Razvoj web aplikacije za upravljanje osobnim zadacima

Periša, Tomislav

Master's thesis / Diplomski rad

2023

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:211:531176

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Periša

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE OSOBNIM ZADACIMA

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Periša

Matični broj: 0016131677

Studij: Baze podataka i baze znanja

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE OSOBNIM ZADACIMA

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2023.

Tomislav Periša

Izjava o izvornosti

Izjavljujem da je moj 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

Ovim radom biti će prikazan razvoj moderne web aplikacije koja će prvenstveno služiti autoru za upoznavanje s cjelokupnim procesom razvoja web aplikacije, uključujući frontend i backend dijelove same aplikacije, kao i popratnim aktivnostima koje su nužne za taj proces. Početak rada baviti će se teorijom razvoja web aplikacija, metodologija, tehnologija koje se mogu koristiti kao i arhitekturama na kojima se web aplikacije mogu zasnivati. Na kraju teorijskog dijela rada biti će definirane funkcionalnosti buduće aplikacije koja će se razvijati u praktičnom dijelu rada. Potom će se rad usmjeriti na sam praktičan dio u kojemu će se pratiti razvoj aplikacije od njenog početka (osnovnog kostura) do završetka u kojemu će biti i prikaz finalne verzije aplikacije. Aplikacija će koristiti *ASP.Net Core* i *Angular* razvojne okvire za backend i frontend respektivno. Za rad je odabrana i *MongoDB* NoSQL baza podataka koja će biti korištena uz *MongoDB Driver* i *Compass* (DBMS).

Ključne riječi: asp.net; c#; nosql; angular; typescript; arhitektura;

Sadržaj

1. Uvod

Tema ovog rada je upoznavanje s modernim razvojnim okvirima za razvoj web aplikacija. Web aplikacije su dio svakodnevice modernog čovjeka i uz pomoć istih, mogu se obavljati razno razni zadaci preko interneta. Kako je razvoj interneta ubrzan, tako su se ubrzali i zahtijevi za web aplikacijama za koje su danas postavljeni vrlo visoki standardi. Takve aplikacije su danas vrlo kompleksne i zahtijevaju jako puno znanja i uloženog truda kako bi se efektivno izgradile i koristile.

Kroz ovaj rad, iskoristiti će se prilika za malo dubljim upoznavanjem web aplikacija i alata koji programerima mogu poslužiti kako bi se izgradile moderne web aplikacije. Početak rada će se baviti prvenstveno pojmom web aplikacije i najpopularnijim modernim softverskim arhitekturama. Potom će se pojasniti pojmovi poput Web API servisa i *Angulara* koji će biti korišteni za izradu praktičnog dijela rada. Pred kraj teorijskog dijela, predstaviti će se funkcionalnosti buduće aplikacije.

Kroz praktičan dio rada, cilj je upoznati se s cjelokupnim razvojem klijentske aplikacije koristeći *Angular* te *ASP.NET Core* razvojne okvire uz pomoć kojih će se izgraditi rješenja za klijentsku aplikaciju i Web API servis koji će predstavljati poslužiteljski servis kojemu će klijentska aplikacija slati zahtjeve za izvršavanje određenih zadataka.

Kao tema aplikacije odabran je sustav za upravljanje osobnim podacima po uzoru na Microsoft OneNote. Microsoft OneNote je vrlo korisna aplikacija koju koristim od početka svog studiranja i uz pomoć koje sam organizirao svoje vrijeme. Uočavam veliku važnost takvih aplikacija koje čovjeku mogu jako puno pomoći u današnjem svijetu i u svakodnevici kako bi korisnik takve aplikacije mogao puno lakše organizirati svoje vrijeme i zadatke, te kako bi ih mogao rasporediti i lakše pratiti svoje obveze. Isto tako, OneNote sam koristio i kao digitalnu bilježnicu u koju je moguće zapisivati vlastite bilješke, bez brige o gubitku podataka ili brige o gubitku fizičkih bilježnica. Upravo to su razlozi zbog kojih sam se odlučio izgraditi baš ovakvu aplikaciju.

Angular i *ASP.NET Core* su jedni od najmoćnijih razvojnih okvira današnjice, uz pomoć kojih programeri mogu izgraditi vrlo korisna rješenja s namjerom dugotrajnog korištenja i lakšeg rješavanja kompleksnih problema.

2. Metode i tehnike rada

Teorijski dio rada zasnivati će se na resursima pronađenim na internetu i *Google Scholar* repozitoriju, kako bi se dublje zašlo u temu web aplikacija i razvoja istih. Potom će se u praktičnom dijelu rada fokus prebaciti na izgradnju moderne web aplikacije korištenjem *Angular* (verzija 15) [1] i *ASP.NET Core* [2] razvojnih okvira, čije su odlične dokumentacije korištene kako bi se bolje upoznala priroda rada s tim razvojnim okvirima. Za testiranje pristupnih točaka API servisa korišten je programski alat *Postman* i preddefinirani paket *Swagger* koji je dio *ASP.NET Core* Web API projekta.

Za bazu podataka, odabran je *Mongo DB* NoSQL tip baze podataka koji je korišten zajedno s *Mongo DB Compass* alatom (*Compass* je sustav za upravljanje *Mongo* bazom podataka). Baza podataka je s API servisom spojena koristeći *Mongo DB Driver*; službeni *Mongo DB nuGet* paket za spajanje i mapiranje entiteta .Net projekta u entitete baze podataka. Isto tako korištena je *Mongo DB* dokumentacija [3] specifično pisana za *C#* programski jezik, odnosno *.NET* platformu.

Od dodatnih resursa, korišten je *Udemy* tutorial [4] te brojni tutorijali i video materijali s platforme YouTube. Uz sve to, korišten je i Bootstrap (verzija 5.3) razvojni okvir na klijentskoj aplikaciji, čija je bogata dokumentacija [5] također korištena prilikom stiliziranja html dokumenata različitih komponenti. Na klijentskoj aplikaciji su korištena dva vanjska paketa, FullCalendar [6] i Angular Editor [7], za jednostavnu implementaciju prikaza kalendara i uređivača teksta.

Ikonice korištene u klijentskoj aplikaciji su preuzete sa stranica Flaticon [8] i Icons8 [9].

3. Web aplikacija

Prema stranici StackPath [10], web aplikacija predstavlja računalni program koji koristi web preglednike i web tehnologije kako bi odradio zadatke preko interneta.

Prema dokumentaciji tvrtke Amazon Web Service [11], web aplikacija je bilo kakav softver koji se izvršava putem internetskog preglednika.

Internet je danas postao jedan od najraširenijih komunikacijskih kanala, a u isto vrijeme je i jedan od najjeftinijih za korištenje. To je jedan od razloga zašto su web aplikacije, koje kao svoj primaran komunikacijski kanal koriste internet, postale toliko popularne i raširene.

Amazon [11] kao glavne prednosti ovakvog tipa aplikacija navodi:

- Dostupnost - web aplikacijama se može pristupiti vrlo jednostavno putem velikog raspona različitih korisničkih uređaja i različitih web preglednika
- Efikasan razvoj - proces razvoja web aplikacija je relativno jednostavan i jeftin, pogotovo za veće organizacije koje imaju i velike benefite posjedovanja vlastitih web aplikacija
- Jednostavnost korištenja – web aplikacije nije potrebno preuzeti i instalirati, a o njihovom ažuriranju i održavanju korisnici nemoraju voditi računa jer sam programski kod ne posjeduju na vlastitom računalu
- Skalabilnost - web aplikacije današnjice su većinom implementirane kao rješenje u oblaku, zbog čega organizacije nemoraju ulagati ogromne resurse u povećanje vlastitih računalnih kapaciteta

Web aplikacije se sastoje od dva važna dijela koja cijelu aplikaciju dijele na dvije strane. Prva strana je strana poslužitelja (eng. *Server side*) koja sadrži logiku za obradu podataka, kao i njihovo trajno skladištenje, brisanje, ažuriranje i dohvaćanje. Druga strana je tzv. korisnička strana (eng. *Client side*) koja u većini slučajeva predstavlja korisničku aplikaciju kojoj korisnik ima direktan pristup i koja korisniku omogućava da izvrši neku željenu radnju.

Generalno se pod pojmom web aplikacije misli na korisničku aplikaciju koja je pisana jezicima koje razumiju internetski preglednici i kojoj se može pristupiti preko interneta.

