Razvoj web rješenja korištenjem klijent – server arhitekture uz primjenu uzoraka dizajna

Mahnet, Domagoj

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:818502

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Mahnet

RAZVOJ WEB RJEŠENJA KORIŠTENJEM KLIJENT – SERVER ARHITEKTURE UZ PRIMJENU UZORAKA DIZAJNA

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Mahnet

Matični broj: 48823

Studij: Informacijsko i programsko inženjerstvo

RAZVOJ WEB RJEŠENJA KORIŠTENJEM KLIJENT – SERVER ARHITEKTURE UZ PRIMJENU UZORAKA DIZAJNA

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2022.

Domagoj Mahnet

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Cilj ovog rada je prikazati proces izrade jednog web rješenja. Rad se sastoji od teorijskog i praktičnog dijela. U teorijskom dijelu biti će objašnjeni moderni arhitekturni uzorci računalnih programa, web servisi i uzorci dizajna. Također će se objasniti ASP.NET Core, Angular i Entity Framework kao tehnologije odabrane za razvoj praktičnog dijela. Praktični dio odnosit će se na izradu web aplikacije korištenjem klijent – server arhitekture. Primijeniti će se i neki od uzoraka dizajna. Tematika izrađene aplikacije odnositi će se na vođenje građevinskih projekata. Objasniti će se arhitektura cijelog rješenja, kao i arhitekture klijentske i serverske aplikacije. Biti će prikazani i najvažniji prikazi aplikacije, te će se prikazati i njihov programski kod.

Ključne riječi: arhitektura, web aplikacija, uzorci dizajna, klijent-poslužitelj, ASP.NET, Angular

Sadržaj

1. Uvod

Tema ovog diplomskog rada proizlazi iz velike želje za potpunim razumijevanjem arhitekture modernih distribuiranih programskih sustava, posebice web aplikacija. Stoga ovaj rad upravo time i započinje, teoretskom podlogom arhitekturnih uzoraka. Proces razvoja računalnih programa sličan je izgradnji kuće. Ako je temelj jak, izdržat će testove vremena i služiti potrebama tijekom svog životnog vijeka. Dobra arhitektura računalnog programa pomaže u definiranju atributa kao što su izvedba, kvaliteta, skalabilnost, mogućnost održavanja, upravljivost i upotrebljivost. Vrlo je važno metodično promišljanje arhitekture softvera za učinkovit razvoj prije pisanja prvog retka izvornog koda.

Nadalje, rad opisuje moderne web servise, s osebitim fokusom na REST API, kao najkorištenijom vrstom API-ja današnjice. Web servisi omogućuju različitim aplikacijama da međusobno komuniciraju i međusobno dijele podatke i usluge. Druge aplikacije također mogu koristiti usluge web servisa. Do evolucije web servisa došlo je kada su sve glavne platforme mogle pristupiti internetu, ali različite platforme nisu mogle međusobno komunicirati. Web servisi su podigli platforme na višu razinu objavljivanjem funkcija, poruka, programa ili objekata na internetu.

Nakon toga slijede uzorci dizajna, koji čine značajan dio ovog rada. Opisati će se svi temeljni uzorci dizajna, te će se obratiti i pozornost na neke novije, modernije uzorke. Uzorak dizajna pruža opće rješenje za uobičajene probleme koji se javljaju u dizajnu računalnih programa. Ideja je ubrzati proces razvoja pružanjem dobro testiranih, dokazanih paradigmi razvoja/dizajna.

Kao teorijska podloga korištenim alatima u praktičnom dijelu, opisati će se .NET i Angular razvojni alati, kao i objektno – relacijsko mapiranje. Korištenjem .NET tehnologije i Angulara, razvojni inženjer može napisati moćnu web aplikaciju koja ispunjava poslovne potrebe na neodređeno vrijeme. Projekti koji uključuju vrlo složenu arhitekturu zahtijevaju česta, ali besprijekorna testiranja. .NET Core aplikacija koja koristi Angular može se dizajnirati kao monolitni projekt, projekt servisa i web klijenta ili čak kao projekt s više klijenata.

U svrhu izrade praktičnog dijela ovog rada, izraditi će se web rješenje temeljeno na klijent – server arhitekturnom uzorku, s time da će serverska aplikacija biti izgrađena na temelju višeslojne arhitekture. Prilikom izrade ovog rješenja, osim arhitekturnih, koristiti će se i uzorci dizajna. Kao tehnologije za razvoj, odabrane su .NET za razvoj serverske aplikacije i Angular za razvoj klijentske aplikacije. Za sustav upravljanja relacijskom bazom podataka odabran je MSSQL.

2. Metode i tehnike rada

Za izradu i uređivanje ovog rada koristi se Microsoft Word program za obradu teksta. Draw.io alat na internetu primjenjuje se za izradu grafičkih prikaza, kao i za izradu dijagrama slučajeva korištenja i dijagrama aktivnosti.

Što se tiče praktičnog dijela aplikacije, za razvoj klijentske aplikacije koristi se Visual Studio Code uređivač koda, a za razvoj serverske aplikacije Visual Studio 2022 integrirano razvojno okruženje. Za potrebe konfiguriranja, upravljanja i administriranja bazom podataka, koristi se Microsoft SQL Management Studio. Pomoću Visual Paradigm Enterprise alata generira se dijagram klasa.

3. Arhitektura računalnih programa

Računalni programi kao proizvodi su kombinacija programskih rutina, procedura, modula ili objekata koji pružaju određenu funkcionalnost. Računalni programi su zapravo jezik koji se pretvara u električne struje unutar procesorske jedinice koja izvodi matematičke izračune. Prevode se programske naredbe odnosno funkcije koje predstavljaju osnovnu operaciju računala koja daje jedan rezultat kada se pozove, time omogućujući manipulaciju podacima. Stoga je bitno da računalni program bude osmišljen tako da odgovori na cijeli niz funkcionalnih ponašanja, koja zajedno čine konačni proizvod.

Arhitektura računalnog programa predstavlja dekompoziciju zahtjeva na funkcije i podfunkcije koje su potrebne za pružanje specificiranog ponašanja i karakteristika izvedbe. Arhitektura softvera odnosi se na umjetnost, struku, vještinu i znanost dizajniranja i implementacije softverskih proizvoda. Sastoji se od tri dijela: 1) zahtjeva za proizvod; 2) funkcionalne arhitekture koja pokazuje funkcionalne karakteristike, karakteristike izvedbe i korištenja resursa; i 3) fizičke arhitekture koja uspostavlja strukturnu konfiguraciju softverskog proizvoda i odnose među strukturnim elementima. Može se reći da je arhitektura softvera analogna skupu inženjerskih crteža i dijagrama na građevinskom području. Kao i kod građevine, implementacija (dizajn, kodiranje i testiranje modula, itd.) računalnog programa ne bi trebala započeti dok se sama arhitektura ne dovrši, dok se ne može pokazati da je u skladu sa zahtjevima i da je autorizirana od strane voditelja projekta. Nije preporučljivo započeti "izgradnju" bez razumijevanja punog opsega inženjerske odgovornosti.

Arhitektura računalnih programa pruža nekoliko tipova dizajna dijagrama, crteža ili pogleda koji predstavljaju nedvosmislenu strukturu i ponašanje računalnih programa kao krajnjih proizvoda. Ova su gledišta nužna za prenošenje arhitektonskih koncepata članovima tima za razvoj računalnih programa i drugim sudionicima. Također i definira mehanizam pomoću kojeg korisnik ili operater može promatrati, komunicirati i kontrolirati računalni program i njegove operacije. Tipični elementi korisničkog sučelja uključuju zvukove, svjetla obavijesti, prozore ili obrasce, dijaloške okvire i formate izvješća. Uz to, pokazuje i na upotrebljivost računalnih programa u smislu jednostavnosti manevriranja između značajki za kontrolu procesa, unos podataka, dohvaćanje podataka, manipulaciju podacima i generiranje izvješća. Arhitektura softvera mora biti dizajnirana tako da najbolje iskoristi dostupne resurse računalnog okruženja, kao što su brzina obrade, brzine prijenosa podataka, memorija, pohrana podataka i komunikacijske mogućnosti [1].

Naziv arhitektonski uzorak, sličan uzorcima dizajna, stvara jedno ime koje služi kao skraćenica između iskusnih arhitekata. Na primjer, kada arhitekt govori o slojevitom monolitu,

njegova sugovornik u razgovoru razumije aspekte strukture, koje vrste karakteristika arhitekture dobro funkcioniraju (a koje mogu uzrokovati probleme), tipične modele implementacije, podatkovne strategije i niz drugih informacija. Stoga bi arhitekti trebali biti upoznati s osnovnim nazivima temeljnih generičkih arhitektonskih stilova.

Nekoliko temeljnih uzoraka pojavljuje se iznova kroz povijest softverske arhitekture jer pružaju korisnu perspektivu organiziranja koda, implementacije ili drugih aspekata arhitekture. Na primjer, koncept slojeva u arhitekturi, koji razdvaja različite probleme na temelju funkcionalnosti, star je koliko i sam softver. Ipak, slojeviti se uzorak i dalje manifestira u različitim oblicima, uključujući i moderne varijante nastale evolucijom uzorka.

Arhitekti odsutnost bilo kakve uočljive arhitektonske strukture nazivaju velikom kuglom od blata (eng. Big Ball of Mud), nazvanom po istoimenom anti-uzorku definiranom u radu koji su 1997. objavili Brian Foote i Joseph Yoder: „Velika lopta od blata nasumično je strukturirana, raširena, traljavo, ljepljiva traka i žica za baliranje, džungla od špageta. Ovi sustavi pokazuju nepogrešive znakove nereguliranog rasta i opetovanog, svrhovitog popravka. Informacije se promiskuitetno dijele među udaljenim elementima sustava, često do točke u kojoj gotovo sve važne informacije postanu globalne ili duplicirane.“

Modernim riječima, velika kugla blata mogla bi opisati jednostavnu aplikaciju za skriptiranje s upraviteljima događajima povezanim izravno s bazom podataka, bez stvarne unutarnje strukture. Mnoge trivijalne aplikacije počinju ovako, a zatim postaju nezgrapne kako nastavljaju rasti. Općenito, arhitekti pod svaku cijenu žele izbjeći ovu vrstu arhitekture, zbog toga što nedostatak strukture čini naknadnu promjenu sve težom. Ova vrsta arhitekture također pati od problema u testiranju, skalabilnosti i performansama.

Arhitekturni stilovi mogu se klasificirati u dvije glavne vrste: monolitni (jedna jedinica za implementaciju cijelog koda) i distribuirani (više jedinica za implementaciju povezanih putem protokola za udaljeni pristup). Iako nijedna klasifikacijska shema nije savršena, sve distribuirane arhitekture dijele zajednički skup izazova i problema koji se ne nalaze u monolitnim arhitektonskim stilovima, što ovu klasifikacijsku shemu čini dobrim odvajanjem između različitih arhitektonskih stilova.

Monolitni uzorci:

- Slojevita arhitektura
- Mikro – jezgrena arhitektura
- Klijent – server arhitektura

Distribuirani uzorci:

- Arhitektura vođena događajima

- Mikroservisna arhitektura
- Arhitektura temeljena na servisima [2]

3.1. Slojevita arhitektura

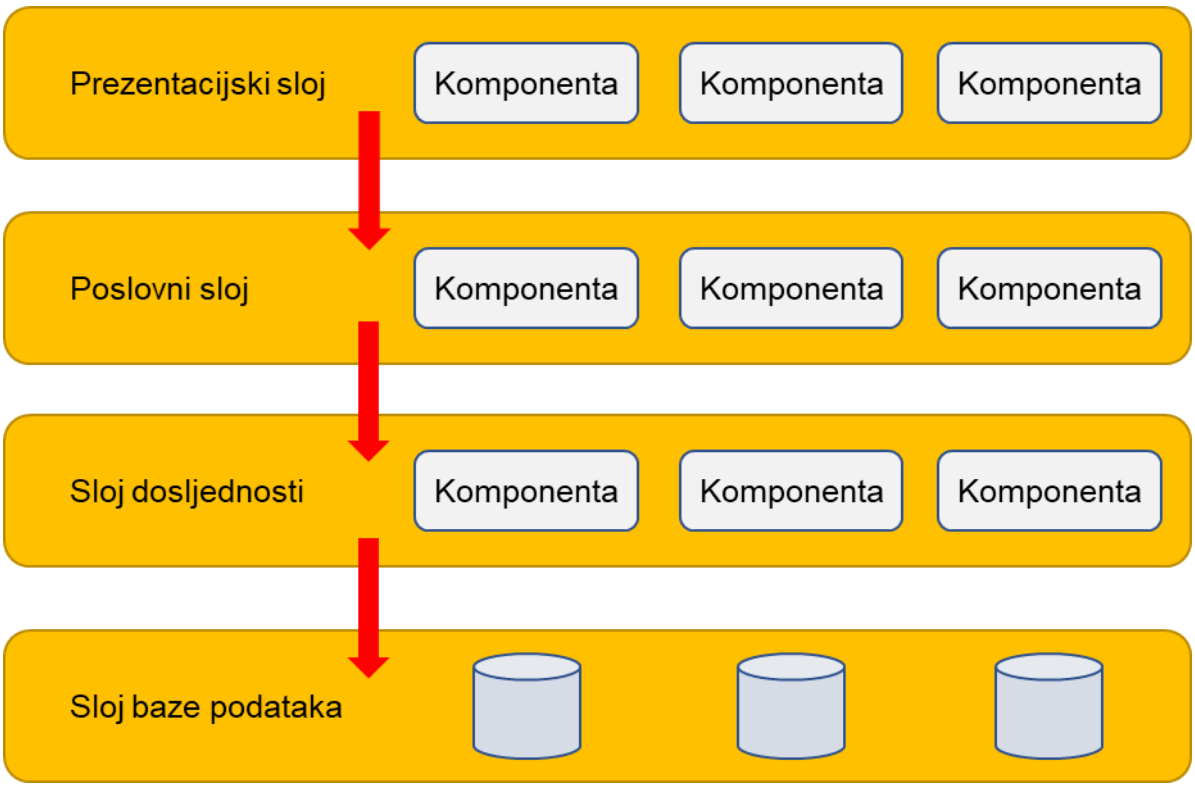
Najčešći arhitektonski uzorak jest slojevita arhitektura, inače poznat kao n-razina (eng. n-tier). Ovaj uzorak je danas standard za većinu aplikacija i stoga je nadaleko poznat većini arhitekata, dizajnera i developera. Slojeviti arhitektonski uzorak usko odgovara tradicionalnoj informacijskoj komunikaciji i organizacijskoj strukturi koje se nalaze u većini tvrtki, što ga čini prirodnim izborom za većinu poslovnih aplikacija.

Komponente unutar slojevitog arhitektonskog uzorka su organizirane u horizontalne slojeve, pri čemu svaki sloj unutar njih ima određenu ulogu u aplikaciji (npr. logika prezentacije ili poslovna logika). Iako slojeviti arhitektonski uzorak ne specificira broj i vrste slojeva koji moraju postojati u uzorku, većina ovog arhitekturnog uzorka sastoji se od četiri standardna sloja: prezentacijski, poslovni, dosljedni (eng. persistence layer) te baza podataka. U nekim slučajevima, poslovni sloj i sloj dosljednosti kombiniraju se u jedan poslovni sloj, par- posebno kada je logika dosljednosti (npr. SQL ili HSQL) ugrađena u sklopu komponentu poslovnog sloja. Dakle, manje aplikacije mogu imati samo tri sloja, a veće i složenije poslovne aplikacije mogu sadržavati pet ili više slojeva.

Svaki sloj ovog uzorka ima specifičnu ulogu i odgovornost unutar aplikacije. Na primjer, prezentacijski sloj je odgovoran za upravljanje korisničkim sučeljem i komunikacijskom logikom preglednika, dok je poslovni sloj odgovoran za izvršavanje specifičnih poslovnih pravila koje se odnose na dani zahtjev. Na primjer, prezentacijski sloj ne mora znati ili brinuti o tome kako doći do podataka o klijentima, samo treba prikazati te informacije na ekranu u određenom formatu. Slično, poslovni sloj ne treba brinuti o tome kako formatirati podatke o klijentu za prikaz na ekranu ili odakle podaci dolaze, samo treba dobiti podatke iz sloja dosljednosti, provesti poslovnu logiku koristeći dobivene podatke i proslijediti te informacije do prezentacijskog sloja.

Jedna od najmoćnijih značajki slojevitog arhitektonskog uzorka jest razdvajanje uloga između komponenti. Komponente unutar određenog sloja bave se samo logikom koja se odnosi na taj sloj. Za na primjer, komponente u sloju prezentacije bave se samo prezentacijskom logikom, dok se komponente koje se nalaze u poslovnom sloju bave samo poslovnom logikom. Ova vrsta klasifikacije komponenti olakšava ugrađivanje učinkovitih uloga i modela odgovornosti u arhitekturu. Definirana sučelja komponenti i njihov ograničeni opseg

olakšavaju razvoj, testiranje, upravljanje te održavanje aplikacija koje koriste ovaj arhitekturni uzorak.



Slika 1: Slojevita arhitektura (Prema: [3])

Kod ovog uzorka treba imati na umu ono što je poznato kao anti-uzorak ponora (eng. sinkhole). Ovaj anti-uzorak opisuje situaciju u kojoj zahtjevi teku kroz više slojeva arhitekture malo ili nimalo izvedene logike unutar svakog sloja. Na primjer, pretpostavimo da prezentacijski sloj odgovara na zahtjev korisnika za dohvaćanje podataka o korisniku. Prezentacijski sloj prosljeđuje zahtjev poslovnom sloju, koji jednostavno prosljeđuje zahtjev sloju dosljednosti, koji zatim šalje jednostavan SQL zahtjev sloju baze podataka za dohvaćanje podataka o korisniku. Podaci zatim prolaze cijelim putem natrag bez dodatnih procesiranja, agregiranja, izračunavanja ili transformacije podataka [3].

3.2. Arhitektura vođena događajima

Uzorak arhitekture vođen događajima popularan je distribuiran asinkroni arhitektonski uzorak koji se koristi za izgradnju visoko skalabilnih aplikacija. Također je vrlo prilagodljiv i može se koristiti za male aplikacije, kao i za velike i složene. Ovaj arhitekturni uzorak sastoji se od visoko odvojenih, jednonamjenskih komponenti koje asinkrono primaju i obrađuju događaje.

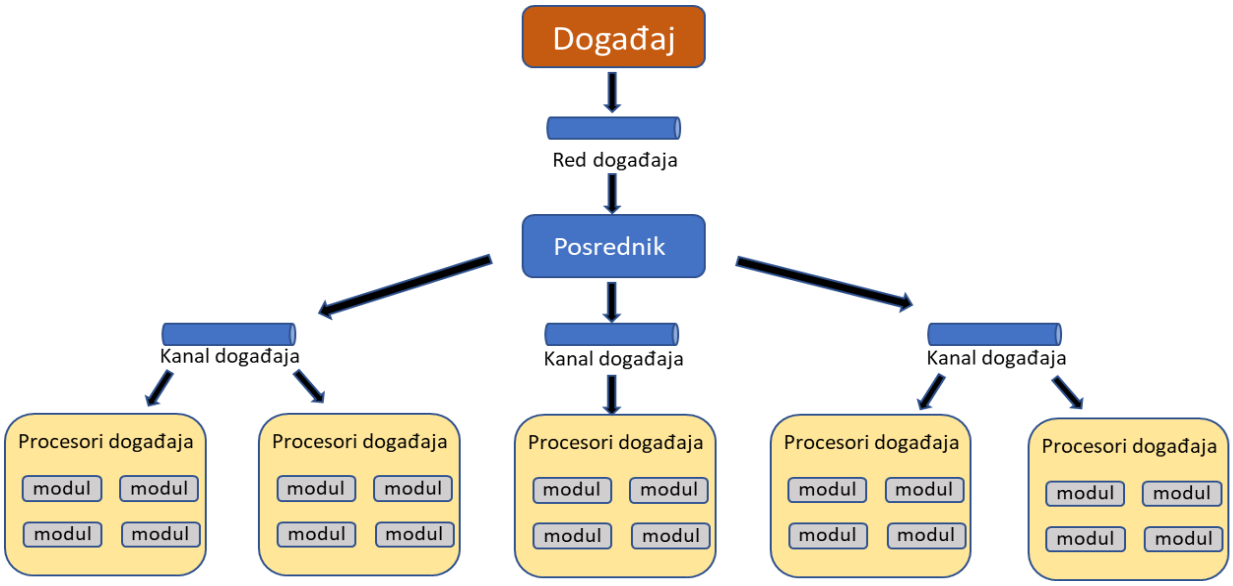
Uzorak arhitekture vođene događajima sastoji se od dvije glavne topologije: posrednik (eng. mediator) i broker. Topologija posrednika obično se koristi kada je potrebno upravljati s više koraka unutar jednog događaja preko središnjeg posrednika, dok se topologija brokera koristi kada je potrebno spojiti događaje bez upotrebe središnjeg posrednika. Budući da se arhitektonske karakteristike i implementacijske strategije uvelike razlikuju između ove dvije topologije, važno je razumjeti svaku od njih kako bi se prepoznala idealna za primjenu u određenoj situaciji.

Uzorak arhitekture vođen događajima relativno je složen uzorak za implementaciju, prvenstveno zbog svoje asinkrone distribuirane prirode. Prilikom implementacije ovog uzorka, potrebno je riješiti različite probleme distribuirane arhitekture, kao što su dostupnost udaljenih procesa, nedostatak odziva i logika ponovnog povezivanja posrednika u slučaju kvara posrednika ili brokera.

3.2.1. Posrednik

Postoje četiri glavne vrste komponenti arhitekture unutar topologije posrednika: redovi događaja (eng. event queues), posrednik događaja (eng. event mediator), kanali događaja (eng. event channels), i procesori događaja (eng. event processors). Tijek događaja započinje klijentovim slanjem događaja u red događaja, koji se koristi za prijenos događaja na posrednik događaja. Posrednik događaja prima početni događaj i šalje dodatne asinkrone događaje na kanale događaja kako bi se izvršio svaki korak procesa. Procesori događaja koji slušaju kanale događaja, primaju događaj od posrednika događaja i izvršavaju specifičnu poslovnu logiku za obradu događaja.

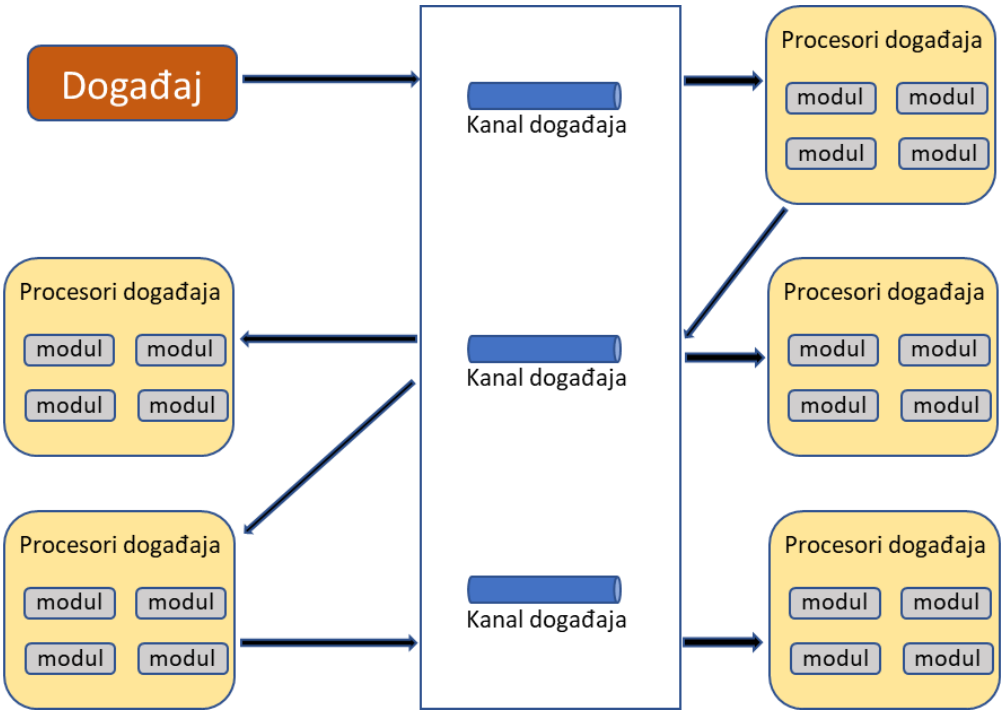
U arhitekturi vođenoj događajima uobičajeno je imati od desetak do nekoliko stotina redova događaja. Uzorak ne specificira implementaciju komponente reda događaja, nego to može biti red poruka, krajnja točka web servisa ili kombinacija. Postoje dvije vrste događaja unutar ovog uzorka: početni događaj i događaj obrade. Početni događaj je izvorni događaj koji je primio posrednik, dok su događaji obrade oni koji generira posrednik i primaju ih komponente za procesiranje događaja.



Slika 2: Topologija posrednika (Prema: [3])

3.2.2. Broker

Topologija brokera razlikuje se od topologije posrednika po tome što ne postoji središnji posrednik događaja; umjesto toga, tok poruka se distribuira kroz komponente procesiranja događaja na ulančan način putem jednostavnog posrednika poruka. Ova topologija je korisna kod korištenja jednostavnog tijeka obrade događaja i središnje upravljanje događajima je nepoželjno ili nepotrebno. Postoje dvije glavne vrste komponenti arhitekture unutar topologije brokera: komponenta brokera i komponenta procesiranja događaja. Komponenta brokera može biti centralizirana ili udružena i sadrži sve kanale događaja koji se koriste unutar toka događaja. Kanali događaja sadržani u komponenti brokera mogu biti redovi poruka, teme poruka ili kombinacija [3].



