање

Након што је корисник успешно пријављен отвара се почетни екран апликације. На врху се све време док је корисник пријављен налази навигациони мени помоћу ког се може мењати тренутно активна страница. Корисник може навигирати између страница за планове, рецепте и састојке.

У горњем дестом углу навигационог менија приказана је иконица профила. Кликом на ту иконицу може се видети корисничко име са којим је корисник тренутно пријављен као и дугме *Odjavite se*. Кликом на ово дугме корисник ће бити одјављен из апликације и преусмерен на страницу за пријављивање.



Слика 5.10: Приказ циљаног броја калорија

На слици 5.14 приказан је навигациони мени као и падајући мени са дугметом за одјављивање.

5.2.3 Управљање састојцима

Предуслов овог случаја употребе јесте да је корисник пријављен на апликацију. Апликација омогућава кориснику да води личну базу састојака. На страници је приказана листа свих састојака који су доступни у бази и основни подаци о њима. Корисник може да претражује састојке по имену и филтрира их према категоријама, чиме се олакшава управљање намирницама које има на располагању. Такође може их и сортирати по свакој од колона. На слици

Слика 5.11: Екран за пријаву

5.15 је приказана страница са листом састојака.

Корисник има могућност додавања новог састојка. Име рецепта је јединствено. На слици 5.16 је приказан модал[1] за додавање састојка у који корисник уноси име састојка, број калорија тог састојка на сто грама и категорију којој тај састојак припада.

5.2.4 Управљање рецептима

Предуслов овог случаја употребе јесте да је корисник пријављен на апликацију. Апликација омогућава кориснику да води личну базу рецепата. На

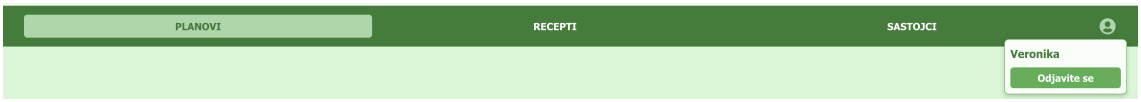
[1]Модал је прозор који се појављује преко главног садржаја апликације и онемогућава интеракцију са осталим деловима док се не затвори или док се не изврши одређена радња.

Слика 5.12: Погрешна лозинка

страници је приказана листа свих рецепата који су доступни у бази и основни подаци о њима. Корисник може да претражује рецепте по имену и филтрира их према категоријама. Такође може их и сортирати по свакој од колона. На слици 5.17 је приказана страница са листом рецепата.

Корисник може да креира рецепте комбинацијом састојака из сопствене базе. Слика 5.18 приказује модал који се отвори кликом на дугме за додавање рецепта. Потребно је унети назив рецепта, број порција овог рецепта која се добије од наведене количине састојака, категорија којој рецепт припада, листа састојака и начин припреме. Уколико је резултат претраге за жељени састојак празан, односно састојак не постоји у корисниковој бази, корисник има могућност да га дода директно у рецепт, као на слици 5.19. Кликом

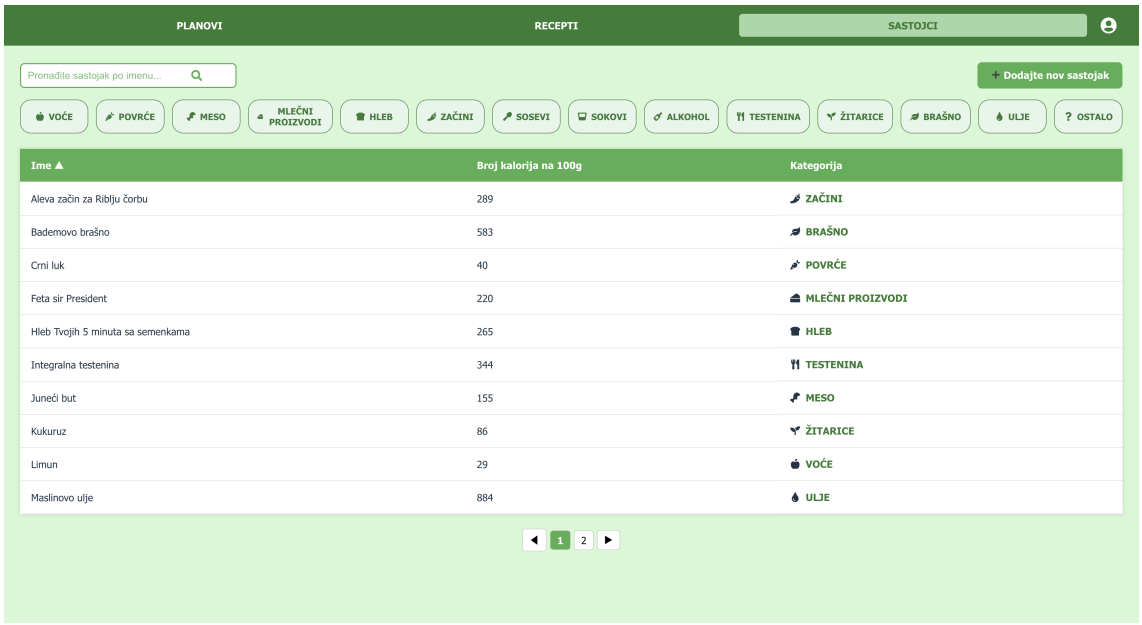
Слика 5.13: Корисник не постоји



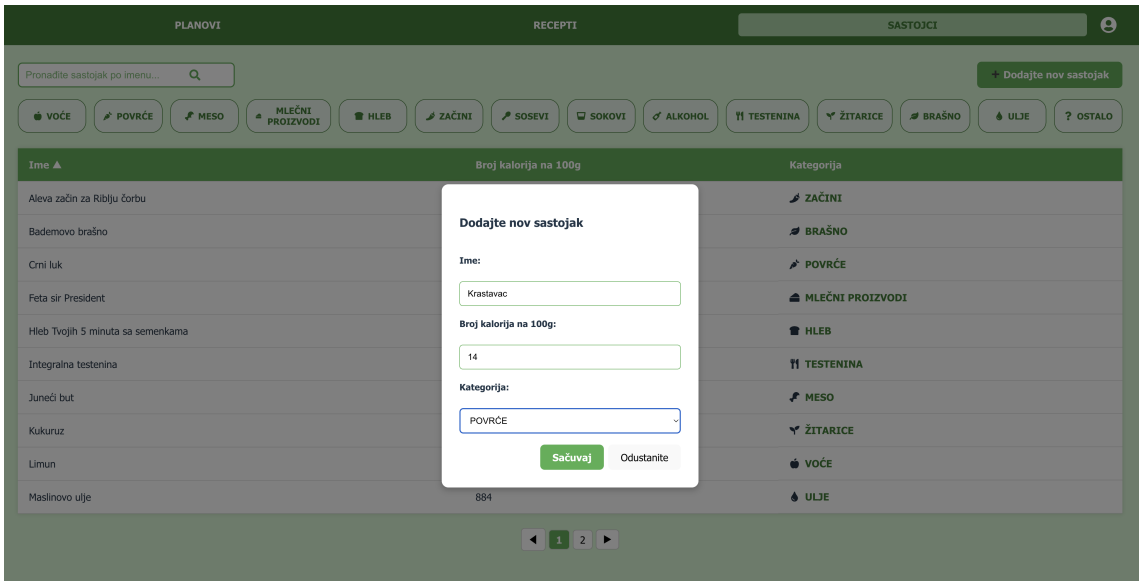
Слика 5.14: Навигациони мени и одјава

на дугме *Dodaj sastojak* отвара се модал за креирање новог састојка. На основу количине и калоријске вредности употребљених састојака апликација аутоматски израчунава калоријску вредност рецепта по порцији.

