УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Александра Биочанин

ПРИМЕНА САВРЕМЕНИХ АРХИТЕКТУРНИХ ОБРАЗАЦА У РАЗВОЈУ ВЕБ ПЛАТФОРМЕ ЗА ОРГАНИЗАЦИЈУ СТРУЧНИХ ДОГАЂАЈА

мастер рад

Београд, 2026.

Ментор:

др Александар КАРТЕЉ, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Владимир ФИЛИПОВИЋ, редовни професор
Универзитет у Београду, Математички факултет

др Сташа ВУЈИЧИЋ СТАНКОВИЋ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Примена савремених архитектурних образаца у развоју веб платформе за организацију стручних догађаја

Резиме: Овај мастер рад разматра примену савремених архитектурних образаца у развоју веб платформе за организацију стручних догађаја. Практични део рада представља апликација *EventOrganizer*, реализована помоћу библиотеке *React* и језика *TypeScript* на клијентској страни и платформе *ASP.NET Core* на серверској страни. У раду је посебан фокус стављен на модуларни монолит, чисту архитектуру (енгл. *Clean Architecture*), убризгавање зависности (енгл. *Dependency Injection*), дизајн вођен доменом (енгл. *Domain-Driven Design*) и раздвајање одговорности команди и упита (енгл. *Command Query Responsibility Segregation*, *CQRS*), које је примењено у апликационом слоју уз заједничку базу података за читање и писање. Поред архитектурне организације система, рад обухвата и питања безбедности, тестирања, препоруке ресурса и могућности даљег проширивања платформе.

Кључне речи: веб апликације, софтверска архитектура, модуларни монолит, чиста архитектура, дизајн вођен доменом, *CQRS*, *React*, *TypeScript*, *ASP.NET Core*

Садржај

Глава 1

Увод

Развој савремених веб апликација не подразумева само реализацију корисничког интерфејса и повезивање са базом података, већ и пажљиво пројектовање начина на који су делови система организовани. Како апликација расте, број функционалности, корисничких улога, пословних правила и зависности постаје све већи. Уколико систем није јасно структуриран, свака наредна измена постаје тежа, а грешке се све чешће јављају као последица нејасних граница између делова апликације. Због тога архитектура софтверског система има важну улогу у његовој дугорочној употребљивости, одржавању и могућности проширивања.

Предмет овог мастер рада је примена савремених архитектурних образаца у развоју веб платформе за организацију стручних догађаја. Практични део рада представља апликација *EventOrganizer*, систем намењен подршци у организацији догађаја као што су семинари, конференције, предавања и други облици стручних окупљања. Платформа обухвата креирање и уређивање догађаја, управљање ресурсима, планирање простора, предавача и опреме, пријављивање учесника, обраду захтева, слање обавештења, прикупљање рецензија и преглед извештаја.

Оваква врста система је погодна за анализу архитектуре зато што обухвата више повезаних целина. С једне стране постоји јавни део апликације, у ком корисници могу да прегледају објављене догађаје и пријаве се за учешће. С друге стране, постоји организациони део, у ком организатори креирају догађаје, бирају потребне ресурсе и прате стање пријава. Поред тога, администратори управљају ресурсима, корисницима и захтевима који захтевају одобравање. Због тога систем мора да подржи различите корисничке улоге,

различита права приступа, више пословних токова и доследну примену правила у свим деловима апликације [31].

Практични део рада реализован је као веб апликација са одвојеним клијентским и серверским делом. Клијентски део је развијен употребом библиотеке *React* и језика *TypeScript*, док је серверски део реализован помоћу *ASP.NET Core* технологије. За приступ подацима користи се *Entity Framework Core*, а као систем за управљање базом података *PostgreSQL*. Архитектурну основу чине модуларни монолит и принципи чисте архитектуре, док су дизајн вођен доменом, *CQRS* и убризгавање зависности примењени ради јасног смештања пословних правила, организације случајева употребе и повезивања слојева. Безбедност је заснована на аутентификацији помоћу *JWT* токена и серверској ауторизацији.

1.1 Предмет и мотивација рада

Организација стручних догађаја подразумева координацију више активности које су међусобно зависне. За један догађај потребно је дефинисати тему, опис, време одржавања, капацитет, буџет, област, број предавача и потребу за додатном опремом. Након тога је неопходно обезбедити одговарајућу салу, предаваче и пакете опреме, водећи рачуна о расположивости, трошковима и могућим преклапањима са другим догађајима. Када се догађај објави, учесници могу да се пријаве, а пријаве може да прати и обрађује организатор тог догађаја, као и администратор.

У једноставним случајевима, део овог процеса може се водити ручно, путем табела, електронске поште или неповезаних алата. Међутим, такав приступ брзо постаје непоуздан када број догађаја, учесника и ресурса порасте. Подаци се тада лако дуплирају, промене нису увек видљиве свим учесницима у процесу, а провера доступности ресурса зависи од ручног рада. Поред тога, различите улоге у систему немају исте одговорности, па је неопходно јасно дефинисати ко може да креира догађај, ко може да одобри захтев за ресурс, ко може да управља корисницима и ко може да види одређене извештаје.

Мотивација за овај рад проистиче из потребе да се на конкретном доменском проблему прикаже како архитектурни обрасци могу да помогну у изградњи система који је разумљив, одржив и спреман за проширивање. Систем је организован тако да сваки део има јасну одговорност. Доменски слој описује

основна правила пословања, апликациони слој координише корисничке случајеве употребе, инфраструктурни слој обезбеђује рад са базом и спољним сервисима, док *API* слој излаже функционалности клијентском делу.

Оваква организација омогућава да се сложеност система контролише. Када се додаје нова функционалност, лакше је одредити где припадају нова правила, где се врши валидација, где се приступа бази података и како се резултат приказује кориснику. То је посебно важно код система који се развијају постепено, јер се захтеви током времена мењају. На пример, платформа за организацију догађаја може се касније проширити интеграцијом са календарима, слањем обавештења електронском поштом, напреднијом аналитиком или додатним критеријумима за препоруку ресурса. Добра архитектура не гарантује да ће такве измене бити једноставне, али значајно смањује ризик да оне наруше постојеће делове система.

1.2 Циљ рада

Основни циљ рада је проучавање савремених архитектурних образаца и разматрање њихове примене у развоју веб апликација, на примеру пројектовања и имплементације веб платформе за организацију стручних догађаја. Реализована апликација служи као практичан пример за анализу начина на који су у софтверском систему организовани подаци, пословна правила, корисничке акције и инфраструктурне зависности.

У остваривању овог циља разматра се како се избором модуларног монолита и применом принципа чисте архитектуре може постићи јасно раздвајање одговорности. Серверски део се покреће као једна апликација, али је изнутра организован према функционалним модулима и кроз више пројеката који представљају *API*, апликациони, доменски и инфраструктурни слој. Смер зависности контролисан је интерфејсима и обрасцем убризгавања зависности, тако да апликациони случајеви употребе не стварају директно инфраструктурне сервисе. Апликациони токови реализовани су применом *CQRS* приступа кроз команде и упите, док доменски модел чува кључна пословна правила.

Посебна пажња посвећена је и организацији клијентске апликације тако да прати функционалне целине система. Компоненте и помоћне функције груписане су по целинама као што су догађаји, ресурси, пријаве, резервације, обавештења и корисници. Такав приступ доприноси прегледности кода и олакшава

рад на појединачним деловима апликације.

Архитектурне одлуке анализирају се и кроз конкретне пословне токове. Посебно важан пример је планирање догађаја, које обухвата избор сале, предавача и опреме, проверу буџета и капацитета, слање захтева за одобравање, одлуку администратора и објављивање догађаја. Овај ток повезује више доменских ентитета и више корисничких улога, па представља добар пример примене раздвајања одговорности у пракси.

На тај начин рад приказује архитектурне обрасце као практичне алате у развоју реалне апликације. Њихова вредност се огледа у лакшем тестирању, јаснијој организацији кода, бољој контроли пословних правила и могућности да се систем даље развија без потпуног преуређивања постојеће структуре.

1.3 Коришћене технологије

Практични део рада реализован је коришћењем технологија које су чест избор у развоју савремених веб апликација. На клијентској страни употребљени су *React* и *TypeScript*. *React* омогућава изградњу интерактивног корисничког интерфејса кроз компоненте, док *TypeScript* додаје статичку типизацију и тиме смањује број грешака које би се иначе откриле тек у току извршавања програма. Развојно окружење и припрема продукционе верзије ослањају се на *Vite* [34]. За рутирање се користи *React Router* [27], за управљање формуларима *React Hook Form* [26], а за опис шема и проверу података *Zod* [37]. За управљање серверским стањем и асинхроним захтевима користи се *TanStack Query*, а за изградњу корисничког интерфејса библиотека *Material UI* (*MUI*).

Серверски део апликације развијен је помоћу *ASP.NET Core* платформе. Ова технологија омогућава изградњу *REST API*-ја, дефинисање контролера и посредничких компоненти (енгл. *middleware*) које учествују у обради *HTTP* захтева, аутентификацију, ауторизацију и конфигурацију зависности [4, 14]. За имплементацију апликационих токова користи се *MediatR*, библиотека која омогућава да се захтеви обрађују кроз команде и упите [12]. За валидацију улазних података користи се *FluentValidation* [29].

За рад са базом података користи се *Entity Framework Core* [17], док је као систем за управљање базом података изабран *PostgreSQL* [25]. За аутентификацију се користе две врсте токена: *JWT access token*, односно приступни токен у *JWT* формату, којим се приступа заштићеним ресурсима, и *refresh*

token, односно токен за обнављање сесије, који омогућава издавање новог *access token*-а без поновне пријаве корисника [8, 24]. Ови токени чине сопствени механизам управљања сесијом апликације, док се управљање корисницима и улогама ослања на *ASP.NET Core Identity*. Тестирање клијентског дела реализовано је помоћу окружења *Vitest* и библиотеке *React Testing Library* [35, 33]. Ове технологије нису у раду посматране изоловано, већ кроз начин на који се уклапају у архитектуру система.

1.4 Структура рада

Рад је организован у седам поглавља. Након увода, друго поглавље представља теоријске основе и архитектурне обрасце примењене у развоју система. Посебна пажња посвећена је избору модуларног монолита, чистој архитектури, убризгавању зависности, раздвајању команди и упита, дизајну вођеном доменом, валидацији, аутентификацији и ауторизацији.

Треће поглавље посвећено је анализи домена платформе *EventOrganizer*. У њему су описане корисничке улоге, основни ентитети и најважнији пословни токови, чиме се успоставља веза између проблема који се решава и архитектуре изабране за његову реализацију.

Четврто поглавље описује имплементацију серверског дела апликације: организацију решења, смер пројектних зависности, регистрацију сервиса помоћу механизма за убризгавање зависности, обраду команди и упита, рад са базом података, управљање корисницима, аутентификацију, ауторизацију и обраду грешака.

Пето поглавље описује организацију *React* апликације, рутирање, контролу приступа, комуникацију са *API*-јем и приказ најважнијих корисничких токова. Као централни пример издвојена је страница за планирање догађаја, јер повезује више делова система и јасно показује интеракцију корисничког интерфејса са серверском логиком.

Шесто поглавље обрађује тестирање и евалуацију решења. Приказани су тестно окружење, врсте и резултати серверских и клијентских тестова, покривеност кода и квантитативно мерење времена одзива препоруке при расту броја ресурса. Разматрају се и архитектурна оцена решења и ограничења спроведене евалуације.

Завршно поглавље повезује циљеве са остварeним резултатима, сумира допринос примењених образаца и наводи правце даљег развоја платформе.

Глава 2

Теоријске основе и архитектурни обрасци

Практични део рада, апликација *EventOrganizer*, обухвата више корисничких улога, различите токове одобравања, управљање ограниченим ресурсима, пријаве учесника, обавештења и извештаје. Због тога је при развоју било важно да систем буде организован тако да се његови делови могу разумети, тестирати и проширивати без сталног нарушавања постојеће структуре [28].

У овом поглављу описани су архитектурни појмови неопходни за разумевање остатка рада, с нагласком на одлуке и принципе директно примењене у пројекту: избор модуларног монолита, чисту архитектуру, управљање зависностима, доменски модел, организацију апликационих токова, безбедност, валидацију и организацију клијентске апликације.

2.1 Савремене веб апликације

Савремене веб апликације се најчешће састоје од клијентског дела, серверског дела и базе података. Клијентски део се извршава у веб прегледачу и задужен је за кориснички интерфејс, навигацију, формуларе и интеракцију. Серверски део прихвата захтеве, примењује пословна правила, проверава идентитет и права корисника, приступа подацима и враћа одговор клијенту. База података чува трајно стање система. Оваква подела представља уобичајену основу за организацију савремених веб апликација [30].

У апликацији *EventOrganizer* клијентски део је реализован помоћу библиотеке *React* и језика *TypeScript*, док је серверски део развијен на платфор-

ми *ASP.NET Core*. Ова два дела комуницирају преко *REST API*-ја. Таква подела омогућава да се кориснички интерфејс и серверска логика развијају релативно независно, али захтева јасно дефинисан уговор о облику података и доступним операцијама.

Ова основна подела, међутим, није довољна за сложенију апликацију. Ако би се сва серверска логика налазила у контролерима, а сва клијентска логика директно у компонентама, систем би временом постао тежак за одржавање. Због тога је потребно увести унутрашњу организацију која одваја различите врсте одговорности.

2.2 Модуларни монолит и разлози избора

EventOrganizer је реализован као модуларни монолит. Серверски део се изграђује, испоручује и покреће као једна *ASP.NET Core* апликација и користи једну *PostgreSQL* базу података. Код је истовремено организован према функционалним модулима и архитектурним слојевима. Монолитни део одлуке односи се на јединствен процес и заједнички поступак испоруке, док модуларни део означава постојање препознатљивих граница између пословних области. Слојеви одређују врсту одговорности, а модули област којој одговорност припада. Зато, на пример, модул догађаја обухвата *API* операције, апликационе случајеве употребе, доменска правила и чување података који се односе на догађаје.

Овај избор најпре одговара структури домена. Најважнији токови јесу повезани, али се могу груписати у кохезивне области као што су догађаји, ресурси, резервације, пријаве, рецензије, обавештења и извештаји. Модуларна организација омогућава да се одговорности и измене задрже у области на коју се односе, док позиви унутар истог процеса избегавају сложеност мрежне комуникације. Границе модула у овом решењу изражене су организацијом контролера, команди, упита, доменских модела и клијентских функционалних целина. Модули нису засебне апликације и сарађују кроз апликационе случајеве употребе и заједничку базу.

Други разлог је потреба за доследношћу података. Објављивање догађаја зависи од стања резервације ресурса, одлучивање о пријави од капацитета догађаја, а право на рецензију од учешћа корисника и завршетка догађаја. У оквиру једне апликације и базе ове промене могу се координисати локалним

позивима и трансакцијама. Подела на засебне сервисе захтевала би дефинисање мрежних уговора, решавање делимичних отказа и додатне механизме за одржавање доследности између података који припадају повезаним токовима.

Трећи разлог односи се на обим система и начин развоја. Не постоје делови платформе за које су утврђени независни захтеви за скалирањем или засебни циклуси испоруке, нити више тимова који морају самостално да управљају различитим сервисима. Једна јединица за испоруку поједностављује конфигурацију, миграције базе, аутентификацију, тестирање и праћење рада апликације. Модуларност истовремено спречава да ова једноставност доведе до мешања одговорности и омогућава да се систем проширује унутар постојећих граница.

Микросервисна архитектура систем организује као скуп независно покретаних и испоручивих сервиса који комуницирају преко мрежних механизама [10]. Њене предности долазе до изражаја када су потребни независно скалирање, технолошка разноликост, аутономни тимови или одвојено објављивање делова система. За *EventOrganizer* те потребе не оправдавају оперативну и развојну сложеност расподељеног система. Приступ у ком се систем најпре развија као добро организован монолит омогућава да се границе функционалности провере и стабилизују пре евентуалног издвајања сервиса [6]. Због тога модуларни монолит представља примеренији однос једноставности и структурисаности. Ако се током даљег развоја појави мерљива потреба за независним скалирањем или испоруком одређене области, постојеће функционалне границе могу послужити као полазиште за њено издвајање у посебан сервис.