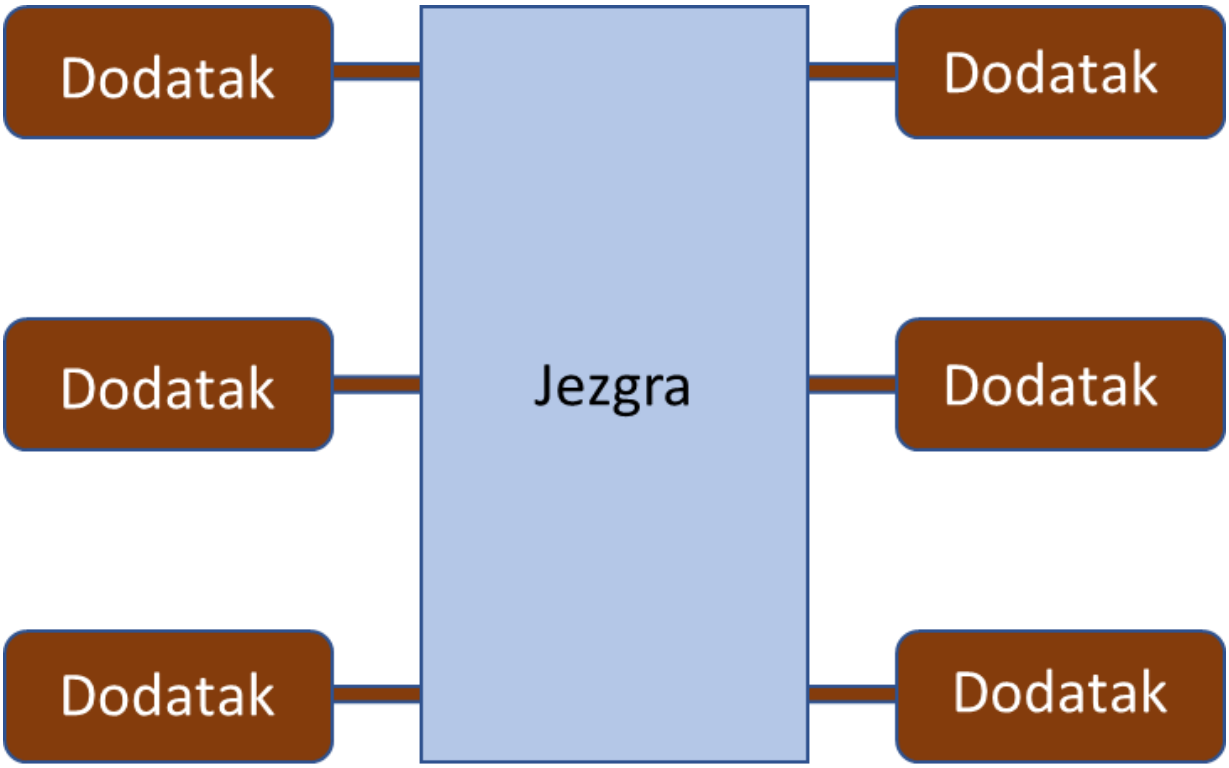
Slika 3: Topologija brokera (Prema: [3])

3.3. Mikro-jezgrena arhitektura

Uzorak arhitekture mikro-jezgre sastoji se od dvije vrste komponenti arhitekture: jezgrenog sustava i modula za dodatke. Aplikacijska logika je podijeljena između neovisnih plug-in modula i osnovnog sustava jezgre, pružajući proširivost, fleksibilnost i izolaciju značajki aplikacije i prilagođenu logiku obrade.

Ovaj uzorak tradicionalno sadrži samo minimalnu funkcionalnost potrebnu da bi sustav bio operativan. Mnogi operacijski sustavi implementiraju uzorak arhitekture mikrojezgre, a otuda i dolazi podrijetlo imena ovog uzorka. Iz perspektive poslovne aplikacije, temeljni sustav je definiran kao opća poslovna logika bez prilagođenog koda za posebne slučajeve, posebna pravila ili složenu obradu.

Dodatni moduli su samostalne, neovisne komponente koje sadrže specijaliziranu obradu, dodatne značajke i prilagođeni kod koji je namijenjen poboljšanju ili proširenju temeljnog sustava kako bi se proizvele dodatne poslovne sposobnosti. Općenito, plug-in moduli trebali bi biti neovisni o drugim plug-in modulima, ali je moguće dizajnirati dodatke koji zahtijevaju ovisnost na druge dodatke. U svakom slučaju, važno je svesti komunikaciju između dodataka na minimum kako bi se izbjegli problemi koje ovisnost o drugim modulima donosi [3].



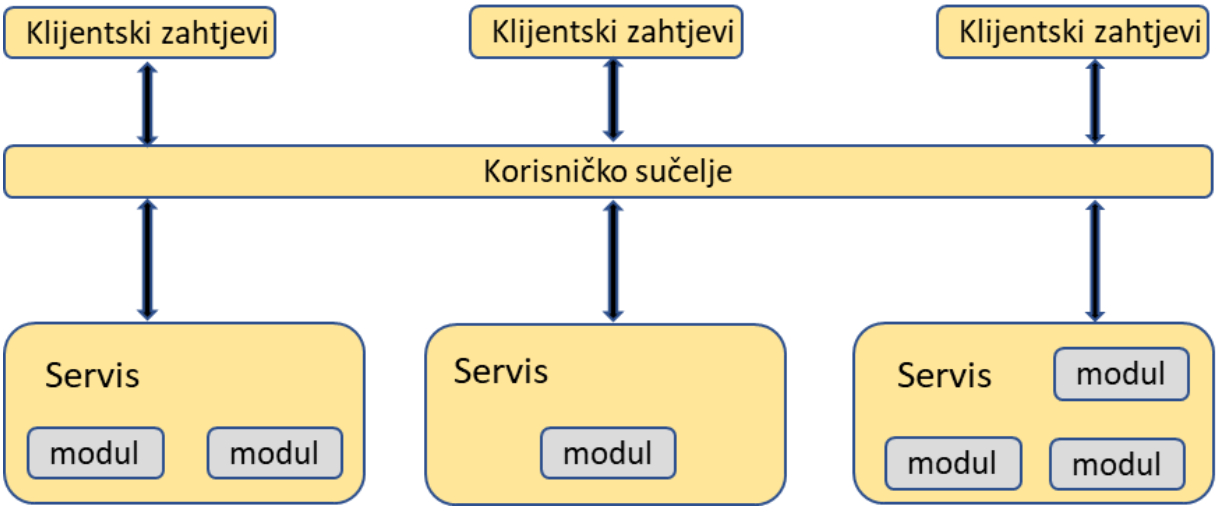
Slika 4: Mikro-jezgrena arhitektura (Prema: [3])

3.4. Mikroservisna arhitektura

Kao što naziv implicira, arhitektura mikroservisa je pristup izgradnji poslužiteljske aplikacije kao skupa malih servisa. To znači da je arhitektura mikroservisa uglavnom orijentirana na pozadinske procese neke aplikacije, iako se ovakav pristup također koristi za razvoj korisničke strane aplikacije. Svaki servis radi u svom vlastitom procesu i komunicira s drugim procesima koristeći protokole kao što su HTTP/HTTPS, WebSockets ili AMQP (Advanced Message Queuing Protocol). Svaki mikroservis implementira specifičnu domenu ili poslovnu sposobnost unutar određene granice konteksta, te se mora razvijati autonomno i samostalno implementirati. Također, svaki mikroservis trebao bi posjedovati svoj model podataka domene i logiku domene i može se temeljiti na različitim tehnologijama pohrane podataka (SQL, NoSQL) i različitim programskim jezicima.

Zašto arhitektura mikroservisa? Mikroservsi omogućuju bolju održivost u složenim, velikim i visoko skalabilnim sustavima omogućujući razvojnim inženjerima da kreiraju aplikacije temeljene na mnogim servisima koje se mogu neovisno implementirati od kojih svaka ima granularni i autonomni životni ciklus, te time pružaju dugoročnu agilnost projekta. Kao dodatna prednost, mikroservis se može samostalno skalirati. Umjesto da ima jednu monolitnu aplikaciju koja se mora skalirati kao jedinica, umjesto toga može se skalirati samo specifičan mikroservis.

Na taj način skalira se samo funkcionalno područje koje treba više procesorske snage ili propusnosti mreže kako bi podržalo potražnju, umjesto da se skaliraju druga područja aplikacije koja se ne moraju skalirati. Ovo donosi uštedu zbog manje potrebe za hardverom [4].



Slika 5: Mikroservisna arhitektura (Prema: [3])

3.5. Klijent – server arhitektura

U arhitektonskom uzorku klijent - server komponenta servera pruža usluge višestrukim komponentama klijenta. Klijentska komponenta zahtijeva usluge od komponente servera. Serveri su stalno aktivni, pritom slušajući klijentske zahtjeve. Zahtjevi se šalju izvan granica procesa i stroja, stoga se mora koristiti neki međuprocesni komunikacijski mehanizam. Klijenti i serveri mogu se nalaziti na različitim računalima, a time i u različitim procesima.

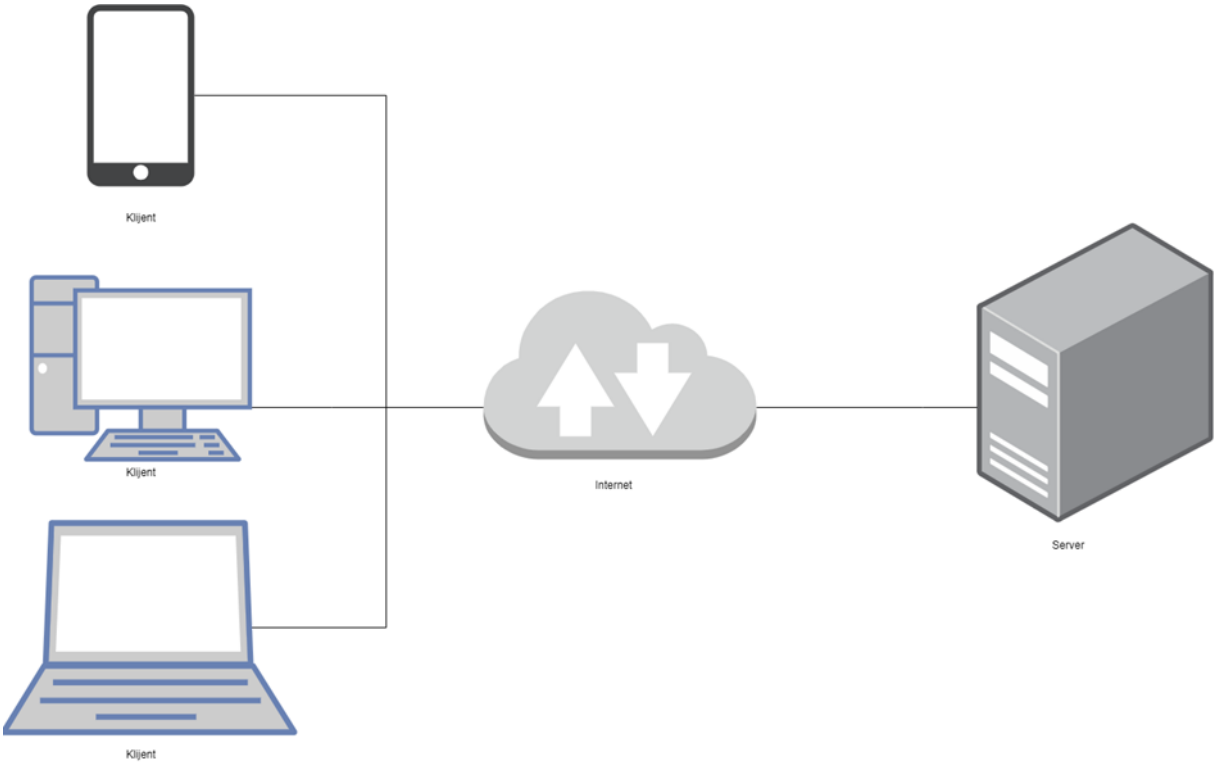
Klijenti i serveri često koriste sesije za očuvanje stanja podataka, a postoje dvije vrste sesija: sa stanjem (stateful) i bez stanja (stateless). Kod servera bez stanja, stanjem sesije upravlja klijent. Stanje klijenta šalje se sa svakim zahtjevom. U web aplikaciji, stanje sesije može biti pohranjeno kao parametri URL-a, u skrivenim poljima forme ili korištenjem kolačića. Ovo je obavezno za REST arhitektonski stil za web aplikacije. Kod servera s stanjem stanja, stanje sesije održava server i povezano je s identifikacijskim ključem klijenta.

Stanje u uzorku klijent - server utječe na transakcije, rukovanje greškama i skalabilnost. Transakcije bi trebale biti atomske, ostaviti konzistentno stanje, biti izolirane (na njih ne smiju utjecati drugi zahtjevi) i trajne, međutim u modernim distribuiranim skalabilnim sustavima ova svojstva je prilično teško pridobiti. Što se tiče upravljanja greškama, stanje koje održava klijent znači na primjer da će sve biti izgubljeno kada dođe do pogreške na klijentskoj strani. Stanje

koje održava klijent također predstavlja sigurnosne probleme jer se osjetljivi podaci moraju poslati serveru sa svakim zahtjevom. Problemi s skalabilnosti pojavljuju se kod upravljanja stanjem servera u memoriji, a postoje brojni klijenti koji istovremeno koriste server, te stoga velik broj stanja mora biti pohranjen u memoriji u isto vrijeme.

REST arhitektura je arhitektura klijent-server, gdje su klijenti odvojeni od servera jedinstvenim sučeljem, a komunikacija se izvodi bez stanja. Server može imati stanje, ali u tom slučaju svako stanje servera treba biti adresibilno (na primjer, URL-om). REST arhitektura je također slojeviti sustav: za klijenta je transparentno je li izravno povezan s serverom ili preko jednog ili više posrednika. Web aplikacije s komunikacijom bez stanja slijede pravila ovog uzorka.

Ovaj arhitekturni uzorak sa sobom donosi i određene nedostatke: zahtjevi se obično obrađuju u zasebnim dretvama na serveru, međuprocesna komunikacija uzrokuje dodatne troškove, zahtjevi i podaci o rezultatima često se moraju transformirati ili rasporediti jer imaju različitu reprezentaciju u klijentu i serveru, distribuirani sustavi s mnogo poslužitelja s istom funkcijom moraju biti transparentni za klijente: ne smije postojati potreba da klijenti razlikuju poslužitelje [1].



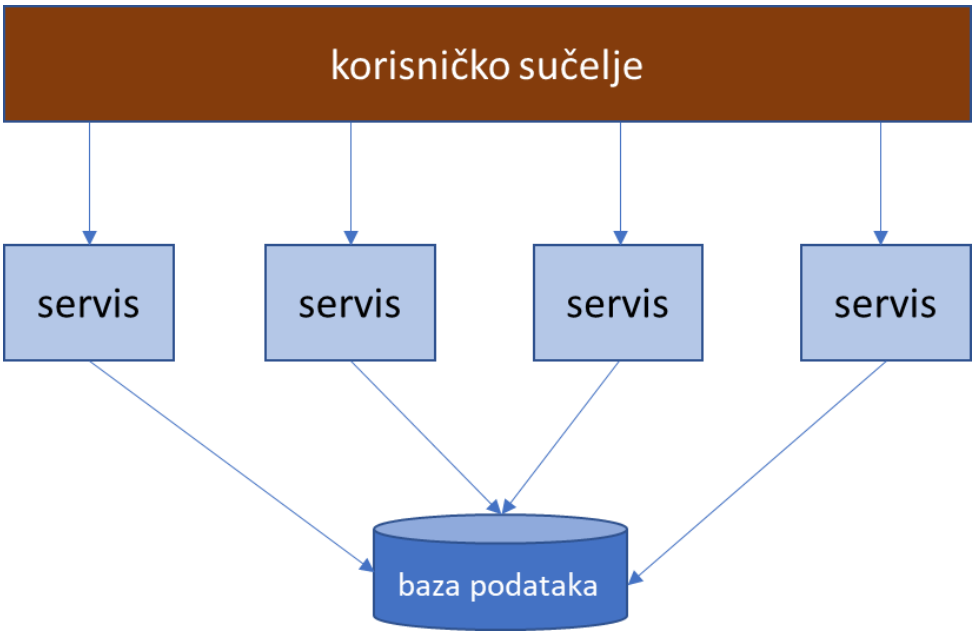
Slika 6: Klijent - server arhitektura [autorski rad]

3.6. Arhitektura temeljena na servisima

Arhitektura temeljena na servisima hibrid je mikroservisnog uzorka arhitekture i smatra se jednim od najpragmatičnijih uzoraka arhitekture, ponajviše zbog svoje arhitektonske fleksibilnosti. Iako je arhitektura temeljena na servisima distribuirana arhitektura, ona nema istu razinu složenosti kao druge distribuirane arhitekture, poput mikroservisne arhitekture ili arhitekture vođene događajima, što je čini vrlo popularnim izborom za mnoge poslovne aplikacije.

Osnovna topologija arhitekture temeljene na servisima slijedi distribuiranu slojevitu strukturu koja se sastoji od zasebno postavljenog (eng. deployed) korisničkog sučelja, zasebno postavljenih servisa i monolitne baze podataka. U većini slučajeva postoji samo jedna instanca svake servisne domene unutar arhitekture koja se temelji na servisima. Međutim, na temelju skalabilnosti, tolerancije grešaka i potreba za propusnošću, sigurno može postojati više instanci servisne domene. Višestruke instance servisa obično zahtijevaju neku vrstu mogućnosti balansiranja opterećenja između korisničkog sučelja i servisa kako bi se korisničko sučelje moglo usmjeriti na dostupnu instancu servisa. Servisima se pristupa udaljeno s korisničkog sučelja pomoću protokola za udaljeni pristup. Dok se REST obično koristi za pristup servisa s korisničkog sučelja, također se može koristiti slanje poruka, udaljeni poziv procedure (eng. Remote Procedure Call) ili čak SOAP.

Jedan važan aspekt arhitekture temeljene na servisima jest da obično koristi centraliziranu bazu podataka. To omogućuje servisima da iskoriste SQL upite i veze prema bazi podataka na isti način na koji bi to učinila tradicionalna monolitna slojevita arhitektura. Zbog malog broja servisa (4 do 12), veze s bazom podataka obično nisu problem u arhitekturi temeljenoj na servisima [2].



Slika 7: Arhitektura temeljena na servisima (Prema: [4])

4. Web servisi

Na World Wide Webu, web servis je standardizirana metoda za širenje poruka između klijentskih i serverskih aplikacija. Web servis je računalni program koji je namijenjen za obavljanje određenog skupa funkcija. Web servisi u računalstvu u oblaku mogu se pronaći i pozvati putem mreže. Web servis isporučuje funkcionalnost klijentu koji je pozvao taj web servis. Web servis ujedno je i skup otvorenih protokola i standarda koji omogućuju razmjenu podataka između različitih aplikacija ili sustava. Mogu ih koristiti računalni programi napisani na raznim programskim jezicima i koji rade na raznim platformama za razmjenu podataka putem računalnih mreža kao što je Internet na sličan način kao međuprocesna komunikacija na jednom računalu.

Svaki softver, aplikacija ili tehnologija u oblaku koja koristi standardizirane web protokole (HTTP ili HTTPS) za povezivanje, međuoperaciju i razmjenu podatkovnih poruka interneta smatra se web-servisom. Web servisi imaju prednost što dopuštaju programima razvijenim na različitim jezicima da se međusobno povežu razmjenom podataka putem web servisa između klijenata i poslužitelja. Klijent poziva web servis podnošenjem XML zahtjeva, na koji servis odgovara na istom jeziku na koji je zahtjev podnešen [5].

Slika 8: Web servisi [6]

4.1. API

Najjednostavnije rečeno, API je akronim za Application Programming Interface, što je softverski posrednik koji omogućuje dvjema aplikacijama da razgovaraju jedna s drugom. Postoji više vrsta API-ja, a najpopularnije su SOAP i REST. Tradicionalno, API-ji su uvijek bili dio razvoja softvera. Operativni sustavi kao što su Microsoft Windows i Mac OS ili mobilne platforme kao što su iOS i Android nude API-je koji programerima omogućuju izradu softvera. Razvojni programeri ovise o postojanju ovih API-ja, očekuju da funkcionira kako je navedeno te dokumentirano i nadaju se da neće prekinuti unaprijed definirane protokole za korištenje bez dostatne obavijesti. Kako je softver prešao s fokusa na stolna računala i web na mobilno računalstvo, došlo je do povećane potražnje za web API-jima. Ovi moderni web API-ji nisu izgrađeni samo za integraciju sustava unutar organizacije. Umjesto toga, omogućuju tvrtkama da dijele poslovne sposobnosti i podatke, grade zajednicu i potiču inovacije. To je dovelo do uspona API ekonomije.

Osim modernih preglednika, došlo je do eksplozivnog rasta mobilnih uređaja kao što su telefoni i tableti. Ovi uređaji imaju pristup internetu s većine lokacija i nude GPS lokaciju i

distribuciju u trgovini aplikacijama. Aplikacije više ne moraju biti web stranice u pregledniku. Umjesto toga, mogu koristiti API-je za pristup podacima i poslovnoj logici kako bi obavile svoju funkciju. Konačno, Internet stvari (IoT) pomiče svijet uređaja, koji su prije zahtijevali ljudsku intervenciju, u autonomne zamjene koje kombiniraju fizički svijet sa svijetom softvera. Kao rezultat toga, API-ji omogućuju IoT uređajima da emitiraju svoje telemetrijske podatke i primaju naredbe od drugih sustava. Povijesno gledano, poduzeća su usvajala tehnologije kao što su SOAP ili XML-RPC za internu integraciju aplikacija i između partnerskih organizacija. Ove tehnologije često su zahtijevale dodatne standarde i specifikacije na vrhu transportnih protokola kao što je HTTP. Međutim, ove su tehnologije namijenjene integraciji sustava gdje su čvrsto definirane specifikacije najvažnije. Moderni web API-ji napuštaju potrebu za ovim složenim standardima, umjesto toga biraju jednostavnije rješenje. Oni potiču korištenje HTTP-a, protokola koji pokreće web, kao temelja za API-je. HTTP specifikacija je dizajnirana da podržava robustan skup glagola zahtjeva (tj. ono što želite učiniti) i kodova odgovora (tj. rezultat zahtjeva). Filozofija web API-ja je izbjegavanje dodatnih standarda i specifikacija, umjesto toga odabir korištenja HTTP standarda za definiranje načina rada web API-ja. Odabirom HTTP-a kao jedinog standarda, svaka aplikacija ili uređaj može koristiti web API. Više nisu potrebna skupa softverska rješenja i komplicirani standardi, a to donosi jednostavnu integraciju za bilo koji uređaj: mobilni telefoni, preglednici ili čak automobili s pristupom mobilnoj mreži mogu konzumirati web API-je. Odabirom HTTP-a za API-je, tvrtke mogu izbjeći izdvajanje velikih proračuna vremena i novca za učenje, izgradnju i održavanje složenih tehnoloških alata. Umjesto toga, ugrađene i otvorene programske biblioteke mogu se koristiti za stvaranje i korištenje raznih web API-ja [7].

4.2. SOAP

Simple Object Access Protocol (SOAP) je protokol za razmjenu poruka koji aplikacijama omogućuje komunikaciju koristeći HTTP i XML. Predstavlja paradigmu jednosmjerne razmjene poruka bez stanja između čvorova. Kombinirajući jednosmjerne razmjene sa značajkama koje pruža temeljni transportni protokol i/ili informacije specifične za aplikaciju, SOAP se može koristiti za stvaranje složenijih interakcija kao što su zahtjev/odgovor ili zahtjev/višestruki odgovor. Proces pozivanja web servisa je vrlo važan, stoga je uspostavljen SOAP protokol za razmjenu poruka između davatelja usluga i potrošača. To je strukturirani format XML poruke za razmjenu podataka u distribuiranom okruženju, a koristi temeljni transportni protokol (HTTP, SMTP itd.).

Postoje tri glavne vrste SOAP čvorova: SOAP pošiljatelj (generira i prenosi SOAP poruku), SOAP primatelj (prima i obrađuje SOAP poruku i također može generirati SOAP

odgovor, poruku ili grešku kao rezultat) te SOAP posrednik, koji je i SOAP primatelj i SOAP pošiljatelj. Prima i obrađuje SOAP blokove zaglavlja usmjerene na njega i ponovno šalje SOAP poruku prema SOAP primatelju. SOAP poruka ima strukturu koju karakteriziraju dva SOAP-specifična podelementa unutar cjelokupne SOAP omotnice (eng. Envelope) odnosno SOAP zaglavlje (eng. header) i SOAP tijelo (eng. body). SOAP je neovisan protokol jer nije važno s kojeg se OS-a ili platforme koristi usluga: odgovara na isti način na bilo kojoj platformi ili OS-u. Sve je to moguće zahvaljujući XML i HTTP protokolima. Postoje dvije vrste zahtjeva za razmjenu SOAP poruka: Remote Procedure Call (RPC) i Document request [8].

4.3. REST

REST (Representational state transfer) arhitektura temelji se na stilu arhitekture klijent - server. Stoga se zahtjevi i odgovori grade na temelju procesa prijenosa resursa. Svi resursi su identificirani jedinstvenim Uniform Resource Identifier (URI), koji obično predstavlja dokument koji bilježi stanje resursa. Općenito, arhitektura REST stila je mnogo jednostavnija u usporedbi sa SOAP-om. Ne zahtijeva da formati poput zaglavlja budu uključeni u poruku, kao što je to potrebno u SOAP arhitekturi. S druge strane analizira JSON (JavaScript Object Notation), jezik dizajniran da omogući razmjenu podataka i olakšava prevođenje i korištenje od strane računala.

REST API-ji su bez stanja, što znači da se pozivi mogu obavljati neovisno jedan o drugom, a svaki poziv sadrži sve podatke potrebne za uspješno dovršenje. REST API se ne bi trebao oslanjati na podatke koji se pohranjuju na poslužitelju ili sesijama da bi odredio što učiniti s pozivom, već se oslanjati isključivo na podatke koji su dati u samom pozivu. Budući da API bez stanja može povećati opterećenje zbog potrebe za rukovanjem velikim brojem dolaznih i odlaznih poziva, REST API bi trebao biti dizajniran tako da potakne pohranu podataka koji se mogu predmemorati. To znači da kada su podaci u predmemoriji, odgovor bi trebao naznačiti da se podaci mogu pohraniti do određenog vremena (istječe u), ili u slučajevima kada podaci trebaju biti u stvarnom vremenu, da odgovor ne bi trebao biti predmemoriran od klijentske strane [8].