Prema ranije spomenutom izvoru [10], web aplikacije mogu biti statične i dinamičke. Statične web aplikacije su one aplikacije koje ne zahtijevaju nikakvu obradu podataka na strani poslužitelja, a dinamičke su one koje većim ili manjim dijelom zahtijevaju pozadinsku potporu poslužiteljske strane kako bi mogle pravilno funkcionirati.

U uobičajenom slučaju, korisnik preko klijentske aplikacije šalje zahtjev poslužiteljskoj strani. Priroda zahtjeva može biti raznovrsna, od dohvaćanja podataka pa sve do spremanja podataka u bazu ili pak obradu i procesiranje podataka. Jednom kada je zahtjev zaprimljen na strani poslužitelja, zahtjev se obrađuje (izvršava se sva pripadna logika zahtijeva), te se po završetku, rezultat obrade šalje natrag klijentskoj strani.

Kao primjer popularnih web aplikacija današnjice mogu se navesti svima poznate Google aplikacije poput *Gmail*-a, Google dokumenata isl. Nadalje, sve internet trgovine koje korisnici koriste kako bi izvršili kupnju ili neki drugi srodan zahtjev, također spadaju u web aplikacije. Društvene mreže se također mogu uvrstiti u web aplikacije itd.

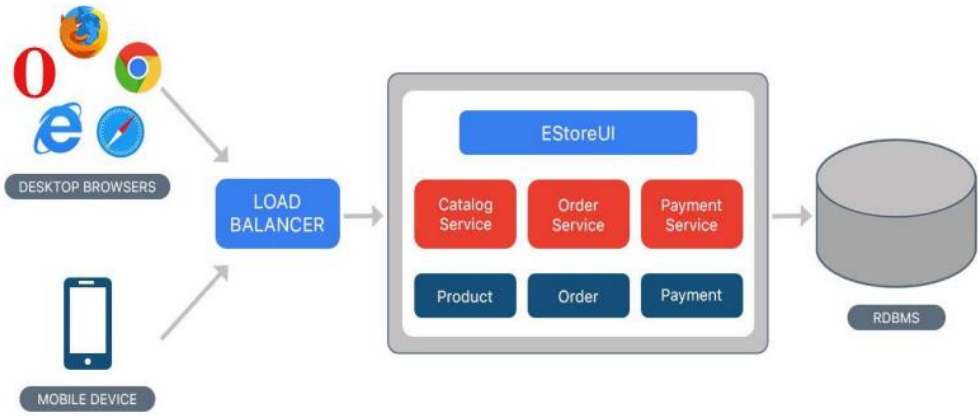
4. Arhitekture web aplikacija

Gos i Zabierowski [12] za softversku arhitekturu pišu da ona predstavlja osnovnu strukturu softvera, koja definira tehničke i operacijske zahtjeve. Arhitektura je zadužena za svaki od glavnih atributa aplikacije, poput efikasnosti, održivosti, skalabilnosti, pouzdanosti, mogućnosti modifikacije, jednostavnosti isporuke itd.

Postoji velik broj arhitektura koje se primjenjuju u današnjem svijetu softvera, no neke od glavnih uključuju monolitnu, mikroservisnu, slojevitu idr.

4.1. Monolitna arhitektura

Prema Gosu i Zabierowski-om [12], monolitna arhitektura je oblik softvera kod kojega su različite komponente aplikacije, poput autorizacije, poslovne logike i sličnih modula, spojene u jedan jedinstveni program baziran na istoj platformi.



Slika 1. Primjer monolitne arhitekture [12]

Na slici iznad moguće je vidjeti prikaz monolitne arhitekture za jednu standardnu web aplikaciju. Korisnici preko web preglednika pristupaju aplikaciji koja se sastoji od nekoliko modula, poput modula za prikaz korisničkog sučelja te pripadnih servisa i modela domene aplikacije. Svi ti moduli su dio jednog te istog programa odnosno platforme.

Prema članku tvrtke Attlasian [13], monolitna arhitektura predstavlja tradicionalan model softverskog programa, koji je građen kao uniformna jedinica neovisna o drugim aplikacijama. Isto tako sam naziv monolit podsjeća na nešto glomazno i veliko, što je uobičajen slučaj kod ovakvih arhitektura. Programi bazirani na ovakvom modelu arhitekture, često

postanu veoma veliki i teški za održavanje. Najveći problem ovakvih arhitektura je što ukoliko dođe do promjene zahtjeva aplikacije, izmjena programskog koda mora se izvršiti na velikom broju modula aplikacije, jer su svi moduli čvrsto vezani, što je ujedno koncept koji se naziva „bliska spojenost“ (eng. *Close coupling*). Unatoč svojim nedostatcima, monolitne arhitekture mogu biti odlično rješenje u ranim fazama projekata, zbog svoje relativno niske kompleksnosti, jednostavnosti upravljanja kodom i jednostavnom odnosno cijelovitom isporukom (monoliti se mogu isporučiti odjednom u svoj svojoj cijelosti).

Prema Attlasian članku [13], glavne prednosti ovakve arhitekture su:

- Jednostavna isporuka - aplikacija se može cijela isporučiti odjednom
- Jednostavan razvoj - postoji samo jedna baza koda pa je razvoj jednostavniji
- Performanse - cijela aplikacija i pozivi API servisa su centralizirani što pozitivno utječe na vrijeme obrade zahtijeva
- Jednostavno testiranje - monoliti su centralizirane jedinice, pa je zbog toga takve aplikacije puno jednostavnije testirati nego više manjih distribuiranih aplikacija
- Jednostavno otklanjanje poteškoća u radu aplikacije - praćenje rada aplikacije je jednostavnije kada je cijeli programski kod na jednom mjestu, jer se zahtjevi mogu pratiti u cijelosti od početka do kraja

Glavni nedostatci su:

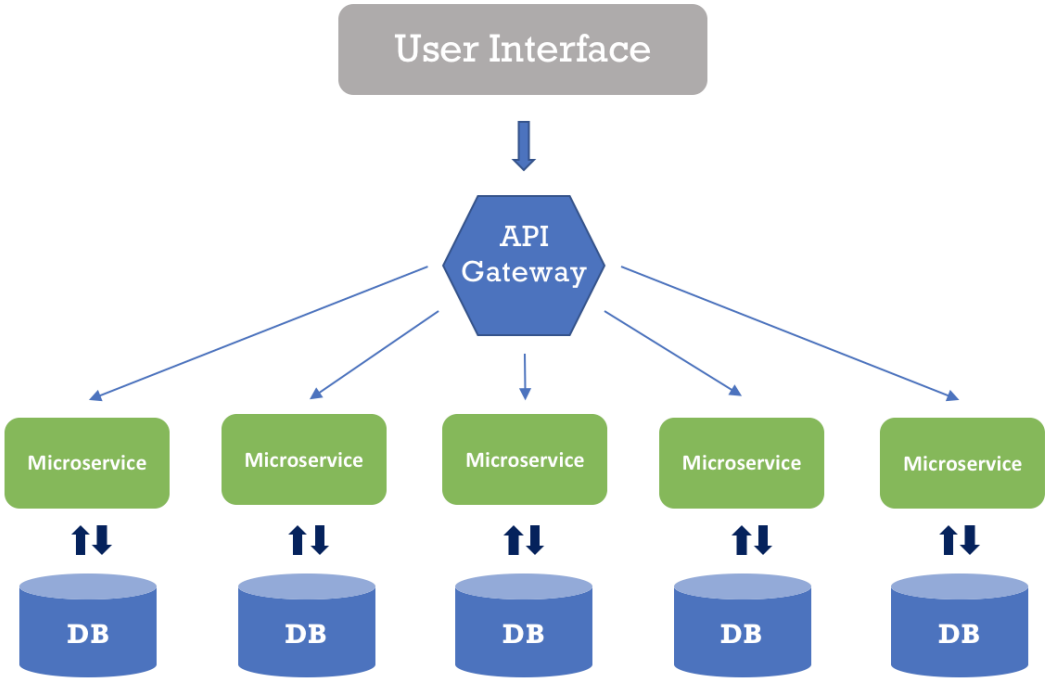
- Spor razvoj – monolit može postati vrlo velik i složen, što razvoj može učiniti dužim
- Skalabilnost – individualne komponente se nemogu odvojeno skalirati jer promjena unutar jedne komponente zahtjeva promjenu i u ostalim povezanim komponentama
- Pouzdanost – greška u jednom dijelu aplikacije može negativno utjecati na ostale povezane dijelove aplikacije
- Teška i skupa promjena tehnologije – kako je monolit baziran na jednoj platformi, promjena tehnologije odnosno platforme zahtjeva prepisivanje čitave aplikacije
- Manjak fleksibilnosti – nedostatak koji se nadovezuje na prethodni gdje je cijeli monolit vezan uz tehnologiju na kojoj je izgrađen
- Isporuka – čak i najmanje promjene u kodu aplikacije zahtjevaju ponovnu isporuku čitave aplikacije što postaje sve neefikasnije kako veličina aplikacije raste

4.2. Mikroservisna arhitektura

Jedan od novijih tipova arhitekture je mikroservisna arhitektura. Mikroservisna arhitektura, kako samo ime kaže, se sastoji od mikro servisa, koji zajedno tvore aplikaciju. Kod ovakve arhitekture, svaki mikro servis je dužan implementirati dio poslovne logike.