2.3 Раздвајање одговорности и слојевита архитектура

Раздвајање одговорности је један од основних принципа доброг софтверског дизајна. Идеја је да сваки део система има јасну улогу и да не преузима послове који му не припадају. У веб апликацији то значи да контролер не треба да садржи целокупну пословну логику, да класе за рад са базом не треба да одлучују о правилима домена, а да кориснички интерфејс не треба да буде једино место на ком се проверавају ограничења.

Слојевита архитектура организује систем у делове са различитим одговорностима. У литератури се овај приступ јавља у више облика, а посебно

је значајан у развоју пословних апликација [7]. Чиста архитектура додатно наглашава да централни део система треба да буде пословни домен, док технички детаљи, као што су база података или радни оквир (енгл. *framework*), треба да остану у спољним слојевима [11]. Њено важно правило односи се на смер зависности: унутрашњи слојеви не треба да зависе од детаља спољних слојева, већ спољни делови треба да се прилагоде уговорима које дефинише језгро система.

У *EventOrganizer* пројекту серверски део је организован кроз четири главна слоја. *API* слој прихвата *HTTP* захтеве и враћа одговоре. Апликациони слој описује корисничке случајеве употребе, као што су креирање догађаја, подношење захтева за резервацију или одобравање пријаве. Доменски слој садржи основне ентитете и правила која се односе на њихова стања. Инфраструктурни слој садржи рад са базом података, корисничким системом, токенима и другим техничким детаљима.

Овај смер видљив је и у референцама између *.NET* пројеката. Доменски пројекат не зависи од осталих слојева, апликациони зависи од доменског, а инфраструктурни имплементира уговоре потребне апликационом слоју. *API* пројекат представља спољну улазну тачку и повезује целину. На тај начин промена начина чувања података или генерисања токена не захтева да се пословни случајеви употребе преместе у инфраструктурни слој.

Оваква подела је важна јер апликација има више токова који се међусобно преплићу. На пример, објављивање догађаја није само промена једног поља у бази. Пре тога треба проверити статус догађаја и стање резервације ресурса. Ако се таква правила налазе на јасном месту, лакше је разумети систем и мања је вероватноћа да ће нека нова функционалност заобићи важно ограничење.

2.4 Убризгавање зависности и смер зависности

Убризгавање зависности је образац којим објекат своје зависности добија споља, уместо да унутар себе непосредно ствара њихове конкретне имплементације. Најчешће се примењује убризгавањем кроз конструктор: класа декларише које су јој сарадничке компоненте потребне, а механизам за конфигурацију апликације обезбеђује одговарајуће инстанце. *ASP.NET Core* садржи

уграђени контејнер за убризгавање зависности за регистрацију сервиса, избор њиховог животног века и разрешавање зависности при извршавању [15].

Образац убризгавања зависности треба разликовати од принципа инверзије зависности (енгл. *Dependency Inversion Principle*). Тај принцип захтева да логика вишег нивоа зависи од апстракција, а не од конкретних техничких детаља, док убризгавање зависности представља механизам којим се такво повезивање остварује при покретању апликације. На пример, класа за обраду команде или упита (енгл. *command handler*, односно *query handler*) може да зависи од интерфејса за приступ подацима или од сервиса за рад са тренутним корисником, док инфраструктурни слој обезбеђује конкретне имплементације тих уговора.

Овај образац је значајан за одрживост и тестирање система. Зависности класе видљиве су у њеном конструктору, имплементација се може заменити без измене пословне логике, а у тесту се може употребити контролисана замена. У *EventOrganizer* апликацији убризгавање зависности повезује слојеве, али не укида њихове границе: место на ком се бирају конкретне имплементације издвојено је у улазну тачку *API* пројекта, док апликациони и доменски код остају усредсређени на случајеве употребе и правила.

2.5 Доменски модел

Доменски модел представља појмове из стварног проблема који апликација решава кроз ентитете, њихова стања и правила која над њима важе. Ова идеја је блиска дизајну вођеном доменом, који наглашава да модел у коду треба да одражава важне појмове пословног домена [3].

У *EventOrganizer* апликацији важни доменски појмови су догађај, ресурс, резервација ресурса, пријава, рецензија и обавештење. Они поред података садрже и понашање повезано са својим стањем. Догађај има животни циклус: може бити у нацрту, објављен, отказан или завршен. Резервација ресурса такође има своја стања: нацрт, поднет захтев, одобрено, одбијено, истекло или отказано. Од тренутног стања зависи које су операције дозвољене.

Смештање пословних правила у доменски модел чини систем поузданијим. На пример, правило да се отказан или завршен догађај не може уређивати не треба да буде препуштено само корисничком интерфејсу. Такво правило мора постојати на серверској страни, у делу система који се увек извршава

без обзира на то одакле је захтев дошао. Слично томе, резервација ресурса не сме да се одобри ако није у стању које дозвољава одлучивање.

У овом раду ће доменски модел бити важан при опису имплементације, јер он показује како су пословна правила преведена у структуру кода. Посебно ће бити значајни модели догађаја и резервације ресурса, пошто се кроз њих види како систем контролише ток планирања догађаја.

2.6 Команде, упити и апликациони токови

Корисничке операције у систему могу се грубо поделити на оне које мењају стање и оне које само читају податке. Креирање догађаја, измена ресурса, подношење захтева за резервацију и одобравање пријаве мењају стање система. С друге стране, приказ листе догађаја, преглед пријава или приказ детаља ресурса представљају читање података. Овај приступ познат је као раздвајање одговорности команди и упита (*CQRS*) [5].

У овом пројекту *CQRS* је примењен као начин организације апликационог слоја, уз заједничку *PostgreSQL* базу података за читање и писање. То значи да систем нема одвојене моделе и базе за читање и измену података, већ да се раздвајање користи пре свега да би свака операција имала јасно место у апликационом коду. Свака значајна операција представљена је командом или упитом, а њена обрада смештена је у посебну класу за обраду. Контролер прима захтев, формира команду или упит и прослеђује га апликационом слоју, док се пословна логика извршава у за то предвиђеним компонентама.

Овакав приступ је користан за *EventOrganizer* зато што систем има много различитих корисничких акција. Организатор креира и планира догађај, учесник се пријављује или отказује пријаву, администратор одобрава резервацију ресурса, а систем бележи обавештења. Када свака од ових операција има своје јасно место, лакше је пратити шта се дешава и писати тестове.

Команде и упити такође помажу да се пословни токови опишу природније. На пример, ток планирања догађаја може се посматрати као низ апликационих операција: измена нацрта резервације, подношење захтева, повлачење захтева, ревизија, одобравање или одбијање. Овај начин организације биће детаљније приказан у поглављу о серверском делу апликације.

2.7 Валидација, безбедност и обрада грешака

Пошто апликација ради са корисничким уносом и различитим правима приступа, валидација и безбедност су важан део архитектуре. Валидација проверава да ли су подаци исправни пре извршавања операције. На пример, догађај мора имати назив, исправан временски интервал, позитиван капацитет и буџет. Клијентски део може да помогне кориснику да исправи грешке у форми, али серверски део мора да буде коначни извор провере.

У *EventOrganizer* апликацији валидација је организована у апликационом слоју, а правила уноса повезана су са командама које представљају конкретне операције. Доменски модел чува правила која зависе од стања објекта. Ова подела омогућава да се разликују једноставне провере улазних података од пословних ограничења.

Безбедност система обухвата аутентификацију и ауторизацију. Аутентификација утврђује идентитет корисника, док ауторизација одређује шта тај корисник сме да уради. У систему постоје различите улоге: учесник, организатор и администратор. Због тога није довољно само знати да је корисник пријављен; потребно је проверити и да ли има право да изврши одређену операцију. *ASP.NET Core* пружа механизме за изградњу и заштиту веб *API*-ја, укључујући аутентификацију корисника, проверу права приступа, конфигурацију сервиса и обраду *HTTP* захтева [14].

Обрада грешака такође треба да буде доследна. Ако сваки контролер сам одлучује како ће вратити грешку, клијентски део добија различите облике одговора и теже их обрађује. Зато је корисно имати централизован механизам који изузетке и грешке преводи у јасне *HTTP* одговоре. Ово је посебно значајно при подношењу захтева за резервацију ресурса, када грешка може да садржи и податке о конфликту са другим догађајем.

2.8 Организација клијентске апликације

React омогућава изградњу корисничког интерфејса кроз компоненте [13], али не прописује како цела апликација треба да буде организована. Код мањих апликација довољно је имати неколико страница и заједничке компоненте. Код апликације као што је *EventOrganizer*, број функционалности је већи, па је важно да клијентски код буде организован на начин који прати доменске

целине.

У пројекту је примењена организација по функционалностима. Делови који се односе на догађаје, ресурсе, резервације, пријаве, рецензије, обавештења и кориснике смештени су у одвојене целине. У оквиру једне целине налазе се компоненте, типови и функције за комуникацију са *API*-јем које припадају тој функционалности. Ово олакшава проналажење кода и смањује мешање неповезаних делова апликације.

Клијентска апликација такође мора да прати корисничке улоге. Неке странице су јавне, неке су доступне само пријављеним корисницима, а неке само организаторима или администраторима. Због тога је рутирање повезано са контролом приступа. Иако сервер остаје коначни извор безбедносних правила, клијентски део треба да води корисника кроз интерфејс који одговара његовој улози.

За рад са подацима добијеним са сервера користи се *TanStack Query*, библиотека која олакшава учитавање, кеширање и освежавање асинхроних података [32]. То је корисно у систему где више страница зависи од података који се могу мењати, као што су статуси догађаја, резервација и пријава. Уместо да свака компонента ручно управља свим детаљима учитавања, грешака и освежавања, део тог посла преузима библиотека.

2.9 Улога образаца у овом раду

Обрасци и технологије описани у овом поглављу делују на различитим нивоима архитектуре и решавају различите врсте проблема. Модуларни монолит одређује начин организације и испоруке система. Чиста архитектура и принцип инверзије зависности одређују границе слојева и дозвољени смер зависности, док убризгавање зависности обезбеђује њихово повезивање при извршавању. Дизајн вођен доменом усмерава моделирање пословних појмова и правила, а *CQRS* организује случајеве употребе као команде и упите у оквиру заједничке базе података за читање и писање. *Entity Framework Core* и *JWT* представљају конкретне инфраструктурне механизме за приступ подацима и аутентификацију.

Заједнички допринос ових одлука јесте прегледна структура у којој се јасно разазнају места *HTTP* комуникације, апликационих операција, пословних правила и инфраструктурних детаља. На клијентској страни исти циљ подр-

жава организација по функционалностима. Вредност такве структуре огледа се у лакшем проналажењу кода, ограниченом утицају измена и могућности да се поједини делови система тестирају на одговарајућем нивоу.

У наредном поглављу биће описан домен платформе: корисничке улоге, основни ентитети и најважнији пословни токови. Након тога ће бити приказано како су принципи из овог поглавља примењени у серверском и клијентском делу апликације.

Глава 3

Анализа домена платформе

Након увођења архитектурних принципа, потребно је описати домен који апликација подржава. Архитектура система има смисла само ако је повезана са проблемом који се решава. У случају *EventOrganizer* платформе, тај проблем је организација стручних догађаја, управљање ресурсима неопходним за њихово одржавање и координација корисника који у том процесу учествују.

Ово поглавље описује основне улоге, ентитете и пословне токове у систему, са циљем да објасни шта систем моделује и због чега су поједини делови домена важни. Тиме се припрема основа за наредна поглавља, у којима ће бити описано како су ови појмови реализовани у серверском и клијентском делу апликације.

3.1 Општи опис система

EventOrganizer је веб платформа намењена организацији стручних догађаја као што су семинари, предавања, радионице и конференције. Основна идеја система је да подржи целокупан ток организације: од креирања догађаја, преко избора простора, предавача и опреме, до пријављивања учесника, праћења стања догађаја и прикупљања повратних информација.

У реалном процесу организације догађаја постоји више ограничења. При избору ресурса потребно је обезбедити простор одговарајућег капацитета, предаваче чија стручна област одговара теми догађаја и, по потреби, доступну опрему, при чему укупни трошкови не смеју премашити буџет. Поред тога, ресурси се могу користити само на једном месту у истом временском периоду, па је потребно спречити преклапања. Управо ова ограничења чине домен

занимљивим за примену јасног доменског модела и апликационих токова.

Систем није намењен само једној врсти корисника. Учесници догађаја користе платформу да би пронашли и пратили догађаје који их интересују. Организатори креирају догађаје и припремају њихово одржавање. Администратори управљају ресурсима, корисничким правима и захтевима који захтевају одобравање. Ова подела улога утиче и на структуру апликације, јер различити корисници имају различит поглед на исти систем.

3.2 Корисничке улоге

У систему постоје три основне улоге: учесник, организатор и администратор. Подела на улоге представља важан део пословних правила, јер одређује ко има право да покрене одређену операцију, а на клијентској страни утиче и на доступност страница и контрола. Њихове функционалне могућности организоване су хијерархијски: организатор поред учешћа на туђим догађајима управља сопственим догађајима, док администратор има најшири опсег права ради надзора над целим системом.

Учесник је основна корисничка улога. Он може да прегледа објављене догађаје, погледа детаље догађаја и пријави се за учешће. Након пријављивања може да прати статус своје пријаве, да је откаже уколико је то дозвољено и да након одржаног догађаја остави рецензију. Из перспективе учесника, систем треба да буде једноставан: важно је да догађаји буду јасно приказани, да процес пријаве буде разумљив и да корисник добије обавештења о битним променама.

Организатор је корисник који има право да креира и управља догађајима. Он дефинише основне податке о догађају, као што су назив, опис, време одржавања, капацитет, буџет, област и број потребних предавача. Организатор затим бира ресурсе неопходне за догађај: салу, предаваче и, по потреби, пакет опреме. Када је план припремљен, организатор подноси захтев за резервацију ресурса. Након одобравања, догађај може бити објављен. Истовремено може да прегледа објављене догађаје других организатора и пријави се за учешће.

Администратор има најшири опсег одговорности. Он управља ресурсима, обрађује захтеве за резервацију, одобрава или одбија захтеве корисника који желе да постану организатори и управља корисничким налозима. Поред тога, може да прегледа и управља свим догађајима, као и да учествује на

догађају који није сам креирао. Администраторска улога је важна јер обезбеђује контролу над ограниченим ресурсима и спречава да више организатора истовремено користи исту салу, предавача или опрему.

Ова подела улога омогућава да се систем понаша различито у зависности од контекста. На пример, исти догађај за учесника представља прилику за пријаву, за организатора представља објекат који треба припремити и објавити, а за администратора може бити повезан са захтевом за одобравање ресурса. Због тога је контрола приступа важан део домена, а не само технички детаљ.

3.3 Догађаји

Догађај је централни појам у систему. Он представља стручни скуп који организатор жели да припреми и објави. Основни подаци о догађају су назив, опис, време почетка и завршетка, капацитет, буџет, област, број потребних предавача и информација о томе да ли је потребна додатна опрема.

Догађај има свој животни циклус. Када се креира, налази се у стању нацрта. У том стању организатор може да мења податке и да припрема ресурсе. Након што су ресурси одобрени, догађај може бити објављен и тада постаје видљив учесницима. Објављен догађај може бити отказан или означен као завршен. Завршени и отказани догађаји више се не третирају као активни за уређивање.

Овакво моделирање статуса је важно јер спречава недоследна стања. На пример, није пожељно да учесници виде догађај који још нема обезбеђену салу или предаваче. Такође, није пожељно да се завршен догађај накнадно мења на исти начин као догађај у припреми. Због тога статус догађаја утиче на доступне операције и на приказ у корисничком интерфејсу.

3.4 Ресурси

Ресурси представљају све оно што је потребно за одржавање догађаја. У систему постоје три типа ресурса: сала, предавач и пакет опреме. Сваки ресурс има назив, опис, цену, оцену квалитета и статус. Статус може означавати да је ресурс доступан, недоступан или архивиран.