Кликом на симбол канте, рецепти се могу брисати уколико више нису потребни. Кликом на симбол ока, отвара се модал са слике 5.20 који приказује детаље тог рецепта. У горњем десном углу овог модала налази се симбол оловке. Кликом на њега отвара се модал за промену рецепта који изгледа исто

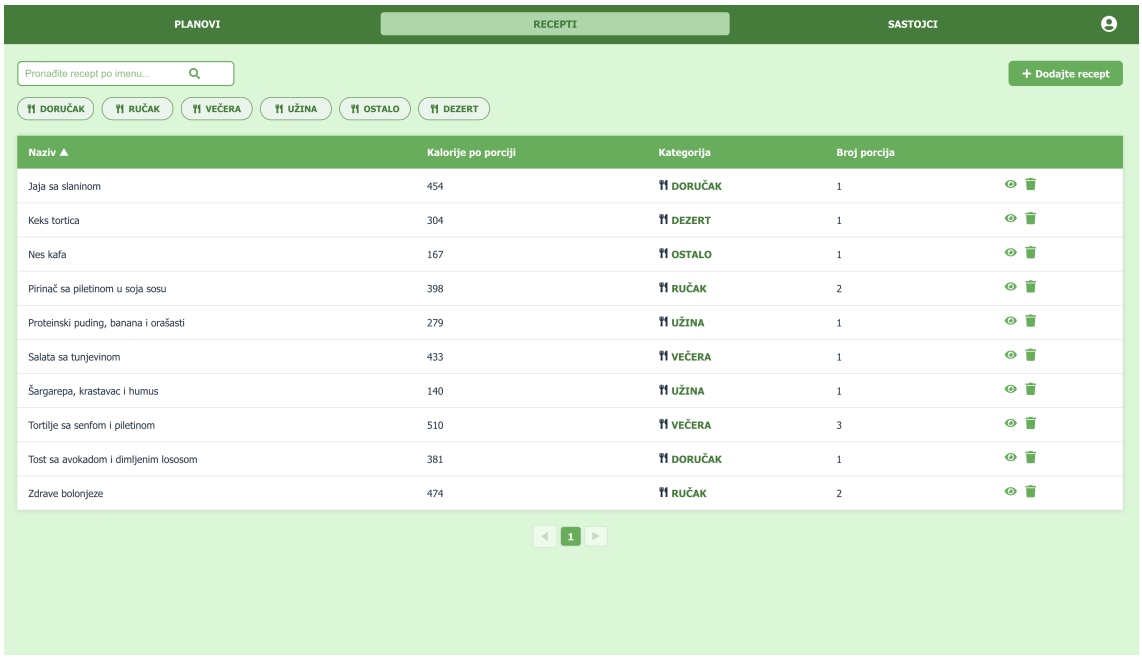


Слика 5.15: Приказ листе састојака

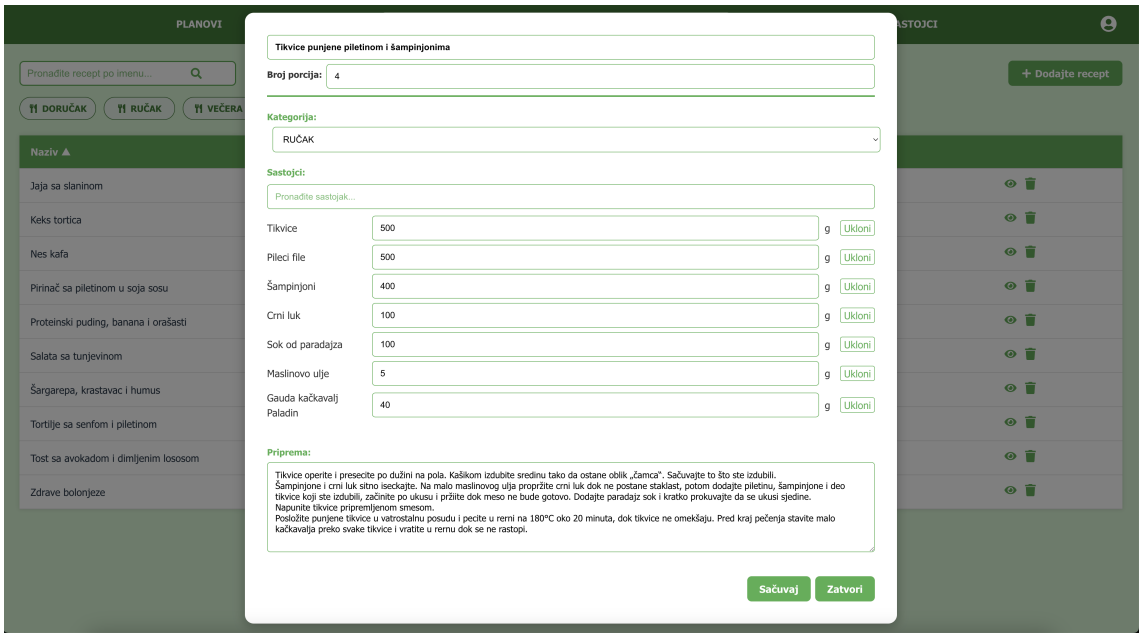


Слика 5.16: Додавање састојка

као и модал за креирање, али овог пута са унапред попуњеним подацима из рецепта који мењате. Овај модал приказан је на слици 5.21. Име рецепта није могуће мењати, а број калорија се поново рачуна користећи нове информације из рецепта.



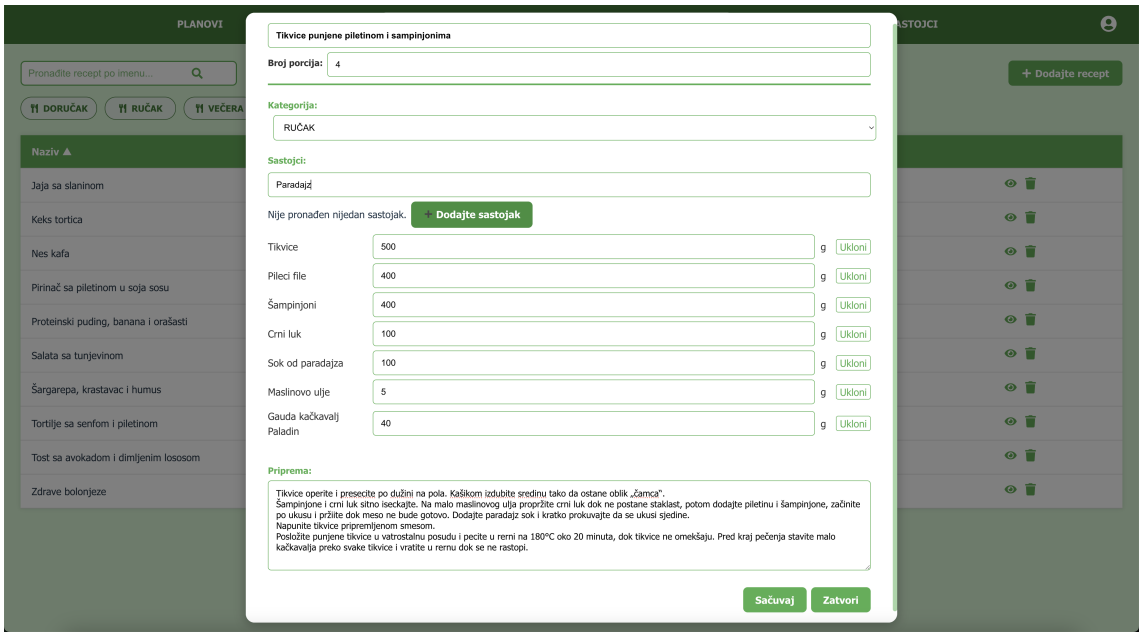
Слика 5.17: Листа рецепата



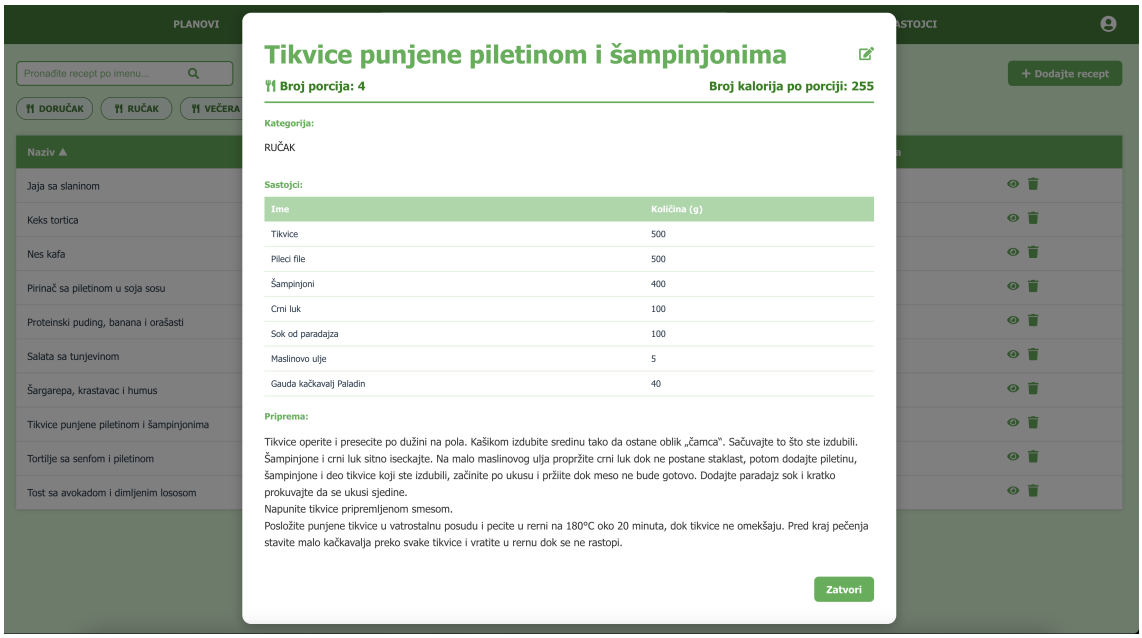
Слика 5.18: Модал за додавање рецепта

5.2.5 Креирање планова исхране

Кључна функционалност апликације јесте могућност креирања планова исхране. Предуслов за овај случај употребе је да је корисник пријављен у апликацију, да већ има потребне рецепте креиране у бази и да је познат податак