5. Uzorci dizajna

„Svaki uzorak opisuje problem koji se uvijek iznova pojavljuje u našem okruženju, a zatim opisuje srž rješenja tog problema, na takav način da ovo rješenje možete koristiti milijun puta, a da to nikada ne učinite na isti način dva puta.“ – riječi su Cristophera Alexandera, prvog čovjeka koji je predložio ideju korištenja uzoraka za arhitekturu zgrada i gradova. Njegove ideje i doprinosi drugih ukorijenjeni su u zajednici objektno orijentiranog softvera. Ukratko, koncept uzorka dizajna u softveru pruža pomoć programerima da iskoriste stručnost drugih vještih arhitekata. Uzorak dizajna sastoji se od četiri glavna elementa:

1. Naziv uzorka – oznaka koja se koristi za opisivanje dizajnerskog problema, njegovih rješenja i posljedica koristeći samo jednu riječ ili dvije. Korištenje vokabularom uzoraka omogućuje programerima međusoban razgovor o uzorcima dizajna i jasnije definiranje i opis dokumentacije. Olakšava razmišljanje o dizajnu i komuniciranje o njima i njihovim prednostima i manama s drugim ljudima.
2. Problem – opisuje kada primijeniti uzorak. Objašnjava problem koji se pokušava otkloniti i njegov kontekst. Može opisivati specifične probleme dizajna (npr. kako predstaviti algoritme kao objekte). Može opisivati klase ili strukture objekata koje su simptomatične za nefleksibilan dizajn. Ponekad problem sadrži popis uvjeta koji moraju biti ispunjeni prije nego što ima smisla primijeniti uzorak.
3. Rješenje – opisuje elemente koji čine dizajn, njihove odnose, odgovornosti i suradnju. Rješenje ne opisuje konkretan dizajn ili implementaciju jer je uzorak poput predloška koji se može primijeniti u mnogo različitih situacija. Umjesto toga, uzorak pruža apstraktni opis problema dizajna i kako ga opći raspored elemenata (klase i objekti) rješava.
4. Posljedice – rezultati i ustupci primjene uzorka. Iako se o posljedicama često ne govori kod opisa odluke o dizajnu, one su kritične za procjenu alternativa dizajna i za razumijevanje troškova i koristi od primjene uzorka [9].

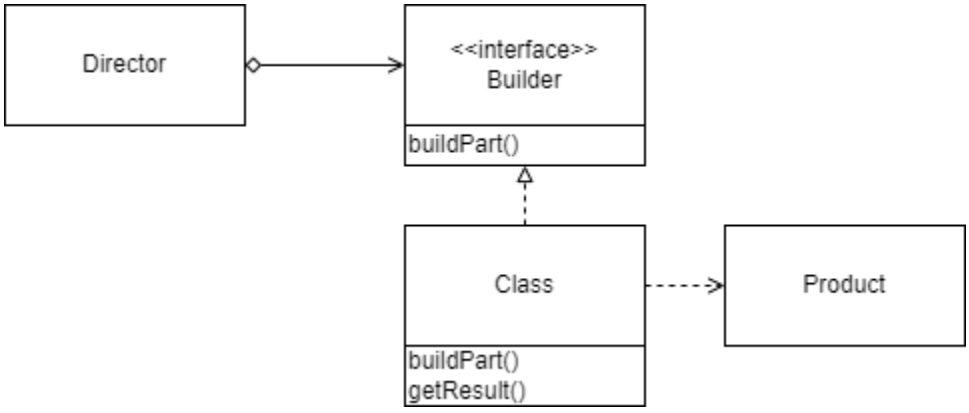
5.1. Uzorci stvaranja

Uzorci stvaranja izdvajaju proces instanciranja. Oni čine sustav neovisnim o tome kako su njegovi objekti kreirani, sastavljeni i predstavljeni. Uzorak stvaranja klasa koristi nasljeđivanje za mijenjanje klase koja je instancirana, dok će uzorak stvaranja objekta delegirati instanciranje drugom objektu. Kreacijski uzorci postaju važni kako sustavi evoluiraju tako da više ovise o sastavu objekta nego o nasljeđivanju klasa. Zbog toga se naglasak pomiče s kodiranja fiksnog skupa ponašanja prema definiranju manjeg skupa temeljnih ponašanja koja

se mogu sastaviti u bilo koji broj složenijih. Stoga stvaranje objekata s određenim ponašanjem zahtijeva više od jednostavnog instanciranja klase [9].

5.1.1. Builder

Namjera Builder uzorka jest odvojiti konstrukciju složenog objekta od njegovog prikaza, tako da isti proces izgradnje može stvoriti različite prikaze. Ova vrsta odvajanja smanjuje veličinu objekta. Ispada da je dizajn više modularan sa svakom implementacijom sadržanom u drugom Builder objektu. Dodavanje nove implementacije (tj. dodavanje novog Buildera) postaje lakše. Proces izgradnje objekta postaje neovisan o komponentama koje čine objekt. To omogućuje veću kontrolu nad procesom izgradnje objekta. U smislu implementacije, svaki od različitih koraka u procesu izgradnje može se deklarirati kao metode zajedničkog sučelja koje će implementirati različiti konkretni Builderi. Klijentski objekt može stvoriti instancu konkretnog Buildera i pozvati skup metoda potrebnih za konstruiranje različitih dijelova konačnog objekta. Ovaj pristup zahtijeva da svaki objekt klijenta bude svjestan logike izgradnje. Kad god se logika izgradnje podvrgne promjeni, svi klijentski objekti moraju biti modificirani u skladu s tim. Kako bi se ovaj problem riješio, ovaj uzorak uvodi drugu razinu odvajanja. Umjesto da objekti klijenta izravno pozivaju različite metode Buildera, ovaj uzorak predlaže korištenje namjenskog objekta koji se naziva direktor (eng. Director), koji je odgovoran za pozivanje različitih metoda Buildera potrebnih za konstrukciju konačnog objekta. Različiti klijentski objekti mogu koristiti direktorski objekt za stvaranje potrebnog objekta. Nakon što je objekt izgrađen, klijentski objekt može izravno zatražiti od Buildera potpuno izgrađeni objekt [9].



Slika 9: Builder (Prema: [9])

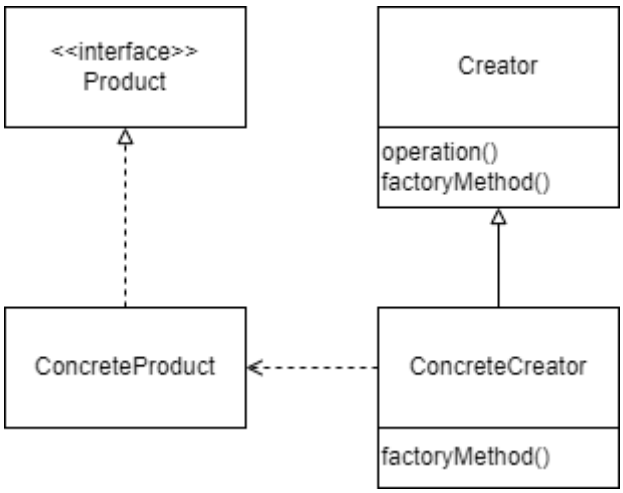
Kada koristiti Builder?

- Algoritam za kreiranje složenog objekta trebao bi biti neovisan o dijelovima koji čine objekt i načinima na koji su sastavljeni.

- Proces kreiranja mora omogućiti različite prikaze za objekt koji se gradi [10].

5.1.2. Factory Method

Factory Method uzorak daje način za enkapsulaciju instanciranja konkretnih tipova. Enkapsulira funkcionalnost potrebnu za odabir i instanciranje odgovarajuće klase unutar određene metode koja se naziva tvornička metoda. Factory Method odabire odgovarajuću klasu iz hijerarhije klasa na temelju konteksta aplikacije i drugih čimbenika utjecaja. Zatim instancira odabranu klasu i vraća je kao instancu tipa nadređene klase. Prednost ovog pristupa je da objekti aplikacije mogu koristiti Factory Method za pristup odgovarajućoj instanci klase. Ovo eliminira potrebu da se aplikacijski objekt bavi različitim kriterijima odabira klase.



Slika 10: Factory Method (Prema: [9])

Kada koristiti Factory Method?

- Klasa ne može predvidjeti klasu objekata koju mora stvoriti.
- Klasa želi da njezine potklase specificiraju objekte koje stvara.
- Klase delegiraju odgovornost jednoj od nekoliko pomoćnih potklasa, a želi se lokalizirati znanje o tome koja je pomoćna potklasa delegat [10].

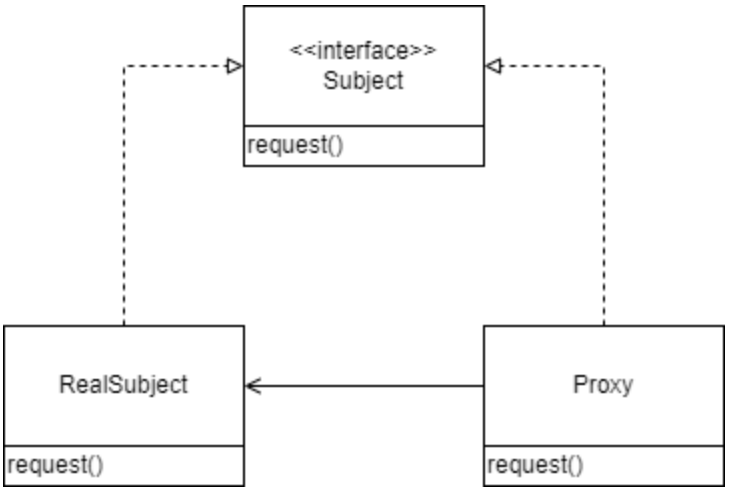
5.2. Strukturni uzorci

Strukturni uzorci bave se načinom na koji su klase i objekti sastavljeni tako da zajedno čine veće strukture. Strukturni uzorci klasa koriste nasljeđivanje za sastavljanje sučelja ili implementacija. Kao jednostavan primjer, razmotrite kako višestruko nasljeđivanje miješa dvije ili više klasa u jednu. Rezultat je klasa koja kombinira svojstva svojih roditeljskih klasa. Ovaj uzorak je posebno koristan za stvaranje zajedničkih neovisno razvijenih biblioteka klasa.

Umjesto sastavljanja sučelja ili implementacija, uzorci strukturnih objekata opisuju načine sastavljanja objekata za realizaciju nove funkcionalnosti. Dodatna fleksibilnost kompozicije objekta dolazi od mogućnosti mijenjanja kompozicije u vrijeme izvođenja, što je nemoguće sa statičnom kompozicijom klase [9].

5.2.1. Proxy

Proxy uzorak podržava objekte koji kontroliraju stvaranje i pristup drugim objektima. Proxy je često mali (javni) objekt koji zamjenjuje složeniji (privatni) objekt koji se aktivira nakon što su određene okolnosti jasne. Proxy uzorak koristi se za stvaranje reprezentativnog objekta koji kontrolira pristup drugom objektu, koji može biti udaljen, skup za izradu ili ga je potrebno osigurati. Jedan od razloga za kontrolu pristupa objektu je odgađanje punog troška njegovog stvaranja i inicijalizacije dok ga stvarno ne budemo trebali koristiti. Proxy može biti vrlo koristan u kontroli pristupa izvornom objektu, posebno kada objekti trebaju imati različita prava pristupa. U Proxy uzorku, klijent ne razgovara izravno s izvornim objektom, on delegira svoje pozive Proxy objektu koji poziva metode izvornog objekta. Važna stvar je da klijent ne zna za Proxy, Proxy djeluje kao izvorni objekt za klijenta [9].



Slika 11: Proxy (Prema: [9])

Kada koristiti Proxy?

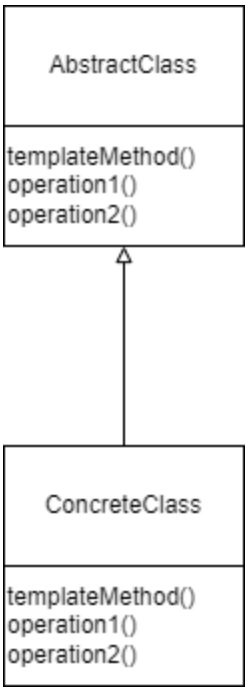
- Udaljeni Proxy pruža lokalnog predstavnika za objekt u drugom adresnom prostoru.
- Virtualni Proxy stvara skupe objekte na zahtjev.
- Zaštitni Proxy kontrolira pristup izvornom objektu. Proxyji zaštite korisni su kada objekti trebaju imati različita prava pristupa [10].

5.3. Uzorci ponašanja

Uzorci ponašanja odnose se na algoritme i dodjelu odgovornosti između objekata. Ovi uzorci ne opisuju samo uzorke objekata ili klasa, već i uzorke komunikacije među njima. Karakteriziraju složeni kontrolni tijek koji je teško pratiti tijekom izvođenja, te odmiču fokus s kontrolnog tijeka kako bi se programer mogao koncentrirati samo na načine na koje su objekti međusobno povezani [10].

5.3.1. Template Method

Template Method uzorak definira kostur algoritma u operaciji, ostavljajući neke korake algoritma potklasama. Template Method omogućuje potklasama da redefiniraju određene korake algoritma bez promjene strukture algoritma. Template Method uzorak može se koristiti u situacijama kada postoji algoritam čiji se neki koraci mogu implementirati na više različitih načina. U takvim scenarijima, Template Method predlaže zadržavanje kostura algoritma u zasebnoj metodi koja se naziva TemplateMethod unutar klase, koja se može nazivati Template Class, izostavljajući specifične implementacije varijantnih dijelova (koraci koji se mogu implementirati na više različitih načina) algoritma u različite potklase ove klase. Klasa predloška ne mora nužno prepustiti implementaciju potklasama u cijelosti. Umjesto toga, kao dio davanja nacrta algoritma, klasa predloška također može pružiti određenu količinu implementacije koja se može smatrati nepromjenjivom u različitim implementacijama. Može čak pružiti zadanu implementaciju za varijante dijelova, ako je prikladno. Samo će se specifični detalji implementirati unutar različitih potklasa. Ova vrsta implementacije eliminira potrebu za dvostrukim kodom, što znači minimalnu količinu koda koju treba napisati [9].



Slika 12: Template Method (Prema: [9])

Kada koristiti Template Method?

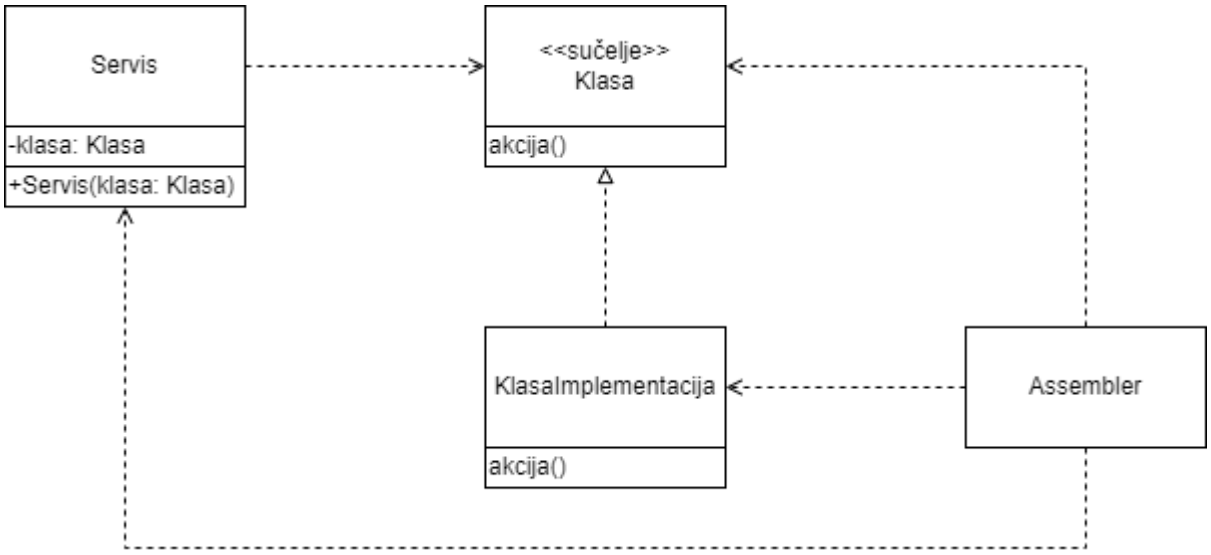
- Kada se želi jednom implementirati nepromjenjive dijelove algoritma i prepustiti potklasama da implementiraju ponašanje koje može varirati.
- Kada se uobičajeno ponašanje među potklasama treba faktorizirati i lokalizirati u zajedničkoj klasi kako bi se izbjeglo dupliciranje koda.
- Za kontrolu ekstenzija potklasa [10].

5.4. Ostali uzorci

5.4.1. Injektiranje ovisnosti

Načelo inverzije ovisnosti (eng. Dependency Inversion) pomaže odvojiti kod osiguravajući da kod ovisi o apstrakcijama, a ne o konkretnim implementacijama. Ovo načelo vrlo je važno za razumijevanje uzoraka dizajna. Injektiranje ovisnosti (eng. Dependency Injection) implementacija je ovog principa. Nazivi inverzija ovisnosti i injektiranje ovisnosti često se koriste naizmjenično, ali oba se odnose na isti proces odvajanja koda. Upotrebom načela inverzije ovisnosti moguće je osigurati da moduli visoke razine ovise o apstrakcijama, a ne o konkretnim implementacijama modula niže razine. Injektiranje ovisnosti jest opskrbljivanje svih klasa koje servis zahtijeva, umjesto prepuštanja odgovornosti servisu za dobivanje zavisnih klasa. Injektiranje ovisnosti obično dolazi u tri oblika:

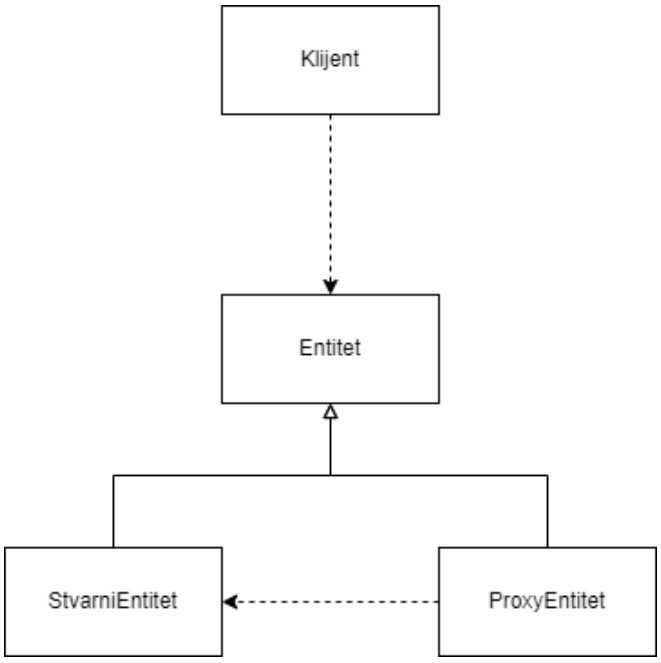
- Konstruktorsko injektiranje (eng. Constructor injection)
- Injektiranje preko postavljača (eng. Setter injection)
- Injektiranje u metode (eng. Method injection)[11]



Slika 13: Injektiranje ovisnosti (Prema: [11])

5.4.2. Lijeno učitavanje

Lijeno učitavanje (eng. Lazy Loading) je uzorak dizajna koji odgađa učitavanje resursa dok on ne bude stvarno potreban. Martin Fowler definirao ga je kao "objekt koji ne sadrži sve podatke koji su vam potrebni, ali zna kako ih dobiti" u Patterns of Enterprise Application Architecture knjizi. Kao primjer mogu se postaviti kupac i narudžbe, prilikom dohvaćanja kupca iz baze podataka možda nije potrebno dohvatiti njegovu cijelu povijest narudžbi, nego samo dio. Ogoda izvršenja dohvaćanja kupčevih narudžbi može povećati brzinu dohvaćanja podataka o kupcu i smanjiti opterećenje poslužitelja baze podataka. Uzorak lijenog učitavanja za izvršenje svojih funkcionalnosti koristi i Proxy uzorak [11].

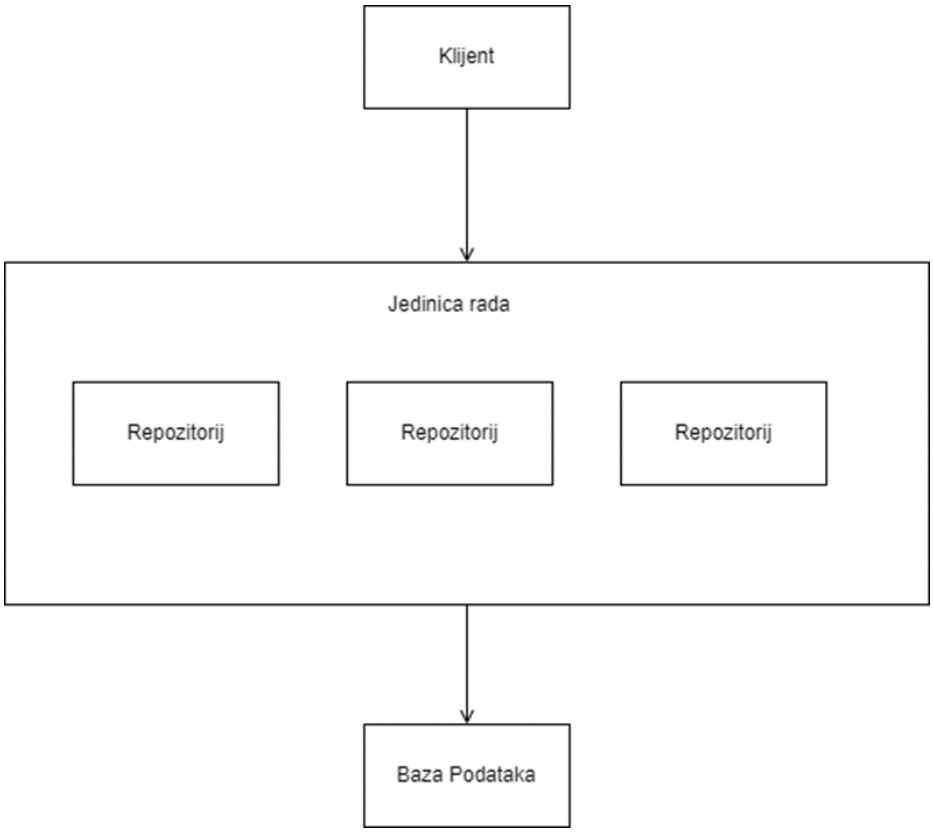


Slika 14: Lijeno učitavanje (Prema: [11])

Na području weba, Lijeno učitavanje uzorak je dizajna za identificiranje resursa kao neblokirajućih (nekritičnih) i učitavanja samo kada su potrebni. To je način da se skrati duljina kritičnog puta prikazivanja, što se prevodi u smanjeno vrijeme učitavanja stranice. Lijeno učitavanje može se dogoditi u različitim trenucima u aplikaciji, ali obično se događa u nekim korisničkim interakcijama kao što su pomicanje i navigacija. JavaScript, CSS i HTML mogu se podijeliti u manje dijelove, a to omogućuje slanje minimalnog koda potrebnog za pružanje tražene funkcionalnosti, poboljšavajući vrijeme učitavanja stranice [12].

5.4.3. Jedinica rada

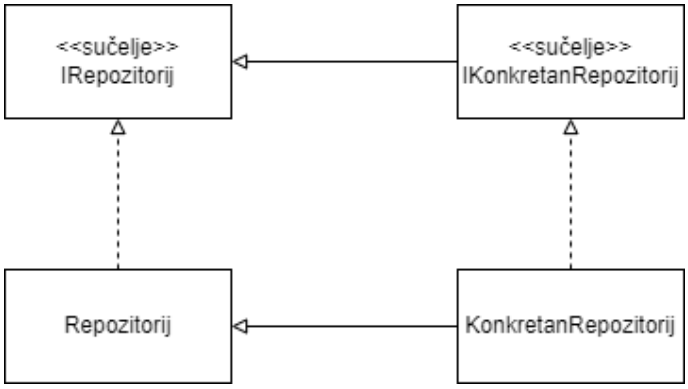
Uzorak jedinice rada (eng. Unit of Work) dizajniran je za održavanje popisa objekata koji su promijenjeni transakcijom, bilo to dodavanjem, uklanjanjem ili ažuriranjem. Radna jedinica zatim koordinira validnost unesenih promjena. Prednost korištenja jedinice rada jest osiguranje integriteta podataka: ako se bilo gdje tijekom transakcije pojavi neki problem, tada se sve promjene vraćaju kako bi se osiguralo da podaci ostanu u važećem stanju [11].



Slika 15: Jedinica rada [autorski rad]

5.4.4. Repozitorij

Repozitorij (eng. Repository) se ponaša kao zbirka u memoriji, potpuno izolirajući poslovne subjekte od temeljne podatkovne infrastrukture. Kada se koristi u projektima koji podržavaju metodologiju dizajna vođenog domenom (eng. Domain Driven Design), repozitorij obično postoji za svaki agregatni korijen (eng. Aggregate root) identificiran unutar danog modela domene [11].



Slika 16: Repozitorij [autorski rad]

5.5. Negativne strane uzoraka dizajna

5.5.1. Negativni problemi dizajna

Korištenje uzoraka dizajna može uzrokovati pojavu negativnih problema s dizajnom. Dobro je poznato da dobar dizajn softvera ima "visoku koheziju i nisku povezanost". Postoji empirijska studija koja pokušava istražiti povećava li uporaba uzoraka dizajna modularnost klase ili ne. Eksperiment je napravljen analizom 5 otvorenih (open-source) Java projekata te njihovo korištenje uzoraka dizajna. Dobiveni rezultati pokazali su da su klase koji koriste uzorke dizajna visoko povezane i manje kohezivne u usporedbi s klasama koje nisu koristile uzorke dizajna [13].

5.5.2. Pogrešna uporaba uzoraka dizajna

Postoji tendencija proučavanja uzoraka dizajna bez prepoznavanja situacija u kojima ih treba primijeniti. Ako je uzorak dizajna primijenjen kao čarobno rješenje na mjestu koje ne odgovara korištenju uzoraka dizajna, tada se stvara više problema nego što ih se rješava korištenjem uzorka dizajna. Kada su uzorci pogrešno primijenjeni, negativno utječu na prenosivost, proširivost i fleksibilnost projekta [14].