Сала је ресурс који обезбеђује простор за догађај. За њу је посебно важан капацитет, јер мора бити усклађен са очекиваним бројем учесника. Предавач је ресурс који је повезан са одређеном облашћу стручности. То омогућава да се приликом планирања догађаја предност да предавачима чија експертиза одговара теми догађаја. Пакет опреме представља техничку подршку, као што су пројектори, озвучење или друга опрема неопходна за одржавање догађаја.

Ресурси су ограничени, па систем мора да води рачуна о њиховој доступности. Поред постојања ресурса и његовог статуса, проверава се да ли је већ резервисан у термину у ком се планира нови догађај. Управљање ресурсима је зато тесно повезано са резервацијама и временом одржавања догађаја.

3.5 Резервације ресурса

Резервација ресурса повезује догађај са ресурсима који су му потребни. Она представља један од најважнијих токова у систему, јер од ње зависи да ли догађај може бити објављен. Организатор најпре прави нацрт резервације, бира салу, предаваче и, ако је потребно, пакет опреме. Док је резервација у нацрту, избор ресурса може да се мења.

Када организатор заврши избор, резервација се подноси на одобравање. У том тренутку добија статус поднетог захтева и чека одлуку администратора. Администратор може да одобри или одбије захтев. Ако га одобри, ресурси се сматрају прихваћеним за тај догађај и организатор може да настави ка објављивању. Ако га одбије, организатор може да ревидира избор и поново поднесе захтев.

У систему постоји и концепт временског задржавања захтева. Када је резервација поднета, она има рок до ког треба да буде обрађена. Ако рок истекне, резервација може прећи у стање истекле резервације. Ово је важно јер спречава да ресурси остану неодређено дуго блокирани без администраторске одлуке.

Резервација ресурса садржи број верзије који се користи за оптимистичко управљање конкурентношћу (енгл. *optimistic concurrency control*). При свакој измени верзија се увећава, а клијент у захтеву шаље верзију података на основу које је корисник донео одлуку. Сервер пре измене пореди очекивану и тренутну верзију; ако се оне разликују, захтев се одбија као конфликт и потребно је освежити податке. Тиме се спречава преписивање новије измене

без дуготрајног закључавања записа [18].

3.6 Пријаве учесника

Када је догађај објављен, пријављени корисник може да се пријави за учешће, без обзира на то да ли има улогу учесника, организатора или администратора. Није дозвољена пријава на догађај који је корисник сам креирао. Пријава повезује корисника са догађајем и има сопствени статус. Почетно стање је пријава на чекању, након чега она може бити потврђена, одбијена или отказана. Овакво моделирање омогућава организаторима да прате интересовање за догађај и да управљају учешћем у складу са капацитетом и правилима догађаја.

Пријава представља пословни објекат са сопственим стањем и правилима. Отказивање је дозвољено само за пријаву на чекању или потврђену пријаву, и то пре почетка догађаја. Одбијена или већ отказана пријава не може поново бити отказана. При одбијању се бележе разлог и корисник који је донео одлуку, што доприноси транспарентности и каснијем праћењу процеса.

Ток пријављивања је значајан и за корисничко искуство. Учесник треба да зна да ли је његова пријава примљена, потврђена или одбијена. Због тога су пријаве повезане са обавештењима о променама стања.

3.7 Рецензије, обавештења и извештаји

Након одржаног догађаја, учесник може оставити рецензију која садржи оцену и коментар. Оцена је целобројна вредност од 1 до 5, док је коментар обавезан, не сме бити празан након уклањања сувишних размака и може имати највише 2000 карактера. Рецензије су важне јер омогућавају прикупљање повратних информација о квалитету догађаја.

Обавештења информишу кориснике о важним променама. Организатор добија информације о одлуци администратора у вези са резервацијом ресурса, а учесник о статусу пријаве, отказивању догађаја и могућности остављања рецензије. Обавештење садржи наслов, поруку, тип и, по потреби, везу ка ентитету на који се односи.

Извештаји и увиди омогућавају организаторима и администраторима да сагледају резултате појединачних догађаја. Приказ обухвата број и статусе

пријава, попуњеност капацитета, просечну оцену, расподелу оцена и последње рецензије. Овај део система је значајан јер показује да платформа не служи само за унос података, већ и за доношење закључака на основу прикупљених информација.

3.8 Препорука ресурса

Дефиниција проблема. Поред ручног избора ресурса, систем садржи и функционалност препоруке. При формирању препоруке узимају се у обзир капацитет и област догађаја, број потребних предавача, потреба за опремом, расположивост ресурса, буџет и оцене њиховог квалитета. Задатак поступка је да за дати догађај предложи комбинацију једне сале, потребног броја предавача и највише једног пакета опреме, који се укључује само ако је догађају потребна опрема. Улаз чине подаци о догађају и скупови кандидата за сваки тип ресурса. Пре покретања оптимизације кандидати се филтрирају према статусу, временском преклапању, капацитету и области догађаја.

Проблем препоруке представља дискретни проблем оптимизације са ограничењима. Из скупа сала V бира се једна сала, из скупа предавача S бира се тачно k предавача, док се из скупа пакета опреме E бира један пакет само ако је догађају потребна опрема. Пошто су доменска ограничења примењена приликом припреме кандидата, комбинација је изводљива ако садржи тражени број ресурса и ако њена укупна цена не прелази буџет догађаја.

Критеријум оптималности. Комбинације се пореде лексикографски. Најпре се максимизује збир оцена квалитета изабраних ресурса, а при једнаком збиру оцена квалитета бира се комбинација са нижом укупном ценом. Ако су и квалитет и цена једнаки, ресурси се пореде по називу и идентификатору како би исти улаз увек дао исти резултат. У псеудокоду је тренутно најбоље решење означено са $R^{\star}$, а релација $R \succ R^{\star}$ значи да је нова комбинација боља према наведеном поретку.

Поступак. Потпуни преглед свих комбинација припремљених кандидата приказан је алгоритмом 1. Ознака k представља број потребних предавача за догађај d, $B(d)$ његов буџет, а $c(R)$ укупну цену комбинације R. Израз $\binom{S}{k}$ означава скуп свих комбинација од k предавача из скупа S.

Алгоритам 1 Препорука ресурса за догађај

Улаз: догађај d и кандидати: сале V, предавачи S и пакети опреме E
Излаз: најбоља изводљива комбинација или разлог неуспеха

```
ако V = ∅ онда
    врати неуспех: нема расположиве сале
ако |S| < k онда
    врати неуспех: нема довољно предавача
ако догађај захтева опрему и E = ∅ онда
    врати неуспех: нема расположивог пакета опреме
E_d ← E ако је опрема потребна, иначе E_d ← {∅}
R* ← ∅
за све v ∈ V уради
    за све P ∈ (S choose k) уради
        за све e ∈ E_d уради
            R ← {v} ∪ P
            ако e ≠ ∅ онда
                R ← R ∪ {e}
            ако c(R) ≤ B(d) и (R* = ∅ или R ≻ R*) онда
                R* ← R
ако R* = ∅ онда
    врати неуспех: нема комбинације у оквиру буџета
врати R*
```

Сложеност. Нека је $q = |E|$ ако је опрема потребна, односно $q = 1$ у супротном случају. Број комбинација које се разматрају износи

$$N = |V| \binom{|S|}{k} q.$$

Пошто се при обради једне комбинације сабирају цене и оцене квалитета k предавача и највише два додатна ресурса, временска сложеност основног потпуног прегледа је

$$O\left(|V| \binom{|S|}{k} qk\right).$$

Ова процена не обухвата претходно филтрирање и једнократно уређивање кандидата. Формирање детерминистичког кључа при потпуном изједначењу

може у најгорем случају додати фактор $\log k$, али не мења комбинаторну природу поступка [1]. Помоћна просторна сложеност генерисања комбинација је $O(k)$, јер се чувају индекси текућег избора и тренутно најбоље решење.

Веза са имплементацијом. Припрема кандидата и оптимизација реализоване су одвојено. Класа `ResourceCandidateProvider` приступа подацима и примењује доменска ограничења, док класа `ConstraintRecommendationOptimizer` добија припремљене скупове кандидата и извршава оптимизациони поступак без директног приступа бази података. Кандидати за сале, предаваче и опрему уређују се само једном, а комбинације предавача генеришу се индексно, без прављења нових низова за преостали део скупа при сваком рекурзивном кораку. Тиме се смањују привремене алокације, али се не мењају критеријум избора ни потпуни преглед кандидатских комбинација. Поступак је зато тачан у односу на задати критеријум, детерминистичан, објашњив и погодан за обим података посматран у овом раду. Оваква подела уједно олакшава независно тестирање припреме података и алгоритамске логике.

3.9 Ток планирања догађаја

Најважнији пословни ток у систему је планирање догађаја. Он почиње када организатор креира догађај у стању нацрта, у ком још није видљив учесницима. Организатор затим уређује податке о догађају и прелази на избор ресурса.

Избор ресурса може бити ручан или заснован на препоруци система и обухвата салу, потребан број предавача и, уколико је неопходан, пакет опреме. Док је резервација у стању нацрта, организатор може да мења избор, а по његовом завршетку подноси захтев за одобравање. Администратор прегледа захтев и може да га одобри, чиме се омогућава објављивање догађаја, или да га одбије, након чега организатор може да измени избор и поново поднесе захтев.

Овај ток повезује догађај, резервацију ресурса, саме ресурсе, корисничке улоге и обавештења, при чему сваки корак зависи од тренутног стања система. Зато доменска правила контролишу дозвољене прелазе и обезбеђују доследност целокупног процеса планирања.

3.10 Однос домена и архитектуре

Анализа домена показује зашто је у пројекту примењен модуларни монолит са слојевитом унутрашњом архитектуром. Систем садржи више ентитета са сопственим животним циклусима, више корисничких улога и више пословних токова који се међусобно повезују. Када би се ова логика налазила само у контролерима или компонентама корисничког интерфејса, било би тешко обезбедити доследност.

Описане пословне области истовремено представљају основу функционалних модула. Догађаји, ресурси и резервације чине језгро планирања, док пријаве, рецензије, обавештења и извештаји обухватају учешће корисника и праћење резултата догађаја. Аутентификација, управљање корисницима и захтеви за улогу организатора обезбеђују идентитет и права приступа осталим токовима. Границе не значе да су ове области изоловане: ток планирања их намерно повезује, али се правила и случајеви употребе задржавају у модулу ком припадају.

Унутар сваког модула доменски појмови издвајају се у доменски слој, апликациони токови у апликациони слој, а технички детаљи у инфраструктурни слој. Клијентски део затим приказује ове токове кориснику кроз странице и компоненте прилагођене његовој улози. У наредном поглављу биће приказано како су слојеви и функционални модули реализовани у серверском делу апликације.

3.11 Резиме анализе домена

Анализа домена показује да платформа обухвата више повезаних пословних целина: догађаје, ресурсе, резервације, пријаве, рецензије, обавештења, извештаје и корисничке улоге. Посебну сложеност уносе правила стања, временска преклапања, ограничени ресурси и различита права приступа. Због тога је домен погодан за примену модуларног монолита, слојевите архитектуре и јасно издвојених апликационих токова.

Глава 4

Имплементација серверског дела

Серверски део апликације *EventOrganizer* реализован је помоћу *ASP.NET Core* платформе. Његова улога је да клијентској апликацији обезбеди *API* за рад са догађајима, ресурсима, резервацијама, пријавама, корисницима, обавештењима и извештајима. Поред тога, серверски део је одговоран за примену пословних правила, проверу права приступа, рад са базом података и обраду грешака.

У складу са избором модуларног монолита, сервер се испоручује и покреће као једна апликација, али његов код није смештен у један пројекат у ком су контролери, приступ подацима и пословна правила помешани. Решење је подељено на више слојева, тако да сваки слој има јасну одговорност. Оваква организација омогућава да се сложенији пословни токови, као што су планирање догађаја или одобравање резервације ресурса, имплементирају на прегледан начин.

4.1 Организација решења

Серверски део је организован кроз четири главна *.NET* пројекта: *EventOrganizer.Api*, *EventOrganizer.Application*, *EventOrganizer.Domain* и *EventOrganizer.Infrastructure*. Они представљају унутрашње границе исте апликације, а не засебне сервисе који се независно покрећу. Ова подела прати идеју да се технички детаљи издвоје од пословних правила и да се сваки део система развија у оквиру своје одговорности.

Пројекат *EventOrganizer.Api* представља улазну тачку серверске апликације. Он садржи конфигурацију апликације, контролере, посредничке компоненте, *CORS* подешавања, *Swagger* конфигурацију и правила ауторизације. Овај слој прима *HTTP* захтеве и претвара их у команде или упите који се прослеђују апликационом слоју.

Пројекат *EventOrganizer.Application* садржи апликационе случајеве употребе. У њему се налазе команде, упити, припадајуће класе за обраду, валидатори, интерфејси ка инфраструктури, сервиси за ауторизацију у оквиру домена и логика која координише више ентитета. Овај слој одговара на питање шта систем треба да уради када корисник покрене одређену акцију.

Пројекат *EventOrganizer.Domain* садржи основне доменске ентитете и правила која важе над њима. Ту се налазе модели као што су догађај, ресурс, резервација ресурса, пријава, рецензија, обавештење и захтев за улогу организатора. Овај слој не треба да зна да ли се подаци чувају у *PostgreSQL* бази, нити да ли се захтеви примају преко *HTTP*-а.

Пројекат *EventOrganizer.Infrastructure* садржи техничке детаље: *Entity Framework Core* контекст, конфигурације ентитета, миграције, *ASP.NET Core Identity*, механизам аутентификације који обухвата *JWT access token* и *refresh token*, као и иницијално попуњавање података. Он имплементира интерфејсе које апликациони слој користи, али не треба да диктира доменска правила.

4.2 Функционални модули система

Четири *.NET* пројекта представљају хоризонталне архитектурне слојеве, док функционални модули представљају вертикалне пословне целине које пролазе кроз те слојеве. Модул, према томе, није исто што и пројекат. Његова граница се препознаје кроз одговарајући *API* контролер, команде и упите, доменске ентитете и правила, конфигурацију чувања података и клијентску функционалну целину. У реализованом систему издвајају се следећи модули и њихове одговорности:

Аутентификација и сесија: регистрација, пријављивање, одјављивање, издавање *JWT access token*-а и обнављање сесије помоћу *refresh token*-а.

Корисници: преглед корисника, суспендовање и поновно активирање налога, као и примена корисничких улога.

Захтеви за улогу организатора: подношење и повлачење захтева, као и одобравање или одбијање захтева од стране администратора.

Догађаји: креирање и измена нацрта, објављивање, отказивање, завршавање и јавни преглед догађаја.

Ресурси: каталог сала, предавача и опреме, њихове цене, квалитет, расположивост и архивирање.

Планирање и резервације: нацрт избора ресурса, провера конфликата и буџета, препорука комбинације ресурса, подношење захтева и административна одлука.

Пријаве учесника: пријављивање на догађај, потврђивање, одбијање и отказивање пријаве, уз контролу капацитета и верзије података.

Рецензије: креирање и измена оцене и коментара након учешћа на завршеном догађају.

Обавештења: стварање и приказ обавештења, број непрочитаних ставки и означавање једног или свих обавештења као прочитаних.

Извештаји и увиди: показатељи пријава, попуњености, оцена и рецензија за појединачне догађаје и збирни преглед доступан овлашћеним улогама.

Неки токови намерно повезују више модула. Планирање догађаја, на пример, користи податке модула догађаја и ресурса, али правила избора, препоруке и одобравања остају у модулу планирања и резервација. Слично томе, модул обавештења реагује на исходе резервација, пријава и промена догађаја, без преузимања њихових пословних правила. Оваква сарадња унутар једног процеса омогућава јасне одговорности без мрежне комуникације између делова система.

4.3 Покретање, конфигурација и убризгавање зависности

Конфигурација серверске апликације налази се у датотеци `Program.cs`, која представља улазну тачку и место повезивања компоненти (енгл. *composition*

root). Ту се региструју контролери, *CORS* политика, *Swagger* документација, апликациони и инфраструктурни сервиси, аутентификација и ауторизација. Овај део кода бира конкретне имплементације и повезује слојеве, али не садржи пословну логику.