Prema već spomenutom članku tvrtke Atlassian [13], mikroservisi sadrže vlastitu poslovnu logiku i pripadnu bazu podataka kako bi obavljali vrlo specifičnu ulogu u cjelokupnoj aplikaciji. Klasične operacije poput ažuriranja, testiranja, isporuke i skaliranja koda se odvijaju zasebno na razini svakog mikro servisa.

Dakle uloga mikro servisa je da cjelokupnu poslovnu logiku, koja je specifična za neku poslovnu domenu, podijele na više međusobno neovisnih baza koda, kod kojih se onda svaka baza koda promatra kao zasebna cjelina. Važno je napomenuti da mikroservisi nužno ne smanjuju kompleksnost (u većini slučajeva zapravo dodaju dodatnu kompleksnost) već čine kompleksnost preglednijom na način da velike i kompleksne zadatke podijele na više manjih koje je onda lakše održavati i koji su međusobno neovisni.



Slika 2. Model mikroservisne arhitekture [14]

Slika 2 sadrži prikaz generalnog modela mikroservisne arhitekture. Moguće je uočiti da inicijalan zahtjev koji dolazi s korisničkog sučelja, dolazi na API pristupnik, odakle se zahtjev preusmjerava na jedan od mikroservisa. Mikroservis koji prihvati zahtjev će taj isti zahtjev

obraditi pri čemu će biti neovisan o ostalim mikroservisima. Valja uočiti i da svaki mikroservis ima pripadnu bazu podataka nad kojom može izvršavati upite, za razliku od monolitne arhitekture kod koje najčešće postoji samo jedna baza kojoj pristupa cijela aplikacija.

Prema stranici HiTechNectar [15], glavne prednosti ovakve arhitekture su:

- Skaliranje: promjene u jednom mikro servisu ne utječu na ostale servise, što omogućava neovisan razvoj i isporuku jednog mikro servisa
- Veća otpornost na pogreške – ukoliko se dogodi kvar u radu jednog mikro servisa, to nužno ne znači da će kvar negativno utjecati na rad ostalih servisa zbog njihove neovisnosti
- Lakše shvaćanje rada zasebnog servisa – kako se svaki mikro servis bavi samo jednim specifičnim problemom, razumijevanje rada specifičnog mikro servisa postaje znatno lakše
- Otvorenost novim tehnologijama – za razliku od monolitne arhitekture, programeri koristeći mikroservisnu arhitekturu imaju mogućnost jednostavno prijeći na novu tehnologiju i eksperimentirati zbog toga što svaki servis može koristiti drugačiju tehnologiju

Iako se na prvi pogled mikroservisna arhitektura čini kao savršeno rješenje pored monolitne ili nekih drugih poznatih arhitektura, ona nije bez mane:

- Kompleksnija međuservisna komunikacija – u ovakvoj arhitekturi postaje puno važnije obraćati pozornost na komunikaicju između servisa i ulazne i izlazne tipove podataka
- Veći zahtjevi za resursima – s većim brojem mikro servisa raste i potreba za dodatnim resursima (npr. upravaljanje bazama podataka postaje složenije, pojavljuje se potreba za rukovanjem transakcijama između servisa itd.)
- Testiranje – u slučaju monolitne arhitekture testiranje je vrlo jednostavno jer se cijela aplikacija može pokrenuti i istestirati u cjelini, dok se u slučaju mikroservisne arhitekture svaki servis mora pokrenuti i testirati zasebno i nakon toga se mora testirati rad nekoliko ili svih servisa u cjelini
- Kvalitetno rješenje samo za velike aplikacije – mikroservisna arhitektura se pokazala učinkovitom samo kod velikih i složenih aplikacija, dok kod manjih aplikacija ovakav pristup može dovesti do nepotrebne kompleksnosti i duljim vremenom razvoja

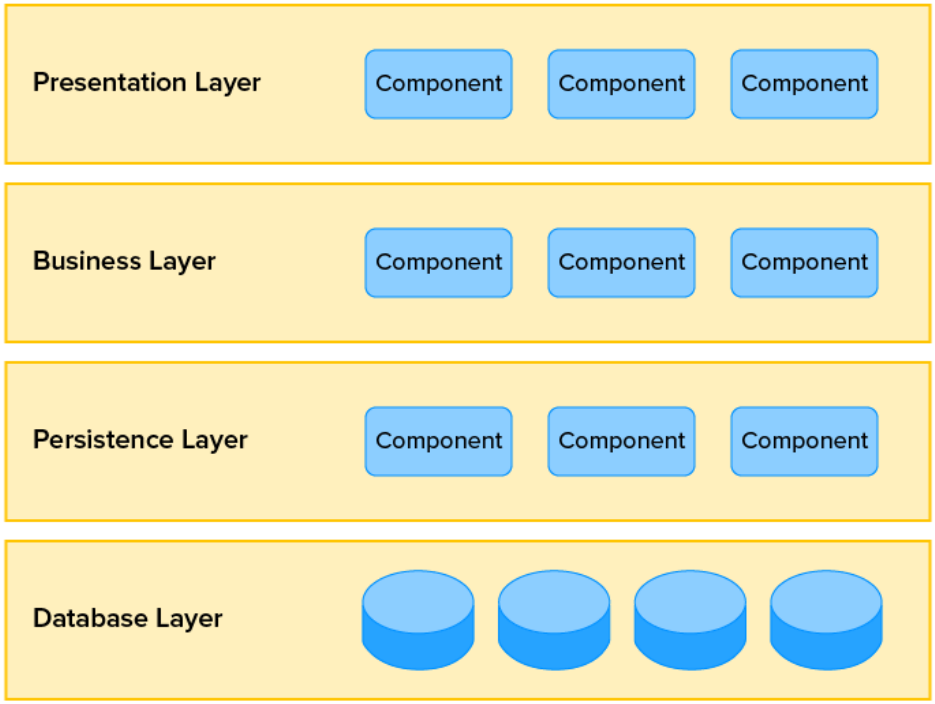
Kako je moguće vidjeti iz priloženog, odabir odgovarajuće arhitekture za razvoj softverskog web rješenja nije nimalo jednostavan zadatak. Postoji velik broj parametara koji se moraju uzeti u obzir kako bi konačno odabrana arhitektura mogla biti korisna i mogla doprinjeti kvaliteti

softverskog proizvoda. U slučaju krive procjene odgovarajuće arhitekture, moguće je učiniti više štete nego koristi.

4.3. Slojevita arhitektura

Prema M. Richardsu [16], slojevita arhitektura je danas jedna od najkorištenijih arhitektura za razvoj softvera. Postoji više naziva za ovakav tip arhitekture uključujući višeslojnu arhitekturu, n-slojna arhitektura idr. Ova arhitektura je isto tako i standardna arhitektura koju poznaje većina programera, pogotovo ako se koriste razvojni okviri poput *Spring*-a ili *Spring Boot*-a i *.NET*-a.

Komponente ovakve arhitekture su horizontalni slojevi, od kojih svaki sloj obavlja neku specifičnu radnju ili zadatak aplikacije. Kako sam naziv arhitekture kaže, ona je višeslojna, što znači da se sastoji od većeg ili manjeg broja slojeva. Broj tih slojeva nije striktno zadan i on ovisi o puno faktora, no standardan broj je 3 ili 4, pa se u tom slučaju govori o troslojnoj ili četveroslojnoj arhitekturi. Uobičajeni slojevi koje većina višeslojnih aplikacija sadržava su: prezentacijski sloj, sloj poslovne logike, sloj za pristup bazi podataka i sama baza podataka. Prema M. Richardsu [16], ponekad se može naići na primjere gdje su slojevi poslovne logike i sloj za pristup bazi podataka spojeni u jedan sloj, ali su takvi slučajevi najčešće vezani uz manje i jednostavnije aplikacije. Kod većih i kompleksnijih aplikacija broj slojeva može biti 5 pa čak i više.