5.5.3. Neučinkovito rješavanje problema

Dobro je poznato da je svrha upotrebe uzoraka dizajna usvajanje standardizirane najbolje prakse za zajednički problem. Međutim, uobičajeni problemi nisu uvijek isti, ali imaju barem jednu zajedničku stvar. Prema tome, primjena uzorka dizajna ne mora uvijek biti rješenje, može dovesti do povećanja složenosti koda ili dupliciranja koda u usporedbi s rješavanjem problema na apstraktan način koji taj uzorak ne definira [14].

6. Tehnologije za realizaciju praktičnog dijela

6.1. Angular

Angular je razvojna platforma izgrađena na TypeScriptu. Kao platforma, Angular uključuje:

- Okvir temeljen na komponentama za izgradnju skalabilnih web aplikacija.
- Zbirku dobro integriranih biblioteka koje pokrivaju širok raspon značajki.
- Paket razvojnih alata koji pomažu u razvoju, izgradnji, testiranju i ažuriranju programskog koda [15].

Modularni i komponentno orijentirani pristup koji koristi Angular zahtijeva jezične značajke koje su dostupne samo u ES6 (ECMAScript 6). Iste funkcionalnosti također se mogu implementirati u standardni JavaScript koji podržava većina preglednika (ECMAScript 5 ili ES5), no to dovodi do vrlo složenog i glomaznog koda. Kako bi se izbjegla složenost i budući da bi prevođenje u ES5 ionako bilo potrebno, Angular tim odlučio je koristiti TypeScript kao jezik po izboru. Uključuje značajke ES6 koje su potrebne za podršku modularnosti okvira, ali također dodaje snažno tipiziranje (eng. Strong typing).

Najvažniji Angular koncepti su:

- Moduli – spremnici koji grupiraju blokove funkcionalnosti koje pripadaju zajedno, kao što su komponente, direktive ili servisi.
- Komponente – definiraju ponašanje jednog dijela ekrana
- Predlošci – HTML datoteke koje definiraju kako se prikazuju komponente
- Povezivanje podataka (eng. Data binding) – proces koji povezuje komponentu s predloškom i omogućuje protok podataka i događaja između njih
- Direktive – Prilagođeni atributi koji poboljšavaju HTML sintaksu i koriste se za definiranje ponašanja određenih elemenata na stranici
- Servisi – Funkcionalnosti za višekratnu upotrebu koje su neovisne o pogledima
- Injektiranje ovisnosti – Način opskrbe klasa (servisi ili komponente) s njihovim ovisnostima (uglavnom servisi)
- Metadata – Upućuje Angular na to kako obraditi klasu (komponentu, modul ili direktivu) ili koje servise treba injektirati

Kao i svi okviri ovakve vrste, ima prilično strmu krivulju učenja zbog mnogih koncepata koje je potrebno u potpunosti shvatiti kako bi se uspješno ovladalo okvirom [16].

6.1.1. Prednosti Angulara

Angular podržava Google, što ga čini pouzdanim i pouzdanim programom koji će najvjerojatnije pratiti Googleova povremena ažuriranja i najave. Postoji detaljna dokumentacija te time Angular postaje pouzdan okvir potkrijepljen velikom količinom korisnih informacija i odgovora na česta pitanja.

Angular pruža velik izbor integracija trećih strana koje se s lakoćom mogu dodati okviru. Ovo razvojnim programerima daje još više alata za poboljšanje cjelokupnog oblika i funkcije njihovih proizvoda.

Angular nudi brže vrijeme učitavanja i povećanu sigurnost upotrebom koncepta poznatog kao kompiliranje unaprijed (eng. Ahead-of-time compiling). Angular kompilira HTML i TypeScript u JavaScript tijekom razvoja, što znači da je sav kod već kompiliran prije nego što preglednik uopće učita web aplikaciju [17].

S elementima i modulima, ovaj je okvir dizajniran da bude potpuno prilagodljiv, dajući veću moć programeru i dizajneru. Elementi se također mogu jednostavno dodati projektima koji su izgrađeni pomoću drugog okvira, što samo pridonosi privlačnosti ovog programa.

6.1.2. Nedostaci Angulara

Jedan od glavnih nedostataka korištenja Angulara su ograničene opcije optimizacije za tražilice i loša dostupnost za tražilice.

Uz složenu mrežu modula, jezika kodiranja, integracija i mogućnosti prilagodbe, Angular može biti veoma teško savladati, te potpuno razumijevanje Angulara definitivno zahtijeva neko vrijeme.

Ako projekt nije odgovarajuće veličine i složenosti za primjenu Angular okvira, korištenje Angulara postaje kontraproduktivno te nepotrebno opterećuje razvojne programere [17].

6.2. ASP.NET Core

ASP.NET Core najnovija je evolucija Microsoftovog popularnog ASP.NET web razvojnog okvira. Nedavne verzije ASP.NET-a doživjele su mnoga inkrementalna ažuriranja, usredotočena na visoku produktivnost programera i dajući prioritet kompatibilnosti s prethodnim verzijama. ASP.NET Core se suprotstavlja tom trendu uvođenjem značajnih arhitektonskih promjena koje promišljaju način na koji je web okvir dizajniran i izgrađen. ASP.NET Core mnogo duguje svom ASP.NET naslijeđu i mnoge su značajke prenesene, no

ASP.NET Core smatra se novim okvirom. Cijela tehnologija je iznova napisana, uključujući i web razvojni okvir i temeljnu platformu. Razvoj ASP.NET Core-a bio je motiviran željom za stvaranjem web razvojnog okvira s četiri glavna cilja:

- Da se pokreće i razvija na više platformi
- Da ima modularnu arhitekturu za lakše održavanje
- Da se u potpunosti razvija kao softver otvorenog koda
- Kako bi bio primjenjiv na trenutne trendove u web razvoju, kao što su aplikacije na strani klijenta i postavljanje na oblak

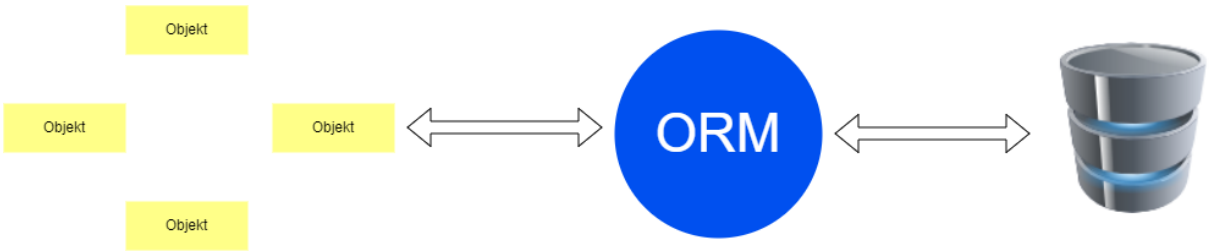
Kako bi se postigli svi ovi ciljevi, Microsoftu je bila potrebna platforma koja bi mogla pružiti temeljne biblioteke za stvaranje osnovnih objekata. Do ove točke razvoj ASP.NET-a uvijek je bio fokusiran i ovisio o .NET Frameworku samo za Windows. Za ASP.NET Core, Microsoft je stvorio laganu platformu koja radi na Windows, Linux i macOS pod nazivom .NET Core. .NET Core dijeli mnogo istih API-ja kao .NET Framework, ali je manji i trenutačno implementira samo podskup značajki koje pruža .NET Framework, s ciljem pružanja jednostavnije implementacije i modela programiranja. To je potpuno nova platforma, a ne nastavak .NET Frameworka, iako koristi sličan kod za mnoge svoje API-je. S .NET Coreom postaje moguće izraditi konzolne aplikacije koje je moguće pokrenuti na više platforma, te je Microsoft stvorio ASP.NET Core kao dodatni sloj na postojeće konzolne aplikacije, te dodavanjem potrebnih biblioteka konzolna aplikacija postaje web aplikacija. ASP.NET Core čini pisanje web aplikacija bržim, lakšim i sigurnijim [18]. ASP.NET Core pruža brojne značajke za izradu web API-ja i web aplikacija:

- Korištenje uzorka model-pogled-kontroler (eng. Model-View-Controller, MVC) pomaže da web API-ji i web aplikacije budu lako testirani.
- Razor Pages model je programiranja temeljen na stranici koji izradu web sučelja čini lakšom i produktivnijom.
- Razor Markup pruža produktivnu sintaksu za Razor stranice i MVC prikaze.
- Tag Helpers omogućuju kodu na strani poslužitelja da sudjeluje u stvaranju i prikazivanju HTML elemenata u Razor datotekama.
- Ugrađena podrška za više formata podataka omogućuje web API-jima da dosegnu širok raspon klijenata, uključujući preglednike i mobilne uređaje.
- Vezanje modela (eng. model binding) automatski preslikava podatke iz HTTP zahtjeva u parametre metode.
- Vezanje modela automatski izvodi provjeru valjanosti modela i na klijentskoj i na poslužiteljskoj strani [19].

6.3. Objektno – relacijsko mapiranje

Objektno – relacijsko mapiranje (eng. Object – relational mapping, ORM) je automatiziran način povezivanja objektnog modela, koji se ponekad naziva i model domene s relacijskom bazom podataka korištenjem metapodataka kao deskriptora objekta i podataka. Važno je razumjeti da postoje mnoge prednosti korištenja ORM-a umjesto drugih tehnika pristupa podacima. Prvo, ORM automatizira konverziju objekta u tablicu i tablice u objekt, što pojednostavljuje razvoj aplikacije. Ovaj pojednostavljeni razvoj dovodi do bržeg izlaska na tržište i smanjenih troškova razvoja i održavanja. Dalje, ORM zahtijeva manje koda u usporedbi s ugrađivanjem SQL upita, rukom pisanim pohranjenim procedurama ili bilo kojim drugim pozivima sučelja s relacijskim bazama podataka. Ovime ORM postiže istu funkcionalnost, no uz manje koda. Posljednje, ali ne i najmanje važno, ORM pruža transparentno predmemoriranje objekata na aplikacijskom sloju, čime se poboljšavaju performanse sustava. ORM alati sadrže nekoliko ključnih aspekata:

- Mapiranje objekta u bazu podataka najvažniji aspekt je ORM alata. ORM alat mora imati mogućnost mapiranja poslovnih objekata u pozadinske tablice baze podataka putem neke vrste mapiranja metapodataka. Ovo je srž objektno-relacijskog mapiranja.
- Predmemoriranje objekata omogućuje predmemoriranje objekata ili podataka za poboljšanje performansi u sloju postojanosti.
- Podrška za više platformi baze podataka – ORM alat nudi prenosivost s jednog sustava upravljanja relacijskom bazom podataka (eng. Relational Database Management System) na drugog.
- Dinamičko postavljanje upita – ORM alat nudi projekcije i besklasno postavljanje upita ili dinamički SQL na temelju korisničkog unosa.
- Lijeno učitavanje – optimiziranje korištenja memorije poslužitelja baze podataka [20].



Slika 17: Objektno - relacijsko mapiranje [autorski rad]

7. Praktični dio

7.1. Funkcionalnosti

Aplikacija svoju temeljnu ideju pronalazi u građevinskom sektoru. Njena primarna namjena jest pružati podršku poslovanju jednom građevinskom poduzeću. Ključna funkcionalnost jest vođenje i evidencija građevinskih projekata. Aplikaciju mogu koristiti tri vrste korisnika: administrator, voditelj projekata te naručitelj radova.

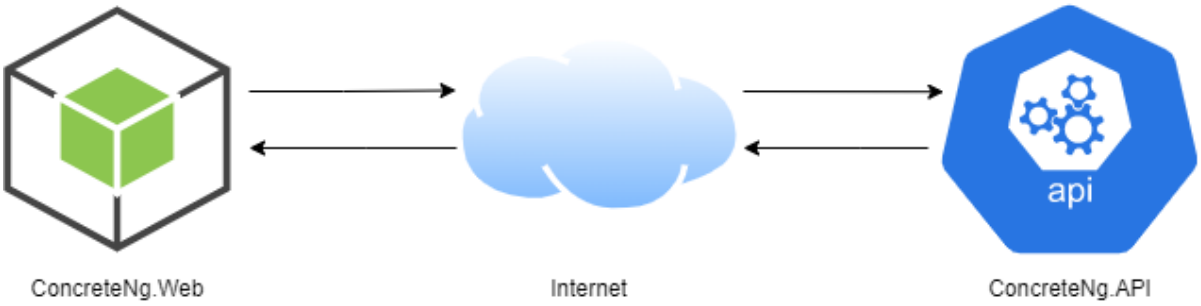
Voditelji projekata imaju pristup pregledu svojim projektima unutar svog poduzeća. Voditelji projekata kroz aplikaciju za svaki projekt definiraju kategorije radova, te određuju zadatke koje je potrebno ispuniti za pojedinu kategoriju. Također svaki od ovih zadataka iziskuje I trošenje novčanih resursa, stoga voditelj projekata putem aplikacije vodi i troškovnik. Također voditelji projekata mogu i unositi slike te dokumente u projektni repozitorij. To mogu biti nacrti, planovi, ili slike gradilišta. Također postoje i partneri poduzeća. Ukoliko postoji potreba, partner poduzeća može izvršiti neke radove u ime poduzeća (npr tvrtka nema bager te mora pozvati svog partnera da iskopa rupu). Voditelji projekata putem aplikacije mogu i voditi dnevnik projekta.

Administratori imaju sve mogućnosti kao i voditelji projekata, ali na razini cijelog poduzeća, a ne samo svojih projekata. Administratori su zaduženi za kreiranje novih projekata, te dodjelu voditelja projekta. Administratori također mogu dodavati nove zaposlenike, bili oni novi administratori ili voditelji projekata. Uz to, administrator mogu i dodavati nove partnere.

Naručitelji projekata, odnosno kupci, imaju pristup pregledu naručenog projekta, kako bi u svakom vremenu mogli znati u kojem stadiju se nalazi.

7.2. Arhitektura

Izrađena web aplikacija, naziva ConcreteNg, koristi server – klijent arhitekturu. Klijentska aplikacija preko HTTP protokola šalje zahtjeve serverskoj aplikaciji koja obrađuje zahtjev te zatim šalje odgovor.



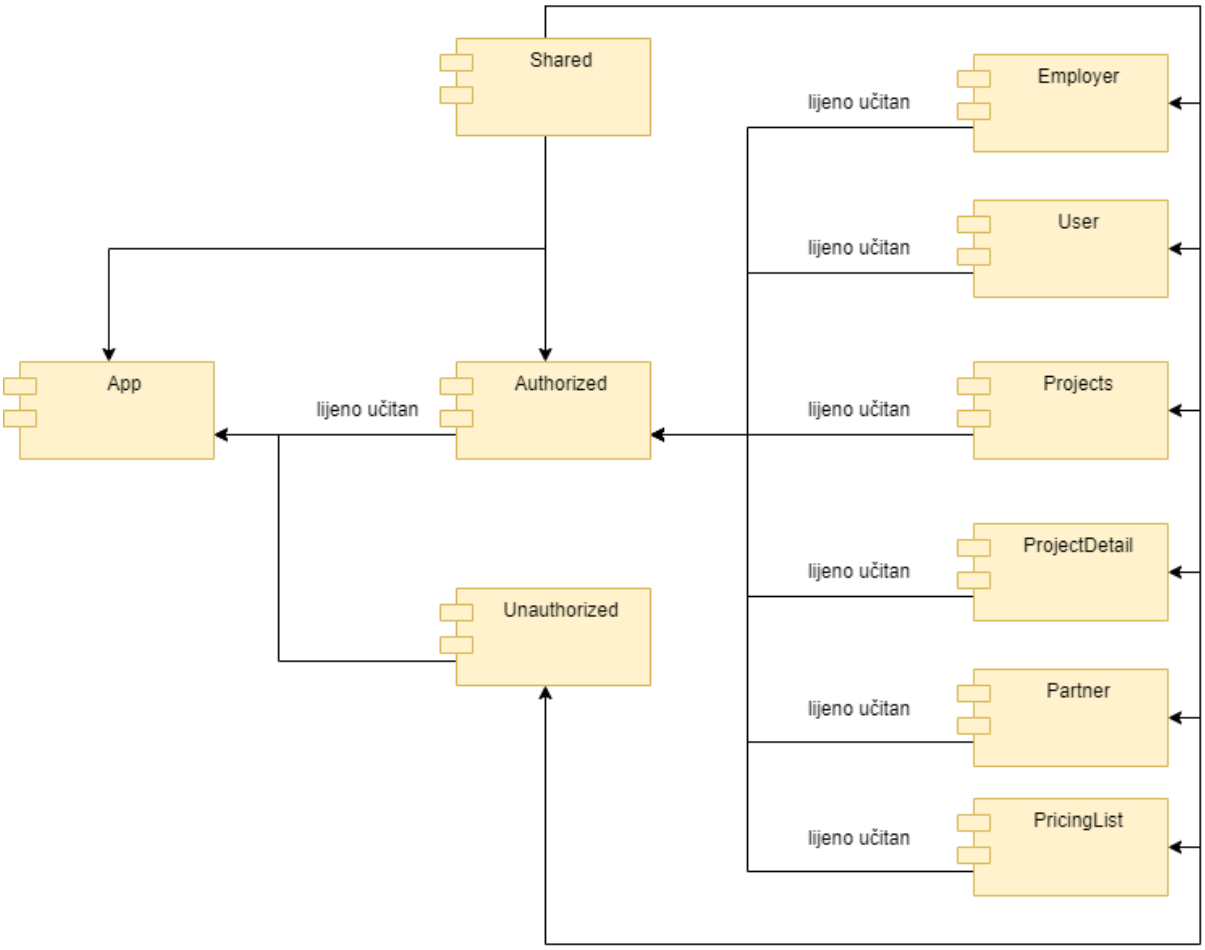
Slika 18: Arhitektura rješenja [autorski rad]

7.2.1. Klijentska aplikacija

Klijentska aplikacija izrađena je korištenjem Angular 13 razvojnog okvira, uz primjenu Angular Material biblioteke za izradu komponenata. Izrađen sustav sastoji se od ukupno deset modula. Od deset modula, njih šest je lijeno učitano s ciljem učitavanja komponenata i servisa tog modula tek kada korisnik to zatraži, čime se smanjuje kompleksnost inicijalnog pokretanja aplikacije. Lijeno učitavanje u Angularu provodi se pozivanjem loadChildren metode prilikom definiranja ruta, gdje se definira koje module je potrebno učitati prilikom aktivacije određene rute.

```
const routes: Routes = [
    { path: '', redirectTo: 'employer-overview'},
    { path: 'employer-overview', loadChildren: () => import('./employer-
overview/employer.module').then(m => m.EmployerModule) },
    { path: 'project-details/:id', loadChildren: () =>
import('./projects/project-details/project-detail.module').then(m =>
m.ProjectDetailModule) },
    { path: 'pricing-list', loadChildren: () => import('./pricing-
list/pricing-list.module').then(m => m.PricingListModule) },
    { path: 'partners', loadChildren: () =>
import('./partners/partner.module').then(m => m.PartnerModule)},
    { path: 'projects', loadChildren: () =>
import('./projects/projects.module').then(m => m.ProjectsModule)},
    { path: 'employees', loadChildren: () =>
import('./users/user.module').then(m => m.UserModule)}
];
```

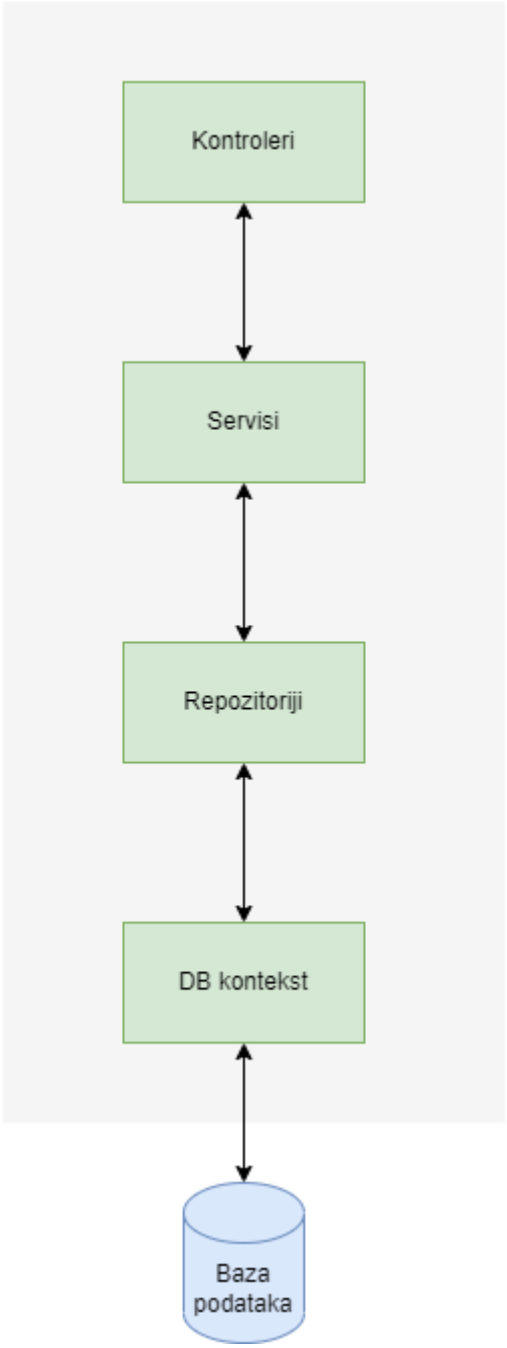
Svaki od ovih modula, osim Shared modula, sadrži komponente, servise i rute potrebne za implementaciju svojih funkcionalnosti. Shared modul sadrži elemente koji su potrebni svim modulima.



Slika 19: ConcreteNg.Web arhitektura [autorski rad]

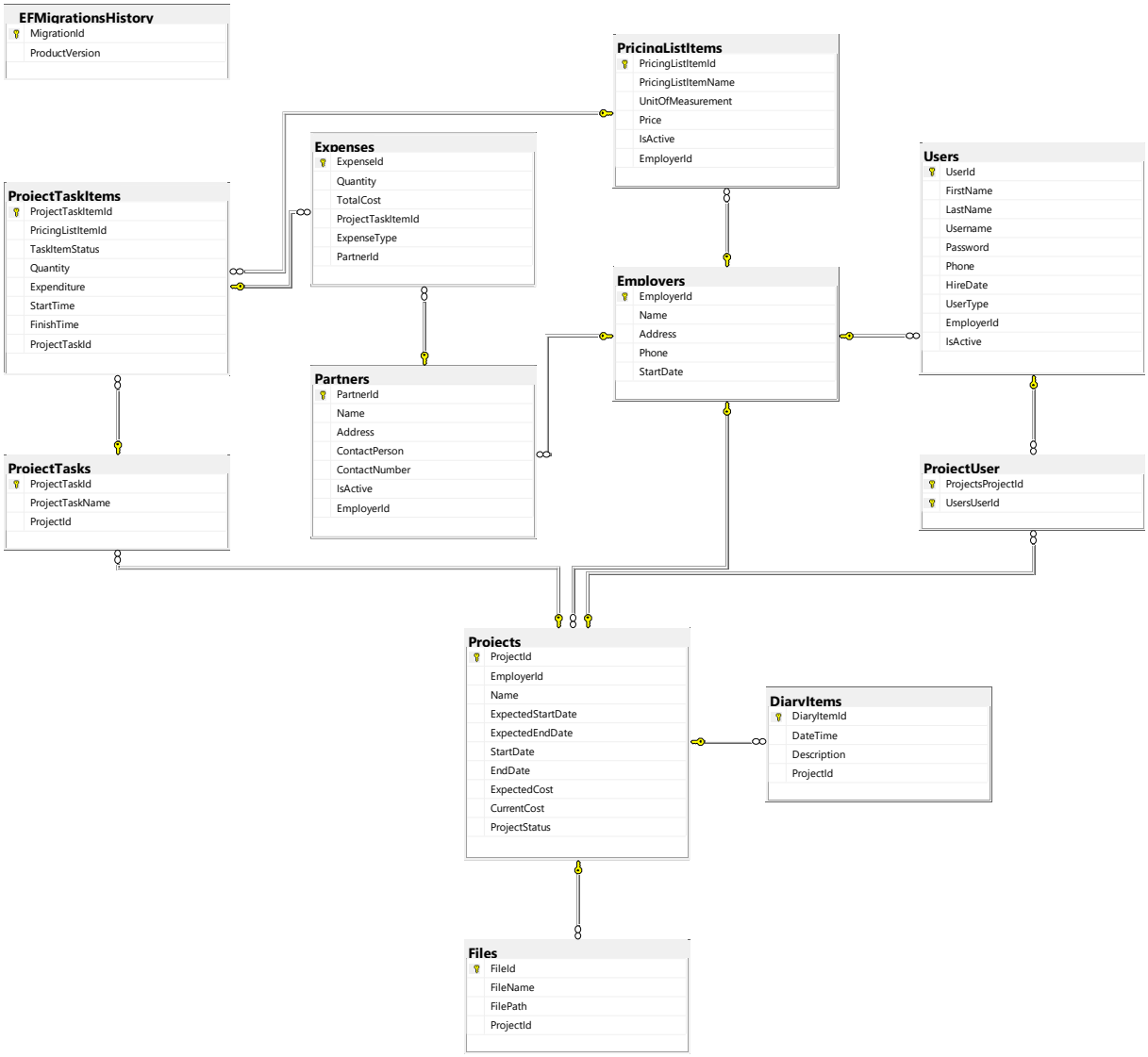
7.2.2. Serverska aplikacija

Serverska aplikacija izrađenja je korištenjem ASP.NET Core tehnologije za izradu API web servisa, uz primjenu Entity Frameworka za objektno – relacijsko mapiranje aplikacijskih modela te entiteta baze podataka. Arhitektura aplikacije temelji se na uzorku slojevite arhitekture. Rad serverske aplikacije počinje primanjem zahtjeva u kontrolerskom sloju, koji potom od servisa traži obradu podataka. Ukoliko je potrebno, servis od repozitorija može zatražiti akcije vezane uz bazu podataka, koji za izvršenje te funkcionalnosti koristi kontekst baze podataka, kao vezu između aplikacijskih modela te entiteta i relacija baze podataka. Nakon obrade, kontroler pošiljatelju zahtjeva vraća kodni broj koji označuje status obrade, te odgovor u JSON obliku.