Регистрације су подељене између апликационог и инфраструктурног слоја. Метода `AddApplication` региструје *MediatR* класе за обраду команди и упита, *FluentValidation* валидаторе и компоненту за валидацију укључену у ток обраде захтева (енгл. *request pipeline*), као и сервисе за обавештења, ауторизацију догађаја и препоруку ресурса. Метода `AddInfrastructure` региструје `AppDbContext` као имплементацију интерфејса `IApplicationDbContext`, сервис за податке о тренутном кориснику, *Identity* сервисе и механизме за издавање и опозив токена.

Зависности се преузимају кроз конструкторе. Класа за обраду команде или упита, на пример, захтева интерфејсе за приступ подацима и подацима о тренутном кориснику. *ASP.NET Core* контејнер за убризгавање зависности при извршавању обезбеђује регистроване имплементације [15]. Контекст базе и сервиси који деле контекст једног *HTTP* захтева регистровани су са животним веком *scoped*, па се у оквиру тог захтева користи иста инстанца сервиса. Класе за обраду које региструје *MediatR*, као и компонента `ValidationBehavior`, имају животни век *transient*, што значи да контејнер при сваком разрешавању зависности ствара нову инстанцу [12]. Оваква конфигурација чини зависности видљивим и омогућава њихову замену у тестовима.

Регистрација компоненти апликационог слоја приказана је у исечку кода 4.1. Експлицитна *transient* регистрација односи се на валидацију у току обраде захтева, док сервиси који учествују у обради једног захтева користе *scoped* животни век.

```
services.AddMediatR(configuration =>
    configuration.RegisterServicesFromAssembly(typeof(DependencyInjection).Assembly));
services.AddValidatorsFromAssembly(typeof(DependencyInjection).Assembly);
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));
services.AddScoped<EventAuthorizationService>();
services.AddScoped<INotificationService, NotificationService>();
services.AddScoped<IResourceCandidateProvider, ResourceCandidateProvider>();
services.AddScoped<IRecommendationOptimizer, ConstraintRecommendationOptimizer>();
```

Исечак кода 4.1: Регистрација компоненти апликационог слоја

CORS политика омогућава да клијентска апликација, која се у развојном окружењу покреће на посебној адреси, комуницира са *API*-јем. *Swagger* се

користи у развојном окружењу за преглед и испробавање крајњих тачака *API*-ја (енгл. *endpoint*). Ово је корисно током развоја јер омогућава брзу проверу захтева, одговора и ауторизационих правила.

При покретању апликације иницијализују се основни кориснички подаци, као што су улоге и почетни администратор. У развојном окружењу може се извршити и попуњавање демонстрационих података. То олакшава тестирање апликације, јер систем одмах има кориснике, догађаје и ресурсе над којима се могу испробати главни токови.

4.4 *API* слој

API слој излаже функционалности система клијентској апликацији. Контролери су организовани око главних доменских целина: аутентификације, догађаја, ресурса, резервација, пријава, рецензија, обавештења, корисника, захтева за улогу организатора и извештаја. Ова организација прати структуру домена, па је лакше разумети који део *API*-ја припада којој функционалности.

Контролери у овом систему имају ограничен скуп одговорности. Они преузимају податке из *HTTP* захтева, формирају одговарајућу команду или упит и тако формиран захтев прослеђују посредством библиотеке *MediatR*. На пример, контролер за креирање догађаја не садржи пословну логику те операције, већ податке из тела захтева пресликава у команду и враћа резултат апликационог слоја.

Овакав приступ смањује количину пословне логике у *API* слоју. То је важно јер контролери треба да остану везани за *HTTP* аспект система: статусне кодове, путање, параметре, тело захтева и тип одговора. Сама правила о томе да ли је догађај исправан, да ли корисник има право да га мења или да ли је резервација у дозвољеном стању припадају нижим слојевима.

API слој такође дефинише правила приступа помоћу политика. На пример, креирање и управљање догађајима доступно је организаторима и администраторима. Организатор управља нацртом резервације за сопствени догађај, док администратор управља ресурсима и одлучује о поднетим захтевима. Поједине јавне крајње тачке *API*-ја доступне су и непријављеним корисницима. Ова подела омогућава да се пословна улога корисника доследно примењује на нивоу сервера.

4.5 Апликациони слој

Апликациони слој представља место на ком се описују кориснички случајеви употребе. Он не садржи само појединачне функције, већ целине које одговарају стварним акцијама у систему: креирање догађаја, измена ресурса, подношење резервације, одобравање резервације, пријављивање учесника, стварање обавештења или читање извештаја.

Операције су организоване као команде и упити. Команда представља захтев који мења стање система, док упит представља читање података. За сваку команду или упит постоји класа за обраду која садржи логику извршавања. Она учитава потребне податке, проверава предуслове, позива доменске методе, чува измене и враћа одговор.

У овом слоју користи се *MediatR*, који омогућава да контролери не зависе директно од конкретних класа за обраду. Контролер шаље команду или упит, а *MediatR* проналази одговарајућу класу за обраду. Ово уводи јасну границу између *API* слоја и апликационе логике.

У овом решењу *MediatR* усмерава команде и упите, али се не користи за објављивање доменских догађаја (енгл. *domain events*). Њима се у доменски вођеном дизајну могу представити значајне промене стања и покренути реакције других делова система [16]. У постојећој реализацији последице промена стања, као што су стварање обавештења након одобравања резервације и отказивање повезаних пријава након отказивања догађаја, непосредно координишу класе за обраду команди у апликационом слоју. Оне позивају доменске методе, координишу измене повезаних ентитета и стварање обавештења, а затим све измене чувају једним позивом методе `SaveChangesAsync`. За посматрани обим система овај приступ задржава ток операције експлицитним, док би увођење доменских догађаја омогућило додатно раздвајање промене доменског стања од реакција других модула и представља могући правац даљег развоја.

Конкретан пример овог *CQRS* тока представља захтев за препоруку ресурса. Контролер посредством библиотеке *MediatR* шаље упит `GetEventRecommendationQuery`. Његова класа за обраду учитава догађај, проверава право управљања, прибавља кандидате и прослеђује их оптимизатору, као што је приказано у исечку 4.2. Тако је пут захтева јасно раздвојен на контролер, упит, класу за обраду, припрему кандидата и оптимизацију.

```
public async Task<EventRecommendationResponse> Handle(
    GetEventRecommendationQuery request,
    CancellationToken cancellationToken)
{
    var eventItem = await _dbContext.Events
        .AsNoTracking()
        .FirstOrDefaultAsync(
            eventItem => eventItem.Id == request.EventId,
            cancellationToken);

    if (eventItem is null)
    {
        throw new NotFoundException(nameof(Event), request.EventId);
    }

    _eventAuthorizationService.EnsureCanManage(eventItem);

    var candidates = await _candidateProvider.GetCandidatesAsync(
        eventItem,
        cancellationToken);

    var recommendation = _optimizer.Optimize(eventItem, candidates);

    return MapRecommendation(recommendation);
}
```

Исечак кода 4.2: Обрада упита за препоруку ресурса

Валидација је такође смештена у апликациони слој. За команде се дефинишу валидатори који проверавају да ли су улазни подаци исправни. На пример, при креирању догађаја потребно је проверити да ли су назив и област задати, да ли је време завршетка након времена почетка и да ли су капацитет, буџет и број предавача позитивни. Ове провере се извршавају у оквиру *MediatR* тока обраде захтева, пре него што захтев стигне до одговарајуће класе за обраду, тако да до пословне операције стижу већ проверени подаци.

Апликациони слој је погодан и за координацију више доменских објеката. На пример, одобравање резервације ресурса не односи се само на промену статуса резервације. Потребно је проверити ко је корисник који доноси одлуку, да ли резервација постоји, да ли је у стању у ком може да се одобри и да ли постоје конфликти са другим догађајима. Ова координација природно припада апликационом слоју, док појединачна правила стања остају у доменским ентитетима.

4.6 Доменски слој

Доменски слој садржи ентитете који представљају основне појмове система. Његова најважнија улога је да очува пословна правила независно од начина на који се систем користи. То значи да правило не треба да важи само када захтев долази са једне странице клијентске апликације, већ увек када се над ентитетом извршава одређена операција.

Ентитет догађаја садржи податке о називу, опису, термину, капацитету, буџету, области, броју потребних предавача и статусу. Он контролише операције као што су ажурирање, објављивање, отказивање и завршавање. На пример, догађај који је отказан или завршен не може се уређивати на исти начин као нацрт.

Ентитет резервације ресурса има сложенији животни циклус. Он почиње као нацрт, затим може бити поднет на одобравање, одобрен, одбијен, истекао или отказан. У оквиру резервације чува се избор сале, предавача и опреме. Правила као што су дозвољеност измена само у стању нацрта, могућност повлачења само поднетог захтева или одобравање само активне поднете резервације налазе се у доменском моделу.

Ресурси су моделовани као сала, предавач и пакет опреме. Сваки ресурс има статус, цену и оцену квалитета. Заједничка правила, као што су архивирање, означавање доступности и провера основних вредности, налазе се у заједничком моделу ресурса. Ово омогућава да различити типови ресурса деле опште понашање, али да истовремено имају своје специфичне податке.

Пријаве, рецензије, обавештења и захтеви за улогу организатора такође имају сопствена правила. Пријава може бити потврђена, одбијена или отказана у зависности од тренутног стања. Рецензија мора имати исправну оцену и коментар. Обавештење може бити означено као прочитано. Захтев за улогу организатора може бити одобрен, одбијен или повучен док је у стању чекања.

Овакав доменски слој смањује ризик да пословна правила буду распршена по систему. Апликационе класе за обраду могу да координишу ток, али правила која припадају ентитетима остају у самим ентитетима.

4.7 Инфраструктурни слој

Инфраструктурни слој садржи техничке механизме неопходне за рад апликације, међу којима централно место има трајно чување података. *PostgreSQL* је релациони систем за управљање базом података, а *Entity Framework Core* се користи као објектно-релациони мапер између *.NET* модела и табела базе [17, 25].

Релациони модел одговара структури домена јер су догађаји, ресурси, резервације, пријаве, рецензије и корисници повезани јасно дефинисаним односима. Примарни и страни кључеви чувају референцијални интегритет, ограничења штите исправност података, а трансакције омогућавају да се повезане измене изврше као једна целина. Ова својства су посебно важна при резервацији ресурса и пријављивању учесника, где више записа мора остати међусобно усклађено.

PostgreSQL је изабран као зрело решење отвореног кода са подршком за трансакције, ограничења, индексе и конкурентни приступ подацима. Његова интеграција са *.NET* окружењем остварена је преко *Npgsql* провајдера за *Entity Framework Core*, који је у инфраструктурном пројекту конфигурисан методом `UseNpgsql` [22]. Ова комбинација омогућава *LINQ* упите, праћење измена и управљање шемом помоћу миграција, уз задржавање могућности употребе *PostgreSQL* механизама када су потребни.

Централна класа за рад са базом је AppDbContext. Она садржи скупове ентитета као што су догађаји, ресурси, резервације, пријаве, рецензије, обавештења, кориснички захтеви и записи о токенима за обнављање сесије. Поред тога, овај контекст наслеђује *Identity* контекст, што омогућава да се корисници и улоге чувају у истој бази података као и доменски подаци.

Конфигурације ентитета издвојене су из самог контекста. Ово је корисно јер AppDbContext остаје прегледнији, док се правила мапирања, односи, ограничења и индекси дефинишу у посебним конфигурационим класама. Миграције се користе за еволуцију шеме базе података током развоја.

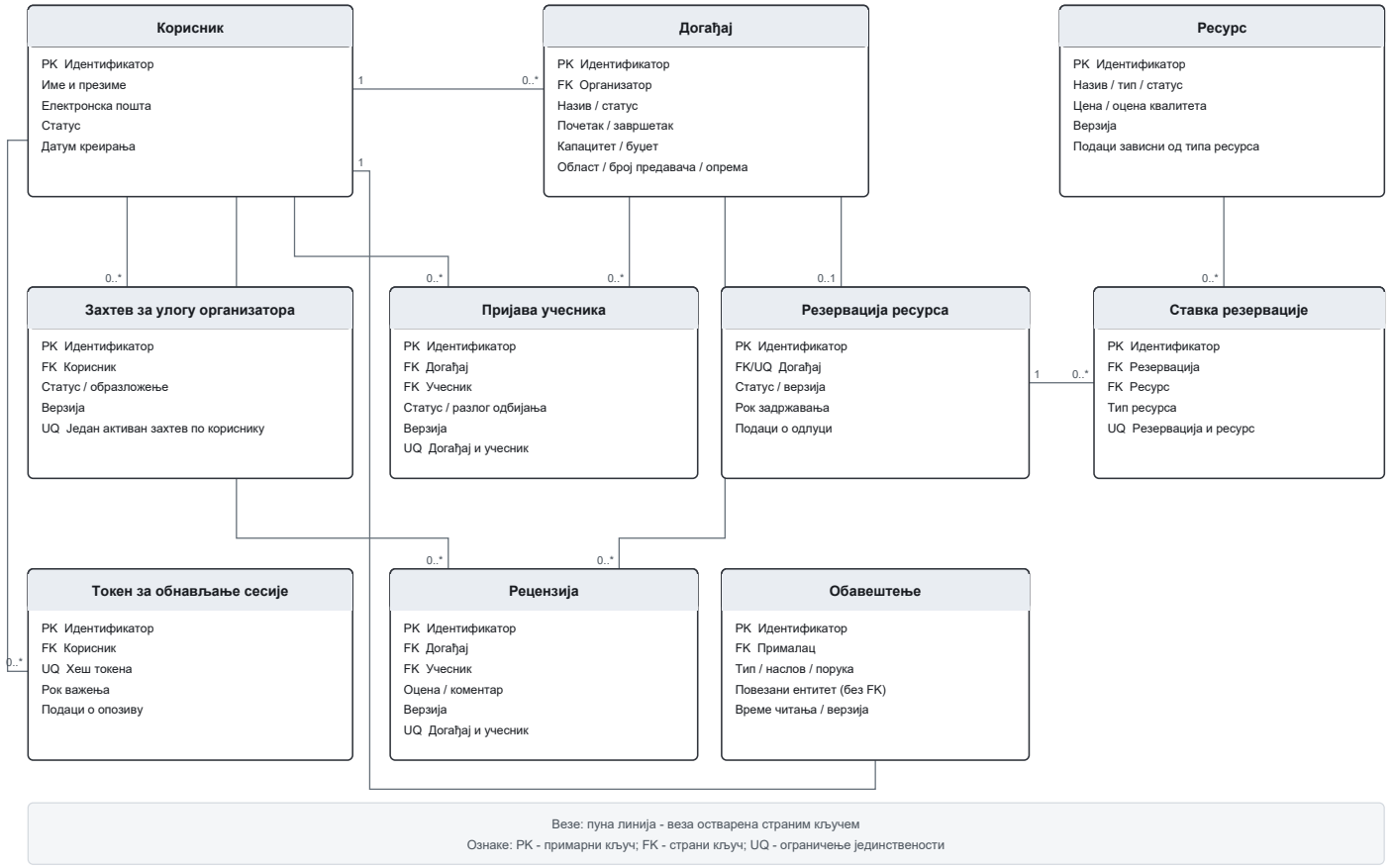
Инфраструктурни слој садржи и имплементацију аутентификације и управљања корисницима. *ASP.NET Core Identity* користи се за корисничке налоге и улоге. *JWT access token* служи за приступ *API*-ју након пријаве, док *refresh token* омогућава обнављање сесије без поновног уноса корисничких података. Вредност *refresh token*-а генерише се помоћу криптографски безбедног гене-

ратора случајних бројева, док се у бази чувају само њен *SHA-256* сажетак и подаци о року важења и опозиву. При успешној обнови претходни *refresh token* се опозива и замењује новим, а активни токени се опозивају и при одјављивању или суспендовању корисника [24].