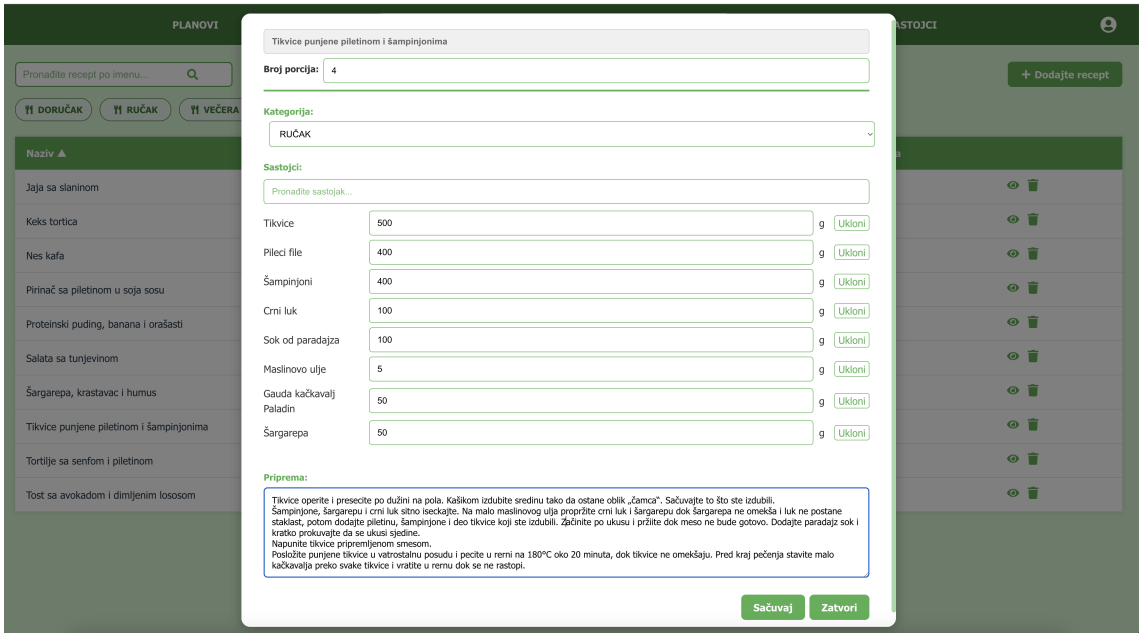
Слика 5.19: Састојак за рецепт не постоји



Слика 5.20: Модал са детаљима рецепта

који је циљани број калорија у дану. На почетном екрану за планирање (Слика 5.22) приказан је изабрани датум и план за тај дан.

Систем аутоматски рачуна укупан број калорија који је већ испланиран и одузима га од укупног дневног циља. На тај начин корисник у сваком тренутку има увид у то колико је калорија преостало за унос. Уколико је корисник



Слика 5.21: Модал за промену рецепта

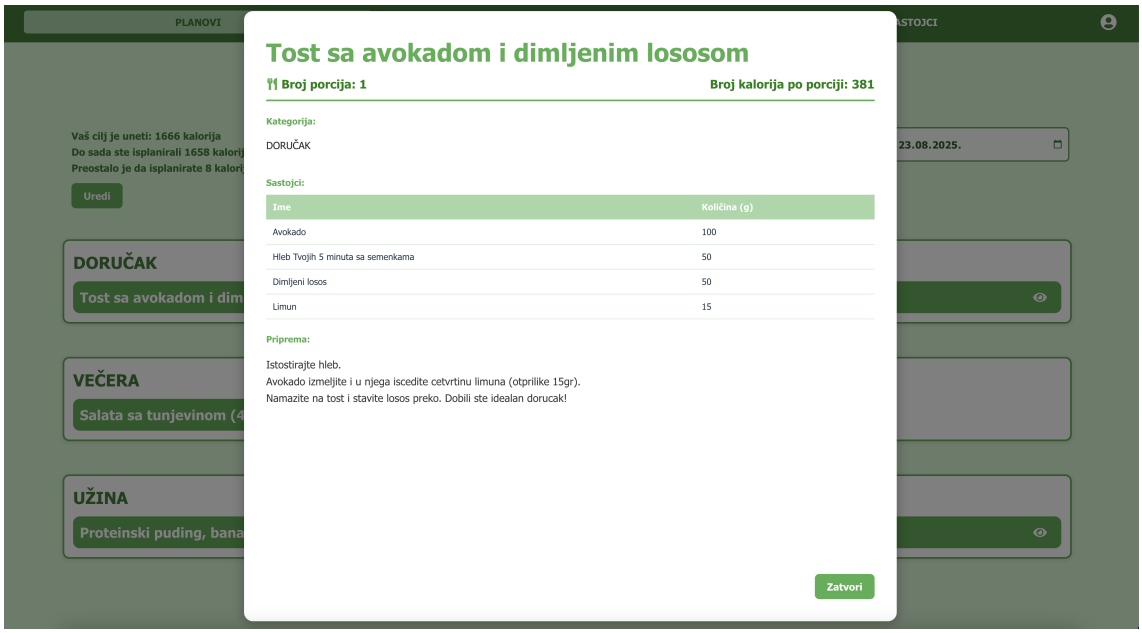
премашио планирани број калорија добиће поруку о томе, као на слици 5.23.

Слика 5.22: Приказ плана оброка за одабрани датум

Осим тога, омогућено је да корисник у сваком тренутку прегледа детаље рецепта који се налази у плану кликом на симбол ока, након чега се отвара приказ као на слици 5.24. За сваки дан у садашњости или будућности корисник има активно дугме *Uredi* и кликом на њега отвара му се могућност да додаје

Слика 5.23: Информација о премашеном дневном уносу калорија

или брише рецепте за појединачне оброке (Слика 5.25). Планови из прошлости доступни су за преглед, али је ово дугме неактивно и они се не могу више прилагођавати.



Слика 5.24: Приказ одабраног рецепта

Слика 5.25: Екран за промену плана за одабрани датум

Глава 6

Примена *BDD* у развоју апликације *Plan&Prep*

У овој глави биће приказано како је апликација *Plan&Prep* развијана коришћењем приступа развоја вођеног понашањем. Циљ ове главе је показати све фазе примене *BDD* приступа — од идентификације пословних потреба и дефинисања сценарија, преко имплементације микросервиса, до интеграције алата и праћења извештаја.

6.1 Дефинисање пословних функционалности

Прва и једна од најважнијих фаза развоја апликације *Plan&Prep* била је идентификација пословних функционалности које сваки микросервис треба да реализује. Уместо да се приступа развоју са техничког аспекта, као што је традиционално у класичном *TDD* приступу, овај процес је започет са пословне стране, са фокусом на потребе крајњих корисника и стварне проблеме које апликација треба да реши.

Пословни аспект подразумевао је да се свака функционалност разматра кроз очекивања корисника и стварне пословне захтеве. Затим су функционалности груписане по микросервисима, а за сваки микросервис написани су детаљни описи очекиваног понашања, укључујући могуће сценарије грешака или необичних ситуација. Ови описи обухватају типичне корисничке операције, услове успешног завршетка, као и случајеве када корисник унесе нетачне податке или дође до конфликта у апликацији.

За сваки микросервис дефинисани су сценарији у *Gherkin* синтакси који

описују понашање система из угла корисника, и који су касније коришћени као основа за *Cucumber* тестове. Пример једног таквог сценарија за проналажење плана на основу корисничког имена и датума приказан је у листингу 6.1.

```
Feature: Get a plan by date and username

  Background:
    Given the system is initialized

  @getByDateAndUsername
  Scenario: Successfully retrieve a plan by date and username
    When I send a GET request to "/plans?username=Admin&date
       =2025-08-28"
    Then I should receive a 200 response
    And the plan response should be like the plan data in JSON
       file "/responses/get_plan_by_name_and_username.json"
```

Листинг 6.1: Сценарио проналажења плана на основу корисничког имена и датума

Сви сценарији су формулисани тако да могу бити директно искоришћени у оквиру *Cucumber* тестова, чиме је успостављена трајна веза између пословне логике и имплементације. У наставку је приказано шта обухватају сценарији, организовани по микросервисима.