Slika 3. Model višeslojne arhitekture [17]

Na slici 3 prikazan je standardan model slojevite arhitekture, koja se sastoji od prezentacijskog sloja, sloja poslovne logike, sloja za pristup bazi podataka i same baze podataka. Jedna od glavnih načela kojom se vodi ovakav tip arhitekture je tzv. SOC (eng. *Separation of concern*) odnosno metoda podijeli pa vladaj. Ta metoda se u samoj arhitekturi očituje na način da je svaki sloj zasebna jedinica koja zna obavljati samo određene zadatke i u isto vrijeme nije svjesna ostalih slojeva. Razlog zašto jedan sloj nezna za ostale slojeve je zato što uopće ne mora znati. Svaki sloj unutar sebe ima enkapsuliranu logiku koja mu je neophodna za izvršavanje specifičnog zadatka i o ostaloj logici u drugim slojevima ne mora voditi brigu.

Prema M. Richardsu [16], prednosti i mane ove arhitekture su slijedeće:

- Jednostavno testiranje – kada se određena komponenta (sloj) želi testirati, ostali slojevi se mogu mock-ati, što testiranje čini poprilično jednostavnim
- Jednostavnost razvoja – arhitekturalni uzorak je relativno jednostavan za implementiranje a i poznat je većini programera što ga čini vrlo pouzdanim i standardnim rješenjem kod odabira arhitekture projekta

- Niska agilnost – negativna strana ove arhitekture je što ne može brzo i precizno odgovoriti na promjene u zahtjevima okoline zbog toga što iako se smatra zasebnom arhitekturom, najčešće naginje monolitnoj
- Složena isporuka softvera – ovaj parametar jako ovisi o načinu kako je višeslojna arhitektura implementirana, ali u nekim slučajevima može dovesti do kompleksnih i dugotrajnih isporuka
- Loše performanse – zbog načina obrade zahtjeva od sloja do sloja kako bi se zahtjev u potpunosti izvršio, aplikacija korištenjem ovakve arhitekture gubi na performansama odnosno brzini
- Niska razina skalabilnosti – ovaj parametar također ovisi o načinu implementacije arhitekturalnog uzorka, no ukoliko slojevi nisu fizički razdvojeni, skaliranje jednog sloja će najčešće dovesti do potrebe za skaliranjem većine ili svih ostalih slojeva aplikacije

U sklopu praktičnog dijela ovog rada, implementirati će se API servis koristeći upravo višeslojnu arhitekturu koja se smatra standardnom arhitekturom. Razlog tomu je što performanse i isporuka aplikacije nisu bitni, pa se negativne strane ove arhitekture mogu zanemariti, a fokus će se staviti na izradu same aplikacije što je jedna od prednosti višeslojne arhitekture.

5. Uvod u izradu aplikacije

Kao praktični dio ovog rada izraditi će se jednostavna aplikacija po uzoru na neke poznatije sustave za upravljanje osobnim zadacima poput *Google* kalendara i *Microsoft OneNote* sustava.

Aplikacija će se sastojati od 3 glavne komponente. Prva komponenta predstavlja korisničku aplikaciju koja će biti izrađena koristeći *Angular* razvojni okvir. Zatim slijedi API RESTful servis koji će biti izrađen pomoću *ASP.NET Core* razvojnog okvira. Posljednja komponenta je baza podataka za koju će se svrhu iskoristiti *Mongo DB* NoSQL baza podataka sa pripadnim serverom i sustavom za upravljanje bazom podataka.

5.1. Angular

Angular je jedan od najpoznatijih frontend razvojnih okvira koji se odlikuje svojim konceptom jedno-straničnih web aplikacija koji se zasniva na modelu komponenti.

Prema Sharma-i [18], pod pojmom *Angular*, razlikuju se dvije glavne verzije:

1. *AngularJS* – ranija verzija angulara izdana 2009. godine koja se bazirala na klasičnom javascript programskom jeziku; neke od glavnih karatkeristika ove verzije su bile validacija i animacije formi, jednostavno rukovanje događajima (engl. Event handling), uzorak dizajna injekcija ovisnosti(eng. *Dependency injection*), vlastiti sustav za preusmjeravanje, vezanje podataka idr.
2. *Angular* - novija verzija angular razvojnog okvira koja je bazirana na *typescript* programskom jeziku (*typescript* je superset javascript jezika s dodanim svojsvom definiranja tipova podataka); glavne karakteristike ove verzije su: napredan DI, poboljšano vezanje podataka, direktive idr.

Angular je razvojni okvir koji se bazira na modelu komponenti. Komponenta je zaseban dio web aplikacije (na strani korisnika) koja sadrži svoju specifičnu logiku, te *html* i *css* datoteke za prikaz. Povezivanjem komponenti korisniku se prikazuje cijela stranica.

Bitno je za naglasiti i da je *angular* sustav za izradu jednostraničnih aplikacija. Dakle prilikom učitavanja stranice preko web preglednika, korisnik učitava samo jednu *.html* datoteku sa svim pripadnim kodom (*typescript* kodom koji je preveden u *javascript* te *html*, *css* i logiku svih pripadnih komponenti koje čine stranicu). Na taj se način ostvaruju odlične performanse jer se kod bilo kakvog preusmjeravanja koje korisnik želi učiniti, ne mora ispočetka učitavati cijela stranica, već se komponente (osnovni gradivni blokovi web stranice) učitavaju dinamički i daju dojam korisniku kao da koristi više stranica.

5.2. API RESTful servis

Kao što je već spomenuto, druga komponenta praktičnog rada je API servis. Prema članku Amazon Web Service trvtke [19], API ili aplikacijsko programsko sučelje je skup pravila koje je potrebno slijediti kako bi se ostvarila komunikacija između dvije aplikacije. Programeri uobičajeno razvijaju API rješenja kako bi druge vanjske aplikacije mogle komunicirati s njihovom aplikacijom programski.

REST oznaka prema članku tvrtke Red Hat označava set arhitekturalnih pravila koje API servis mora poštivati kako bi se smatrao REST servisom [20]. Neke glavne karakteristike odnosno pravila REST servisa su:

- Arhitektura klijent-poslužitelj koja se sastoji od klijentskih aplikacija, poslužitelja (servisa) i resursa (podaci koji se šalju kao zahtjev ili odgovor na zahtjev) koji sudjeluju u komunikaciji preko HTTP protokola
- Komunikacija između klijenta i poslužitelja gdje se informacije o klijentu ne pamte niti na koji način između različitih zahtjeva, odnosno svaki zahtjev je element sam za sebe
- Uniformno sučelje između komponenti tako da se informacija prenosi u standardnom obliku
- Višeslojni sustav koji svaku vrstu poslužitelja uključenog u dohvaćanje traženih podataka organizira u hijerarhije nevidljive klijentu

Prema već spomenutom članku tvrtke Amazon [19], glavne prednosti RESTful API servisa su:

- Skalabilnost – efikasno skaliranje jer REST servisi optimiziraju interakciju između klijenta i poslužitelja
- Fleksibilnost – korištenjem RESTful servisa, moguće je postići potpuno odvajanje klijenta i poslužitelja, gdje se onda svaka od te dvije komponente može promatrati i graditi neovisno o drugoj
- Neovisnost – primarno se misli na neovisnost o tehnologiji jer se klijentska aplikacija može razviti korištenjem jedne, a API servis sasvim druge tehnologije

Komunikacija se između klijenta i API servisa ostvaruje na način da klijent preko HTTP-a šalje zahtjev servisu (poslužitelju) koji zahtjev mora zaprimiti, obraditi i konačni rezultat obrade vratiti nazad klijentu.

Konkretno, koriste se HTTP metode kroz koje klijent API servisu može točno reći kakav tip obrade nad podatkom zahtjeva. 4 su osnovne HTTP metode:

- **Get** – dohvaćanje resursa
- **Post** – slanje podataka servisu (najčešće s ciljem kreiranja novog resursa na poslužitelju)
- **Put** – ažuriranje postojećih resursa
- **Delete** – brisanje postojećih resursa

Isto tako, HTTP zahtjev sadrži zaglavlje u kojemu se mogu nalaziti metapodaci koji se razmjenjuju između klijenta i poslužitelja. Unutar zaglavlja, moguće je poslati i cjelovite podatke, primarno kod POST i PUT HTTP metoda. Zahtjev može sadržavati i dodatne parametre s ciljem detaljnijeg specificiranja poslanog zahtjeva. Postoje 3 glavne vrste parametara: parametri u putanji (dio su URL-a), parametri upita kojima je moguće zatražiti više informacija o nekom resursu i parametri s kolačićima najčešće radi autentifikacije korisnika.