Посебна пажња посвећена је и конкурентним изменама. Код операција које могу бити осетљиве на истовремене промене, као што су резервације ресурса, важно је да систем не дозволи недоследно стање. Због тога инфраструктурни слој подржава трансакције и обраду грешака које могу настати због конкурентних промена у бази.

Поједностављени приказ главних табела базе података дат је на слици 4.1. Ради читљивости нису приказане помоћне табеле механизма *ASP.NET Core Identity*, нити секундарне везе према администратору који је обрадио захтев или пријаву. Приказ обухвата табелу корисника јер се на њу ослањају главни пословни токови, као и табелу токена за обнављање сесије која подржава аутентификацију.

Сваки догађај има једног организатора и може имати највише једну резервацију ресурса. Резервација у стању нацрта може садржати нула или више ставки, док свака ставка припада једној резервацији и упућује на један ресурс. Приликом подношења захтева апликациона правила додатно проверавају да ли су изабрани сала, потребан број предавача и, када је неопходан, пакет опреме. Пријаве и рецензије повезују догађај са учесником, док обавештење припада једном кориснику. Различити типови ресурса чувају се у једној табели применом мапирања хијерархије у једну табелу (енгл. *Table-per-Hierarchy*, *TPH*), при чему дискриминаторска колона одређује конкретан тип ресурса [17].



Слика 4.1: Поједностављени приказ главних табела базе података

4.8 Аутентификација и ауторизација

Пријава корисника реализована је тако да сервер након успешне провере података издаје *JWT access token*. Клијент тај токен шаље уз наредне захтеве, а сервер на основу њега препознаје корисника и његове улоге. Овај приступ је погодан за апликације у којима су клијентски и серверски део одвојени.

У систему постоје три основне улоге: *Participant*, *Organizer* и *Admin*. Њихове функционалне могућности су хијерархијски организоване, па организатор и администратор, поред својих посебних права, могу да се пријаве на туђи објављени догађај и оставе рецензију након потврђеног учешћа. Ниједан корисник не може да се пријави на догађај који је сам креирао. Само организатор или администратор могу да креирају и управљају догађајима; организатор управља сопственим догађајима, док администратор има увид у све догађаје, мења каталог ресурса, одлучује о поднетим резервацијама и управља корисничким налозима. Ове политике су регистроване у *API* слоју и примењују се над контролерима или појединачним крајњим тачкама.

Важно је нагласити да ауторизација на серверу не зависи од тога шта је приказано на клијенту. Клијентска апликација може сакрити дугме кориснику који нема одговарајућу улогу, али сервер и даље мора да одбије захтев ако корисник нема право да изврши операцију. На тај начин се безбедносна правила примењују доследно.

4.9 Обрада грешака и одговори *API*-ја

У веб апликацији грешке могу настати из више разлога: неисправни улазни подаци, непостојећи ресурс, недостатак права приступа, конфликт при резервацији или неочекивана грешка на серверу. Ако се ове грешке обрађују појединачно у сваком контролеру, *API* постаје недоследан и тежи за одржавање.

Због тога *EventOrganizer* користи централизовану посредничку компоненту за обраду изузетака. Она хвата изузетке који настану током обраде *HTTP* захтева и претвара их у *JSON* одговор са *HTTP* статусом, насловом грешке, листом порука и, када је потребно, детаљима конфликта. На пример, грешка валидације враћа статус 400, неуспешна аутентификација статус 401, недостатак права приступа статус 403, непостојећи ресурс статус 404, а конфликт

статус 409.

Овај приступ је користан и за клијентску апликацију. Пошто грешке имају уједначен облик, клијент може на исти начин да прикаже поруку кориснику без обзира на то која је операција покренута. Посебно је значајно што одговор са статусом конфликта може да садржи детаље о ресурсима који се преклапају са другим догађајем.

4.10 Пример тока: планирање и објављивање догађаја

Планирање догађаја добро показује сарадњу свих серверских слојева. Организатор најпре креира догађај. *API* контролер прима захтев и прослеђује команду апликационом слоју. Валидатор проверава основне податке, а класа за обраду креира доменски ентитет и чува га кроз инфраструктурни слој.

Након тога организатор уређује нацрт резервације ресурса. Апликациони слој проверава да ли корисник има право да управља догађајем и да ли је резервација у стању које дозвољава измену. Доменски модел контролише промене стања и избор ресурса. Када организатор поднесе захтев, резервација добија статус поднете резервације и рок задржавања.

Администратор затим прегледа поднете резервације. Ако одобрава захтев, апликациони слој проверава да ли је резервација и даље активна и да ли нема конфликта са другим догађајима. Ако су услови испуњени, доменски модел мења статус резервације у одобрено. Након тога организатор може да објави догађај. Објављивање мења статус догађаја и он постаје доступан учесницима.

Овај пример показује серверски ток који укључује више корисника и ентитета, проверу права приступа, валидацију, доменска правила и рад са базом података. Слојевита организација омогућава да сваки део тог процеса буде имплементиран у оквиру одговарајуће одговорности.

4.11 Репозиторијум и покретање апликације

Изворни код практичног дела рада доступан је у *GitHub* репозиторијуму на адреси:

`https://github.com/biocanin01/event-organizer`

За локално покретање потребни су *.NET SDK*, *Node.js*, *PostgreSQL* и алат *dotnet-ef*. Подаци за повезивање са базом задају се изван репозиторијума, преко механизма *.NET User Secrets*. Следећа команда поставља развојну вредност кључа `ConnectionStrings:DefaultConnection`; ознаку `<lozinka>` треба заменити лозинком локалног корисника базе:

```
dotnet user-secrets set "ConnectionStrings:DefaultConnection" "Host=localhost;
    Port=5432;Database=event_organizer_demo;Username=postgres;Password=<
    lozinka>" --project backend/src/EventOrganizer.Api
```

Након креирања празне базе, миграције се примењују, а сервер покреће следећим командама из кореног директоријума репозиторијума:

```
dotnet ef database update --project backend/src/EventOrganizer.Infrastructure
    --startup-project backend/src/EventOrganizer.Api
dotnet run --project backend/src/EventOrganizer.Api
```

У развојном окружењу сервер је доступан на адреси `http://localhost:5117`, крајње тачке апликације налазе се испод путање `http://localhost:5117/api`, а документација *Swagger* на адреси `http://localhost:5117/swagger`.

Клијентски део покреће се из директоријума `frontend`, након инсталирања зависности:

```
cd frontend
npm install
npm run dev
```

Подразумевана адреса клијентске апликације је `http://localhost:5173`. Из кореног директоријума репозиторијума серверски и клијентски тестови могу се једнократно покренути командама:

```
dotnet test backend/EventOrganizer.slnx
npm --prefix frontend test -- --run
```

4.12 Резиме серверске имплементације

Серверски део подељен је на *API*, апликациони, доменски и инфраструктурни слој, чиме су правила одвојена од техничких детаља, а случајеви употребе изражени командама и упитима.

Глава 5

Имплементација клијентског дела

Клијентски део система представља једностраничну веб апликацију (енгл. *single-page application*, *SPA*) развијену помоћу библиотеке *React* и језика *TypeScript* [13, 20]. За развојно окружење и припрему продукционе верзије коришћен је *Vite* [34], док су за изградњу корисничког интерфејса употребљене компоненте библиотеке *Material UI* [21]. Ова комбинација омогућила је типски безбедан развој, поновну употребу компоненти и јасно раздвајање приказа од комуникације са сервером. *React* се у апликацији користи за декларативно формирање интерфејса и управљање његовим стањем, у складу са компонентним приступом који ова библиотека подржава.

Клијентски део приказује податке, омогућава покретање дозвољених операција и на разумљив начин представља резултат сваког захтева. Серверски део остаје одговоран за коначну проверу овлашћења, исправности података и прелаза између стања доменских објеката.

5.1 Организација клијентске апликације

Изворни код је организован према функционалним целинама које прате модуле серверског дела. Већина страница, типова и функција за приступ *API*-ју налази се унутар целина за аутентификацију, догађаје, ресурсе, резервације, пријаве учесника, рецензије, обавештења, извештаје, кориснике и захтеве за улогу организатора. На тај начин исти пословни појмови остају препознатљиви са обе стране *API* уговора.

Свака функционална целина садржи само оне делове који су јој потребни: *TypeScript* типове, функције за комуникацију са сервером, странице, дијалоге и помоћне компоненте. На пример, модул за догађаје обухвата приказ јавних и управљивих догађаја, формулар за креирање и измену, типове података, шеме валидације и *API* функције. Клијентска целина за планирање има интеграциону улогу: она на једној страници повезује догађаје, ресурсе, резервације и препоруке, али не дуплира правила тих серверских модула. Оваква организација олакшава проналажење кода и ограничава утицај измена на један део апликације.

Заједнички делови смештени су ван функционалних модула. Ту припадају конфигурација апликације, тема корисничког интерфејса, главни распоред странице, централни *API* клијент и компоненте за приказ статуса, датума и новчаних вредности. На самом врху хијерархије налазе се провајдери за тему, рутирање, аутентификацију и управљање серверским стањем. На тај начин све странице користе исту инфраструктуру, без понављања иницијализације.

TypeScript типови описују структуру захтева и одговора које клијент размењује са сервером. Њихова улога је да грешке у називима поља и употреби вредности буду уочене током развоја. Ипак, ти типови постоје само у време превођења, па се подаци које корисник уноси додатно проверавају шемама валидације, а сервер задржава улогу коначног ауторитета.

5.2 Рутирање, распоред и контрола приступа

Навигација је реализована библиотеком *React Router* [27]. Јавни део обухвата почетну страницу, преглед објављених догађаја, детаље изабраног догађаја, пријављивање и регистрацију корисника. Странице за пријављивање и регистрацију доступне су само анонимном кориснику, док се већ пријављени корисник преусмерава на почетну страницу за пријављене кориснике.

Заштићене руте су организоване према улогама описаним у анализи домена. Сваки пријављени корисник може да отвори страницу „Почетна“ и странице сопствених пријава и рецензија. Учеснику и организатору доступан је „Преглед догађаја“, док организатор на страници „Моји догађаји“ управља сопственим догађајима. Администратор на страници „Догађаји“ има увид у све догађаје у систему. Организатор и администратор могу да приступе ресурсима, планирању догађаја, управљању пријавама и извештајима, док су

управљање корисницима, одлучивање о захтевима за улогу организатора и одобравање резервација ресурса ограничени на администратора.

Компонента задужена за заштиту рута најпре чека да се утврди стање корисничке сесије. Ако сесија не постоји, корисник се преусмерава на страницу за пријављивање. Ако је корисник пријављен, али нема тражену улогу, преусмерава се на страницу „Почетна“. Ова провера побољшава корисничко искуство и спречава приказ недоступних страница, али не замењује серверску ауторизацију.

Након пријављивања приказује се заједнички распоред апликације са навигацијом прилагођеном улогама корисника. На већим екранима навигација је стално видљива у бочној траци, док се на мањим екранима отвара као привремени панел. Главни распоред садржи и податке о пријављеном кориснику, одјављивање и центар за обавештења. Захваљујући прилагодљивом распореду, исти токови рада остају доступни и на рачунару и на мобилном уређају.

5.3 Аутентификација и комуникација са сервером

Стање аутентификације чува се у *React* контексту. После успешног пријављивања или регистрације, клијент добија *JWT access token* и основне податке о кориснику. При покретању апликације, *refresh token* који сервер поставља у *HttpOnly* колачић користи се за обнављање сесије. Овај колачић није непосредно доступан *JavaScript* коду, већ га прегледач аутоматски шаље одговарајућем *API* захтеву.

Пошто прегледач колачиће шаље аутоматски, код оваквог механизма потребно је размотрити напад типа *CSRF* (енгл. *Cross-Site Request Forgery*), у ком злонамерна страница покушава да наведе прегледач пријављеног корисника да пошаље нежељени захтев другој апликацији [23]. У реализованом систему пословне операције захтевају *access token* у `Authorization` заглављу, које прегледач не додаје аутоматски, док се колачић са *refresh token*-ом користи само за обнављање и прекид сесије. Колачић има поставку *HttpOnly*, која онемогућава непосредан приступ из *JavaScript* кода, и *SameSite=Lax*, која ограничава његово слање у захтевима са других сајтова; поставка *Secure* примењује се уз протокол *HTTPS* [24]. За продукционо окружење овај механизам може се додатно ојачати посебним токеном за заштиту од *CSRF* напада или

провером заглавља *Origin* и *Referer*. Ове додатне провере нису део тренутне имплементације.

За комуникацију са сервером направљена је заједничка функција заснована на уграђеном `fetch` механизму. Она придружује основну адресу *API*-ја, поставља заглавље за *JSON* садржај, додаје *JWT access token* у `Authorization` заглавље и укључује слање колачића. Одговори без садржаја и *JSON* одговори обрађују се на једном месту. Неуспешан одговор претвара се у типизирани објекат грешке који, поред *HTTP* статуса и поруке, може да садржи грешке валидације и податке о конфликтима.

Заштићени захтеви пролазе кроз посебну функцију која користи тренутни *JWT access token*. Ако сервер врати статус 401, клијент једном покушава да обнови сесију и понови првобитни захтев са новим токеном. Уколико и обнављање не успе, локална сесија се уклања и корисник се сматра одјављеним. Овај поступак је централизован, па га појединачне странице не имплементирају засебно.

API функције су распоређене по функционалним модулима и представљају танак слој између компоненти и *HTTP* позива. Компонента, на пример, позива функцију за креирање догађаја или преузимање ресурса, без познавања детаља адресе и начина серијализације. Такво раздвајање смањује повезаност корисничког интерфејса са транспортним слојем.

5.4 Управљање серверским и локалним стањем

За податке преузете са сервера употребљена је библиотека *TanStack Query*. Упити се идентификују кључевима, а добијени резултати се привремено чувају у меморији. После успешне измене, одговарајући упит се поништава како би следеће учитавање приказало актуелно стање. Овај механизам је примењен на догађаје, ресурсе, резервације, пријаве, рецензије, извештаје и обавештења. Библиотека на тај начин раздваја серверско стање од локалног стања компоненти и поједностављује обраду учитавања, грешака и поновног преузимања података [32].

На нивоу целе апликације подешено је да се резултат упита сматра свежим 30 секунди, да се не освежава аутоматски при сваком повратку на прозор и да се неуспели упит понови једном. Поједини модули могу да прилагоде ово

понашање. За број непрочитаних обавештења примењено је периодично испитивање сервера (енгл. *polling*) сваких 30 секунди док је корисник пријављен и апликација активна, док се комплетна листа обавештења учитава тек када корисник отвори панел. Овај приступ је изабран јер се преноси само мала бројчана вредност, а кашњење до 30 секунди прихватљиво је за ову врсту обавештења. За посматрани обим једноставнији је од одржавања трајне *WebSocket* или *SSE* везе, док се заустављањем у позадини избегавају непотребни захтеви.

Локално стање остаје у компонентама када нема потребе да се дели са остатком апликације. Оно обухвата отвореност дијалога, изабране филтере, тренутни избор ресурса и привремене поруке о грешци. Оваква подела спречава стварање једног великог глобалног складишта и задржава стање што ближе месту на којем се користи.

5.5 Формулари и валидација уноса

Формулари за пријављивање, регистрацију, догађаје, ресурсе и захтев за улогу организатора реализовани су помоћу библиотеке *React Hook Form* [26]. Правила за проверу података дефинисана су *Zod* шемама [37] и повезана са формуларима преко одговарајућег адаптера. Корисник зато добија повратну информацију пре слања захтева када обавезно поље није попуњено, вредност није у дозвољеном опсегу или два поља нису међусобно усклађена.

Код догађаја се, између осталог, проверавају назив, опис, почетак и крај, капацитет, буџет, област и потребан број предавача. За ресурсе се правила мењају према изабраном типу. За салу су обавезни подаци о локацији и капацитету, за предавача област стручности и контакт, а за пакет опреме подаци о добављачу, подржаном капацитету и садржају пакета. Један формулар тиме подржава више сродних варијанти, без мешања њихових специфичних правила.