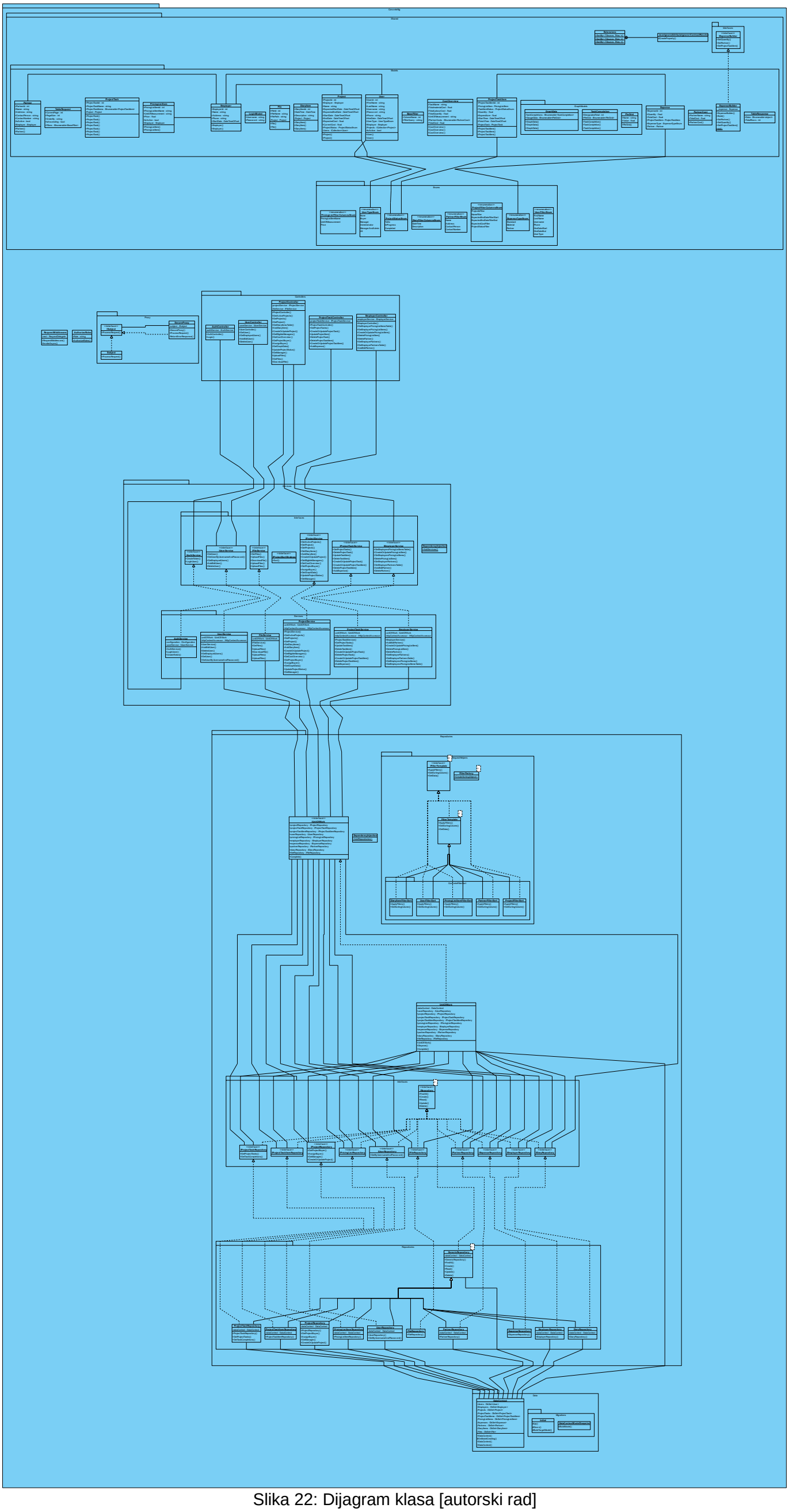
Slika 20: ConcreteNg.API arhitektura [autorski rad]

7.3. Baza podataka



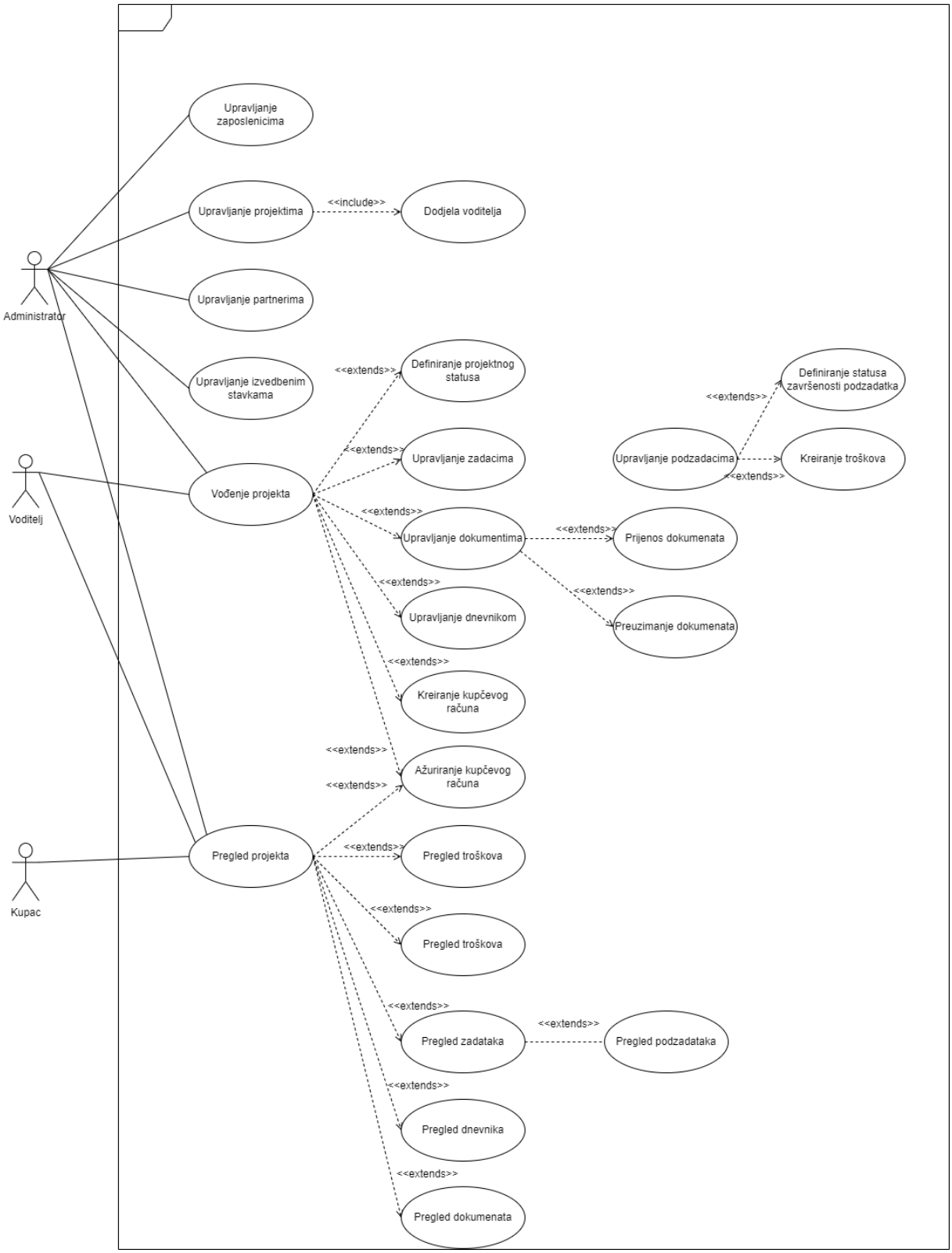
Slika 21: ERA model [autorski rad]

7.4. Dijagram klasa



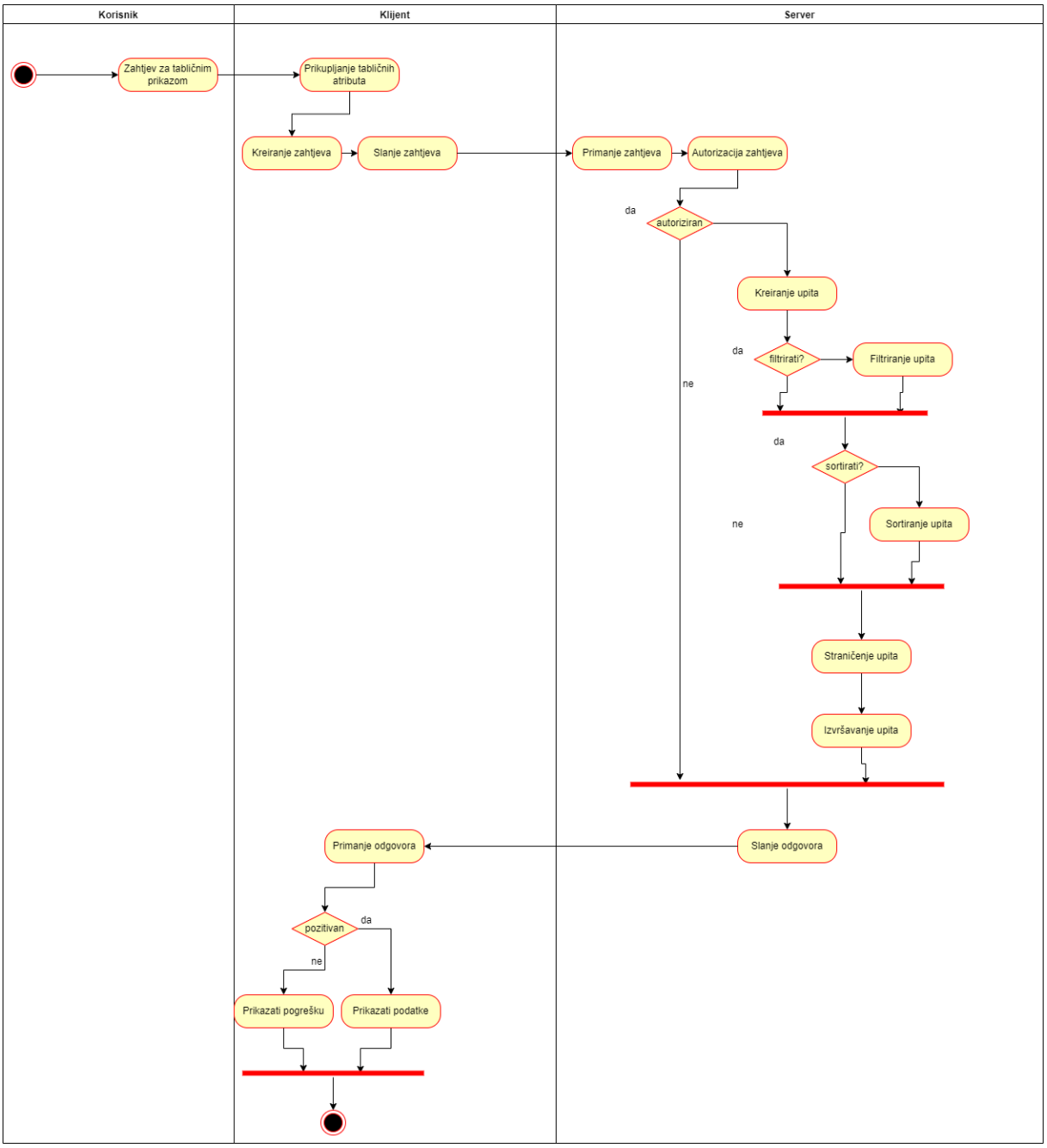
Slika 22: Dijagram klasa [autorski rad]

7.5. Dijagram slučajeva korištenja

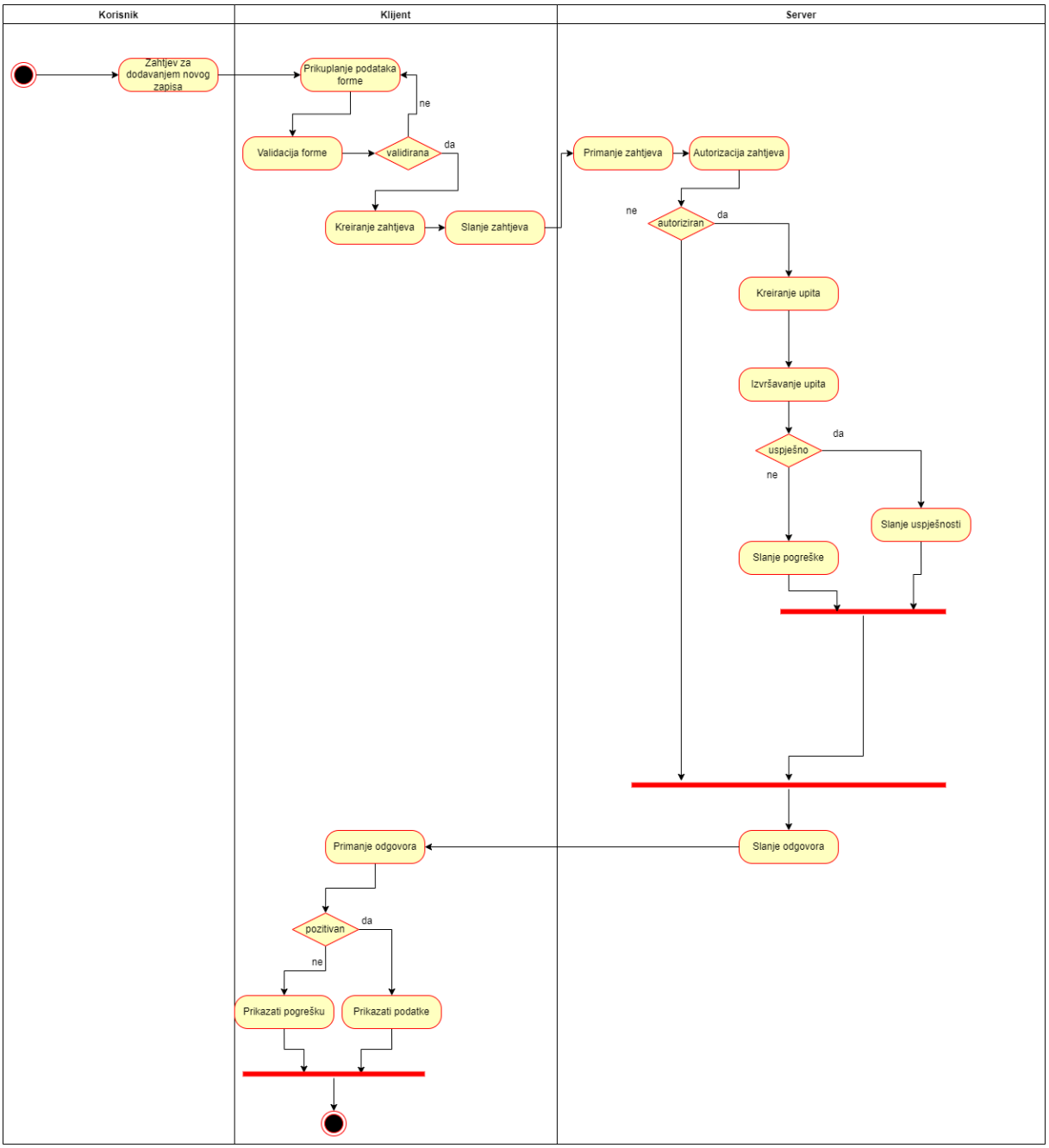


Slika 23: Dijagram slučajeva korištenja [autorski rad]

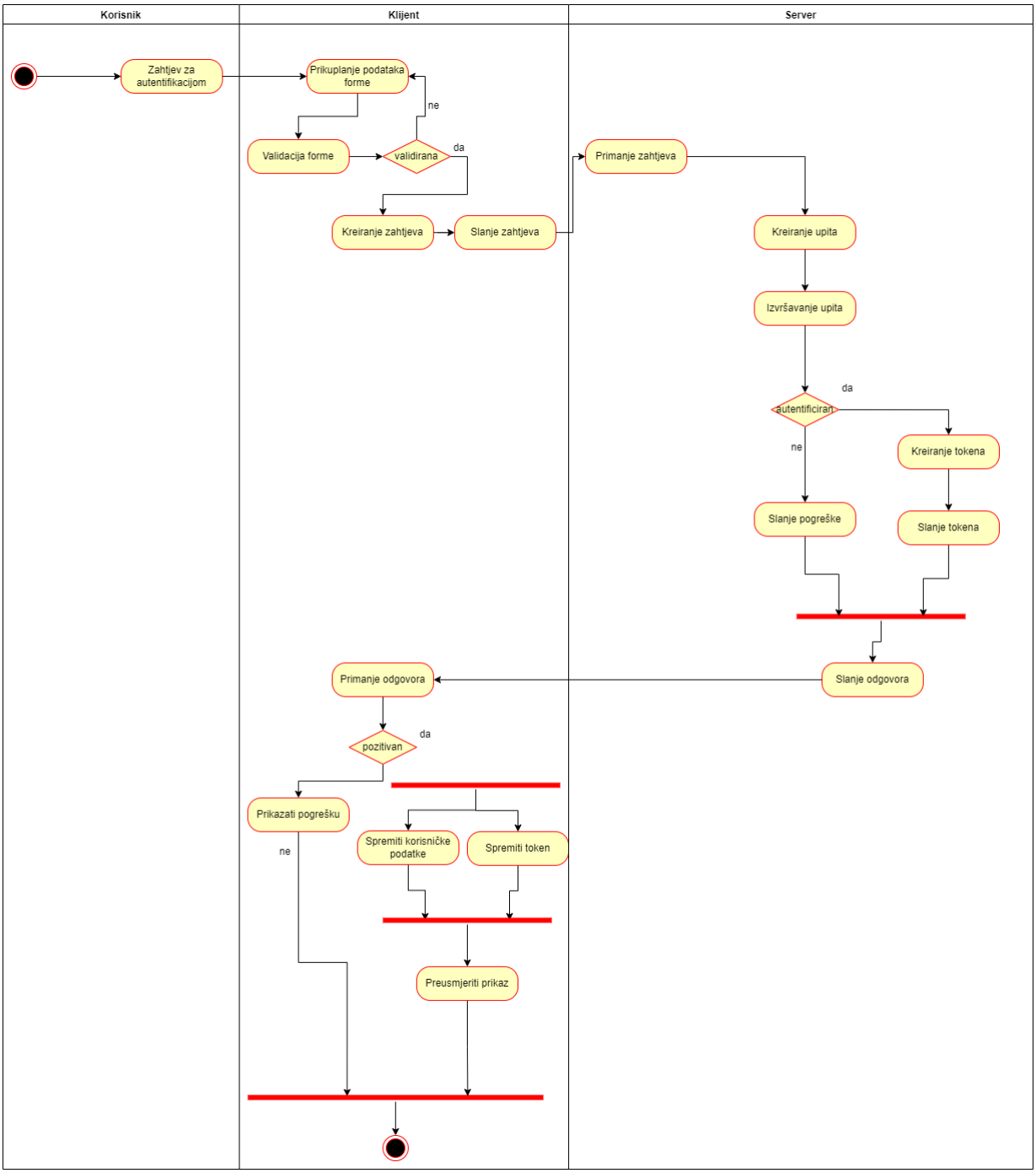
7.6. Dijagrami aktivnosti



Slika 24: Dijagram aktivnosti tabličnog prikaza [autorski rad]



Slika 25: Dijagram aktivnosti unosa novog zapisa [autorski rad]



Slika 26: Dijagram aktivnosti prijave [autorski rad]

7.7. Injektiranje ovisnosti

I klijentska i serverska aplikacija koriste načelo injektiranja ovisnosti koristeći uzorak konstruktorskog injektiranja. Kako bi neki servis mogao biti injektiran u Angularu, potrebno mu je dodati @Injectable svojstvo:

```
@Injectable({
    providedIn: 'root'
})
```

Sada je moguće traženi servis injektirati u neku komponentu, ili drugi servis:

```
constructor(private employerService: EmployerService,{ }
```

Sličan postupak provodi se i u serverskoj aplikaciji, gdje je potrebno registrirati servise i repozitorije prilikom pokretanja aplikacije:

```
public static IServiceCollection AddRepositories(this IServiceCollection
services)
{
    services.AddScoped<IUserRepository, UserRepository>();
    services.AddScoped<IProjectRepository, ProjectRepository>();
    services.AddScoped<IProjectTaskRepository, ProjectTaskRepository>();
    services.AddScoped<IProjectTaskItemRepository,
ProjectTaskItemRepository>();
    services.AddScoped<IPricingListRepository,
PricingListItemRepository>();
    services.AddScoped<IEmployerRepository, EmployerRepository>();
    services.AddScoped<IPartnerRepository, PartnerRepository>();
    services.AddScoped<IExpenseRepository, ExpenseRepository>();
    services.AddScoped<IDiaryRepository, DiaryRepository>();
    services.AddScoped<IUnitOfWork, UnitOfWork>();
    return services;
}

public static IServiceCollection AddServices(this IServiceCollection
services)
{
    services.AddScoped<IUserService, UserService>();
    services.AddScoped<IAuthService, AuthService>();
    services.AddScoped<IProjectService, ProjectService>();
    services.AddScoped<IProjectTaskService, ProjectTaskService>();
    services.AddScoped<IEmployerService, EmployerService>();
    return services;
}

builder.Services.AddRepositories();
builder.Services.AddServices();
```

Primjer konstruktorskog injektiranja:

```
private readonly IProjectService projectService;

public ProjectController(IProjectService _projectService)
{
    projectService = _projectService;
}
```

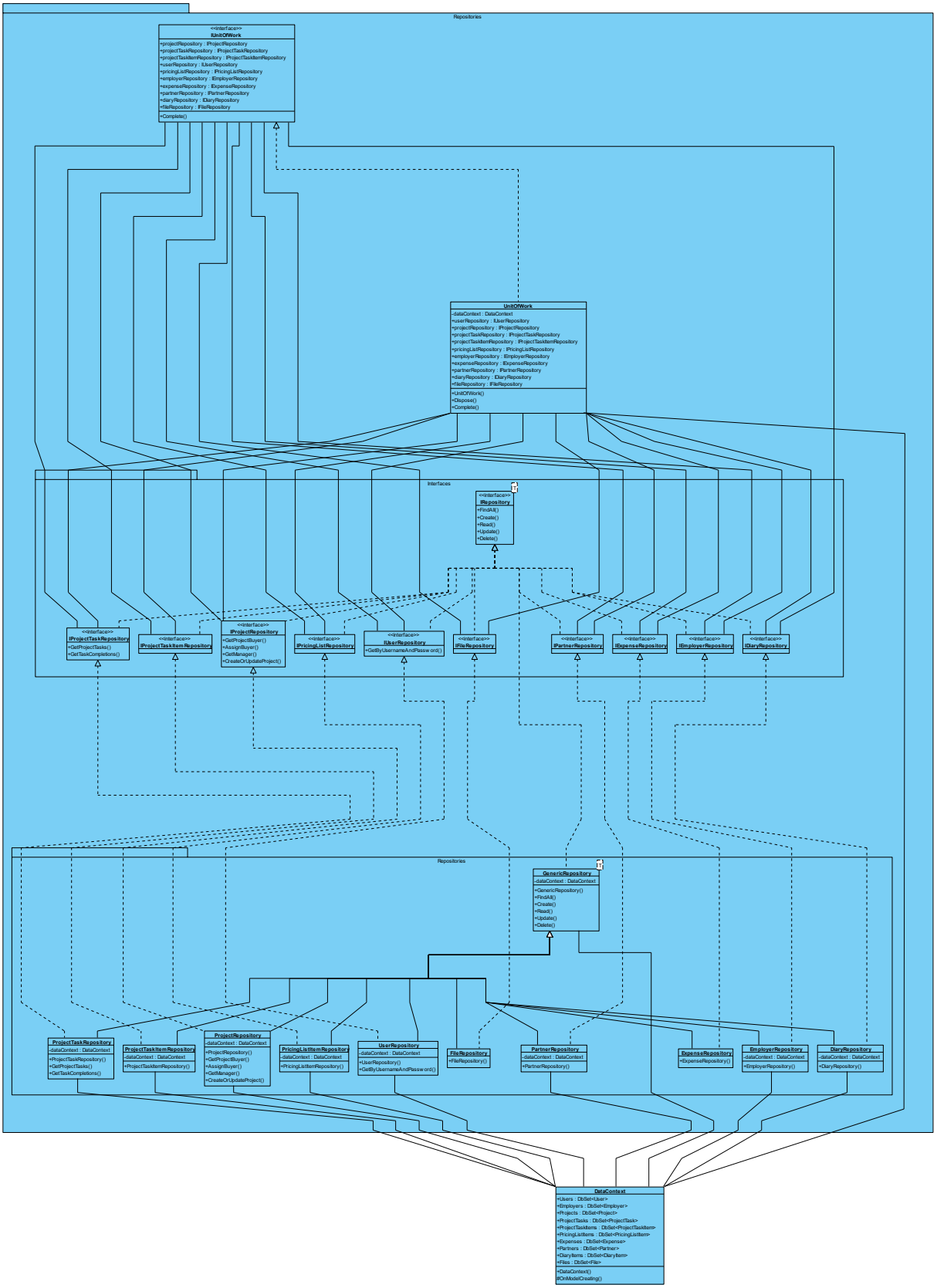
Svi servisi i repozitoriji registrirani su tako da se kreiraju jednom za svaki klijentski zahtjev. Tako će dvije klase, koje injektiraju isti servis, injektiranjem dobiti isti objekt. Injektirani objekti servisa i repozitorija uništavaju se na kraju zahtjeva. Korištenjem injektiranja ovisnosti, u aplikaciji je smanjena povezanost između klasa i njihovih ovisnosti, te kod postaje lakši za održavanje i ponovno korištenje.

7.8. Rad s bazom podataka

Kako serverska aplikacija koristi Entity Framework, potrebno je, kao i servise, registrirati kontekst baze podataka za injektiranje ovisnosti.

```
builder.Services.AddDbContext<DataContext>(options =>
{
   options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultC
   onnection"));
});
```

Za stvaranje dodatnog sloja apstrakcije između sloja poslovne logike i sloja pristupa podataka, implementiran je repozitorij uzorak dizajna za svaki entitet baze podataka. Još jedna prednost korištenja repozitorija jest laka promjena alata za objektno relacijsko mapiranje. Ukoliko se u nekom trenutku odluči za korištenje nekog drugog alata, potrebno je izmijeniti samo repozitorije, a ne cijeli sloj poslovne logike. Ova aplikacija sastoji se od jedanaest repozitorija: deset implementacijskih i jedan generički.