U ovom radu za izradu API servisa koristiti će se *ASP.NET Core* razvojni okvir za izradu Web API projekta.

5.3. Funkcionalnosti aplikacije

Aplikacija koja će se izraditi biti će vrlo jednostavna web aplikacija za više korisnika koji će kroz tu aplikaciju moći organizirati svakodnevne zadatke i voditi bilješke.

Kako bi aplikaciju mogao koristiti veći broj korisnika, biti će potrebno kao jedan od prvih koraka, implementirati funkcionalnost registracije (za kreiranje računa) i prijave (funkcionalnost autentifikacije). Korisnik će prilikom uspješne registracije ili prijave u sustav, u odgovoru na HTTP zahtjev, dobiti token koji će zapravo biti JWT tip tokena u kojemu će se nalaziti kriptirani podaci o korisniku koji posjeduje token. Na taj način će API servis moći autentificirati korisnika prilikom zapraimljenog zahtjeva kako bi mogao odlučiti ima li korisnik pristup resursu ili ne. Uz te dvije funkcionalnosti, korisnik će moći i obrisati vlastiti korisnički račun.

Nadalje, korisnik će kroz aplikaciju moći voditi brze bilješke preko tekstualnog uređivača. Konkretan modul koji će se koristiti je @kolkov angular-editor, koji je po svojoj prirodi otvorenog koda.

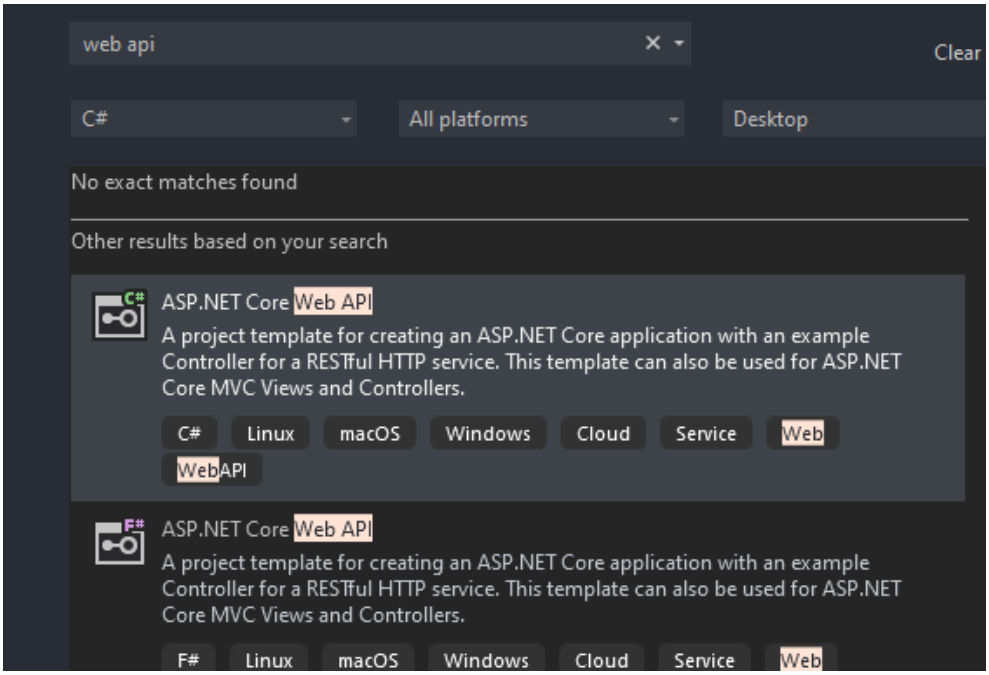
Iduća funkcionalnost koju će korisnik moći koristiti je dodavanje zadataka, za koje će se moći definirati konkretno vrijeme. Korisnik će moći upravljati svojim zadacima, što uključuje dodavanje novih te izmjenu i brisanje postojećih zadataka. Funkcionalnost koja se nadovezuje na ovu, je prikaz zadataka u obliku kalendara. Kalendar će se kao i tekstualni zređivač uvesti u aplikaciju a konkretan modul koji će biti korišten je FullCalendar koji je također otvorenog koda.

Posljednja funkcionalnost koja će se implementirati je upravljanje vlastitim bilježnicama (dokumentima) koje će biti vrlo slične brzim bilješkama. Korisnik će moći dodavati, ažurirati, brisati i pregledavati svoje bilježnice.

6. Početna struktura backend-a

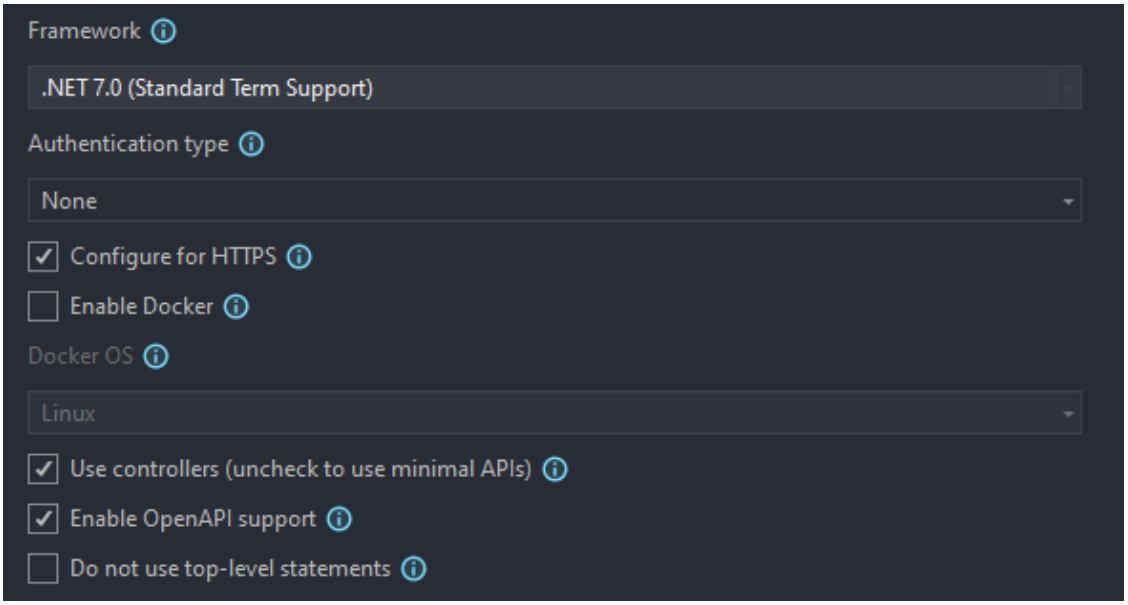
6.1. Izrada projekta

S razvojem backend dijela aplikacije započeti će se stvaranjem novog *ASP.NET* Web Api projekta.



Slika 4. Odabir tipa projekta [Autorski rad]

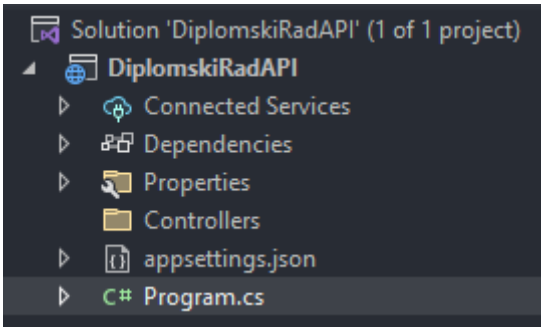
Potom se odabire naziv aplikacije te putanja u koju će se projekt spremiti. Na poslijetku je potrebno postaviti konfiguraciju same aplikacije, odnosno servisa.



Slika 5. Postavljanje početne konfiguracije [Autorski rad]

Jednom kada je projekt stvoren, pobrisati će se dvije klase koje su dio inicijalnog projekta kako bi se projekt očistio od nebitnih datoteka, *WeatherForecast.cs* i *WeatherForecastController.cs*.