Клијентска валидација служи бржој повратној информацији, али исти подаци се поново проверавају на серверу. Грешке које сервер врати приказују се у оквиру странице или дијалога. Посебно су обрађени конфликти приликом резервисања ресурса, јер поред опште поруке садрже назив заузетог ресурса и термин догађаја са којим је дошло до преклапања.

5.6 Реализација главних корисничких токова

Јавни део апликације омогућава преглед објављених догађаја и отварање њихових детаља без пријављивања. Пријављени корисник може да поднесе пријаву за туђи догађај, прати њен статус и откаже је. Ове могућности доступне су учеснику, организатору и администратору, док управљачке странице зависе од њихове улоге. Организатор на страници „Моји догађаји“ управља сопственим догађајима и преко странице „Преглед догађаја“ проналази туђе објављене догађаје. Администратор на страници „Догађаји“ има увид у све догађаје у систему. По завршетку догађаја корисник са потврђеним учешћем може да унесе оцену и коментар, као и да накнадно измени сопствену рецензију у оквиру правила система.

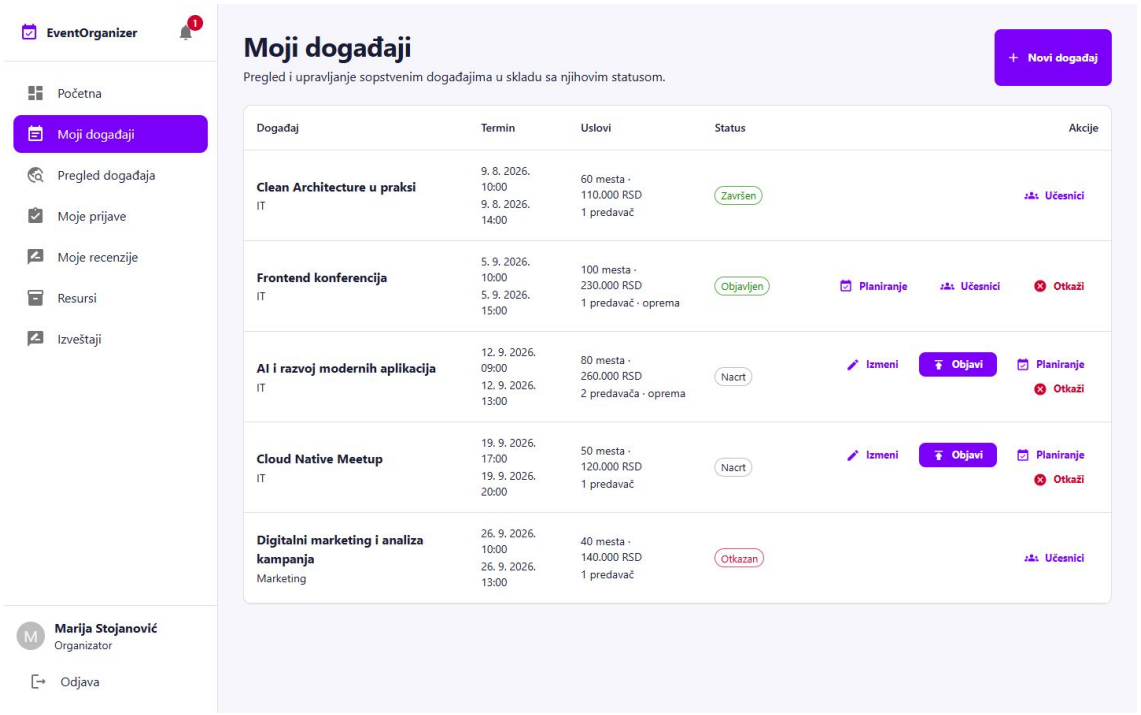
Организатор на страници догађаја добија приказ догађаја којима сме да управља и акције које зависе од њиховог тренутног статуса. Поред креирања и измене нацрта, доступни су отварање странице за планирање, преглед пријава учесника, објављивање, отказивање и означавање догађаја као завршеног. На страници пријава организатор може да потврди или одбије захтев учесника, при чему се одговарајуће листе освежавају после завршене операције.

На слици 5.1 приказан је преглед догађаја организатора. Поред основних података и статуса, у последњој колони приказују се само акције дозвољене у тренутном стању догађаја.

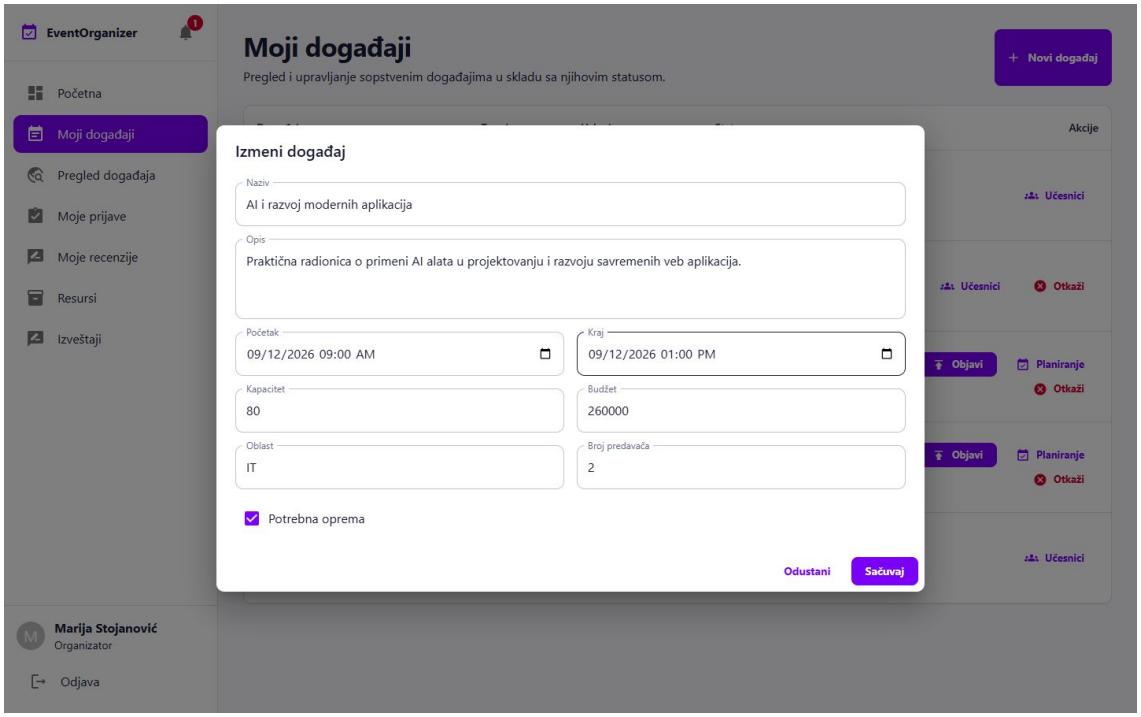
Формулар за измену догађаја приказан је на слици 5.2. Јединствен приказ обухвата основне податке, временски интервал и услове који се касније користе при планирању ресурса.

Модул ресурса приказује сале, предаваче и пакете опреме. Организатору је омогућен преглед података потребних за планирање, док администратор има и операције за додавање, измену, промену доступности и архивирање ресурса. Администратор такође обрађује поднете захтеве за резервацију ресурса и захтеве учесника за добијање улоге организатора, као и управљање корисничким налозима.

Центар за обавештења повезује више токова рада. Корисник види број непрочитаних ставки, може да означи једну или све ставке као прочитане, а избор обавештења води га до повезане странице. Обухваћена су обавештења о одлукама за улогу организатора, резервацијама ресурса, пријавама учесника, отказивању догађаја и доступности рецензије. Организатору и администра-



Слика 5.1: Преглед догађаја којима организатор управља



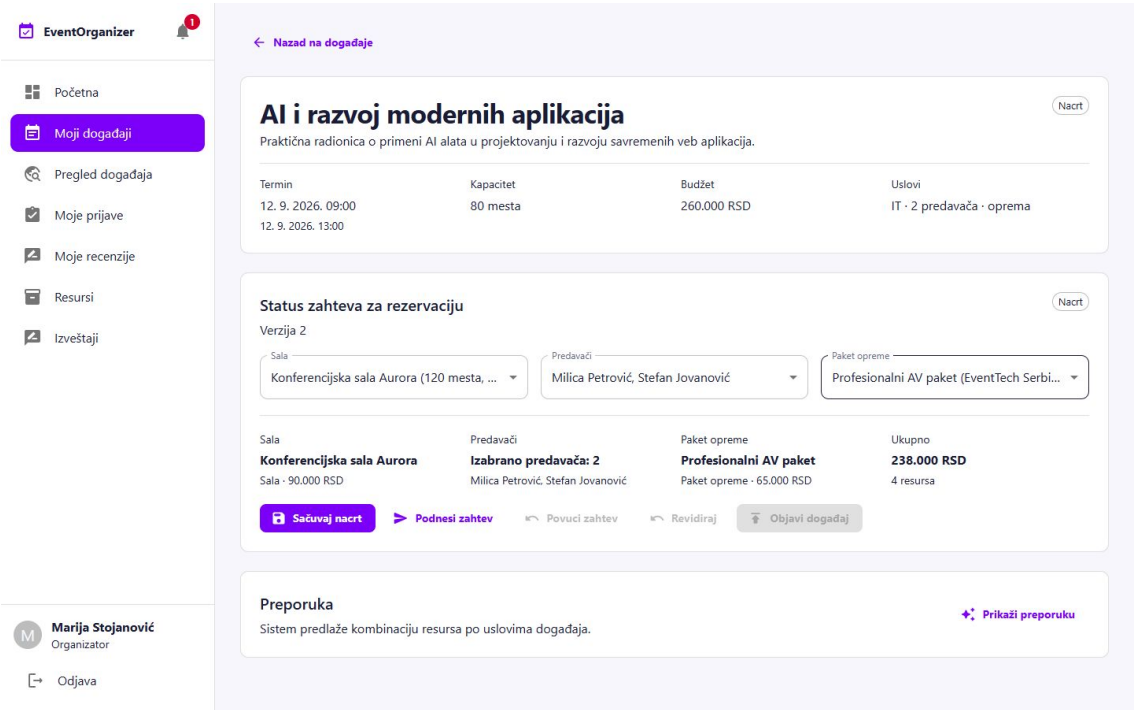
Слика 5.2: Формулар за измену основних података о догађају

тору доступни су и извештаји са бројем пријава, процентом попуњености, просечном оценом, расподелом оцена и последњим рецензијама за изабрани догађај.

5.7 Планирање догађаја као централни пример

Страница за планирање догађаја најпотпуније приказује сарадњу клијентског и серверског дела. При отварању странице паралелно се учитавају подаци о догађају, текући захтев за резервацију и расположиви ресурси. Кориснику су приказани термин, капацитет, буџет, област, број потребних предавача и податак о томе да ли је опрема неопходна. На основу тога бира салу, једног или више предавача и, када је потребно, пакет опреме.

Избор сале, предавача и пакета опреме приказан је на слици 5.3. На истој страници видљиви су услови догађаја, појединачни трошкови, укупна цена и акције које одговарају статусу резервације.



Слика 5.3: Избор ресурса приликом планирања догађаја

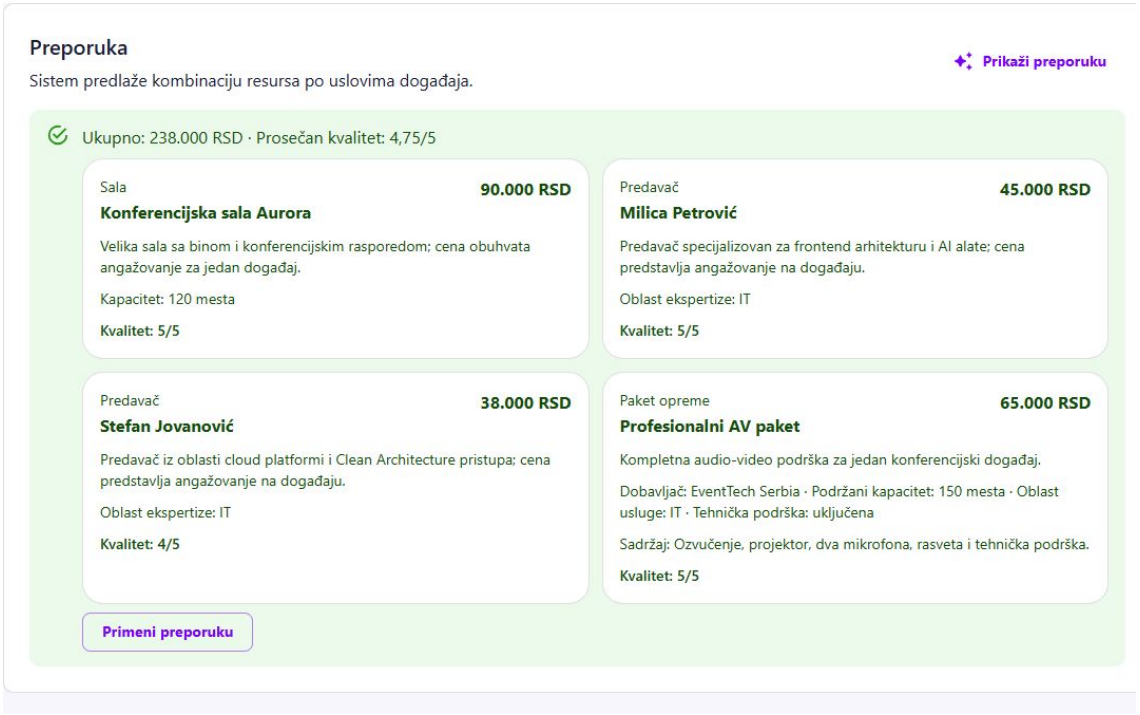
Избор се чува као локално стање све док корисник не покрене операцију чувања нацрта. Уз изабране ресурсе приказују се њихови основни подаци и укупна цена. Недоступни ресурси се не нуде за нови избор, али већ изабран ресурс остаје видљив како би постојећи нацрт могао исправно да се прикаже. Могућност измене и доступне акције зависе од статуса догађаја и захтева за резервацију.

Ток резервације прати стања дефинисана на серверу. Организатор може да сачува нацрт и поднесе га на одобравање. Док је захтев поднет, може да га повуче, а одбијен или истекао захтев може да врати на ревизију. Догађај се може објавити тек када је захтев за ресурсе одобрен. Клијент онемогућава акције које у датом стању немају смисла, док сервер поново проверава све предуслове и верзију објекта.

Поред ручног избора, страница нуди захтев за препоруку ресурса. Сервер враћа комбинацију која одговара условима догађаја или разлоге због којих изводљива комбинација не постоји. Алгоритам пореди комбинације према збиру оцена квалитета, док се кориснику ради лакшег тумачења приказује њихова просечна оцена на скали од 1 до 5. Ова два критеријума дају исти поредак зато што свака изводљива комбинација за посматрани догађај садржи исти број ресурса. Корисник може да примени препоруку на тренутни избор, али се она не чува аутоматски. Тиме систем задржава корисника као доносиоца коначне одлуке.

Слика 5.4 приказује успешан резултат препоруке, са предложеним ресурсима, укупном ценом и оценом квалитета. Дугме за примену преноси предложену комбинацију у текући избор, након чега организатор и даље самостално одлучује да ли ће је сачувати и поднети.

Ако сервер приликом подношења открије временско преклапање, клијент приказује сваки спорни ресурс и термин повезаног догађаја. После успешног чувања или промене статуса, подаци о резервацији се поново преузимају. Овај ток обједињује рутирање, контролу приступа, локално и серверско стање, обраду конфликата и условљене акције, па представља практичан пример архитектурних принципа разматраних у претходним главама.



Слика 5.4: Резултат препоруке ресурса

5.8 Резиме клијентске имплементације

Клијентски део апликације организован је тако да компоненте корисничког интерфејса, комуникација са сервером и управљање стањем имају јасно раздвојене одговорности. Функционална подела кода, заштићене руте, централизована обрада захтева, контролисани формулари и кеширање серверских података доприносе одрживости решења, док сервер остаје одговоран за коначну проверу пословних правила.