Slika 27: Repozitoriji [autorski rad]

Generički repozitorij definira osnovne akcije svih repozitorija, a implementacijski repozitoriji implementiraju te akcije za svoj entitet, te po potrebi imaju i dodatne metode. Osnovne akciju uključuju upit za sve zapise, dodavanje novog zapisa, brisanje zapisa, čitanje zapisa na temelju identifikacijskih ključeva, te ažuriranje zapisa. Primjer implementacije UserRepository repozitorija:

```
public interface IRepository<T> where T : class
{
    IQueryable<T> FindAll();
    void Create(T entity);
    T Read(object keys);
    void Update(T entity);
    void Delete(T entity);
}
public interface IUserRepository : IRepository<User>
{
    User GetByUsernameAndPassword(LoginModel loginModel);
}
public abstract class GenericRepository<T> : IRepository<T>
    where T : class
{
    private readonly DataContext dataContext;
    public GenericRepository(DataContext dbContext)
    {
        dataContext = dbContext;
    }
    public virtual IQueryable<T> FindAll()
    {
        return dataContext.Set<T>();
    }
    public virtual void Create(T entity)
    {
        dataContext.Set<T>().Add(entity);
    }
    public virtual T Read(object keys)
    {
        return dataContext.Set<T>().Find(keys);
    }
    public virtual void Update(T entity)
    {
        dataContext.Entry(entity).State = EntityState.Modified;
    }

    public virtual void Delete(T entity)
    {
        dataContext.Set<T>().Remove(entity);
    }
}

public class UserRepository : GenericRepository<User>, IUserRepository
{
    private readonly DataContext dataContext;
    public UserRepository(DataContext dbContext) : base(dbContext)
    {
        dataContext = dbContext;
    }
    public User GetByUsernameAndPassword(LoginModel loginModel)
    {
```

```
     return dataContext.Users.Include(x => x.Employer).FirstOrDefault(u =>
u.Username == loginModel.Username && u.Password == loginModel.Password);
    }
}
```

Zajedno sa repozitorijima, implementiran je i uzorak dizajna naziva jedinica rada. Ovime se u aplikaciji postiže izolacija od promjene podataka, odnosno ili se pohranjuju podaci ako su svi upiti uspješni, ili se ne pohranjuje ni jedan rezultat upita.

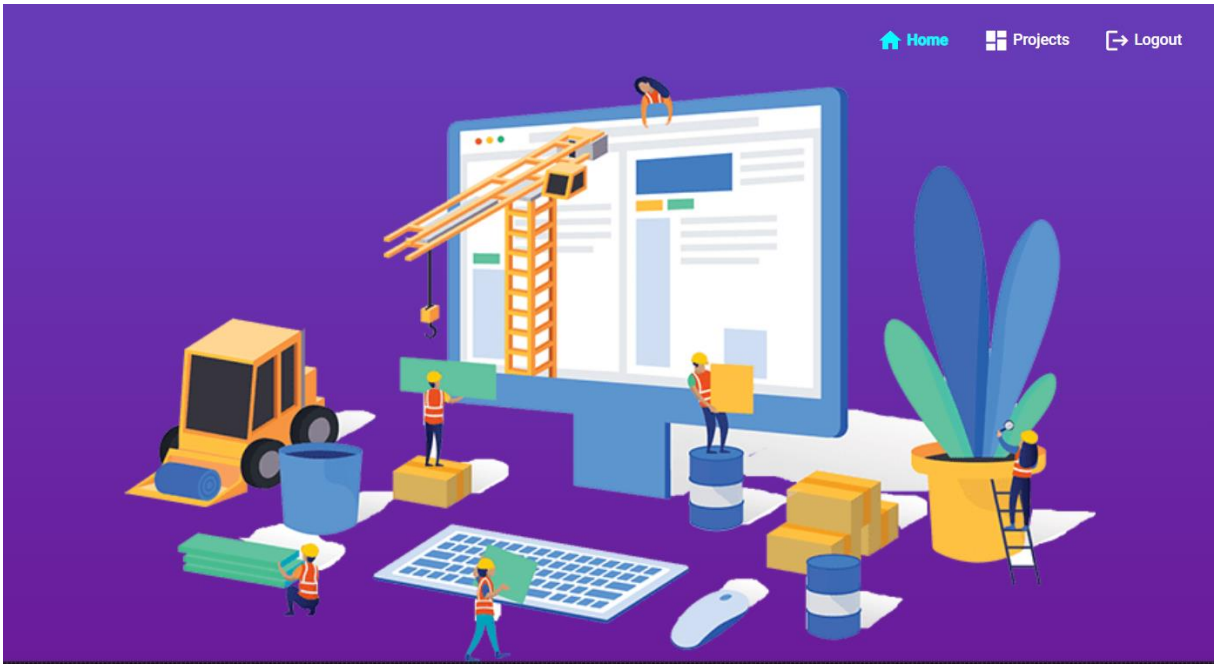
```
public interface IUnitOfWork : IDisposable{
    IProjectRepository projectRepository { get; }
    IProjectTaskRepository projectTaskRepository { get; }
    IProjectTaskItemRepository projectTaskItemRepository { get; }
    IUserRepository userRepository { get; }
    IPricingListRepository pricingListRepository { get; }
    IEmployerRepository employerRepository { get; }
    IExpenseRepository expenseRepository { get; }
    IPartnerRepository partnerRepository { get; }
    IDiaryRepository diaryRepository { get; }
    IFileRepository fileRepository { get; }
    int Complete();
 }

 public class UnitOfWork : IUnitOfWork
 {
     private readonly DataContext dataContext;
     public IUserRepository userRepository { get; }
     public IProjectRepository projectRepository { get; }
     public IProjectTaskRepository projectTaskRepository { get; }
     public IProjectTaskItemRepository projectTaskItemRepository { get; }
     public IPricingListRepository pricingListRepository { get; }
     public IEmployerRepository employerRepository { get; }
     public IExpenseRepository expenseRepository { get; }
     public IPartnerRepository partnerRepository { get; }
     public IDiaryRepository diaryRepository { get; }
     public IFileRepository fileRepository { get; }

     public UnitOfWork(
         DataContext dbContext,
         IUserRepository _userRepository,
         IProjectRepository _projectRepository,
         IProjectTaskRepository _projectTaskRepository,
         IProjectTaskItemRepository _projectTaskItemRepository,
         IPricingListRepository _pricingListRepository,
         IEmployerRepository _employerRepository,
         IExpenseRepository _expenseRepository,
         IPartnerRepository _partnerRepository,
         IDiaryRepository _diaryRepository,
         IFileRepository _fileRepository)
     {
         dataContext = dbContext;
         userRepository = _userRepository;
         projectRepository = _projectRepository;
         projectTaskRepository = _projectTaskRepository;
         projectTaskItemRepository = _projectTaskItemRepository;
         pricingListRepository = _pricingListRepository;
         employerRepository = _employerRepository;
         expenseRepository = _expenseRepository;
         partnerRepository = _partnerRepository;
         diaryRepository = _diaryRepository;
```

```
            fileRepository = _fileRepository;
        }

        public void Dispose()
        {
            if (dataContext == null) return;
            dataContext.Dispose();
        }
        public int Complete()
        {
            return dataContext.SaveChanges();
        }
}
```

Svaki servis koji ima potrebu za komuniciranje s bazom podataka, injektira IUnitOfWork, te preko njega pristupa repozitorijima koji su mu potrebni.

7.9. Početna stranica

Početna stranica vrlo je jednostavnog izgleda, sadrži tek sliku koja korisniku vizualno opisuje tematiku aplikacije.



Slika 28: Početna stranica [autorski rad]

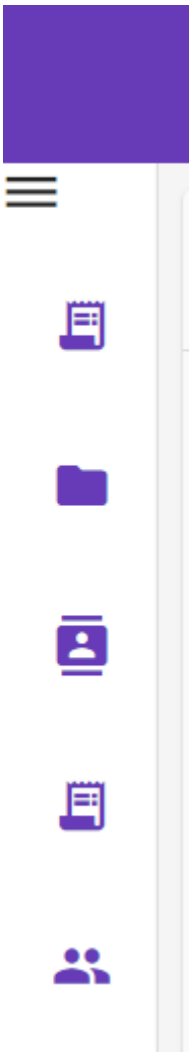
7.10. Navigacija

Klijentska aplikacija sadrži dva navigacijska izbornika. Jedan izbornik nalazi se u zaglavlju, i namijenjen je za pristup početnoj stranici te stranici za prijavu. Ukoliko se korisnik uspješno prijavi pojavljuju se dvije opcije: opcija za pristup projektima, te opcija za odjavu, a opcija za prijavu je skrivena sve dok se korisnik ne odjavi.

Slika 29: Zaglavlje [autorski rad]

```
<mat-toolbar color="primary" class="app-header header">
    <span class="nav-tool-items">
        <a mat-button [routerLink]="['/landing-page']"
routerLinkActive="active-route">
            <mat-icon>home</mat-icon>
            Home
        </a>
        <a *ngIf="!LoggedIn"
            mat-button [routerLink]="['/login']"
routerLinkActive="active-route">
            <mat-icon>login</mat-icon>
            Log In
        </a>
        <a *ngIf="LoggedIn"
            mat-button [routerLink]="['/authorized']"
routerLinkActive="active-route">
            <mat-icon>dashboard</mat-icon>
            Projects
        </a>
        <a *ngIf="LoggedIn"
            mat-button (click)="LogOut()">
            <mat-icon>logout</mat-icon>
            Logout
        </a>
    </span>
</mat-toolbar>
```

Drugi navigacijski izbornik pojavljuje se nakon uspješne prijave u obliku izbornika na bočnoj traci.



Slika 30: Sporedni izbornik [autorski rad]

```
<section [class.sidenav]="isExpanded">
    <div class="toggle">
      <mat-icon (click)="toggleMenu.emit(null)">
        {{ isExpanded ? "keyboard_backspace" : "dehaze" }}
      </mat-icon>
    </div>
    <mat-list class="nav" *ngFor="let route of routeLinks">
      <a
        mat-list-item
        routerLinkActive="active-link"
        class="hover"
        routerLink="{{ route.link }}"
      >
        <mat-icon
            color="primary"
            mat-list-icon
            [matTooltip]="!isExpanded ? route.name : ''"
            matTooltipPosition="right"
        >
            {{ route.icon }}</mat-icon
        >
```

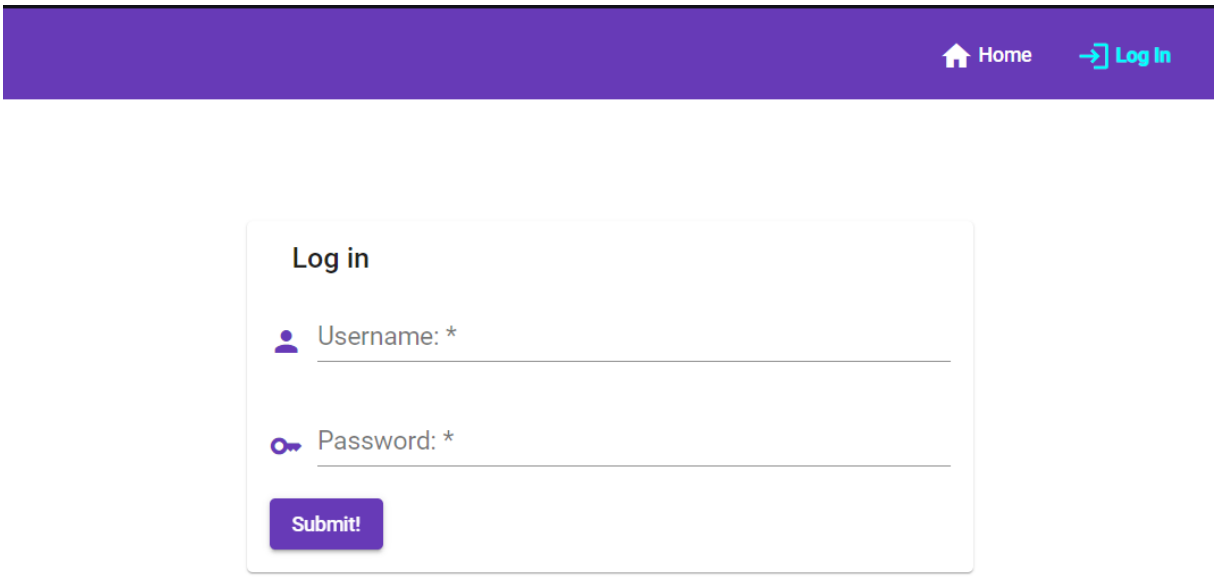
```
        <p matLine *ngIf="isExpanded">{{ route.name }}</p>
      </a>
    </mat-list>
  </section>
```

Elementi se u ovom izborniku pojavljuju ovisno o vrsti korisničkog računa trenutno prijavljenog korisnika, stoga se injektira AccountService servis u ovu komponentu kako bi komponenta znala što mora prikazati.

```
constructor(private accountService: AccountService) { }

ngOnInit(): void {
    if(this.accountService.userValue?.userType !== UserTypeEnum.Buyer){
        this.routeLinks = [...this.routeLinks, ...[
            { link: "partners", name: "Partners", icon: "contacts" },
            { link: "pricing-list", name: "Pricing List", icon:
"receipt_long" },]
        ]
    }
    if(this.accountService.userValue?.userType ===
UserTypeEnum.Administrator){
        this.routeLinks = [...this.routeLinks, ...[
            { link: "employees", name: "Employees", icon: "group" },]
        }
    }
}
```

7.11. Autentifikacija i autorizacija

Proces autentifikacije započinje slanjem zahtjeva serverskoj aplikaciji na odgovarajuću adresu, slanjem korisničkog imena i lozinke.



Slika 31: Forma za prijavu [autorski rad]

```
<div class="login-wrapper" fxLayout="row" fxLayoutAlign="center center">
    <mat-card class="box">
```

```
            <mat-card-header>
                <mat-card-title>Log in</mat-card-title>
            </mat-card-header>
            <form [formGroup]="LoginForm" (ngSubmit)="onSubmit()"
   class="login-form">
                <div class="input-container">
                    <mat-icon color="primary">person</mat-icon>
                    <mat-form-field class="login-full-width">
                        <mat-label>Username:</mat-label>
                        <input matInput placeholder="Username"
   formControlName="username">
                    </mat-form-field>
                </div>
                <div class="input-container">
                    <mat-icon color="primary">key</mat-icon>
                    <mat-form-field class="login-full-width">
                        <mat-label>Password:</mat-label>
                        <input type="password" matInput
   placeholder="Password" formControlName="password">
                    </mat-form-field>
                </div>
                <button mat-raised-button color="primary">Submit!</button>
            </form>
        </mat-card>
   </div>
```

Klikom na gumb za prijavu klijentska aplikacija kreira zahtjev sa potrebnim podacima prema serverskoj aplikaciji za prijavu korisnika korištenjem injektiranog HttpClient servisa.

```
constructor(
        private http: HttpClient,
        private toastr: ToastrService,
        private router: Router,
        private accountService: AccountService
        ) {
}
userUrl: string = environment.BASE_URL+ "api/Auth";
LogUser(username: string, password: string) {
        const data = {"Username" : username, "Password" : password};
        this.http.post<any>(this.userUrl + "/login", data).subscribe({
                next: (token: any) =>{
                    this.accountService.JwtTokenValue = token;
                    this.http.get(environment.BASE_URL +
   "api/User").subscribe((data: any)=>{
                    this.accountService.userValue = data;
                    this.router.navigate(['authorized/employer-overview']);
                 })
                },
                error: err => {
                if(err.status === 404){
                this.toastr.error("Invalid Credentials", "",{
                positionClass: 'toast-top-full-width'
                })
            }
        },
   })
}
```

Serverska aplikacija prima zahtjev te kontroler koji je primio zahtjev poziva odgovarajući injektirani servis, u ovom slučaju IAuthService, za obradu zahtjeva.

```
private IAuthService authService;
public AuthController(IAuthService _authService)
{
    authService = _authService;
}
```

Serverska aplikacija provjerava postoji li korisnik s tim korisničkim imenom i lozinkom pozivanjem injektiranog IUserService servisa, koji potom poziva UserRepository preko injektiranog IUnitOfWork uzorka dizajna. Ukoliko ne postoji, kontroler vraća status NotFound sa porukom „Invalid credentials“. Ukoliko postoji, AuthService servis kreira autentifikacijski JWT (Json Web Token) token.

```
[HttpPost("login")]
public async Task<ActionResult<string>> Login([FromBody]LoginModel
loginModel)
{
    string token = authService.LogInUser(loginModel);
    if (string.IsNullOrEmpty(token))
    {
         return NotFound("Invalid credentials");
    }
    return Ok(JsonSerializer.Serialize(token));
}
private IConfiguration configuration;
private readonly IUserService userService;

public AuthService(IConfiguration _configuration, IUserService userService)
{
    configuration = _configuration;
    userService = _userService;
}
public string LogInUser(LoginModel loginModel)
{
    User user = userService.GetUserByUsernameAndPassword(loginModel);
    if(user == null)
    {
        return string.Empty;
    }
    return CreateToken(user);
}
public User GetUserByUsernameAndPassword(LoginModel loginModel)
{
    return unitOfWork.userRepository.GetByUsernameAndPassword(loginModel);
}
public User GetByUsernameAndPassword(LoginModel loginModel)
{
        return dataContext.Users.Include(x => x.Employer).FirstOrDefault(u
=> u.Username == loginModel.Username && u.Password == loginModel.Password);
}
public string CreateToken(User user)
{
    List<Claim> claims = new List<Claim>
    {
       new Claim(ClaimTypes.NameIdentifier, user.UserId.ToString()),
       new Claim(ClaimTypes.Role, user.UserType.ToString()),
       new Claim(type: "EmployerID", value:
   user.Employer.EmployerId.ToString())
    };
    var key = new SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes(
```

```
                    configuration.GetSection("AppSettings:Token").Value)
            );
    var credentials = new SigningCredentials(key,
SecurityAlgorithms.HmacSha512Signature);
    var token = new JwtSecurityToken(
        claims: claims,
        expires: DateTime.Now.AddDays(1),
        signingCredentials: credentials
    );
   var jwt = new JwtSecurityTokenHandler().WriteToken(token);
   return jwt;
}
```

Taj token sadrži osnovne podatke o korisniku: njegov identifikacijski ključ, vrstu korisnika, te identifikacijski ključ njegovog poduzeća. Token je zaštićen simetričnim ključem na temelju zapisa u konfiguracijskoj datoteci, potpisan je SHA512 algoritmom te vrijedi jedan dan. Kod pozitivnog odgovora, klijentska aplikacija korištenjem AccountService servisa postavlja token te korisničke podatke u spremnik sesije (eng. Session Storage) preglednika, odakle dohvaća podatke za kasnije potrebe.

```
LoggedIn = new Subject();
public get userValue(): any {
    return JSON.parse(sessionStorage.getItem('currentUser') || '{}') as
User;
}
public set userValue(user: User) {
    sessionStorage.setItem('currentUser', JSON.stringify(user));
}
public get JwtTokenValue(): any{
    return sessionStorage.getItem('token') as string;
}
public set JwtTokenValue(token: string) {
    this.LoggedIn.next(true)
    sessionStorage.setItem('token', token);
}
```

Kako bi klijentska aplikacija za svaki zahtjev, osim autentifikacijskog, slala autentifikacijski token, zaustavlja svaki zahtjev koji šalje preko HTTP protokola, kopira ga, dodaje mu token u zaglavlje te ga šalje prema serveru.

```
constructor(private accountService: AccountService) { }

intercept(request: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    const token = this.accountService.JwtTokenValue
    if (token) {
       request = request.clone({
           setHeaders: { Authorization: `bearer ${token}` }});
       }
    return next.handle(request);
}
```

Klijentska aplikacija također zabranjuje navigiranje ukoliko korisnik nije prijavljen, odnosno preko injektiranog AccountService servisa se provjerava postoji li u spremniku sesije autentifikacijski token..

```
export class AuthGuard implements CanActivate {
    constructor(
        private accService: AccountService,
        private router: Router)
    { }
    canActivate(
        route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {
            const isAuthenticated = (this.accService.JwtTokenValue !==
null);
            const userRole = this.accService.userValue?.userType;
            if (!isAuthenticated) {
                this.router.navigate(['/home']);
            }
            return isAuthenticated;
    }
}
```

Ova provjera se definira kod definiranja ruta, te preusmjerava korisnika na stranicu za prijavu ukoliko korisnik nije autentificiran. Primjer ograničavanja rute:

```
{ path: 'authorized', component:BaseContainerComponent, canActivate:
[AuthGuard]},
```

Proces autorizacije implementiran je korištenjem Proxy uzorka dizajna. Svaki kontroler za određenu krajnju točku može imati atribut koji sadrži definirane vrste korisnika koji smiju pristupiti. Primjer AuthorizeRoles atributa:

```
[AuthorizeRoles(UserTypeEnum.ManagerAndAdmin)]
[HttpGet]
[Route("")]
public async Task<ActionResult> Method(){}
```

Prije nego što kontroler počne sa obradom zahtjeva, poziva se međuprogram (eng. Middleware).

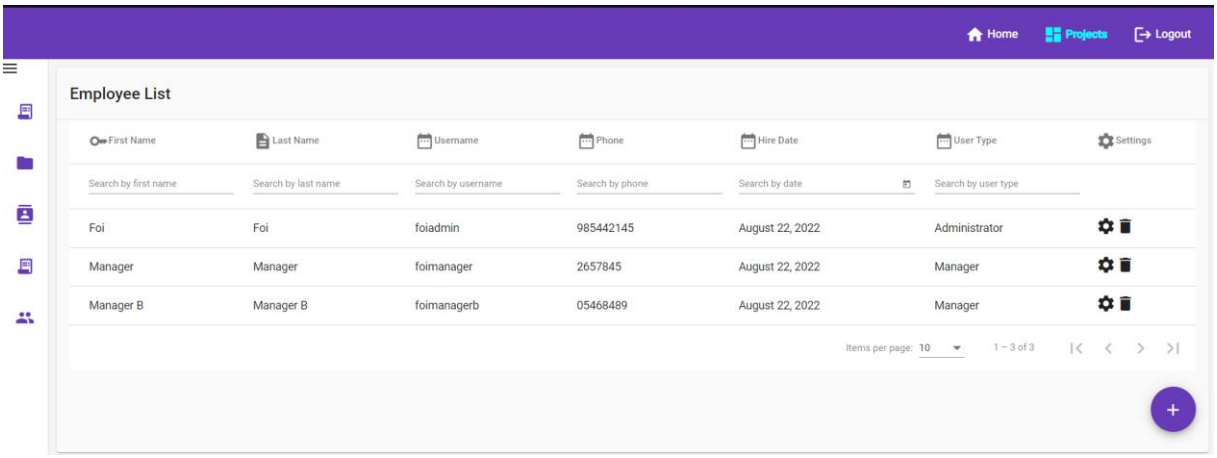
```
public class RequestMiddleware
{
    private readonly RequestDelegate next;
    public RequestMiddleware(RequestDelegate _next)
    {
        next = _next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        var secureProxy = new SecureProxy(new Subject());
        UserTypeEnum? userType = null;
        if (!context.Request.Path.ToString().Contains("/login"))
        {
            userType = (UserTypeEnum)Enum.Parse(typeof(UserTypeEnum),
context.User.Claims.FirstOrDefault(x => x.Type ==
ClaimTypes.Role)?.Value);
        }
         await secureProxy.ProcessRequest(userType, next, context);
    }
}
```

Ovaj međuprogram za svaki zahtjev kreira proxy objekt, koji provjerava ima li zahtjev u vrijednostima tokena onu korisničku vrstu koja je potrebna za izvršavanje danog zahtjeva. Ukoliko nema, proxy kreira Forbidden odgovor. Ukoliko ima, proxy dopušta subjektu da nastavi sa obradom zahtjeva.

```
public class SecureProxy : ISubject
{
    private readonly ISubject subject;
    public SecureProxy(ISubject _subject)
    {
        subject = _subject;
    }
    public async Task ProcessRequest(UserTypeEnum? userType,
   RequestDelegate nextRequest, HttpContext context)
    {
        bool isAuthorized = true;
        var controllerActionDescriptor = context
                  .GetEndpoint()
                  .Metadata
                  .GetMetadata<ControllerActionDescriptor>();

        if (controllerActionDescriptor.EndpointMetadata.Any(x => x is
AuthorizeRoles))
        {
            var authorizeAttribute =
   (AuthorizeRoles)controllerActionDescriptor.EndpointMetadata.FirstOrDefau
   lt(x => x is AuthorizeRoles);
            var role = (UserTypeEnum)Enum.Parse(typeof(UserTypeEnum),
   authorizeAttribute.Role);
            if (!role.HasFlag(userType))
            {
                isAuthorized = false;
                await ReturnErrorResponse(context);
            }
        }
        if(isAuthorized)
        {
            await subject.ProcessRequest(userType, nextRequest, context);
        }
    }
    private async Task ReturnErrorResponse(HttpContext context)
    {
        context.Response.ContentType = "application/json";
        context.Response.StatusCode = (int)HttpStatusCode.Forbidden;
        context.Response.WriteAsJsonAsync(new { message = "Forbidden" });
        await context.Response.StartAsync();
    }
}
```

7.12. Tablični prikazi

Klijentska aplikacija ukupno sadrži pet tabličnih prikaza. Prikaz projekata, prikaz korisnika, prikaz partnera, prikaz izvedbenih stavki te dnevnik projekta. Svaki od ovih prikaza implementira straničenje, te ima mogućnost filtriranja i sortiranja po stupcima.