Početna struktura izgleda ovako:



Slika 6. Početne datoteke [Autorski rad]

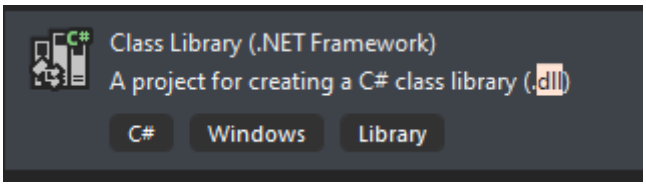
Početna struktura projekta sastoji se od nekoliko datoteka, od kojih su najvažnije *Controllers*, *appsettings.json* i *Program.cs*. *Controllers* datoteka sadrži c# klase koje predstavljaju API pristupne točke. Kada korisnik iz nekog sučelja kreira neki HTTP zahtjev, taj zahtjev dolazi na jedan od mogućih (implementiranih) pristupnih točaka i okida se kod koji se unutar te točke nalazi, odnosno možemo raći da se zahtjev obrađuje. Nakon što je zahtjev obrađen, odgovor se vraća korisniku.

Appsettings.json je datoteka koja sadržava konfiguraciju projekta, i kasnije će sadržavati podatke poput *connection string* vrijednosti za pristup bazi isl.

Program.cs je glavna datoteka preko koje se pokreće servis, odnosno stvara se nova instanca aplikacije (API servisa). Svaki novi servis koji se želi koristiti unutar aplikacije, mora se registrirati kao servis kako bi ga aplikacija mogla koristiti, a to se ostvaruje *Dependency injection*-om kojime se registrira željeni servis i dodaje u DI Container.

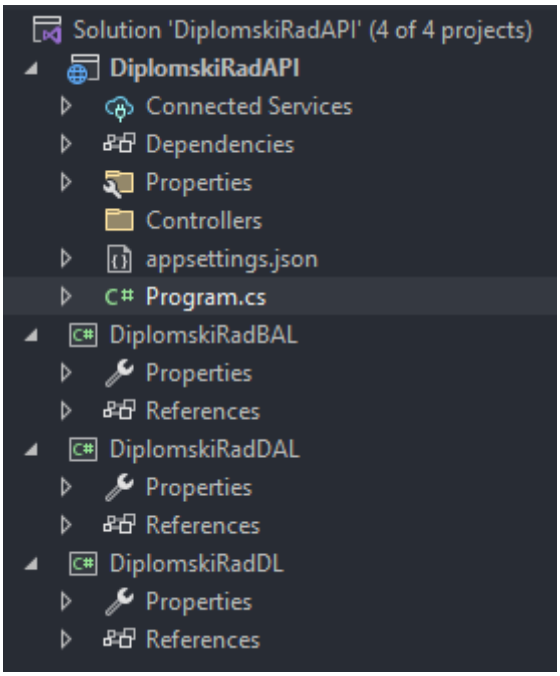
Jedan od prvih koraka novoga projekta je postaviti željenu strukturu projekta, odnosno postaviti slojeve aplikacije (prvi sloj je API, zatim slijede BLL business logic layer i DAL data access layer).

Pojedinačni slojevi su dodani kao posebni pod projekti u obliku dll-a.



Slika 7. Vrsta projekta aplikacijskog sloja [Autorski rad]

Struktura, nakon dodanih podprojekata izgleda ovako:



Slika 8. Početna struktura [Autorski rad]

Zatim je potrebno postaviti reference, odnosno hijerarhiju podprojekata. API (glavni projekt) ima reference na sve preostale pod projekte, BAL ima reference na DAL i DL, DAL ima reference samo na DL i DL nema nikakvih referenci.

Razlog zašto je potrebno postaviti reference je zbog toga što će se svaki sloj oslanjati na resurse iz preostalih slojeva (izuzetak je DL sloj koji služi samo za definiranje domene aplikacije).

6.2. Definiranje početnih klasa

Kao početne funckionalnosti koje su bitne za aplikaciju, implementirati će se login i registracija korisnika. Za to je prvo potrebno dodati entitet korisnika kao model.

6.2.1. Klasa korisnika

Unutar DL sloja, napraviti će se nova mapa *Data* i unutar nje mapa *Models*. Unutar mape *Models* dodati će se nova c# klasa *User* koja će predstavljati entitet korisnika aplikacije.

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public string Email { get; set; }
    public string Username { get; set; }
    public byte[] PasswordHash { get; set; }
    public byte[] PasswordSalt { get; set; }
}
```

Ta klasa će imati niz atributa koji će se nadograđivati kako funckionalnosti aplikacije budu rasle, no za sada će imati osnovne atribute koji su potrebni kako bi se implementirale funckionalnosti prijave i registracije korisnika.

6.2.2. Postavke Mongo baze podataka

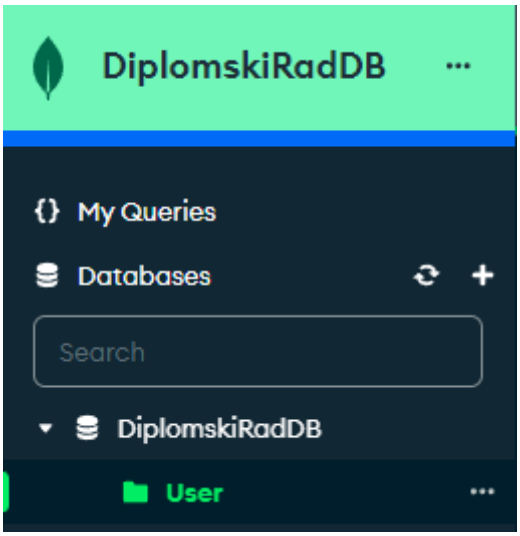
Jednom kada je entitet kreiran, potrebno je uspostaviti vezu između aplikacije i baze podataka, u ovom slučaju s *MongoDB* bazom. Kako bi se uspostavila veza s bazom, potrebno je učiniti nekoliko koraka. Prvo je potrebno kreirati novu klasu unutar DAL sloja koja će sadržavati postavke za uspostavljanje veze (konekcije) s *MongoDB* bazom podataka.

```
public class MongoDBSettings
{
    public string ConnectionURI { get; set; } = null;
```

```
        public string DatabaseName { get; set; } = null;
        public string CollectionName { get; set; } = null;
    }
```

Ova klasa sadrži informacije poput connection URI-a same baze, zatim naziva specifične baze te naziva kolekcije (nešto poput relacije odnosno tablice u klasičnoj SQL bazi podataka) koja će sadržavati dokumente (pojedinačne zapise).

Potom je potrebno unutar *MongoDB Compass* sučelja kreirati novu bazu podataka i dodati novu kolekciju.



Slika 9. Dio sučelja Mongo DB Compass-a [Autorski rad]

Compass pruža vrlo jednostavan način dohvaćanja Connection string vrijednosti koji je potreban API servisu kako bi znao za lokaciju baze podataka koju će koristiti. Uz sam connection string potrebno je dodati i nuget pakete za rad s *MongoDB* bazom podataka. Potrebni paketi su: MongoDB.Driver i MongoDB.BSON.

Prethodno definirana klasa *MongoDbSettings* podatke će vući iz konfiguracijske datoteke *appsettings.json*. Trenutno ta datoteka ne sadrži nikakve informacije o bazi podataka, pa je te podatke potrebno unijeti.

```
"MongoDB": {
  "ConnectionURI": "mongodb://localhost:27017",
  "DatabaseName": "DiplomskiRadDB",
  "CollectionName": "User"
}
```

ConnectionURI je preuzet iz aplikacije *MongoDB Compass* i specifičan je za svaku bazu. Naziv baze podataka i kolekcije se također moraju definirati kako bi MongoDB.Driver mogao znati koja baza podataka i koja specifična kolekcija je cilj, pa te vrijednosti moraju biti iste kao i u bazi podataka (prethodno pojašnjen korak).

6.2.3. Kontekst klasa

Slijedeći korak je definiranje konteksta baze podataka. To će biti klasa koja će služiti kao sučelje prema izrađenoj bazi podataka i preko nje će se slati zahtjevi.

```
    public class MongoDBService
    {
        public readonly IMongoCollection<User> userCollection;

        public MongoDBService(IOptions<MongoDBSettings> mongoDBSettings)
        {
            MongoClient client = new
MongoClient(mongoDBSettings.Value.ConnectionURI);
            IMongoDatabase database =
client.GetDatabase(mongoDBSettings.Value.DatabaseName);
            userCollection =
database.GetCollection<User>(mongoDBSettings.Value.CollectionName);
        }
    }
```

Ovu klasu je potrebno inicijalizirati, ali u slučaju Web API servisa, inicijalizacija ovakvih klasa ne odrađuje se „ručno“. Klasu je potrebno definirati unutar *Program.cs* datoteke. Kao što je već ranije spomenuto, svi servisi koje aplikacija koristi moraju biti registrirani unutar DI container-a.