Сервис за кориснике — Сценарији обухватају:

- успешна пријава корисника са важећим корисничким именом и лозинком;
- руковање неуспешним покушајима пријаве — укључује случајеве неисправне лозинке, непостојећег корисничког имена, као и недостајућих обавезних поља;
- регистрација новог корисника са различитим комбинацијама параметара као што су пол, ниво активности, циљна тежина и слично;
- валидација података током регистрације — обухвата проверу да ли корисничко име већ постоји, као и неважеће вредности за корисничко име, лозинку, висину, тежину и године;

Сервис за састојке — Сценарији обухватају:

- додавање новог састојка уз проверу да ли је име састојка унето, да број калорија буде већи од нуле и да име састојка није већ у употреби;
- преглед свих састојака повезаних са корисничким налогом, укључујући случај када корисник нема ниједан састојак;
- проналажење састојака на основу листе идентификатора;
- претрага састојака по имену и корисничком налогу, као и руковање ситуацијама када састојак није пронађен;

Сервис за рецепте — Сценарији обухватају:

- креирање рецепта са валидним подацима и руковање грешкама када име није унето или већ постојећи назив рецепта;
- брисање рецепта по имену или идентификатору уз обраду грешке када рецепт не постоји;
- проналажење свих рецепата за одређено корисничко име, укључујући и случај када корисник нема рецепте;
- проналажење свих рецепата у систему, укључујући и случај када нема рецепата;
- израчунавање укупних калорија на основу прослеђених идентификатора рецепата;
- проналажење рецепта по имену уз обраду грешке када рецепт не постоји;
- проналажење броја калорија за рецепт на основу идентификатора;
- ажурирање рецепта по имену или идентификатору, укључујући и руковање грешкама уколико рецепт не постоји;

Сервис за планове — Сценарији обухватају:

- креирање дневног плана са једним или више рецепата уз аутоматско израчунавање калоријске вредности плана;
- ажурирање постојећег плана уз аутоматско израчунавање калоријске вредности плана;

- проналажење плана по датуму и корисничком имену, са руковањем случајевима када план не постоји;
- управљање ситуацијом у којој неки од рецепата у плану више не постоји у бази рецепата - у том случају аутоматски се уклањају референце на обрисане рецепте, чиме се одржава конзистентност података;

6.2 Имплементација на основу сценарија

Након дефинисања пословних функционалности и креирања *Gherkin* сценарија, развој апликације *Plan&Prep* настављен је имплементацијом микросервиса на основу тих сценарија. Сценарији прецизно описују очекивано понашање апликације у различитим ситуацијама, што омогућава да сваки корак у пословном процесу буде директно повезан са конкретном имплементацијом у коду.

Сваки микросервис користи два конфигурациона профила: један за развој и један за *Cucumber* тестове. За сваки профил, сваки микросервис има своју независну базу података са идентичном структуром табела, што омогућава потпуно одвајање тестних података од података за развој. На овај начин, сви позиви на репозиторијуме у тестовима се понашају идентично као у реалном развојном окружењу, чиме се обезбеђује додатна прецизност и поузданост тестова. Ово решење омогућава да се тестови изводе без ризика од утицаја на развојне податке, а истовремено гарантује да тестови верно симулирају понашање апликације у реалним условима.

Претварање сценарија у имплементацију подразумева идентификацију кључних корака сценарија и њихово мапирање на методе и класе у Јава микросервисима. Сваки корак *Given*, *When* или *Then* у сценарију добија своју реализацију у класи са дефиницијом корака, која позива одговарајуће сервисе апликације. На тај начин сваки корак сценарија је директно повезан са конкретним кодом, а *Cucumber* тестови обезбеђују аутоматску проверу исправности имплементације.

Пример сценарија и имплементације

Сценарији у *Gherkin* формату служе као водич за имплементацију функционалности у микросервисима. Следећи пример илуструје процес креирања новог састојка у апликацији *Plan&Prep*. У овом примеру, сценарио у *Gherkin* формату приказује очекивано понашање апликације приликом креирања новог састојка. Сценарио је приказан у листингу 6.2 и дефинише следеће кораке:

Припрема података: Апликација очекује да се у *JSON* формату доставе подаци о новом састојку, као што су име, број калорија на 100 грама, категорија и корисник који додаје састојак. Ови подаци се користе као захтев који се шаље сервису за састојке.

Слање захтева: Када се подаци припреме, апликација шаље *POST* захтев на одговарајућу приступну тачку (ендпоинт) сервиса за састојке. Овим се иницира креирање новог састојка у бази података.

Провера одговора: Очекује се да сервис врати статусни кôд 201, што означава да је нови састојак успешно креиран. Поред тога, садржај одговора мора одговарати подацима који су послати у захтеву – то јест, име, број калорија, категорија и корисник морају бити идентични.

```
1 Feature: Create a new ingredient
2
3   @create
4   Scenario: Successfully create a new ingredient
5     Given the ingredient data in JSON file "/requests/
          create_ingredient_request.json"
6     When I send a POST request to "/ingredients"
7     Then I should receive a 201 response
8     And the ingredient response should be like the ingredient
          data in JSON file "/responses/create_ingredient_response.
          json"
```

Листинг 6.2: Сценарио креирања новог састојка

Фајлови који се користе у корацима сценарија садрже податке који се очекују приликом слања захтева и примања одговора од *API-ја*. Очекујемо да

тело захтева и одговора садрже идентичне информације, приказане у листингу 6.3.

```
{
  "name": "Apple",
  "calorieNumber": 20,
  "addedBy": "Admin",
  "category": "FRUIT"
}
```

Листинг 6.3: Тело захтева и одговора приказано у *json* формату

Сценарио се реализује кроз дефиниције корака у Јава коду приказане у листингу 6.4.

```
@Given("the ingredient data in JSON file {string}")
public void theIngredientDataInJsonFile(String fileName) throws
    IOException {
     ObjectMapper objectMapper = new ObjectMapper();
     ingredientDTO =
         objectMapper.readValue(
             new File(RESOURCES_PATH + fileName), IngredientDTO.
                 class);
}

@When("I send a POST request to {string}")
public void iSendAPostRequestTo(String path) {
     String url = BASE_PATH + path;
     try {
         response = restTemplate.postForEntity(url, ingredientDTO
             , String.class);
     } catch (HttpClientErrorException | HttpServerErrorException
         e) {
         response = ResponseEntity.status(e.getStatusCode()).body
             (e.getResponseBodyAsString());
     }
}

@Then("I should receive a {int} response")
```

```
20 public void iShouldReceiveAResponse(int statusCode) {
21     assertEquals(HttpStatus.valueOf(statusCode), response.
           getStatusCode());
22 }
23
24 @And("the ingredient response should be like the ingredient data
       in JSON file {string}")
25 public void
       theIngredientResponseShouldBeLikeTheIngredientDataInJsonFile(
       String fileName)
26     throws IOException {
27     ObjectMapper objectMapper = new ObjectMapper();
28     IngredientDTO expectedIngredientDTO =
29         objectMapper.readValue(
30             new File(RESOURCES_PATH + fileName), IngredientDTO.
                   class);
31     IngredientDTO actualIngredientDTO =
32         objectMapper.readValue(response.getBody(), IngredientDTO
               .class);
33
34     assertEquals(expectedIngredientDTO.getName(),
           actualIngredientDTO.getName());
35     assertEquals(expectedIngredientDTO.getCalorieNumber(),
           actualIngredientDTO.getCalorieNumber());
36     assertEquals(expectedIngredientDTO.getCategory(),
           actualIngredientDTO.getCategory());
37 }
```

Листинг 6.4: Дефиниције корака из *Cucumber* сценарија

Функционалност унутар микросервиса се имплементира кроз контролер, сервис и репозиторијум који се позива у сервису. Имплементација контролер слоја приказан је у листингу 6.5, док је имплементација сервисног слоја приказана у листингу 6.6.

```
1 @PostMapping
2 public ResponseEntity<IngredientDTO> createIngredient(
      @RequestBody IngredientDTO ingredientDTO)
```

```
3     throws BadRequestException {
4     IngredientDTO response = ingredientService.createIngredient(
          ingredientDTO);
5     return ResponseEntity
6          .status(HttpStatus.CREATED)
7          .body(response);
8 }
```