Slika 32: Tablični prikaz korisnika [autorski rad]

```
<mat-card class="mat-card-custom-color">
    <h2>Employee List</h2>
    <mat-divider></mat-divider>
    <table
        mat-table
        matSort
        (matSortChange)="sortChanged($event)"
        [dataSource]="dataSource" style="width: 100%;"
        >
        <ng-container matColumnDef="firstName">
            <th mat-header-cell *matHeaderCellDef mat-sort-header
class="table-header"><mat-icon>key</mat-icon>First Name</th>
            <td mat-cell *matCellDef="let element">
{{element.firstName}} </td>
        </ng-container>
        <ng-container matColumnDef="lastName">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>description</mat-icon>Last Name</th>
            <td mat-cell *matCellDef="let element"> {{element.lastName}}
</td>
        </ng-container>
        <ng-container matColumnDef="username">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>date_range</mat-icon>Username</th>
            <td mat-cell *matCellDef="let element">
{{element.username}}</td>
        </ng-container>
        <ng-container matColumnDef="phone">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>date_range</mat-icon>Phone</th>
            <td mat-cell *matCellDef="let element">
{{element.phone}}</td>
        </ng-container>
        <ng-container matColumnDef="hireDate">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>date_range</mat-icon>Hire Date</th>
            <td mat-cell *matCellDef="let element"> {{element.hireDate |
date:'longDate'}}</td>
        </ng-container>
        <ng-container matColumnDef="userType">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>date_range</mat-icon>User Type</th>
```

```
            <td mat-cell *matCellDef="let element">
{{typeEnum[element.userType]}}</td>
        </ng-container>

        <ng-container matColumnDef="settings">
            <th mat-header-cell *matHeaderCellDef mat-sort-header><mat-
icon>settings</mat-icon>Settings</th>
            <td mat-cell *matCellDef="let element">
                <mat-icon (click)="OpenAddEditItemDialog(element)"
matTooltip="Edit item">settings</mat-icon>
                <mat-icon (click)="deleteItem(element.userId)"
matTooltip="Edit item">delete</mat-icon></td>
        </ng-container>

        <ng-container matColumnDef="firstName-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by first name</mat-label>
                    <input matInput (keyup)="keyup($event,
userListFilterColumnsEnum.FirstName)">
                </mat-form-field>
            </th>
         </ng-container>

         <ng-container matColumnDef="lastName-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by last name</mat-label>
                    <input matInput (keyup)="keyup($event,
userListFilterColumnsEnum.LastName)">
                </mat-form-field>
            </th>
         </ng-container>
         <ng-container matColumnDef="username-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by username</mat-label>
                    <input matInput (keyup)="keyup($event,
userListFilterColumnsEnum.Username)">
                </mat-form-field>
            </th>
         </ng-container>
         <ng-container matColumnDef="phone-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by phone</mat-label>
                    <input matInput (keyup)="keyup($event,
userListFilterColumnsEnum.Phone)">
                </mat-form-field>
            </th>
         </ng-container>
         <ng-container matColumnDef="hireDate-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by date</mat-label>
                    <mat-date-range-input [rangePicker]="picker">
                        <input matStartDate placeholder="Start date"
                            (dateChange)="dateChange($event,
userListFilterColumnsEnum.HireDateStart)">
                        <input matEndDate placeholder="End date"
```

```
                            (dateChange)="dateChange($event,
userListFilterColumnsEnum.HireDateEnd)">
                    </mat-date-range-input>
                    <mat-datepicker-toggle matSuffix
[for]="picker"></mat-datepicker-toggle>
                    <mat-date-range-picker #picker></mat-date-range-
picker>
                </mat-form-field>
            </th>
         </ng-container>
         <ng-container matColumnDef="userType-filter">
            <th mat-header-cell *matHeaderCellDef>
                <mat-form-field>
                    <mat-label>Search by user type</mat-label>
                    <input matInput (keyup)="keyup($event,
userListFilterColumnsEnum.UserType)">
                </mat-form-field>
            </th>
         </ng-container>
         <ng-container matColumnDef="settings-placeholder">
            <th mat-header-cell *matHeaderCellDef>
            </th>
         </ng-container>
        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
        <tr mat-header-row
*matHeaderRowDef="displayedColumnFilters"></tr>
        <tr mat-row *matRowDef="let row; columns:
displayedColumns;"></tr>
    </table>
    <mat-paginator [pageSizeOptions]="[10, 20, 30]" showFirstLastButtons
(page)="pageChanged($event)"></mat-paginator>
    <mat-spinner *ngIf="isLoading"></mat-spinner>
    <div class="fab-container" *ngIf="userRole ===
typeEnum.Administrator">
        <button mat-fab class="mat-fab-button" color="primary"><mat-
icon  matTooltip="Add Item" (click)="OpenAddEditItemDialog()">add</mat-
icon></button>
    </div>
</mat-card>
```

Svaka tablica definira svoja zaglavlja, te svoje filtere. Primjer definiranja za tablicu korisnika:

```
   displayedColumns: string[] = [
           "firstName",
           "lastName",
           "username",
           "phone",
           "hireDate",
           "userType",
           "settings"
       ];

       displayedColumnFilters: string[] = [
           "firstName-filter",
           "lastName-filter",
           "username-filter",
           "phone-filter",
           "hireDate-filter",
           "userType-filter",
           "settings-placeholder"
       ];

       filters: BaseFilter[] = [
           {columnName: UserListFilterEnum.FirstName, filterQuery: ""},
           {columnName: UserListFilterEnum.LastName, filterQuery: ""},
           {columnName: UserListFilterEnum.Username, filterQuery: ""},
           {columnName: UserListFilterEnum.Phone, filterQuery: ""},
           {columnName: UserListFilterEnum.HireDateStart, filterQuery: ""},
           {columnName: UserListFilterEnum.HireDateEnd, filterQuery: ""},
           {columnName: UserListFilterEnum.UserType, filterQuery: ""},
       ];
```

Svaki tablični prikaz prilikom prvog učitavanja inicijalizira tablične atribute, te šalje zahtjev za dobivanjem podataka prema serverskoj aplikaciji pozivanjem odgovarajućeg injektiranog servisa za slanje zahtjeva.

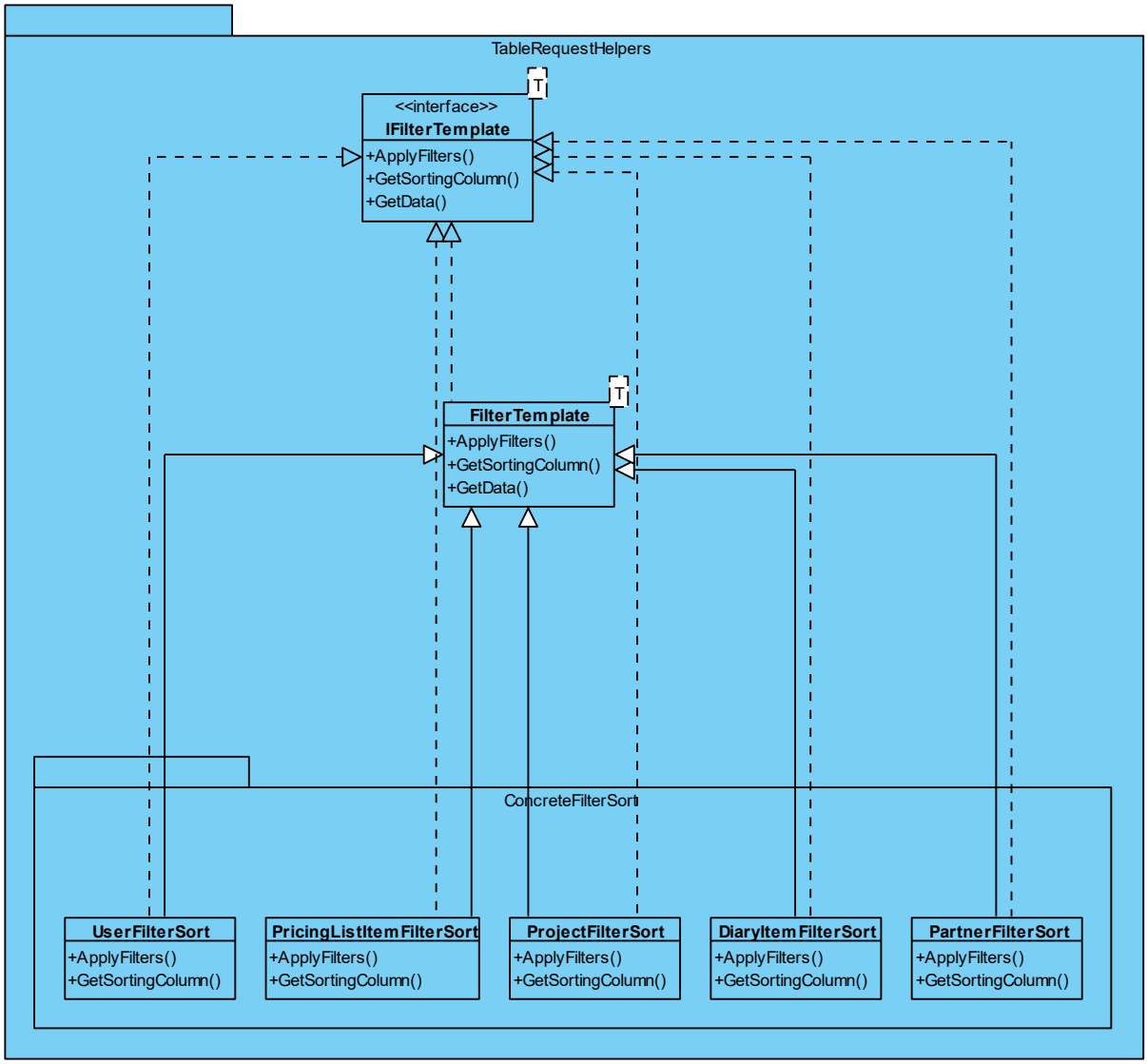
```
ngOnInit(): void {
    this.userRole = this.accountService.userValue?.userType;
    this.initializeTable();
    this.loadData();
}

ngAfterViewInit() {
    this.dataSource.paginator = this.paginator;
    this.dataSource.sort = this.sort;
}
initializeTable() {
    this.pageSize = 10;
    this.currentPage = 0;
    this.sortByColumn = this.defaultOrderColumn;
    this.isAscending = true;
}
loadData() {
    this.isLoading = true;
    let tableRequest: TableRequest = {
        currentPage: this.currentPage,
        pageSize: this.pageSize,
        orderBy: this.sortByColumn,
        isAscending: this.isAscending,
        filters: this.filters
    }
```

```
   this.employerService.getUsersTable(tableRequest).subscribe((response:
   TableResponse) => {
       this.dataSource.data = response.data;
       setTimeout(() => {
           this.paginator.pageIndex = this.currentPage;
           this.paginator.length = response.totalRows;
       });
      this.isLoading = false;
    })
}
```

Odgovarajući kontroler na serverskoj aplikaciji prima zahtjev, te poziva potreban servis za obradu zahtjeva.

```
[AuthorizeRoles(UserTypeEnum.Administrator)]
[HttpPost]
[Route("users")]
public async Task<ActionResult<TableResponse>> GetEmployedUsers([FromBody]
TableRequest tableRequest)
{
   var items = userService.GetEmployedUsers(tableRequest);
   return Ok(items);
}
```

Pošto postoji više sličnih tablica, te svaka od njih šalje vrlo sline zahtjeve, obrada zahtjeva za dohvat tabličnih podataka implementirana je korištenjem kombinacije Factory Method i Template Method uzoraka dizajna.



Slika 33: Template Method [autorski rad]

```
public TableResponse GetEmployedUsers(TableRequest tableRequest)
{
    TableResponse tableResponse = new TableResponse();
    var query = unitOfWork.userRepository.FindAll().Where(x =>
   x.Employer.EmployerId ==
   int.Parse(httpContextAccessor.HttpContext.User.FindFirst("EmployerID").V
   alue) && x.UserType != UserTypeEnum.Buyer);
    tableResponse.TotalRows = query.Count();

    IFilterTemplate<User> filterTemplate =
   FilterFactory<User>.CreateSortingObject();
    tableResponse.Data = filterTemplate.GetData(query, tableRequest);
    return tableResponse;
}
```

Factory Method uzorak dizajna ovisno o tipu tablice kreira objekte za obradu straničenja, filtriranja i sortiranja dobivenih podataka.

```
public static class FilterFactory<T> where T : class
{
```

```
    public static IFilterTemplate<T> CreateSortingObject()
    {
        string dataSourceName = typeof(T).Name;
        switch (dataSourceName)
        {
             case nameof(Project):
                 return (IFilterTemplate<T>) new ProjectFilterSort();
             case nameof(PricingListItem):
                 return (IFilterTemplate<T>) new
   PricingListItemFilterSort();
             case nameof(DiaryItem):
                  return (IFilterTemplate<T>)new DiaryItemFilterSort();
             case nameof(Partner):
                  return (IFilterTemplate<T>)new PartnerFilterSort();
             case nameof(User):
                  return (IFilterTemplate<T>)new UserFilterSort();
             default:
                   throw new ArgumentException();
        }
   }
}
```

Međutim, kako je za sve tablice potrebno straničenje, filtriranje i sortiranje, Template Method uzorak dizajna definira kostur metode za implementaciju tih funkcionalnosti. Straničenje je definirano direktno unutar predloška, dok su filtriranje i sortiranje ostavljene za implementatore.

```
public abstract class FilterTemplate<T> : IFilterTemplate<T> where T :
class
{
    public abstract IQueryable<T> ApplyFilters(IQueryable<T> query,
BaseFilter filter);
    public abstract object GetSortingColumn(T x, string orderBy);

    public List<T> GetData(IQueryable<T> query, TableRequest tableRequest)
    {
        foreach (var filter in tableRequest.Filters.Where(x =>
   !string.IsNullOrEmpty(x.FilterQuery)))
        {
            query = ApplyFilters(query, filter);
        }
   return query
       .SortBy(tableRequest.IsAscending, x => GetSortingColumn(x,
   tableRequest.OrderBy))
        .Skip(tableRequest.PageSize * tableRequest.CurrentPage)
        .Take(tableRequest.PageSize)
        .ToList();
    }
}
public class UserFilterSort : FilterTemplate<User>, IFilterTemplate<User>
{
    public override IQueryable<User> ApplyFilters(IQueryable<User> query,
BaseFilter filter)
    {
        switch ((UserFilterEnum)filter.ColumnName)
        {
            case UserFilterEnum.FirstName:
                query = query.Where(x =>
   x.FirstName.Contains(filter.FilterQuery));
                break;
            case UserFilterEnum.LastName:
```

```
                query = query.Where(x =>
   x.LastName.Contains(filter.FilterQuery));
                break;
            case UserFilterEnum.Username:
                query = query.Where(x =>
x.Username.Contains(filter.FilterQuery));
                break;
            case UserFilterEnum.HireDateStart:
                var dateFrom =
   DateTimeOffset.Parse(filter.FilterQuery).UtcDateTime;
                query = query.Where(x => x.HireDate >= dateFrom);
                break;
            case UserFilterEnum.HireDateEnd:
                var dateTo =
   DateTimeOffset.Parse(filter.FilterQuery).UtcDateTime.AddDays(1);
                query = query.Where(x => x.HireDate <= dateTo);
                break;
            case UserFilterEnum.UserType:
                query = query.Where(x =>
   x.UserType.ToString().Contains(filter.FilterQuery));
                break;
            case UserFilterEnum.Phone:
                query = query.Where(x =>
   x.Phone.ToString().Contains(filter.FilterQuery));
                break;
            default:
                break;
             }
        return query;
        }

    public override object GetSortingColumn(User x, string orderBy)
    {
         switch (orderBy)
         {
              case "firstName":
                  return x.FirstName;
              case "LastName":
                  return x.LastName;
              case "username":
                  return x.Username;
              case "hireDate":
                  return x.HireDate;
              case "userType":
                  return x.UserType;
              case "phone":
                 return x.Phone;
             default:
                  return x.FirstName;
        }
    }
}
```

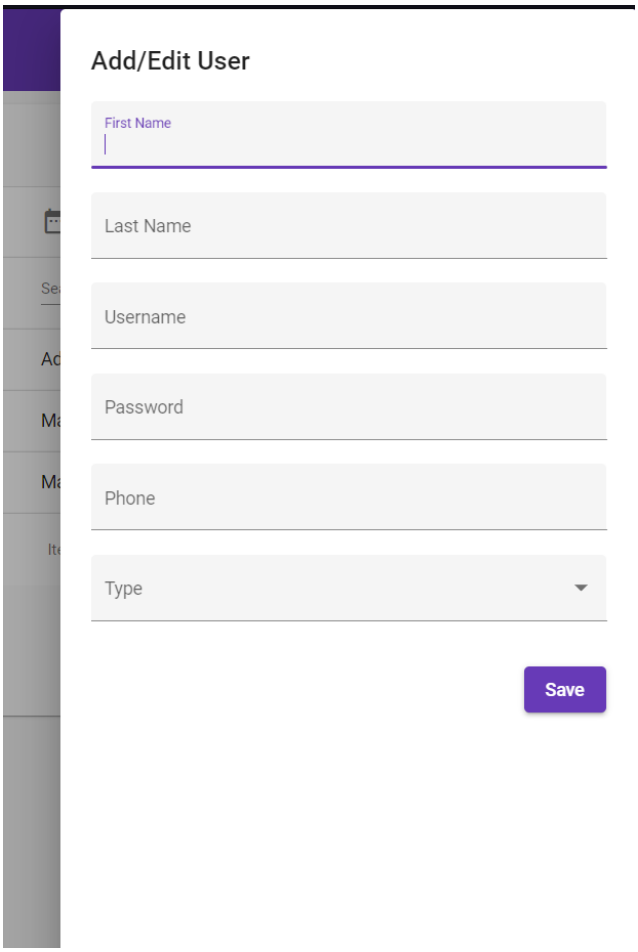
Kod svake promjene tabličnih atributa, bilo to unos filtera, promjena stranice ili sortiranje, klijentska aplikacija šalje ponovan zahtjev za dohvat tabličnih podataka.

```
pageChanged(event: PageEvent) {
    this.pageSize = event.pageSize;
    this.currentPage = event.pageIndex;
    this.loadData();
}
```

```
sortChanged(sortState: Sort) {
    if (sortState.direction) {
        this.sortByColumn = sortState.active;
        this.isAscending = sortState.direction === "asc" ? true : false;
    } else {
        this.sortByColumn = this.defaultOrderColumn;
        this.isAscending = true;
    }
    this.loadData();
}
keyup(event: KeyboardEvent, columnName: UserListFilterEnum) {
    let filter = this.filters.find(f => f.columnName === columnName);
    if (filter) {
        filter.filterQuery = (event.target as HTMLInputElement).value;
    }
    this.loadData();
}
dateChange(event: any, columnName: UserListFilterEnum){
    let filter = this.filters.find(f => f.columnName === columnName);
    if (filter) {
        filter.filterQuery = (event.target as HTMLInputElement).value !==
   null ? new Date((event.target as HTMLInputElement).value).toUTCString()
   : "";
        this.loadData();
    }
}
```

7.13. Unos/ažuriranje zapisa

Klikom na plus ikonicu na dnu tablice, ili klikom na ikonicu za postavke u jednom retku tablice otvara se skočni prozor na desnoj strani ekrana. Ovisno o lokaciji sa koje je pozvan, skočni prozor može biti prazan, za potrebe dodavanja novog zapisa, ili ispunjen podacima, za potrebe uređivanja postojećeg zapisa.



Slika 34: Skočni prozor za unos/ažuriranje zapisa [autorski rad]

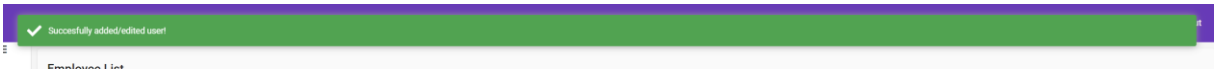
```
<h2 mat-dialog-title>Add/Edit User</h2>
<mat-dialog-content
    [formGroup]="form">
    <mat-form-field appearance="fill">
        <mat-label>First Name</mat-label>
        <input
            matInput
            formControlName="firstName"
        >
    </mat-form-field>
    <mat-form-field appearance="fill">
        <mat-label>Last Name</mat-label>
        <input matInput formControlName="lastName">
    </mat-form-field>

    <mat-form-field appearance="fill">
        <mat-label>Username</mat-label>
        <input matInput formControlName="username">
    </mat-form-field>
    <mat-form-field appearance="fill">
        <mat-label>Password</mat-label>
        <input matInput formControlName="password" type="password">
    </mat-form-field>
    <mat-form-field appearance="fill">
        <mat-label>Phone</mat-label>
```

```
            <input matInput formControlName="phone">
        </mat-form-field>
        <mat-form-field appearance="fill" *ngIf="!data.isBuyerAccount">
            <mat-label>Type</mat-label>
            <mat-select formControlName="userType">
                <mat-option *ngFor="let item of options"
[value]="item">{{typeEnum[item]}}</mat-option>
            </mat-select>
        </mat-form-field>
    </mat-dialog-content>
    <mat-dialog-actions>
        <button style="margin-left: auto;" class="mat-raised-button mat-
primary" (click)="saveChanges()">Save</button>
    </mat-dialog-actions>
```

Prilikom otvaranja komponente, injektiraju se potrebni servisi, kao i opcionalni podaci, zavisno radi li se o akciji kreiranja i brisanja.

```
    constructor(
            @Optional() @Inject(MAT_DIALOG_DATA) public data: {user: User,
isBuyerAccount: boolean},
            private dialogRef: MatDialogRef<AddEditUserComponent>,
            private formBuilder: FormBuilder,
            private employerService: EmployerService,
            private toastr: ToastrService
        ) { }

        ngOnInit(): void {
            if(this.data){
                this.form = this.formBuilder.group({
                    firstName: [this.data.user.firstName],
                    lastName: [this.data.user.lastName],
                    username: [this.data.user.username],
                    password: [this.data.user.password],
                    phone: [this.data.user.phone],
                    userType: [this.data.user.userType],
                });
            }
        }
```

Ukoliko je korisnik zadovoljan unosom, komponenta prema podacima iz forme kreira odgovarajući objekt te poziva pripadajući injektirani servis, koji šalje POST metodu za kreaciju/ažuriranje zapisa u bazi. Ukoliko je zapis uspješno dodan/ažuriran, pojavljuje se poruka uspješnosti na ekranu. Ukoliko nije, pojavljuje se poruka greške.



Slika 35: Poruka uspješnosti [autorski rad]

```
    saveChanges(){
            let user: User = {
                userId: this.data.user === undefined ? -1 :
this.data.user.userId,
                firstName: this.form.get("firstName")?.value,
                lastName: this.form.get("lastName")?.value,
                username: this.form.get("username")?.value,
                password: this.form.get("password")?.value,
```

```
            phone: this.form.get("phone")?.value,
            userType: this.data.isBuyerAccount === true ?
   UserTypeEnum.Buyer : this.form.get("userType")?.value,
            hireDate: new Date(),
            isActive: true
        }
        this.employerService.createOrUpdateUser(user).subscribe({
            next: (res: any) =>{
                this.toastr.success("Succesfully added/edited user!",
   "",{
                    positionClass: 'toast-top-full-width'
                });
                this.dialogRef.close(res)
            },
            error: err => {
                this.toastr.error(err.error, "",{
                    positionClass: 'toast-top-full-width'
                })
            },
        })
    }
```

Serverska aplikacija prima zahtjev, kontroler poziva odgovarajući injektirani servis te se kreira/ažurira zapis. Ukoliko je došlo do pogreške promjene se ne spremaju u bazu podataka.

```
[AuthorizeRoles(UserTypeEnum.Administrator)]
[HttpPost]
[Route("user")]
public async Task<ActionResult<User>> AddEditUser([FromBody] User user)
{
    var result = userService.AddEditUser(user);
    if(result != null)
    {
        return Ok(result);
    }
    else
    {
        return BadRequest("Username already exists!");
    }
}

public User AddEditUser(User user)
{
     bool error = false;
     if(user.UserId == -1)
     {
         if (unitOfWork.userRepository.FindAll().Any(x => x.Username ==
   user.Username))
         {
             error = true;
         }
         else
         {
             Employer employer =
   unitOfWork.employerRepository.Read(int.Parse(httpContextAccessor.HttpCon
   text.User.FindFirst("EmployerID").Value));
             user = new User(user.FirstName, user.LastName, user.Username,
   user.Password, user.Phone, DateTime.UtcNow, user.UserType, employer,
   true);
             unitOfWork.userRepository.Create(user);
         }
```

```
     }
     else
     {
         var userToUpdate = unitOfWork.userRepository.Read(user.UserId);
         if(userToUpdate.Username != user.Username)
         {
            if(unitOfWork.userRepository.FindAll().Any(x => x.Username ==
user.Username))
             {
                 error = true;
             }
             else
             {
                 userToUpdate.FirstName = user.FirstName;
                 userToUpdate.LastName = user.LastName;
                 userToUpdate.Username = user.Username;
                 userToUpdate.Password = user.Password;
                 userToUpdate.Phone = user.Phone;
                 userToUpdate.HireDate = user.HireDate;
                 userToUpdate.UserType = user.UserType;
                 user = userToUpdate;
                 unitOfWork.userRepository.Update(user);
              }
         }
    }
   if(!error)
    {
         if(unitOfWork.Complete() > 0)
         {
             return user;
         }
    }
    return null;
}
```

Na sličan način implementira se dodavanje novog zapisa za svaki entitet, osim za dodavanje novog troška, koji za svoju kreaciju koristi Builder uzorak dizajna, iz razloga što trošak sadrži neke atribute koji nisu uvijek potrebni za njegovu kreaciju.

```
public class ExpenseBuilder : IExpenseBuilder
    {
        private Expense _expense = new Expense();
        public ExpenseBuilder(float cost, ExpenseTypeEnum expenseType)
        {
            _expense.TotalCost = cost;
            _expense.ExpenseType = expenseType;
        }

        public static ExpenseBuilder Init(float cost, ExpenseTypeEnum
expenseType)
        {
            return new ExpenseBuilder(cost, expenseType);
        }

        public Expense Build() => _expense;

        public ExpenseBuilder SetPartner(Partner partner)
        {
            _expense.Partner = partner;
            return this;
```

```
        }

        public ExpenseBuilder SetQuantity(float quantity)
        {
            _expense.Quantity = quantity;
            return this;
        }

        public ExpenseBuilder SetProjectTaskItem(ProjectTaskItem
projectTaskItem)
        {
            _expense.ProjectTaskItem = projectTaskItem;
            return this;
        }
    }
public Expense AddExpense(Expense expense, int taskItemId, int?
pricingListItemId, int? partnerId)
    {
        var item = unitOfWork.projectTaskItemRepository.Read(taskItemId);
        if(expense.ExpenseType == Shared.Enums.ExpenseTypeEnum.Standard)
        {
            var cost =
unitOfWork.pricingListRepository.Read(pricingListItemId).Price;
            expense.TotalCost = (float)(cost * expense.Quantity);
        }
        var expenseToAdd = new ExpenseBuilder((float)expense.TotalCost,
expense.ExpenseType);
        if(expense.ExpenseType == Shared.Enums.ExpenseTypeEnum.Partner)
        {
            var partner = unitOfWork.partnerRepository.Read(partnerId);
            expenseToAdd.SetPartner(partner);
        }
        if(expense.ExpenseType == Shared.Enums.ExpenseTypeEnum.Standard)
        {
            expenseToAdd.SetQuantity((float)expense.Quantity);
        }
        result = expenseToAdd.Build();
        result.ProjectTaskItem = item;
        unitOfWork.expenseRepository.Create(result);
        if (unitOfWork.Complete() == 1)
        {
            return result;
        }
        throw new Exception();
 }
```