```
builder.Services.Configure<MongoDBSettings>(builder.Configuration.GetSectio
n("MongoDB"));
builder.Services.AddSingleton<MongoDBService>();
```

Ovim kodom postavljena je konfiguracija za bazu podataka te je klasa *MongoDBService* instancirana automatski kao singleton (kako bi se osigurala samo jedna instanca konekcije prema bazi podataka).

Iduća stvar je zapravo izmijena modela *User*. U jednom od prošlih koraka definirana je klasa *User* koja predstavlja entitet korisnika aplikacije. Jedan od atributa te klase je Id odnosno identifikacijski ključ koji je jedinstvena oznaka svakog korisnika aplikacije. Tip podatka je postavljen kao int, no *MongoDB* id ključeve zapisuje kao string i to ne kao klasičan JSON atribut već kao BSON, Binary JSON. Iz tog je razloga potrebno promijeniti tip podatka tog atributa.

```
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }
```

Ovime je efektivno implementirana komunikacija između API servisa i pripadne *MongoDB* baze podataka. Isto tako, kreirana je kolekcija *User* unutar same baze s kojom je

sada moguće raditi. Kako projekt prati višeslojnu arhitekturu, sada je potrebno napisati tri klase koje će predstavljati kontroler te pripadajući servis i repozitorij baze podataka a započeti će se s repozitorijem.

6.2.4. Repozitorij korisnika

Unutar DAL sloja potrebno je kreirati novu mapu *Repositories* u koju će se smjestiti svi repozitoriji koje servis koristi. Svaki repozitorij predstavlja sučelje prema jednoj od kolekcija u bazi podataka.

```
    public class UserRepository
    {
        private readonly MongoDBService _mongoDBService;

        public UserRepository(MongoDBService mongoDBService)
        {
            _mongoDBService = mongoDBService;
        }

        public async Task<User> AddUser(User user)
        {
            await _mongoDBService.userCollection.InsertOneAsync(user);
            return user;
        }

        public async Task<User> GetUserByUsernameAndEmail(string username,
string email)
        {
            var user = await
_mongoDBService.userCollection.AsQueryable().Where(x => x.Username ==
username || x.Email == email).FirstOrDefaultAsync();
            if (user != null) return user;
            return null;
        }

        public async Task<User> DeleteUser(string id)
        {
            var user = await
_mongoDBService.userCollection.AsQueryable().Where(x => x.Id ==
id).FirstOrDefaultAsync();
            if (user != null)
            {
                FilterDefinition<User> filter =
Builders<User>.Filter.Eq("Id", id);
                var result = await
_mongoDBService.userCollection.DeleteOneAsync(filter);
                return user;
            }
            else
            {
                throw new ArgumentException();
            }
        }

        public async Task<User> GetUserByUsernameOrEmail(string
usernameOrEmail)
        {
```

```
            var user = await
_mongoDBService.userCollection.AsQueryable().Where(x => x.Username ==
usernameOrEmail || x.Email==usernameOrEmail).FirstOrDefaultAsync();
            if (user != null) return user;
            return null;
        }
    }
```

Klasa *UserRepository* sadrži jedno privatno polje u kojemu će biti pohranjen kontekst baze podataka. Taj atribut se dodaje preko Dependency injection metode u konstruktoru same klase. Moguće je uočiti kako se kroz DI prosljeđuje tip podatka *MongoDBService* koji je ranije dodan u DI Container same aplikacije pa ga nije potrebno ručno inicijalizirati, već je on automatski inicijaliziran kao singleton. Repozitorij sadrži i metode za dodavanje novog korisnika (funkcionalnost registracije), brisanje korisnika te prijavu u sustav (funkcionalnost prijave).

U svakoj od tih metoda se koristi *_mongoDBService* koji je prosljeđen kroz DI metodu kako bi se željena akcija prema bazi podataka mogla izvršiti.

6.2.5. Servis klasa korisnika

Iduća klasa je servis za funkcionalnosti korisnika koja će koristiti prethodno definirani repozitorij. Ta klasa će biti implementirana u BAL sloju.

```
    public class UserService
    {
        private readonly UserRepository _userRepository;
        private readonly ITokenService _tokenService;

        public UserService(UserRepository userRepository,ITokenService
tokenService)
        {
            _userRepository = userRepository;
            _tokenService = tokenService;
        }

        public async Task<User> RegisterUser(RegisterVM newUser)
        {
            var existingUser= await UserExists(newUser.Username,
newUser.Email);
            if (existingUser.Item1)
            {
                throw new ArgumentException();
            }
            var hmac = new HMACSHA512();
            var user = new User
            {
                Name = newUser.Name,
                Surname = newUser.Surname,
                Username = newUser.Username,
                Email = newUser.Email,
```

```
                PasswordHash =
hmac.ComputeHash(Encoding.UTF8.GetBytes(newUser.Password)),
                PasswordSalt = hmac.Key
            };
            hmac.Dispose();
            var createdUser = await _userRepository.AddUser(user);
            return createdUser;
        }

        public async Task<User> DeleteAccount(string id)
        {
            var user = await _userRepository.DeleteUser(id);
            return user;
        }

        public async Task<User> Login(string usernameOrEmail, string
password)
        {
            var existingUser = await UserExists(usernameOrEmail);
            if(existingUser.Item1)
            {
                var hmac = new HMACSHA512(existingUser.Item2.PasswordSalt);
                var computedHash =
hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
                for(int i = 0; i < computedHash.Length; i++)
                {
                    if (computedHash[i] !=
existingUser.Item2.PasswordHash[i]) throw new ArgumentException();
                }
                hmac.Dispose();
                return existingUser.Item2;
            }
            throw new ArgumentException();
        }

        private async Task<Tuple<bool, User>> UserExists(string
usernameOrEmail, string email = null)
        {
            User user;
            if (email != null)
            {
                user = await
_userRepository.GetUserByUsernameAndEmail(usernameOrEmail, email);
            }
            user = await
_userRepository.GetUserByUsernameOrEmail(usernameOrEmail);
            if (user != null)
            {
                return new Tuple<bool, User>(true, user);
            }
            return new Tuple<bool, User>(false, null);
        }
    }
```

Ovdje je moguće primjetiti da se opet koristi DI metoda kako bi se klasi servisu prosljedio prethodno definirani repozitorij. Servis također sadrži tri metode za registraciju i prijavu kao i brisanje korisnika. Razlog zašto je ova klasa dodana kao još jedna razina apstrakcije je zato što će u budućnosti ovdje biti implementirana poslovna logika servisa.

6.2.6. Kontroler za korisnike

Na posljetku potrebno je definirati i sam API endpoint za ove funkcionalnosti. To se rješava uz pomoć kontrolera servisa.

```
    [Route("api/[controller]")]
    [ApiController]
    public class UserController : ControllerBase
    {
        private readonly UserService _userService;
        public UserController(UserService userService)
        {
            _userService = userService;
        }

        [HttpPost("/register")]
        public async Task<IActionResult> RegisterUser([FromBody] RegisterVM
newUser)
        {
            try
            {
                var createdUser = await _userService.RegisterUser(newUser);
                return Ok(createdUser);
            }
            catch (Exception)
            {
                return BadRequest();
            }
        }

        [HttpDelete("id")]
        public async Task<IActionResult> DeleteUser(string id)
        {
            try
            {
                var deletedUser = await _userService.DeleteAccount(id);
                return Ok(deletedUser);
            }
            catch (Exception)
            {
                return BadRequest();
            }

        }

        [HttpPost("/login")]
        public async Task<IActionResult> Login([FromBody] LoginVM
loginParams)
        {
            try
            {
                var user = await
_userService.Login(loginParams.UsernameOrEmail, loginParams.Password);
                return Ok(user);
            }
            catch (Exception)
            {
                return NotFound();
```

```
            }
        }
    }
```

Svaki endpoint u kontroleru je zapravo metoda koja ima određeni dekorator. 4 su osnovna dekoratora za endpointe: HttpGet (za dohvaćanje podataka), HttpPut(za ažuriranje podataka), HttpPost (za unos podataka) te HttpDelete (za brisanje podataka). Svaki endpoint ima i svoju adresu prema kojoj se odvija „routing“ zahtjeva.

Ovdje se također koristi DI kako bi se prosljedio odgovarajući servis. Potom se u svakom od endpointa koriste resursi servisa kako bi se obavila željena radnja.

6.2.7. DI kontejner

Posljednji korak je zapravo dodati prethodno implementirane klase u DI Container servisa.