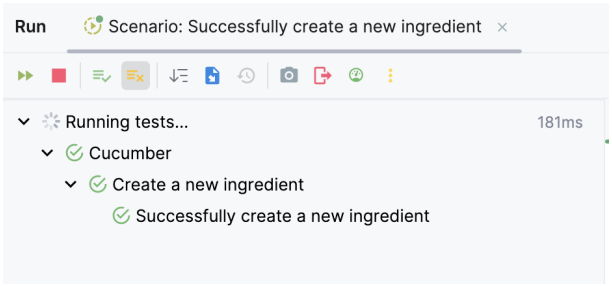
Листинг 6.5: Контролер за креирање састојка

```
1 public IngredientDTO createIngredient(@Valid IngredientDTO
      ingredientDTO)
2     throws BadRequestException {
3     if(ingredientRepository.findByNameAndAddedBy(ingredientDTO.
          getName(), ingredientDTO.getAddedBy()).isPresent()) {
4          throw new BadRequestException(
               INGREDIENT_NAME_ALREADY_EXIST + ingredientDTO.getName
               ());
5     }
6
7     IngredientEntity ingredientEntity = ingredientDTO.
          mapToEntity();
8     IngredientEntity savedEntity = ingredientRepository.save(
          ingredientEntity);
9     return savedEntity.mapToDto();
10 }
```

Листинг 6.6: Сервисни слој за креирање састојка

Приликом покретања теста *Cucumber* нас обавештава да је сценарио прошао и тиме потврђујемо исправност наше имплеметације. Приказ резултата за тест је на слици 6.1

На овај начин је могуће приказати директну везу између сценарија, корака, имплементације и очекиваног формата података за микросервис који управља састојцима.



Слика 6.1: Резултат теста

6.3 Аутоматизовани тестови и извештаји

Једна од најважнијих предности примене алата *Cucumber* јесте могућност аутоматизованог покретања тестова и генерисања извештаја. Овакви извештаји нису корисни само за тим програмера који прате стабилност система, већ представљају и врсту документације која описује функционалности система и начин на који су тестиране. На овај начин извештаји повезују технички и пословни део развоја апликације.

6.3.1 Покретање тестова

Тестови се покрећу уз помоћ алата *Maven*, који је коришћен и за управљање зависностима у целом пројекту, помоћу команде приказане у листингу 6.7.

```
1 mvn clean test
```

Листинг 6.7: Команда која се користи за покретање тестова

Ова команда аутоматски пронађе све *Feature* фајлове, изврши дефинисане кораке и генерише извештај о резултатима тестирања. У случају пада теста, у извештају се јасно приказује у ком сценарију и на којем кораку је дошло до грешке.

На слици 6.2 је приказан део излаза који је исписан када су тестови покренути у сервису за састојке. У првом тесту сви кораци су успешни, док други тест има грешку у `Then` кораку. Из грешке се јасно може видети која је разлика између добијеног и очекиваног резултата (`java.lang.AssertionError: expected:<200 OK> but was:<403 FORBIDDEN>`), на којој линији у класи са дефиницијом корака је дошло до грешке (`IngredientStepDefs.java:91`). Та-

кође имамо и информацију о томе у којој линији *Feature* фајла се налази корак који не пролази (`get_ingredient_by_name.feature:9`).

Слика 6.2: Део резултата покретања свих тестова

6.3.2 Cucumber извештаји

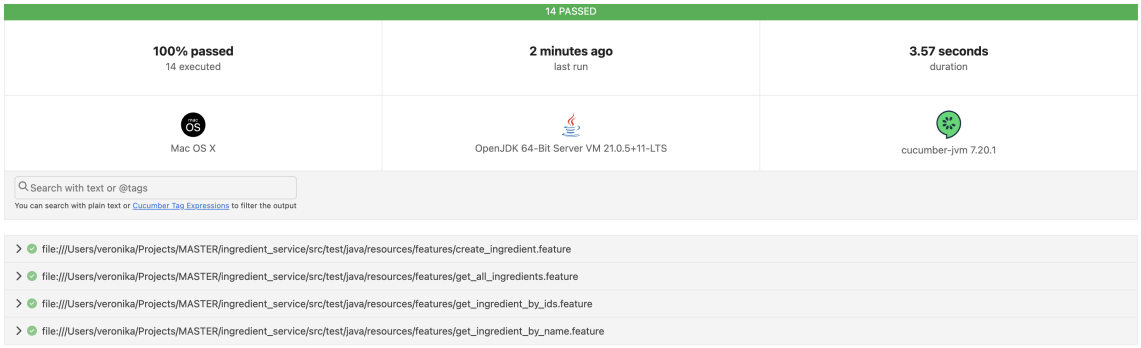
По завршетку тестирања, *Cucumber* генерише извештаје у облику *HTML* или *JSON* фајлова. *HTML* извештај је посебно погодан јер даје визуелни приказ резултата:

- зелена боја означава успешно извршене сценарије,
- црвена боја означава тестове који су се нису успешно извршили,
- наводи се тачан сценарио, корак и разлог неуспеха.

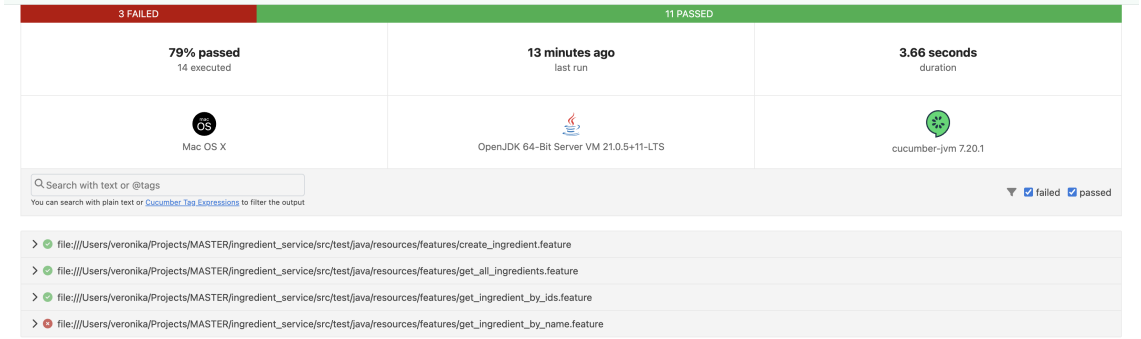
На слици 6.3 је приказан пример како изгледа један *Cucumber* извештај када се сви тестови успешно извршавају, док је на слици 6.4 приказано како изгледа када је неки сценарио неуспешан. Извештај садржи следеће целине:

- Заглавље (*summary*) — приказује укупан број извршених сценарија, проценат успешно извршених тестова, оперативни систем, верзију Јаве, верзију *Cucumber*-а, као и укупно време трајања тестова.
- Филтер и претрага — омогућавају преглед тестова по ознакама (@tag) или по садржају корака.

- Листа *Feature* фајлова — сви *Feature* фајлови су приказани као секције које се могу проширити, а унутар њих се налазе сценарији.
- Детаљи сценарија — за сваки сценарио се приказују сви кораци (`Given`, `When`, `Then`), њихов статус (успешан, неуспешан, прескочен), као и време извршавања. У случају грешке приказује се и детаљна порука са детаљима о грешци. Пример сценарија са корацима без грешке приказан је на слици 6.5, док је пример сценарија са грешкама приказан на слици 6.6.