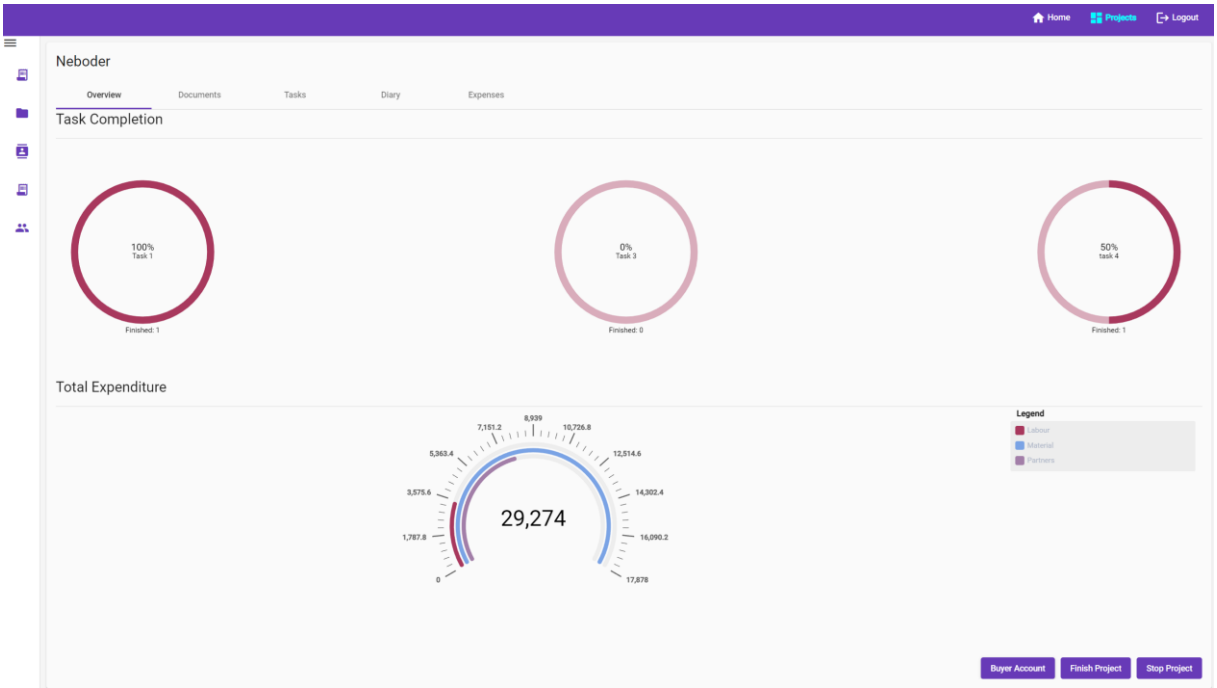
7.14. Projektni detalji

Klikom na gumb za pregled detalja o projektu u tabličnom prikazu projekata otvara se novi prikaz, koji se sastoji od ukupno pet dijelova: opći pregled, dokumenti, zadaci, dnevnik i troškovi. Ovi dijelovi podijeljeni su u kartice (eng. tabs).

```
   <mat-card class="mat-card-custom-color">
       <h1>{{project.name}}</h1>
       <mat-tab-group>
           <mat-tab label="Overview">
               <ng-template matTabContent>
```

```
                <app-project-overview [(project)]="project"></app-
project-overview>
            </ng-template>
        </mat-tab>
        <mat-tab label="Documents">
            <ng-template matTabContent>
                <app-documents [(project)]="project"></app-documents>
            </ng-template>
        </mat-tab>
        <mat-tab label="Tasks">
            <ng-template matTabContent>
                <app-project-tasks [(project)]="project"></app-project-
tasks>
            </ng-template>
        </mat-tab>
        <mat-tab label="Diary">
            <ng-template matTabContent>
                <app-diary [(project)]="project"></app-diary>
            </ng-template>
        </mat-tab>
        <mat-tab label="Expenses">
            <ng-template matTabContent>
                <app-expenses [(project)]="project"></app-expenses>
            </ng-template>
        </mat-tab>
    </mat-tab-group>
</mat-card>
```

7.14.1. Opći pregled



Slika 36: Opći pregled [autorski rad]

Opći pregled sadrži vizualne prikaze završenosti kategorija zadataka, u obliku postotka završenosti te broja završenih zadataka jedne kategorije zadataka. Također se nalazi i vizualni

prikaz raspoređenosti vrste troškova, kao i ukupan trošak projekta. Vrste troškova mogu biti trošak rada, materijal ili partnerski trošak. Kupac na ovom prikazu može urediti svoj korisnički račun, a voditelji projekata i administratori, uz ovu funkcionalnost, mogu i mijenjati status završenosti projekta.

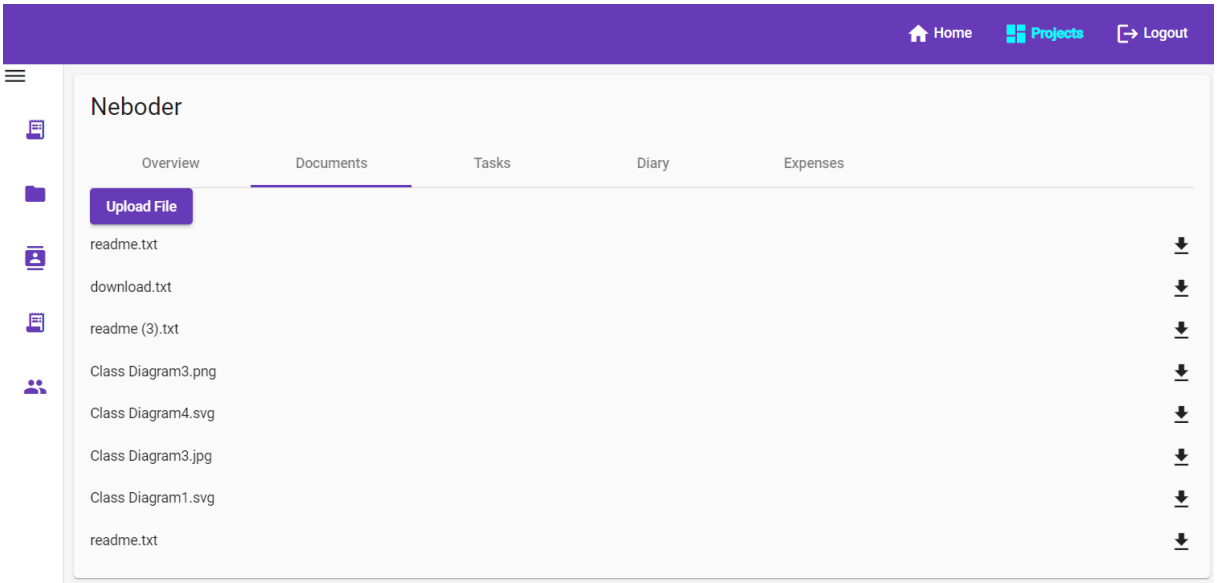
```
<h1>Task Completion</h1>
<div style="display: flex; flex-wrap: wrap; justify-content: space-
between;">
    <mat-divider></mat-divider>
    <div #containerRef *ngFor="let item of graphData.taskCompletions">
        <ngx-charts-pie-grid
        [view]="[containerRef.offsetWidth, 400]"
        [results]="item.pieGrid"
        [designatedTotal]="item.designatedTotal"
        label="Finished">
        </ngx-charts-pie-grid>
    </div>
</div>
<h1>Total Expenditure</h1>
<div style="display: flex; min-height: 400px;">
    <mat-divider></mat-divider>
    <ngx-charts-gauge
    [results]="graphData.gaugeData"
    [legend]="legend"
    [legendPosition]="legendPosition"
    >
</ngx-charts-gauge>
</div>
<div style="display: flex; padding-top: 20px; margin-left: auto; gap:
10px;">
    <button mat-raised-button color="primary"
(click)="OpenAddEditItemDialog()">Buyer Account</button>
    <ng-container [ngSwitch]="project.projectStatus" *ngIf="userRole !==
userTypeEnum.Buyer">
        <ng-container *ngSwitchCase="projectStatusEnum['In Progress']">
            <button mat-flat-button class="mat-flat-button"
color="primary"
(click)="updateProjectStatus(projectStatusEnum['Completed'])">Finish
Project</button>
            <button mat-flat-button class="mat-flat-button"
color="primary" (click)="updateProjectStatus(projectStatusEnum['To
Do'])">Stop Project</button>
        </ng-container>
        <ng-container *ngSwitchCase="projectStatusEnum['To Do']">
            <button  mat-flat-button class="mat-flat-button"
color="primary" (click)="updateProjectStatus(projectStatusEnum['In
Progress'])">Start Project</button>
        </ng-container>
        <ng-container *ngSwitchCase="projectStatusEnum['Completed']">
            <button mat-flat-button class="mat-flat-button"
color="primary" (click)="updateProjectStatus(projectStatusEnum['To
Do'])">Stop Project</button>
        </ng-container>
    </ng-container>
</div>

ngOnInit(): void {
        this.userRole = this.accountService.userValue?.userType;
```

```
        this.projectService.getProjectGraphData(this.project.projectId).sub
scribe((res) =>{
            this.graphData = res;
        })
}

getProjectGraphData(projectId: number){
        return this.http.get<GraphData>(this.projectApiUrl + "/graphData/"
+ projectId);
}
```

Serverska aplikacija prima zahtjev te pozivom ProjectService servisa kreira objekte potrebne za implementaciju vizualnih prikaza

```
public GraphData GetGraphData(int projectId)
{
    var taskCompletions =
unitOfWork.projectTaskRepository.GetTaskCompletions(projectId);
    var expenses = unitOfWork.expenseRepository.FindAll().Where(x =>
x.ProjectTaskItem.ProjectTask.Project.ProjectId == projectId);
    var labourExpenses = (float)expenses.Where(x => x.ExpenseType ==
ExpenseTypeEnum.Standard).Select(x => x.TotalCost).Sum();
    var materialExpenses = (float)expenses.Where(x => x.ExpenseType ==
ExpenseTypeEnum.Material).Select(x => x.TotalCost).Sum();
    var partnerExpenses = (float)expenses.Where(x => x.ExpenseType ==
ExpenseTypeEnum.Partner).Select(x => x.TotalCost).Sum();
    List<PieGrid> gaugeData = new List<PieGrid>() { new PieGrid("Labour",
labourExpenses), new PieGrid("Material", materialExpenses), new
PieGrid("Partners", partnerExpenses), };


    return new GraphData(taskCompletions, gaugeData);
}
```

7.14.2. Dokumenti



Slika 37: Dokumenti [autorski rad]

Prikaz dokumenata sadrži listu dokumenata koje se odnose na traženi projekt, te je moguće dodati nove dokumente, ili preuzeti već dodane.

```
<button type="button" mat-raised-button color="primary"
(click)="fileInput.click()">Upload File</button>
<input hidden (change)="onFileSelected($event)"  #fileInput type="file"
id="file">
<div style="display: flex; padding-top: 10px;" *ngFor="let item of
files">
    <p>{{item.fileName}}</p>
    <mat-icon style="margin-left: auto;"
(click)="downloadFile(item.fileId, item.fileName)">download</mat-icon>
</div>
```

Prijenos dokumenata provodi se dodavanjem dokumenata u podatke forme (eng. FormData), nakon čega se šalje zahtjev:

```
uploadFile(fileList: FileList, projectId: number){
    const formData: FormData = new FormData();
    Array.from(fileList).forEach(element => {
        formData.append('files', element, element.name);
    });
    return this.http.post(this.projectApiUrl + "/upload/" + projectId,
formData);
}
```

Serverska aplikacija prima zahtjev, te prenesene dokumente sprema, nakon čega se u bazu zapisuju podaci o prenesenim dokumentima.

```
public int UploadFiles(List<IFormFile> files, int projectId)
{
    var project = unitOfWork.projectRepository.Read(projectId);
    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();
            unitOfWork.fileRepository.Create(new File(formFile.FileName,
filePath, project));
            using (var stream = System.IO.File.Create(filePath))
            {
                formFile.CopyToAsync(stream);
            }
        }
    }
    return unitOfWork.Complete();
}
```

Kod zahtjeva za preuzimanje dokumenta, serverska aplikacija čita podatke o dokumentu iz baze, učitava dokument te ga šalje klijentu.

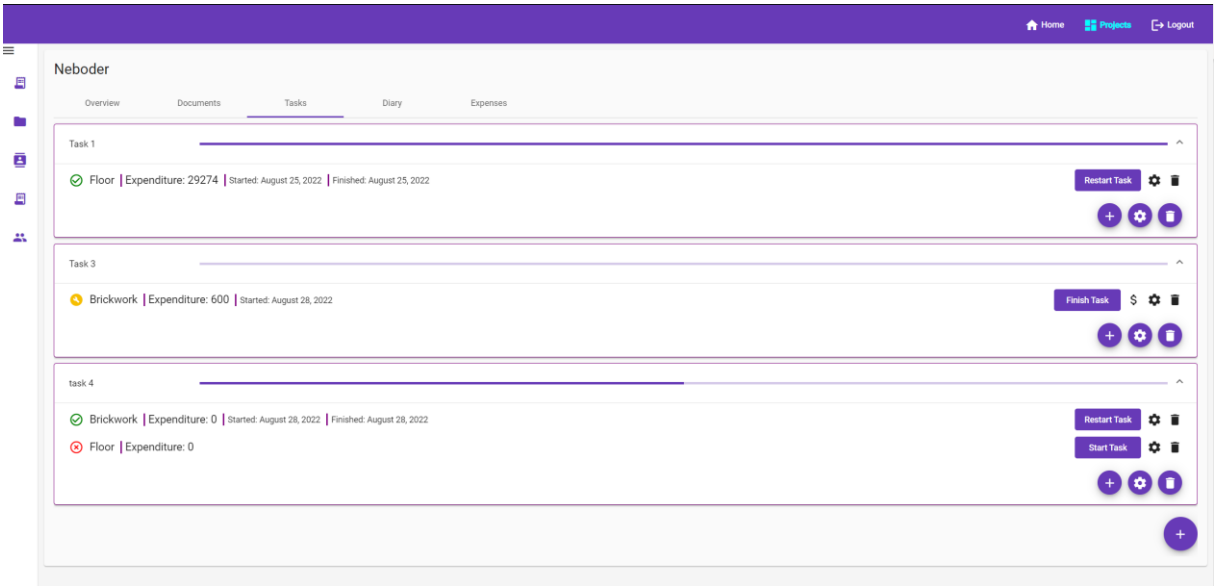
```
[HttpGet]
[Route("download/{id}")]
public async Task<IActionResult> DownloadFile(int id)
{
    var file = fileService.DownloadFile(id);
    if(file != null)
    {
        var bytes = await System.IO.File.ReadAllBytesAsync(file.FilePath);
```

```
        Response.Headers.Add("Content-Disposition", "attachment");
        return File(bytes, "text/plain", file.FileName);
    }
    else
    {
        return BadRequest();
    }
}
```

Klijentska aplikacija prima odgovor te sprema dobiveni dokument u mapu za preuzimanje definiranu prema postavkama preglednika u kojem je pokrenuta.

```
downloadFile(fileId: number, fileName: string){
    this.http.get<any>(this.projectApiUrl + "/download/" + fileId, {
observe: 'response', responseType: 'blob' as 'json' }).subscribe((res) => {
        FileSaver.saveAs(res.body, fileName);
    });
}
```

7.14.3. Projektni zadaci



Slika 38: Projektni zadaci [autorski rad]

Prikaz projektnih zadataka sastoji se od tri komponente: ProjectTasks, ProjectTask i ProjectTaskItem.

ProjectTasks:

```
<div>
    <ng-container *ngFor="let task of projectTasks">
        <app-project-task [task]="task" [project]="project"
(taskChange)="OpenAddEditItemDialog($event)"></app-project-task>
    </ng-container>
    <div class="fab-container" *ngIf="userRole !== userTypeEnum.Buyer">
        <button mat-fab class="mat-fab-button" color="primary"><mat-
icon  matTooltip="Add Task" (click)="OpenAddEditItemDialog()">add</mat-
icon></button>
    </div>
```

```
</div>
```

ProjectTask:

```
<mat-expansion-panel>
    <mat-expansion-panel-header>
        <mat-panel-title style="max-width: 200px;">
            {{task.projectTaskName}}
        </mat-panel-title>
        <mat-panel-description>
            <mat-progress-bar style="width: 100%;" mode="determinate"
[value]="progressPercentage"></mat-progress-bar>
        </mat-panel-description>
    </mat-expansion-panel-header>
    <mat-divider></mat-divider>
    <ng-container matExpansionPanelContent>
        <app-project-task-item *ngFor="let taskItem of
task.projectTaskItems; let i = index"
            [projectTaskItem]="task.projectTaskItems[i]"
            [projectTaskId]="task.projectTaskId"
            [project]="project"
        >
        </app-project-task-item>
    </ng-container>
    <div class="fab-container" *ngIf="userRole !== userTypeEnum.Buyer">
        <button mat-mini-fab class="mat-fab-button"
color="primary"><mat-icon  matTooltip="Add item"
(click)="OpenAddEditItemDialog()">add</mat-icon></button>
        <button mat-mini-fab class="mat-fab-button"
color="primary"><mat-icon  matTooltip="Edit task" >settings</mat-
icon></button>
        <button mat-mini-fab class="mat-fab-button"
color="primary"><mat-icon  matTooltip="Delete task"
(click)="deleteProjectTask()">delete</mat-icon></button>
    </div>
</mat-expansion-panel>
```

ProjectTaskItem:

```
<div class="task-item-container">
    <ng-container [ngSwitch]="projectTaskItem.taskItemStatus">
        <ng-container *ngSwitchCase="projectStatusEnum['In Progress']">
            <mat-icon class="material-icons
color_yellow">build_circle</mat-icon>
        </ng-container>
        <ng-container *ngSwitchCase="projectStatusEnum['To Do']">
            <mat-icon class="material-icons
color_red">highlight_off</mat-icon>
        </ng-container>
        <ng-container *ngSwitchCase="projectStatusEnum['Completed']">
            <mat-icon class="material-icons
color_green">check_circle_outline</mat-icon>
        </ng-container>
    </ng-container>
    <div>
        <a style="font-size:
large;">{{projectTaskItem.pricingListItem.pricingListItemName}}</a>
    </div>
    <div class="content">
        <a style="font-size: large;">Expenditure:
{{projectTaskItem.expenditure}}</a>
    </div>
```

```
    <div class="content" *ngIf="projectTaskItem.taskItemStatus !==
projectStatusEnum['To Do']">
        <a>Started: {{projectTaskItem.startTime | date:'longDate'}}</a>
    </div>
    <div class="content" *ngIf="projectTaskItem.taskItemStatus ===
projectStatusEnum['Completed']">
        <a>Finished: {{projectTaskItem.finishTime |
date:'longDate'}}</a>
    </div>

    <div class="right-content-container">
        <ng-container [ngSwitch]="projectTaskItem.taskItemStatus"
*ngIf="userRole !== userTypeEnum.Buyer">
            <ng-container *ngSwitchCase="projectStatusEnum['In
Progress']">
                <button mat-flat-button class="mat-flat-button"
color="primary" (click)="updateTaskStatus()">Finish Task</button>
                <mat-icon (click)="OpenAddExpenseDialog()"
matTooltip="Add expense">attach_money</mat-icon>
            </ng-container>
            <ng-container *ngSwitchCase="projectStatusEnum['To Do']">
                <button  mat-flat-button class="mat-flat-button"
color="primary" (click)="updateTaskStatus()">Start Task</button>
            </ng-container>
            <ng-container
*ngSwitchCase="projectStatusEnum['Completed']">
                <button mat-flat-button class="mat-flat-button"
color="primary" (click)="updateTaskStatus()">Restart Task</button>
            </ng-container>
        </ng-container>
        <mat-icon *ngIf="userRole !== userTypeEnum.Buyer"
(click)="OpenAddEditItemDialog()" matTooltip="Edit item">settings</mat-
icon>
        <mat-icon *ngIf="userRole !== userTypeEnum.Buyer"
(click)="deleteTask()" matTooltip="Delete item">delete</mat-icon>
    </div>
</div>
```

8. Zaključak

U ovom radu obrađeni su temeljni koncepti arhitekture računalnih programa, te su definirane prednosti, ali i mane svake pojedine arhitekture. Nadalje su opisani web servisi, te njihove temeljne osobine. Uz njih, opisani su i neki od uzoraka dizajna. Kombinacija tih uzoraka, REST web servisa te klijent – server arhitekture računalnih web programa čini teorijsku podlogu izrađenoj aplikaciji u praktičnom dijelu. Također su obrađene i odabrane razvojne tehnologije za izradu praktičnog dijela.

Cilj ovog rada bio je prikazati arhitekturni proces izrade programskog rješenja, te kako kreirati suvremene web servise i njihove klijente. Prikazan je i moderan autentifikacijski proces u obliku JWT tokena. I korisnička i serverska strana za implementaciju svojih funkcionalnosti primjenjuju uzorke dizajna, te su dani čvrsti razlozi za njihovo korištenje. Jasno definirana arhitektura rješenja, te primijenjeni uzorci dizajna zajedno čine čvrst i stabilan temelj, te pružaju vrlo preglednu, skalabilnu i lako nadogradivu aplikaciju.

ASP.NET Core i Entity Framework pokazali su se kao izvrsni razvojni okviri za konstruiranje serverske aplikacije i baze podataka. ASP.NET Core uvelike olakšava razvojni proces jedne serverske aplikacije, te nudi mnoštvo funkcionalnosti za izradu modernog REST API-ja. Entity Framework korištenjem objektno – relacijskog mapiranja omogućio je modeliranje same baze podataka kroz kod, te eliminirao potrebu za pisanjem SQL upita. Na klijentskoj strani, Angular aplikaciji daje moderan izgled te omogućavanjem ponovnog korištenja komponenata ubrzava razvojni proces.

Najveći razlog izbora ove teme bio je dodatno usavršavanje kritičkog razmišljanja o arhitekturi programskog rješenja, te smatram da mi je ovaj rad u tome uvelike pomogao. Uz to, neke tehnologije sam po prvi put koristio te ovaj rad daje odličnu odskočnu dasku za napredak tehničkih vještina. Shvatio sam važnost dizajniranja arhitekture nekog rješenja prije pisanja koda, kako bi krajnji produkt bio što kvalitetniji.

Sav aplikacijski kod izrađenog rješenja dostupan je na sljedećoj poveznici: https://github.com/domagojmahnet/ConcreteNg. Dostupan je svima na pregled, preuzimanje, modifikaciju i korištenje.

Popis literature

[1] A.Bijlsma, B.J. Heeren, E.E. Roubtsova, S. Stuurman, *Software Architecture*, FTA, 2011.

[2] M. Richards, N. Ford, *Fundamentals of Software Architecture*, O'Reilly Media, 2020.

[3] M. Richards, *Software Architecture Patterns*, O'Reilly Media, 2017.

[4] C. de la Torre, B. Wagner, M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*, Microsoft, 2022.

[5] K. T. Tran, *Introduction to Web Services with Java,* Bookboon, 2013.

[6] Web servisi [Slika] (bez dat.) Dostupno: https://media.ttmind.com/Media/tech/article_82_12-10-201812-46-13PM.jpg

[7] J. Higginbotham, *Designing Great Web APIs*, O'Reilly Media, 2015.

[8] E.Ramadani, F. Halili, *Web Services: A Comparison of Soap and Rest Services,* [Na internetu]. Dostupno: Academia.edu, https://doi.org/10.5539/MAS.V12N3P175 [pristupano 30.06.2022.]

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994.

[10] R. Joshi, *Java Design Patterns: Reusable Solutions To Common Problems,* Exelixis Media, 2015.

[11] S. Millett, *Professional ASP.NET Design Patterns*, Wiley Publishing, 2010.

[12] Mozilla, *Lazy loading,* [Na internetu]. Dostupno: Mozilla, https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading [pristupano 27.08.2022.]

[13] M. Mohammed, M. Elish, A. Qusef, *Empirical insight into the context of design patterns: Modularity analysis*, [Na internetu]. Dostupno: IEEE Xplore, https://ieeexplore.ieee.org/document/7549474 [pristupano 13.08.2022.]

[14] M. Yener, A. Theedom, *Professional Java EE Design Patterns*, John Wiley & Sons, Inc., 2015.

[15] Angular, *What is Angular?,* [Na internetu]. Dostupno: Angular, https://angular.io/guide/what-is-angular [pristupano 15.08.2022.]

[16] S. Chiaretta, *Front-end Development with ASP.NET Core, Angular and Bootstrap,* John Wiley & Sons, Inc., 2018.

[17] Pluralsight, *Angular 101: Pros, cons, features and more,* [Na internetu]. Dostupno: Pluralsight, https://www.pluralsight.com/blog/software-development/angular-101 [pristupano 16.08.2022.]

[18] A.Lock, *ASP.NET Core In Action*, Manning Publications, 2018.

[19] Microsoft, *Overview to ASP.NET Core*, [Na internetu]. Dostupno: Microsoft, https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-

6.0, [pristupano 18.08.2022.]
[20] V. P. Mehta, *Pro LINQ Object Relational Mapping with C# 2008*, Apress, 2008.

Popis slika

Popis tablica

Prilozi (1, 2, …)