Глава 6

Тестирање и евалуација

Тестирање система обухвата доменска правила, апликационе токове, рад са базом података, *HTTP* комуникацију и понашање корисничког интерфејса. Овакав приступ је важан за *EventOrganizer* јер, поред изолованих грешака у коду, проверава неисправне прелазе стања, недоследна правила приступа и неусклађеност клијентског и серверског дела.

Аутоматизовани тестови организовани су у складу са архитектуром апликације. Серверски део проверава се помоћу *xUnit* окружења [36], док се за клијентски део користе *Vitest* и *React Testing Library* [35, 33]. Тестови су усмерени на критична пословна правила и главне корисничке токове, како би омогућили поуздану проверу најважнијих понашања и рано откривање регресија приликом измена система.

6.1 Верзије технологија и окружење за тестирање

Тестови и мерења покретани су у локалном развојном окружењу на оперативном систему *Microsoft Windows*, верзија 10.0.26200, на рачунару са процесором *Intel Core Ultra 7 255H*, 16 логичких процесора и 64 *GB* радне меморије. Коришћене верзије основних технологија и алата приказане су у табели 6.1. Верзије су утврђене командама `dotnet -info`, `node -version` и `npm -version`, као и провером датотека `.csproj`, `package.json` и `package-lock.json`. Датотека `package-lock.json` бележи тачне верзије *npm* пакета употребљене при инсталацији.

Табела 6.1: Верзије коришћених технологија и алата

Технологија или алат	Верзија
.NET SDK	10.0.301
ASP.NET Core	10.0.9
Entity Framework Core	10.0.9
Npgsql Entity Framework Core Provider	10.0.2
PostgreSQL	18
Microsoft Entity Framework Core SQLite	10.0.9
MediatR	14.1.0
FluentValidation	12.1.1
Node.js	24.16.0
npm	11.13.0
React	19.2.8
TypeScript	6.0.3
Vite	8.2.0
React Router	8.3.0
React Hook Form	7.84.0
Zod	4.4.3
TanStack Query	5.101.4
Material UI	9.2.0
xUnit	2.9.3
Vitest	4.1.10
React Testing Library	16.3.2
jsdom	30.0.1

PostgreSQL је база података саме апликације, док се *SQLite* провајдер за *Entity Framework Core* користи искључиво за интеграционе тестове и алат за мерење [19]. За сваки тест и сваку мерну серију формира се нова изолована база, без остатака претходног извршавања, потребе за спољном инстанцом базе и чишћења дељеног стања. Тиме се добијају бржа и поновљивија извршавања у једнаким почетним условима.

6.2 Организација тестова

Серверски део

Серверски тестови налазе се у засебном пројекту *EventOrganizer.Tests* и прате границе слојева продукционог кода. Обухваћени су јединични тестови доменског модела, тестови валидатора и класа за обраду команди и упита,

интеграциони тестови са базом података, инфраструктурни сервиси и *API* операције. Ова подела омогућава да се узрок грешке лакше локализује и да се сваки ниво система проверава на начин који одговара његовој одговорности.

Доменска правила и валидација

Доменски тестови непосредно проверавају ентитете, без покретања веб сервера и без приступа бази података. На тај начин се потврђује да основна пословна правила не зависе од инфраструктуре. Тестирани су ентитети догађаја, ресурса, резервација, пријава учесника, рецензија, обавештења и захтева за улогу организатора.

Код догађаја се, на пример, проверава да исправни подаци доводе до креирања нацрта, да време завршетка мора бити после времена почетка и да капацитет, буџет и број потребних предавача морају бити позитивни. Посебно се проверавају дозвољени и недозвољени прелази статуса, као што су објављивање нацрта и покушај објављивања отказаног догађаја. Сличан приступ примењен је на остале ентитете: тестови обухватају успешне операције, граничне вредности и покушаје нарушавања правила.

Валидатори апликационих захтева тестирани су независно од класа за обраду. Тако се проверавају обавезна поља, дозвољени опсези вредности и међусобна зависност улазних података. Додатно је проверен *MediatR* ток обраде захтева: неисправан захтев зауставља се пре извршавања пословне операције, док се исправан захтев прослеђује наредном кораку. Тиме се потврђује да је валидација централизована и да појединачни случајеви употребе не морају да понављају исте основне провере.

Апликациони и инфраструктурни тестови

За тестирање апликационог слоја користи се *SQLite* база у меморији. За сваки тест формира се изоловани контекст и креира потребна шема базе, без посебне конфигурације или покретања спољног процеса. Овакав приступ је бржи од покретања засебне *PostgreSQL* инстанце, али као релациона база и даље омогућава проверу *Entity Framework Core* упита, релација, уписа измена и поновног учитавања података. Због тога ови тестови представљају комбинацију тестова апликационе логике и лакших интеграционих тестова.

Тестови класа за обраду обухватају креирање, измену, објављивање, отказивање и завршавање догађаја, као и управљање ресурсима. Значајан део

тестова односи се на целокупан ток резервације. Проверава се чување непотпуног нацрта, подношење исправне резервације, обавезан избор опреме када је она потребна, откривање временског преклапања ресурса, поштовање буџета, рок задржавања и контрола верзије података. Обухваћене су и одлуке администратора, враћање резервације на дораду и спречавање измена које нису дозвољене у тренутном статусу.

Посебни тестови проверавају алгоритам за препоруку ресурса. Када постоји више изводљивих комбинација, бира се комбинација са највећим укупним квалитетом, а при једнаком квалитету предност има решење са нижом ценом. Проверавају се и случајеви у којима нема доступне сале, нема довољно предавача, недостаје обавезна опрема или све комбинације премашују буџет. Тиме се тестирају и успешан резултат и објашњив неуспех препоруке.

Инфраструктурни тестови проверавају да *JWT access token* садржи очекиване податке, да се за *refresh token* генеришу јединствене вредности које се исправно хеширају, као и да је њихов рок важења добро подешен. Проверена је и контрола статуса корисничког налога. Тестирани су сервиси који обезбеђују податке о тренутном кориснику, иницијално креирање улога и почетних података, као и трајност обавештења, пријава, резервација и рецензија. На овај начин провера не остаје само на резултату у меморији, већ обухвата и стање које се заиста чува у бази.

Интеграциони тестови *API*-ја

API се тестира помоћу класе *WebApplicationFactory*, која покреће апликацију у тестном окружењу. Продукциона база замењена је *SQLite* базом у меморији, док посебан тестни механизам аутентификације омогућава задавање идентитета и улога корисника. Захтеви се затим шаљу преко *HttpClient* објекта, на исти начин на који би их слао клијентски део апликације.

Ови тестови проверавају комплетан пут од *HTTP* захтева до одговора. За заштићене операције потврђује се разлика између статуса 401, када корисник није пријављен, и статуса 403, када је пријављен али нема одговарајућу улогу. Проверавају се и успешни одговори, статус 404 за недоступан ресурс и статус 409 за конфликте као што су застарела верзија података, попуњен капацитет или преклапање ресурса.

Интеграциони тестови обухватају управљање догађајима и ресурсима, токове резервација и пријава, рецензије, обавештења, извештаје, администра-

цију корисника и захтеве за улогу организатора. Поред појединачних *API* операција, проверени су и повезани токови. Један такав тест потврђује да потврђени учесник може да оцени завршен догађај и да се нова рецензија затим појављује у извештају. Тестови сесије проверавају да одјава поништава *refresh token* и спречава његову поновну употребу, као и да суспендовани корисник не може да настави да користи раније издату сесију.