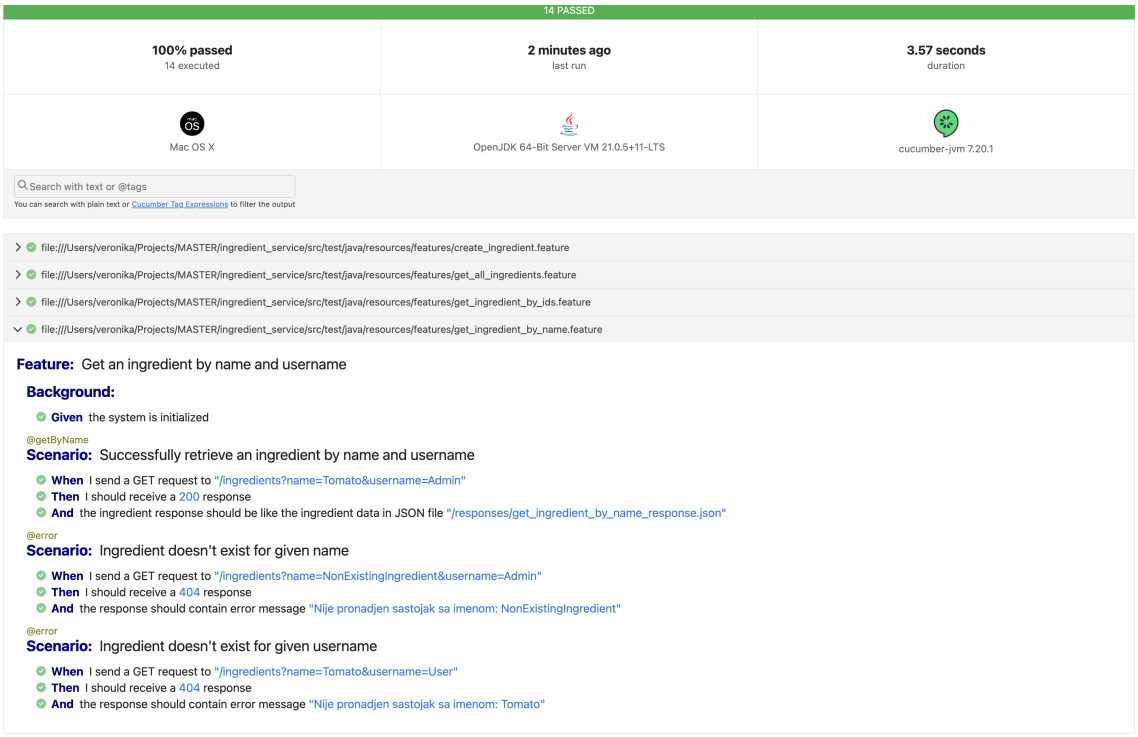


Слика 6.3: Извештај када се сви сценарији успешно извршавају

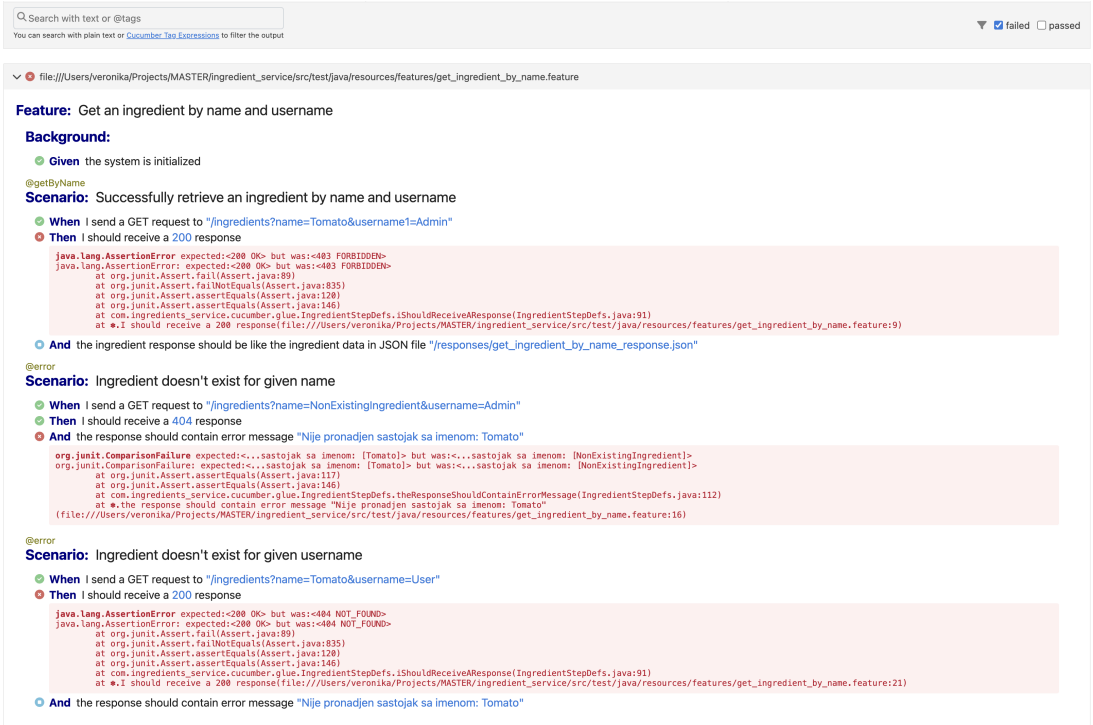


Слика 6.4: Извештај када има грешака у сценаријима

На основу генерисаних *Cucumber* извештаја може се закључити да се сви дефинисани сценарији успешно извршавају, без присуства грешака, чиме се потврђује исправност имплементираних функционалности. Овакви извештаји омогућавају да и чланови тима који нису програмери (нпр. пословни аналитичари) могу лако да разумеју шта је тестирано и какав је исход. Поред тога, извештаји могу да служе као жива документација – описују како систем тренутно функционише у складу са дефинисаним пословним захтевима.



Слика 6.5: Сценарио где се сви кораци успешно извршавају



Слика 6.6: Сценарији са грешкама

6.3.3 *Jacoco* извештај о покривености кода

Да би се прецизно измерила покривеност кода током извршавања тестова, у пројекат је интегрисан алат *Jacoco* [3]. *Jacoco* је алат за мерење покривености кода тестовима у Јава пројектима, који омогућава праћење које линије, методе и гране кода су заиста извршене током тестирања. Он је најпогоднији за употребу јер се лако интегрише са *Maven* и *Gradle* пројектима, генерише детаљне HTML извештаје, подржава све врсте тестова (укључујући *JUnit* и *Cucumber*) и омогућава прецизну идентификацију непокривених делова кода, што директно доприноси побољшању квалитета и одрживости софтвера. Пошто су *Cucumber* тестови једини тестови који постоје у овом пројекту, *Jacoco* омогућава праћење који делови кода су заправо извршени током извршавања сценарија. На тај начин се могу идентификовати делови кода који нису обухваћени постојећим тестовима, што пружа основу за разматрање да ли су ти делови заиста потребни или представљају непотребан кôд који се може редуковати или допунити новим тестовима.

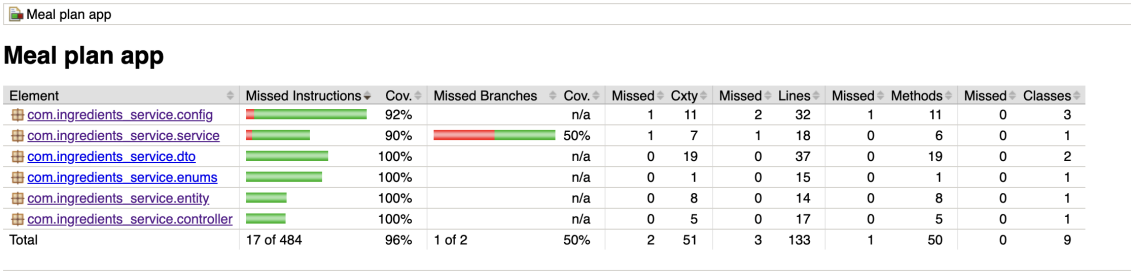
Jacoco мери:

- проценат извршеног кода током тестирања,
- покривеност грана у условним изразима,
- број метода и класа који су тестирани.

Алат је додат као *Maven* софтверски додатак, а у конфигурацији је подешено да се приликом покретања тестова аутоматски генерише извештај унутар фасцикле `target/jacoco-report`. Извештај је у *HTML* формату и приказује процентуалну покривеност сваког пакета, класе и методе, као и посебно покривеност грана у условним изразима.

Пример како изгледа извештај када неки део кода није покривен тестовима приказан је на слици 6.1. Кликом на директоријум отвара се преглед свих класа унутар њега и информација о њиховој покривености, а затим можете отворити класу и добити покривеност по функцијама као на слици 6.8. У овом примеру можемо видети да функција `createIngredient` једну грану која није покривена и да је покривеност ове функције 75%. Када кликнемо на функцију видимо приказ као на слици 6.9 са ког можемо видети да је имплементиран услов да уколико рецепт са одређеним именом за датог корисника већ постоји треба приказати грешку, али да сценарио за овај случај не постоји. Сада

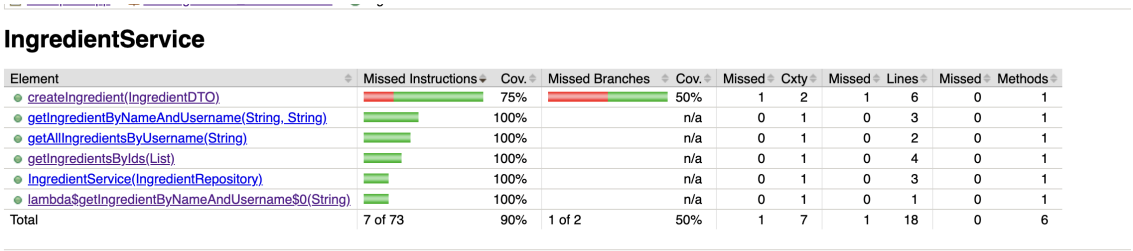
треба размислити да ли је ова функционалност вишак, с обзиром да није обухваћена сценаријима, или је ово нешто што нам је потребно, али смо га превидели када смо дефинисали жељене функционалности. Уколико нам је потребно додајемо сценарио када очекујемо овакво понашање, а уколико нам није потребно уклањамо овај део кода.


Meal plan app

Meal plan app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ingredients_service.config		92%		n/a	1	11	2	32	1	11	0	3
com.ingredients_service.service		90%		50%	1	7	1	18	0	6	0	1
com.ingredients_service.dto		100%		n/a	0	19	0	37	0	19	0	2
com.ingredients_service.enums		100%		n/a	0	1	0	15	0	1	0	1
com.ingredients_service.entity		100%		n/a	0	8	0	14	0	8	0	1
com.ingredients_service.controller		100%		n/a	0	5	0	17	0	5	0	1
Total	17 of 484	96%	1 of 2	50%	2	51	3	133	1	50	0	9

Слика 6.7: Приказ *Jacoco* извештаја сервиса за састојке


IngredientService

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
createIngredient(IngredientDTO)		75%		50%	1	2	1	6	0	1
getIngredientByNameAndUsername(String, String)		100%		n/a	0	1	0	3	0	1
getAllIngredientsByUsername(String)		100%		n/a	0	1	0	2	0	1
getIngredientsByIds(List)		100%		n/a	0	1	0	4	0	1
IngredientService(IngredientRepository)		100%		n/a	0	1	0	3	0	1
lambda$getIngredientByNameAndUsername$0(String)		100%		n/a	0	1	0	1	0	1
Total	7 of 73	90%	1 of 2	50%	1	7	1	18	0	6

Слика 6.8: Приказ функција класе која није стопостотно покривена

Комбиновањем извештаја из *Cucumber*-а и *Jacoco*-а добија се свеобухватан увид у квалитет апликације: *Cucumber* показује функционалну испуњеност захтева, док *Jacoco* омогућава идентификацију делова кода који нису обухваћени тестовима и пружа основу за доношење одлука о њиховој релевантности. На овај начин, извештаји служе не само програмерима, већ и као средство за комуникацију са свим заинтересованим странама у пројекту, доприносећи већој поузданости, одрживости софтвера и аутоматизованој документацији.

IngredientService.java

```java
package com.ingredients_service.service;

import com.ingredients_service.dto.IngredientDTO;
import com.ingredients_service.entity.IngredientEntity;
import com.ingredients_service.repository.IngredientRepository;
import jakarta.validation.Valid;
import java.util.List;
import org.apache.coyote.BadRequestException;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.springframework.validation.annotation.Validated;

@Service
@Validated
public class IngredientService {

    private static final String INGREDIENT_NOT_FOUND_ERROR = "Nije pronadjen sastojak sa imenom: ";
    private static final String INGREDIENT_NAME_ALREADY_EXIST = "Vec postoji sastojak sa imenom: ";


    private final IngredientRepository ingredientRepository;

    public IngredientService(IngredientRepository ingredientRepository) {
        this.ingredientRepository = ingredientRepository;
    }

    public IngredientDTO createIngredient(@Valid IngredientDTO ingredientDTO)
        throws BadRequestException {
      if (ingredientRepository.findByNameAndAddedBy(ingredientDTO.getName(),
          ingredientDTO.getAddedBy()).isPresent()) {
            throw new BadRequestException(INGREDIENT_NAME_ALREADY_EXIST + ingredientDTO.getName());
        }

        IngredientEntity ingredientEntity = ingredientDTO.mapToEntity();
        IngredientEntity savedEntity = ingredientRepository.save(ingredientEntity);
        return savedEntity.mapToDto();
    }


  public List<IngredientDTO> getAllIngredientsByUsername(String username) {
    List<IngredientEntity> listOfIngredientsEntity = ingredientRepository.findAllByAddedBy(
        username);
        return listOfIngredientsEntity.stream().map(IngredientEntity::mapToDto).toList();
    }

  public IngredientDTO getIngredientByNameAndUsername(String name, String username) {
    IngredientEntity ingredientEntity = ingredientRepository.findByNameAndAddedBy(name, username)
        .orElseThrow(() -> new UsernameNotFoundException(INGREDIENT_NOT_FOUND_ERROR + name));
        return ingredientEntity.mapToDto();
    }

    public List<IngredientDTO> getIngredientsByIds(List<Long> ids) {
        return ingredientRepository.findAllById(ids)
            .stream()
            .map(IngredientEntity::mapToDto)
            .toList();
    }

}
```

Слика 6.9: Приказ дела класе који није покривен тестовима

6.3.4 Резултати покривености за апликацију *Plan&Prep*

Након извршавања свих *Cucumber* тестова, алат *Jacoco* је показао да су све функционалности унутар појединачних сервиса, као и обрада могућих грешака и граничних случајева, успешно покривене тестовима. Покривене су основне операције, валидације података, као и случајеви у којима се очекују грешке.

Једини део кода који није обухваћен *Jacoco* извештајем односи се на *REST* клијенте који служе за комуникацију између микросервиса. Током извршавања

Cucumber тестова се нису извршавали стварни *HTTP* позиви ка другим сервисима, већ су њихови одговори били симулирани. Због тога *Jacoco* није могао да региструје извршавање тих делова кода, па се у извештају воде као непокривени.

Иако клијенти нису покривени тренутним тестовима, њихова логика своди се на прослеђивање података. Уколико би било потребно, покривеност ових делова могла би се постићи увођењем јединичних тестова. Међутим, с обзиром на то да је фокус тестирања био на *Cucumber* сценаријима и развоју вођеном понашањем, симулирање понашања клијената се показало као одговарајуће решење. Оваква анализа показује да је пословна логика апликације темељно тестирана и да постоји јасна свест о деловима који нису покривени, уз разумљив разлог и могућа решења за њихово касније тестирање.

Резултати покривености сервиса за кориснике приказан је на слици 6.10, док је резултат покривености сервиса за састојке приказан на слици 6.11. На сликама 6.12 и 6.13 се може видети покривеност сервиса за рецепте и инфомрација о томе да *IngredientClient* није покривен тестовима. Слично се може видети и на сликама 6.14 и 6.15 када је у питању сервис за планове и *RecipeClient*.

Users service

Users service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.user_service.service		100%		100%	0	13	0	45	0	7	0	1
com.user_service.config		100%		n/a	0	14	0	39	0	14	0	4
com.user_service.dto		100%		n/a	0	27	0	41	0	27	0	3
com.user_service.enums		100%		n/a	0	6	0	18	0	6	0	2
com.user_service.entity		100%		n/a	0	13	0	22	0	13	0	1
com.user_service.controller		100%		n/a	0	3	0	7	0	3	0	1
Total	0 of 655	100%	0 of 12	100%	0	76	0	172	0	70	0	12

Слика 6.10: Извештај о покривености сервиса за кориснике

Ingredient service

Ingredient service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ingredients_service.config		100%		n/a	0	11	0	32	0	11	0	3
com.ingredients_service.dto		100%		n/a	0	19	0	37	0	19	0	2
com.ingredients_service.enums		100%		n/a	0	1	0	15	0	1	0	1
com.ingredients_service.service		100%		100%	0	7	0	18	0	6	0	1
com.ingredients_service.entity		100%		n/a	0	8	0	14	0	8	0	1
com.ingredients_service.controller		100%		n/a	0	5	0	17	0	5	0	1
Total	0 of 484	100%	0 of 2	100%	0	51	0	133	0	50	0	9

Слика 6.11: Извештај о покривености сервиса за састојке

Recipes service

Recipes service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.recepies_service.entity		74%		n/a	3	27	7	51	3	27	1	3
com.recepies_service.service		98%		100%	1	24	0	84	1	22	0	1
com.recepies_service.dto		100%		n/a	0	35	0	71	0	35	0	4
com.recepies_service.config		100%		n/a	0	11	0	32	0	11	0	3
com.recepies_service.enums		100%		n/a	0	2	0	23	0	2	0	2
com.recepies_service.controller		100%		n/a	0	11	0	33	0	11	0	1
Total	41 of 1,033	96%	0 of 4	100%	4	110	7	294	4	108	1	14