Исечак кода 6.1 приказује репрезентативан интеграциони тест препоруке ресурса. Тест формира аутентификованог организатора и догађај са потребним ресурсима, шаље стварни *HTTP* захтев тестном серверу и проверава статус одговора. На овај начин једна провера обухвата рутирање, тестну аутентификацију, ауторизацију, приступ бази и извршавање апликационог случаја употребе.

```
[Fact]
public async Task GetRecommendation_WithOrganizerRoleAndOwnEvent_ReturnsOk()
{
    var organizerUserId = Guid.NewGuid();
    var client = await CreateAuthenticatedClientAsync(
        organizerUserId,
        ApplicationRoles.Organizer);
    var eventId = await CreateEventWithRecommendationResourcesAsync(
        organizerUserId);

    var response = await client.GetAsync(
        $"/api/events/{eventId}/recommendation");

    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

Исечак кода 6.1: Интеграциони тест приступа препоруци ресурса

Клијентски део

Клијентски тестови покрећу се помоћу *Vitest* окружења, док *React Testing Library* омогућава приказ компоненти и интеракцију са њима из перспективе корисника. *DOM* окружење обезбеђено је библиотеком *jsdom* [9]. Тестови углавном проналазе елементе према њиховој улози и видљивом називу, извршавају корисничку акцију и проверавају резултат који се појављује на екрану, чиме се провера усмерава на видљиво понашање компоненте.

Заједнички *API* клијент тестира се независно од страница. Мрежна комуникација замењује се контролисаном имплементацијом функције `fetch`, што омогућава проверу формирања адресе, *HTTP* методе, *JSON* садржаја, `Authorization` заглавља и слања колачића. Обухваћени су одговори са и без тела, структуриране *API* грешке и неочекивани формати одговора. Посебно је проверен ток у ком сервер враћа статус 401: клијент покушава да обнови сесију, добија нови *JWT access token* и само једном понавља првобитни захтев. Ако обнављање не успе, локална сесија се уклања и корисник се преусмерава на пријављивање.

Тестови рутирања и главне апликационе компоненте проверавају јавне и заштићене странице, приказ навигације према улози и спречавање приступа недозвољеним функционалностима. Обухваћени су главни токови управљања догађајима, резервацијама, пријавама, рецензијама, корисницима и извештајима. На пример, проверава се да организатор може да пошаље резервацију, да администратор може да донесе одлуку, да се пријава учесника потврђује уз исправну верзију података и да учесник не може да отвори извештаје намењене организатору и администратору.

Центар за обавештења тестиран је као засебна компонента. Проверава се приказ броја непрочитаних обавештења, учитавање листе тек након отварања центра, означавање једног или свих обавештења као прочитаних и навигација до повезаног садржаја. Обухваћен је и празан приказ на мобилном распореду. Ови тестови су значајни јер компонента повезује серверске податке, кеш библиотеке *TanStack Query*, корисничке акције и рутирање.

6.3 Резултати тестирања

Евалуација реализованог решења у овој глави има две целине. Прва целина је функционална провера аутоматизованим тестовима, уз мерење трајања целих тестних скупова и покривености кода. Друга целина је квантитативно мерење времена одзива препоруке ресурса при расту броја расположивих ресурса.

Актуелна верзија пројекта проверена је покретањем комплетног скупа серверских и клијентских тестова. Серверски тестови покренути су у оптимизованој конфигурацији *Release*, након претходног превођења серверских пројеката, док су клијентски тестови извршени у једнократном режиму алата *Vitest*.

У 5 независних покретања мерено је укупно протекло време извршавања, а у табели 6.2 приказана је медијана времена целог серверског, односно клијентског скупа. Сви тестови су успешно извршени, без прескочених случајева.

Табела 6.2: Резултати аутоматизованих тестова по нивоима провере

Ниво провере	Успешно / укупно	Медијана времена
Доменски тестови	82/82	5,87 *s*
Апликациони, инфраструктурни и лакши интеграциони	258/258	5,87 *s*
Интеграциони тестови *API*-ја	93/93	5,87 *s*
Клијентски тестови	73/73	23,29 *s*
Укупно	506/506	–

Вредност 5,87 *s* односи се на заједничко покретање 433 серверска теста, па је наведена уз сваки серверски ниво провере. Вредност 23,29 *s* односи се на целокупан скуп од 73 клијентска теста. Покривеност је намерно издвојена у табелу 6.3. Наведена времена описују посматрано локално окружење и не представљају гаранцију трајања на другом рачунару.

Сам број тестова није доказ квалитета архитектуре. Информативнији налаз је њихова расподела по нивоима и могућност да се сваки ниво проверава одговарајућим механизмом: доменска правила без базе и веб сервера, апликациони токови уз заменљиву инфраструктуру, целокупан *HTTP* пут кроз интеграционе тестове и видљиво понашање клијентске апликације. Тестови не обухватају само успешне сценарије, већ и недостатак права приступа, неисправне прелазе стања, конфликте ресурса, застареле верзије података и неуспешно обнављање сесије.

Покривеност продукционог кода мерена је алатима *Coverlet* на серверској и *V8 coverage* на клијентској страни [2, 35]. Из обрачуна су искључени тестни код, генерисане миграције, *Designer* датотеке и улазна датотека клијентске апликације. Табела 6.3 даје детаљан приказ покривености по слојевима. Покривеност не показује исправност тестова, али помаже у откривању делова система који су слабије проверени.

Табела 6.3: Детаљна покривеност продукционог кода аутоматизованим тестовима

Компонента	Линије (%)	Гране (%)
Доменски слој	95,21	79,03
Апликациони слој	91,64	79,12
Инфраструктурни слој	92,84	75,62
API слој	57,83	8,94
Серверски део укупно	87,91	63,44
Клијентски део укупно	78,82	70,98

Највећа серверска покривеност линија остварена је у доменском и инфраструктурном слоју, док је најнижа у *API* слоју. Ниска покривеност грана *API* слоја указује да постоје комбинације конфигурације и помоћне гране контролера које нису непосредно активиране постојећим тестовима. На клијенту је покривено 78,82% линија, при чему су заједнички *API* клијент и критични токови боље покривени од појединих формулара и страница.

6.4 Квантитативна евалуација препоруке ресурса

Квантитативна евалуација извршена је над тестним *API*-јем и *SQLite* базом у меморији, како би мерење било поновљиво и одвојено од локалних пословних података. Изолована база смањује утицај стања и оптерећења спољног процеса базе, па мерење јасније обухвата пролазак кроз *HTTP* слој и рад алгоритма препоруке. Посматрани догађај захтевао је једну салу, једног предавача и један пакет опреме. У сваком скупу 5% ресурса чиниле су сале, 5% пакети опреме, а преосталих 90% предавачи. Сви ресурси испуњавали су основне услове догађаја, а буџет није ограничавао избор, па је алгоритам морао да упореди све изводљиве комбинације.

За сваку величину скупа извршено је 5 независних серија. Свака серија покренута је над новом тестном базом и новом инстанцом тестне *API* апликације, без других паралелних оптерећења. У свакој серији првих 5 захтева није улазило у мерење, након чега је извршено 20 секвенцијалних *HTTP* захтева чије је време одзива забележено. Време припреме и попуњавања тестне базе није укључено у резултат. За сваку серију израчуната је медијана времена

одзива, а као коначан резултат приказана је медијана 5 медијана добијених у независним серијама. Свих 100 мерених захтева по величини скупа завршено је статусом 200.

Табела 6.4: Време одзива препоруке при расту броја ресурса

Ресурси	Сале	Предавачи	Опрема	Медијана времена одзива (*ms*)
100	5	90	5	1,57
200	10	180	10	2,11
400	20	360	20	10,51
800	40	720	40	64,21

Резултати показују нелинеаран раст. При удвостручавању скупа са 400 на 800 ресурса коначна медијана времена одзива порасла је приближно 6,1 пута. То је у складу са поступком који за посматрани сценарио обилази производ броја кандидата за салу, предавача и опрему. Једнократно сортирање кандидата и индексно генерисање комбинација смањују непотребне алокације, а време од 64,21 *ms* за 800 ресурса потврђује интерактиван одзив у целом посматраном опсегу.

6.5 Архитектурна оцена на основу проверљивих особина

Резултати тестирања показују практичну корист раздвајања одговорности. Доменска правила могу да се провере без базе и веб сервера, јер доменски слој не зависи од инфраструктуре. Класе за обраду команди и упита могу да се тестирају са стварним *Entity Framework Core* контекстом, док *API* тестови додају проверу рутирања, серијализације, аутентификације и ауторизације. Иста пословна операција тако може бити проверена на више нивоа, али сваки тест има јасну сврху.

Убризгавање зависности додатно подржава овакво тестирање. Пошто класе за обраду и сервиси примају зависности кроз конструктор, у тесту им се могу проследити контролисане имплементације интерфејса или контекст базе намењен конкретном сценарију. Тест зато не мора да покреће целу апликацију када проверава један случај употребе, а списак зависности сваке класе остаје јасно видљив.

Организација по функционалним целинама такође ограничава утицај измена. Додавање нове операције обично подразумева нову команду или упит, класу за обраду, валидатор и одговарајуће тестове, без потребе за изменом неповезаних модула. На клијентској страни, издвојени *API* клијент и контекст аутентификације омогућавају да се механизам комуникације тестира једном, док се странице усредсређују на конкретне корисничке токове.

Избор модуларног монолита омогућио је да се повезани токови проверавају у оквиру једне апликације и трансакционог контекста, без тестне инфраструктуре за више удаљених сервиса. Истовремено, тестови су груписани око области као што су догађаји, резервације, пријаве, рецензије и обавештења, па се грешка може локализовати у функционалном модулу ком понашање припада.

Централизована правила додатно смањују ризик од недоследности. Валидација пролази кроз *MediatR* ток обраде захтева, серверска ауторизација примењује политике независно од приказа на клијенту, а грешке се преводе у уједначене *HTTP* одговоре. Због тога тестови могу да очекују стабилно понашање и када захтев пролази кроз више слојева.

6.6 Ограничења

Иако постојећи тестови покривају велики број пословних сценарија, реализовани приступ има ограничења. Интеграциони тестови користе *SQLite* базу у меморији ради изолације, брзине и поновљивости, док продукционо окружење користи *PostgreSQL*. Зато резултати потврђују понашање апликационе логике и преносивих *Entity Framework Core* операција, али не и идентично понашање или перформансе продукционе базе. Разлике у типовима података, *SQL* дијалекту, ограничењима и конкурентном извршавању захтевају додатну проверу над *PostgreSQL* инстанцом; *SQLite* није део продукционе архитектуре.

Границе функционалних модула изражене су структуром директоријума, простора имена (енгл. *namespace*) и апликационих случајева употребе, а не засебним *.NET* склоповима са строго ограниченим јавним уговорима. То одговара обиму пројекта, док би се у већем систему границе могле ојачати провером зависности или издвајањем модулских интерфејса. Интерфејс `IApplicationDbContext` истовремено излаже типове `DbSet` библиотеке *Entity Framework Core*, па апликациони слој није у потпуности независан од иза-

браног *ORM* окружења. Ово практично поједностављење смањује количину посредничког кода, а апстракција се по потреби може сузити на операције појединачних случајева употребе.

Клијентски тестови користе *jsdom* и контролисане одговоре функције `fetch`. Они поуздано проверавају компоне и токове *React* апликације, али не замењују тестове у стварном прегледачу. Даље унапређење могло би да обухвати мањи скуп тестова који би покренули клијент, *API* и *PostgreSQL* базу и проверили критичне сценарије кроз кориснички интерфејс.

Спроведено мерење препоруке обухвата један секвенцијални сценарио и 5 независних серија, па не представља тест оптерећења. Нису обухваћени конкурентни захтеви, дуготрајан рад, мрежно кашњење, пенетрационо тестирање ни понашање продукционе *PostgreSQL* базе. Резултате покривености треба тумачити заједно са значајем тестираних сценарија, а не као самосталну меру исправности.

6.7 Резиме тестирања и евалуације

Тестирање је обухватило доменска правила, апликационе токове, инфраструктурне сервисе, *API* операције и главне токове клијентске апликације. Свих 506 тестова успешно је извршено, кључни слојеви имају високу покривеност, а интеграциони тестови проверавају цео *HTTP* пут. Могућност изоловане провере домена и замене базе у тестном окружењу представља конкретан показатељ проверљивости реализованог решења. Квантитативна евалуација допунила је ове налазе и потврдила интерактиван одзив препоруке у посматраном опсегу.

Глава 7

Закључак

У овом раду проучени су савремени архитектурни обрасци и размотрена је њихова примена у развоју веб апликација кроз реализацију платформе за организацију стручних догађаја. Посебна пажња посвећена је утицају архитектурних одлука на конкретне пословне токове, организацију кода, проверу исправности и могућност даљег развоја система.

Практични резултат рада је апликација *EventOrganizer*, која повезује управљање догађајима, ресурсима и корисницима са процесима резервације, пријављивања учесника, рецензирања, обавештавања и извештавања. Реализација ових функционалности показала је да домен организације догађаја није сведен на једноставне операције уноса и приказа података. Значајан део система чине правила стања, временска и капацитетска ограничења, различита права приступа и токови у којима учествује више корисничких улога.

7.1 Преглед урађеног

Систем је реализован као веб апликација са одвојеним клијентским и серверским делом, при чему је за серверску архитектуру изабран модуларни монолит. Клијентска апликација развијена је употребом технологија *React* и *TypeScript*, док је серверски део заснован на платформи *ASP.NET Core*. Подаци се чувају у *PostgreSQL* бази, а приступ подацима и управљање изменама шеме реализовани су помоћу *Entity Framework Core* окружења.

На серверској страни примењени су принципи чисте архитектуре, са јасно издвојеним *API*, апликационим, доменским и инфраструктурним слојем. *API* слој задужен је за *HTTP* комуникацију и правила приступа. Апликациони

слој описује случајеве употребе кроз команде и упите, док доменски слој чува стања ентитета и правила која над њима важе. Инфраструктурни слој обухвата рад са базом, корисничким налозима, токенима и другим техничким механизмима.

Раздвајање команди и упита, уз *MediatR*, омогућило је да свака значајна операција има јасно место у апликационом слоју. *CQRS* је примењен као организациони образац у оквиру једне апликације и заједничке *PostgreSQL* базе за читање и писање, а не као расподељено решење са одвојеним складиштима података. *FluentValidation* и *MediatR* ток обраде захтева обезбеђују да се неисправни захтеви зауставе пре извршавања пословне операције. Доменске методе затим штите правила као што су дозвољени прелази статуса, исправност временских интервала и услови под којима се догађај, резервација, пријава или рецензија могу изменити.

Убризгавање зависности повезује ове слојеве у улазној тачки апликације. Апликациони случајеви употребе зависе од интерфејса, док инфраструктурни слој обезбеђује имплементације за приступ подацима, рад са идентитетом корисника и издавање токена. Тиме су техничке зависности учињене видљивим и заменљивим, што је допринело прегледности структуре и лакшем тестирању.

Посебно значајан део система представља планирање ресурса. Организатор може да формира нацрт резервације, изабере салу, предаваче и опрему, затражи препоруку расположивих ресурса и поднесе захтев администратору. При томе се проверавају расположивост, преклапање термина, капацитет, област, број предавача, потреба за опремом и расположиви буџет. Овај ток је показао практичну вредност издвајања доменских правила од *HTTP* комуникације и приказа на клијенту.

Клијентска апликација организована је према функционалним целинама. Странице и компоненте прилагођене су улогама учесника, организатора и администратора, док сервер остаје коначни извор правила ауторизације. Заједнички *API* клијент централизује формирање захтева, обраду грешака и обнављање сесије, а *TanStack Query* управља серверским стањем, кеширањем и освежавањем података. На тај начин су приказ, комуникација и локално стање раздвојени без дуплирања пословних правила на клијенту.

Исправност решења проверена је аутоматизованим тестовима на више нивоа. Доменски тестови проверавају правила ентитета, апликациони и инфра-

структурни тестови обухватају класе за обраду команди и упита, валидацију и рад са базом, док интеграциони тестови проверавају *API* уговоре и ауторизацију. На клијентској страни тестирани су *API* клијент, рутирање, обнављање сесије, обавештења и главни кориснички токови. Свих 506 тестова успешно је извршено, без прескочених случајева. Овај резултат потврђује проверена понашања у обухваћеним сценаријима и показује да се делови система могу проверавати изоловано и интеграционо; сам број тестова не доказује погодност архитектуре.

Евалуација препоруке ресурса допунила је функционалне тестове квантитативним мерењем. У 5 независних серија тестног *API* окружења коначна медијана времена одзива порасла је са 1,57 *ms* за 100 ресурса на 64,21 *ms* за 800 ресурса. Резултат потврђује да оптимизована имплементација задржава интерактиван одзив у посматраном опсегу, уз детерминистичан избор истог квалитета и цене за исте улазе.

На основу реализованог система може се закључити да су постављени циљеви рада остварени. Одговорности су раздвојене, пословна правила су смештена у одговарајуће слојеве, а кључни токови могу да се мењају и тестирају без ослањања на једну велику и тесно повезану целину. Примењени обрасци нису уклонили сложеност домена, али су омогућили да она буде распоређена на места на којима се може лакше разумети и контролисати.

7.2 Правци даљег развоја

Даљи развој платформе може се одвијати у више праваца. Први правац односи се на комуникацију са корисницима. Постојећи систем обавештења може се проширити слањем електронске поште и подсетника пред почетак догађаја. Интеграција са календарским сервисима омогућила би да потврђени догађаји и пријаве буду додати у лични календар корисника, уз ажурирање у случају промене термина или отказивања.

Други правац је унапређивање планирања и препоруке ресурса. Постојећи алгоритам користи расположивост, капацитет, област, квалитет и цену. Мерење скалирања показало је да би пре рада са знатно већим скуповима требало смањити број комбинација раним одбацивањем, гранањем са ограничењима или применом специјализованог оптимизационог поступка. Модел се затим може проширити додатним критеријумима, као што су локација, исто-

ријска поузданост ресурса, приоритети организатора и сложенија ограничења опреме. Значајно би било и приказивање више рангираних предлога, како би организатор могао да упореди цену и квалитет алтернативних комбинација.

Извештавање се може проширити праћењем трендова кроз време, поређењем области и типова догађаја, као и извозом података у формате погодне за даљу анализу. За употребу у већим организацијама корисни би били детаљнији ревизиони записи, историја административних одлука и финије дефинисана права приступа.

Са техничке стране, следећи корак представљало би увођење тестова целог система у стварном прегледачу и интеграционих тестова са *PostgreSQL* базом. Тиме би се допуниле постојеће провере које користе *jsdom*, контролисане мрежне одговоре и *SQLite* базу у меморији. Пре продукционе употребе потребно је размотрити и тестирање оптерећења, безбедносне провере, централизовано праћење грешака, резервне копије и аутоматизован поступак испоруке апликације.

Архитектура реализованог решења оставља простор за наведена проширења без промене основне организације система. Нови случајеви употребе могу се додавати унутар постојећих функционалних и слојевитих граница, док се технички сервиси могу заменити или допунити кроз дефинисане интерфејсе. Управо та могућност постепеног развоја, уз очување јасних одговорности и проверљивости система, представља главни практични резултат примене архитектурних образаца разматраних у овом раду.

Литература

[1] Thomas H. Cormen и др. *Introduction to Algorithms*. 4th издање. MIT Press, 2022.

[2] Coverlet contributors. *Coverlet documentation and source repository*. url: `https://github.com/coverlet-coverage/coverlet` (посећено 24. 8. 2026).

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[4] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures". Докторска теза. University of California, Irvine, 2000.

[5] Martin Fowler. *CQRS*. 2011. url: `https://martinfowler.com/bliki/CQRS.html` (посећено 24. 8. 2026).

[6] Martin Fowler. *Monolith First*. 2015. url: `https://martinfowler.com/bliki/MonolithFirst.html` (посећено 24. 8. 2026).

[7] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[8] Michael Jones, John Bradley и Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. Internet Engineering Task Force, 2015. doi: `10.17487/RFC7519`. url: `https://www.rfc-editor.org/rfc/rfc7519` (посећено 24. 8. 2026).

[9] jsdom contributors. *jsdom: A JavaScript implementation of various web standards*. url: `https://github.com/jsdom/jsdom` (посећено 24. 8. 2026).

[10] James Lewis и Martin Fowler. *Microservices*. 2014. url: `https://martinfowler.com/articles/microservices.html` (посећено 24. 8. 2026).

[11] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[12] MediatR contributors. *MediatR documentation and source repository*. url: https://github.com/LuckyPennySoftware/MediatR (посећено 24. 8. 2026).

[13] Meta Open Source. *React documentation*. url: https://react.dev/ (посећено 24. 8. 2026).

[14] Microsoft. *ASP.NET Core documentation*. url: https://learn.microsoft.com/aspnet/core/ (посећено 24. 8. 2026).

[15] Microsoft. *Dependency injection in ASP.NET Core*. url: https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection (посећено 24. 8. 2026).

[16] Microsoft. *Domain events: Design and implementation*. url: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation (посећено 28. 8. 2026).

[17] Microsoft. *Entity Framework Core documentation*. url: https://learn.microsoft.com/ef/core/ (посећено 24. 8. 2026).

[18] Microsoft. *Handling Concurrency Conflicts*. Јул 5, 2023. url: https://learn.microsoft.com/en-us/ef/core/saving/concurrency (посећено 24. 8. 2026).

[19] Microsoft. *SQLite EF Core Database Provider*. url: https://learn.microsoft.com/en-us/ef/core/providers/sqlite/ (посећено 24. 8. 2026).

[20] Microsoft. *The TypeScript Handbook*. url: https://www.typescriptlang.org/docs/handbook/ (посећено 24. 8. 2026).

[21] MUI contributors. *Material UI documentation*. url: https://mui.com/material-ui/getting-started/ (посећено 24. 8. 2026).

[22] Npgsql Development Team. *Npgsql Entity Framework Core Provider documentation*. url: https://www.npgsql.org/efcore/ (посећено 24. 8. 2026).

[23] OWASP Foundation. *Cross-Site Request Forgery Prevention Cheat Sheet*. url: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html (посећено 24. 8. 2026).

[24] OWASP Foundation. *Session Management Cheat Sheet*. url: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html (посећено 24. 8. 2026).

[25] PostgreSQL Global Development Group. *PostgreSQL documentation*. url: `https://www.postgresql.org/docs/` (посећено 24. 8. 2026).

[26] React Hook Form contributors. *React Hook Form documentation*. url: `https://react-hook-form.com/` (посећено 24. 8. 2026).

[27] React Router contributors. *React Router documentation*. url: `https://reactrouter.com/` (посећено 24. 8. 2026).

[28] Mark Richards и Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[29] Jeremy Skinner и contributors. *FluentValidation documentation*. url: `https://docs.fluentvalidation.net/` (посећено 24. 8. 2026).

[30] Steve Smith. *Architect Modern Web Applications with ASP.NET Core and Azure*. Верзија 8.0. Јан. 18, 2025. url: `https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/` (посећено 24. 8. 2026).

[31] Ian Sommerville. *Software Engineering*. 10th издање. Pearson, 2016.

[32] TanStack. *TanStack Query documentation*. url: `https://tanstack.com/query/latest` (посећено 24. 8. 2026).

[33] Testing Library contributors. *React Testing Library documentation*. url: `https://testing-library.com/docs/react-testing-library/intro/` (посећено 24. 8. 2026).

[34] Vite contributors. *Vite documentation*. url: `https://vite.dev/guide/` (посећено 24. 8. 2026).

[35] Vitest contributors. *Vitest documentation*. url: `https://vitest.dev/guide/` (посећено 24. 8. 2026).

[36] xUnit.net project. *xUnit.net documentation*. url: `https://xunit.net/` (посећено 24. 8. 2026).

[37] Zod contributors. *Zod documentation*. url: `https://zod.dev/` (посећено 24. 8. 2026).

Биографија аутора

Александра Биочанин рођена је 9. јануара 2001. године у Крушевцу. Основну школу „Нада Поповић“ завршила је у родном граду. Средњошколско образовање истовремено је стицала у Гимназији у Крушевцу, на природно-математичком смеру, и у Музичкој школи „Стеван Христић“, такође у Крушевцу, а обе школе завршила је 2019. године.

Исте године уписала је Математички факултет Универзитета у Београду. Основне академске студије на смеру Математика и рачунарство завршила је у фебруару 2024. године, након чега је на истом смеру уписала мастер академске студије. Професионалну каријеру започела је у фебруару 2025. године као софтверски инжењер, радећи на развоју клијентског и серверског дела веб апликација (енгл. *full-stack developer*).