Слика 6.12: Извештај о покривености сервиса за рецепте

Recipes service > com.recepies_service.entity

com.recepies_service.entity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
IngredientClient		0%		n/a	3	3	7	7	3	3	1	1
RecipeEntity		100%		n/a	0	18	0	35	0	18	0	1
RecipeIngredientEntity		100%		n/a	0	6	0	9	0	6	0	1
Total	36 of 140	74%	0 of 0	n/a	3	27	7	51	3	27	1	3

Слика 6.13: Информација о покривености класе *IngredientClient*

Plans service

Plans service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.plans_service.entity		74%		n/a	3	22	5	43	3	22	1	3
com.plans_service.service		100%		100%	0	12	0	46	0	8	0	1
com.plans_service.config		100%		n/a	0	10	0	30	0	10	0	3
com.plans_service.dto		100%		n/a	0	26	0	51	0	26	0	3
com.plans_service.enums		100%		n/a	0	1	0	7	0	1	0	1
com.plans_service.controller		100%		n/a	0	3	0	11	0	3	0	1
Total	30 of 621	95%	0 of 8	100%	3	74	5	188	3	70	1	12

Слика 6.14: Извештај о покривености сервиса за планове

Plans service > com.plans_service.entity

com.plans_service.entity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RecipeClient		0%		n/a	3	3	5	5	3	3	1	1
PlanEntity		100%		n/a	0	12	0	23	0	12	0	1
PlanRecipeEntity		100%		n/a	0	7	0	15	0	7	0	1
Total	30 of 116	74%	0 of 0	n/a	3	22	5	43	3	22	1	3

Слика 6.15: Информација о покривености класе *RecipeClient*

Глава 7

Закључак

У оквиру рада реализована је микросервисна апликација за планирање оброка, која се састоји од четири сервиса за које је укупно написано 80 сценарија. У сценаријима су тестирани и успешни захтеви и могуће грешке, а по извршавању свих тестова показало се да све сценарији успешно извршавају. Ово потврђује да је апликација имплементирана у складу са унапред дефинисаним захтевима. Процена покривености кода алатом *Jacoco* показала је високу покривеност за сваки од сервиса - 96% за сервис за рецепте, 95% за сервис за планове и 100% за сервисе за кориснике и састојке.

Извештаји добијени из *Cucumber*-а, заједно са извештајима о покривености кода, омогућили су транспарентан увид у степен тестираног кода и обухваћених функционалности. На тај начин демонстрирано је да интеграција развоја вођеног понашањем у развој микросервисне архитектуре води ка већем квалитету софтвера, систематичнијем процесу тестирања и одрживијем развоју у целини. Допринос овог рада огледа се у томе што је приказано како се један савремени методолошки приступ може успешно применити у конкретном, практичном систему, чиме се показује његова вредност и применљивост и у ширем контексту развоја софтвера.

Постоји више праваца за даље унапређење и примену развоја вођеног понашањем у апликацији. Један од могућих праваца је проширење апликације додавањем нових сервиса и функционалности, као што су генерисање списка за куповину за један дан или целу недељу, као и могућност праћења напретка корисника и статистичка анализа о поштовању планираних оброка. Такви додаци омогућили би корисницима да прате свој напредак, да боље управљају исхраном и да добијају повратне информације о томе у којој мери се

придржавају планова, што значајно повећава практичну вредност апликације.

За комуникацију између микросервиса тренутно је коришћен *HTTP* протокол, али у будућности би се могли размотрити алати који подржавају асинхрону размену порука, као што су *Apache Kafka* [6] или *RabbitMQ* [18]. Коришћење ових приступа могло би да повећа ефикасност и поузданост размене података, смањи оптерећење сервиса и омогући боље скалирање апликације. Поред тога, могуће је унапређење безбедности кроз додавање механизама аутентификације и ауторизације заснованих на токенима (нпр. *JWT*) или кроз интеграцију са *OAuth2* протоколом, чиме би се обезбедила контролисана размена података и већа сигурност апликације.

Ради додатне сигурности и свеобухватније провере апликације, могу се увести и друге врсте тестова попут јединичних, интеграционих и тестова перформанси, што би омогућило детаљнију оцену квалитета кода и смањило ризик од неочекиваних грешака у раду апликације. Ови кораци показују да развој вођен понашањем пружа снажну основу за даљи раст апликације, омогућавајући истовремено већу поузданост система и бољу корисничку подршку.

Библиографија

[1] The official Spring documentation, Aspect Oriented Programming with Spring. on-line at: https://docs.spring.io/spring-framework/reference/core/aop.html.

[2] The official Spring documentation, Dependencies. on-line at: https://docs.spring.io/spring-framework/reference/core/beans/dependencies.html.

[3] The official Jacoco site. on-line at: https://www.jacoco.org/.

[4] The official Jenkins site. on-line at: https://www.jenkins.io/.

[5] The official Jetty site. on-line at: https://jetty.org/.

[6] The official apache kafka site. on-line at: http://kafka.apache.org/.

[7] Veronika Marinkovic. Frontend part of Plan&Prep application, 2025. on-line at: https://github.com/veronika1996/mealplan-app-front.

[8] Veronika Marinkovic. Ingredients service, 2025. on-line at: https://github.com/veronika1996/ingredient_service.

[9] Veronika Marinkovic. Plan&Prep application, 2025. on-line at: https://github.com/veronika1996/mealplan-app.

[10] Veronika Marinkovic. Plans service, 2025. on-line at: https://github.com/veronika1996/plans_service.

[11] Veronika Marinkovic. Recipes service, 2025. on-line at: https://github.com/veronika1996/recipes_service.

[12] Veronika Marinkovic. Users service, 2025. on-line at: https://github.com/veronika1996/users_service.

[13] The official Maven documentation. on-line at: `https://maven.apache.org/`.

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

[15] The official Next.js documentation. on-line at: `https://nextjs.org/`.

[16] Oracle. Java Documentation. on-line at: `https://docs.oracle.com/en/java/`.

[17] The official postgresql site. on-line at: `https://www.postgresql.org/`.

[18] The official rabbitmq site. on-line at: `https://www.rabbitmq.com/`.

[19] The official React documentation. on-line at: `https://react.dev/`.

[20] Amazon Web Services. What is SDLC? Software Development Life Cycle Explained, 2025. on-line at: `https://aws.amazon.com/what-is/sdlc/`.

[21] John Ferguson Smart and Jan Molak. *BDD in Action, Second Edition: Behavior-Driven Development for the Whole Software Lifecycle*. Manning, 2023.

[22] The official Spring documentation. on-line at: `https://spring.io/`.

[23] The official Spring documentation, Spring Boot. on-line at: `https://spring.io/projects/spring-boot`.

[24] The official Spring documentation, Spring Web MVC. on-line at: `https://docs.spring.io/spring-framework/reference/web/webmvc.html`.

[25] The official Spring documentation, Spring Security. on-line at: `https://spring.io/projects/spring-security`.

[26] The official Spring documentation, Spring WebFlux. on-line at: `https://docs.spring.io/spring-framework/reference/web/webflux.html`.

[27] The official swagger site. on-line at: `https://swagger.io/`.

[28] The Cucumber Open Source Project. Cucumber Expressions. on-line at: `https://github.com/cucumber/cucumber-expressions#readme`.

[29] The Cucumber Open Source Project. The official Cucumber documentation. on-line at: `https://cucumber.io/docs`.

[30] The official Tomcat site. on-line at: `https://tomcat.apache.org/`.

[31] The official Travis CI site. on-line at: `https://www.travis-ci.com/`.

[32] TutorialsPoint. SDLC - Agile Model. on-line at: `https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm`.

[33] TutorialsPoint. SDLC - Spiral Model. on-line at: `https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm`.

[34] TutorialsPoint. SDLC - V Model. on-line at: `https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm`.

[35] TutorialsPoint. SDLC - Waterfall Model. on-line at: `https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm`.

Биографија аутора

Вероника Маринковић рођена је 25.09.1996. године у Крушевцу. Са одличним успехом завршила је основну школу „Аца Алексић” и средњу школу „Свети Трифун” у Александровцу. Године 2015. уписује основне студије на Математичком факултету у Београду на смеру Математика и модулу Рачунарство и информатика. Основне студије завршава у јулу 2020. године, након чега уписује мастер студије на истој катедри.

У новембру 2020. године запошљава се у компанији *Enjoy.ing*, где је најпре започела праксу током које је савладала основе развоја апликација у *Spring Boot* окружењу и принципе микросервисне архитектуре. Након праксе радила је на пројектима у финансијском сектору за швајцарске банке, као и на развоју платформе за онлајн продају едукативних садржаја. У априлу 2024. године запошљава се у компанији *Exxeta*, где је ангажована на пројектима у области аутомобилске индустрије и где наставља рад и данас.