```
builder.Services.AddScoped<UserRepository>();
builder.Services.AddScoped<UserService>();
```

Ovdje je moguće uočiti da se klase ne dodaju kao singletoni već kao Scoped klase. U ASP.NET Core Web API tipu projekta, klase je moguće dodati u DI container u tri različite verzije: Transient, Scoped i Singleton. Razlika je u vremenu trajanja (životnom vijeku) dodane klase. Transient klase se kreiraju prilikom svakog novog poziva klase. Scoped klase se inicijaliziraju za svaki pojedinačni Http zahtjev i traju dok se zahtjev ne obradi u potpunosti. Singleton klase se inicijaliziraju samo jednom, prilikom njihovog prvog poziva u aplikaciji i traju koliko i sam servis, što znači da svaki slijedeći poziv klase koristi istu instancu te klase.

7. Početna struktura klijentske aplikacije

Nakon što je backend dio gotov (logika za prijavu i registraciju korisnika), krenuti će se s radom na korisničkoj aplikaciji koristeći *Angular* razvojni okvir. *Angular* je prvo potrebno instalirati koristeći *Node Package Manager* (skraćeno NPM). Verzija *Angular* razvojnog okvira koji će se koristiti u ovom radu je 15.0.0. Nakon što je *Angular* uspješno instaliran, koristeći naredbu:

ng new <naziv projekta> može se kreirati novi projekt. *Angular* je razvojni okvir za izradu jedno straničnih (eng. *Single page*) aplikacija i osnovni gradivni blok takvih aplikacija je komponenta. Komponenta se može zamisliti kao određeni dio web stranice koji predstavlja neku smislenu cjelinu (npr. navigacijska traka, glavni zaslon, izbornik, modal isl.). Svaka se komponenta u angularu sastoji od svoje pripadne *HTML* datoteke (u kojoj je definirana struktura komponente), *CSS* datoteke (za stiliziranje komponente) i *typescript* datoteke u kojoj je definirana logika komponente. Takvim je pristupom onda moguće upravljati različitim komponentama stranice i korisnik ima dojam da koristi više različitih web stranica iako zapravo koristi samo jednu stranicu čiji se dijelovi (komponente) mijenjaju korisničkom interakcijom.

Prilikom kreiranja novog *Angular* projekta, kreirana je korijenska komponenta *app-root* koja se prikazuje u *index.html* datoteci. Koristeći naredbu „ng serve“, moguće je izgraditi i pokrenuti aplikaciju.

Jedan od prvih koraka u izradi korisničke aplikacije je izrada korisničkog sučelja, pa će idući korak biti kreiranje korisničkog sučelja za funkcionalnosti prijave i registracije korisnika. Za izradu korisničkog sučelja koristiti će se *Bootstrap* okvir kako bi proces bio brži. Klasičan *Bootstrap* je baziran na *javascript* programskom jeziku, ali za potrebe ove aplikacije koristi se *Angular* koji je baziran na *typescript* jeziku. Iz tog je razloga uz sam *Bootstrap* potrebno koristiti i *Angular* modul koji pruža funkcionalnosti rada s Bootstrapom koji se naziva *ngx-bootstrap*.

Ngx-bootstrap se može instalirati naredbom:

```
npm install ngx-bootstrap@10 –legacy-peer-deps
```

Potrebno je instalirati i sam *bootstrap*:

```
npm install bootstrap@5 –legacy-peer-deps
```

Unutar *angular.json* datoteke potrebno je ažurirati styles sekciju:

```
"styles": [
      "./node_modules/ngx-bootstrap/datepicker/bs-datepicker.css",
      "./node_modules/bootstrap/dist/css/bootstrap.min.css",
```

```
        "./node_modules/font-awesome/css/font-awesome.min.css",
        "./node_modules/ngx-toastr/toastr.css",
        "src/styles.css"
],
```

Unutar *app.module.ts* datoteke potrebno je uvesti *BrowserAnimationsModule*.

Jednom kada su paketi instalirani, može se krenuti s izradom korisničkog sučelja. Prva stranica koja će se dizajnirati je naslovna stranica, potom stranice za prijavu, registraciju i glavno sučelje.

Kao što je već ranije spomenuto, *Angular* je okvir za izradu jednostraničnih aplikacija, pa će stoga svaka od navedenih „stranica“ zapravo biti implementirana kao komponenta. Nove se komponente u angularu kreiraju naredbom ng generate component <naziv komponente>.

7.1. Preusmjeravanje

Jednom kada su kreirane komponente (s pripadajućim *html* i *css* kodom), potrebno je dodati navigaciju. Prilikom kreiranja novog *Angular* projekta, korisnik ima mogućnost odabrati korištenje *Angular Routing* modula za preusmjeravanje. Uz pomoć tog modula, implementacija navigacije je vrlo jednostavna. Prvi korak je definiranje putanja do svake komponente i to u *app-routing.module.ts* datoteci koja je automatski kreirana prilikom inicijalnog pokretanja aplikacije ukoliko je korisnik odabrao opciju korištenja *Angular Routing* modula.

```
const routes: Routes = [
        {path: '',component: CoverPageComponent},
        {path: 'login',component: LoginComponent},
        {path: 'register',component: RegisterComponent},
        {path: '**',component: CoverPageComponent, pathMatch: 'full'},
];
```

Kao što je moguće vidjeti, prva putanja je zadana putanja koja vodi na naslovnicu (komponenta „*CoverPageComponent*“), a zatim su definirane putanje do komponenata za prijavu i registraciju korisnika. Posljednja definirana putanja je tzv. „wild card“ putanja koja predstavlja svaki drugi URL kojeg korisnik unese a ne podudara se niti s jednom prethodno definiranom putanjom i u tom slučaju će se korisnik preusmjeriti na naslovnu stranicu (komponentu).

Sada je pitanje gdje se točno pozivaju komponente kada korisnik unese njihov pripadajući URL. Unutar *app.component.html* datoteke (u korijenu projekta), potrebno je postaviti *Angular* komponentu zvanu „*router-outlet*“. Ta komponenta će se dinamički zamijeniti onom komponentom čiji je URL odabran.

Ostalo je još samo definirati putanje kod elemenata stranice koji nose ulogu navigacije. *Angular* za tu potrebu, umjesto klasičnog *html* atributa za navigaciju „*href*“, koristi vlastiti atribut „*routerLink*“, kojime je moguće definirati URL na kojeg će se korisnik preusmjeriti jednom kada se interaktira s elementom.

8. Funkcionalnost prijave i registracije

Kako bi se komunikacija između aplikacije i API servisa mogla uspješno odvijati, u API projekt je prvo potrebno dodati CORS postavke, odnosno potrebno je omogućiti frontend aplikaciji da koristi resurse API servisa.

Unutar *Program.cs* datoteke, potrebno je dodati Cors kao Servis (u DI container), i potrebno je prosljediti opcije metodi *UseCors* kroz koje se može registrirati adresa korisničke aplikacije kako bi tu adresu API servis prepoznao kao validnu.

```
builder.Services.AddCors();
app.UseCors(builder=>
builder.AllowAnyHeader().AllowAnyMethod().WithOrigins("http://localho
st:4200"));
```

8.1. Json Web Token

Za potrebe autentifikacije i autorizacije, klasično rješenje koje se koristi su tzv. tokeni. U ovom slučaju, koristiti će se jedan od najpopularnijih tipova tokena zvan JWT (eng. *Json Web token*).

8.1.1. Token servis

Jedan od prvih koraka je implementacija klase za kreiranje tokena. Prvo će se kreirati *ItokenService* klasa koja će služiti kao sučelje koje će onda implementirati konkretna klasa *TokenService*. U sloju domene, kreirati će se mapa *Contracts* u koju će se smještati sučelja, i unutar nje će se kreirati novo sučelje.

```
public interface ITokenService
{
    string CreateToken(User user);
}
```

Ovo sučelje definira samo jednu metodu koja je potrebna u ovom trenutku, a to je kreiranje tokena. Kao što je već spomenuto, klasa koja će implementirati ovo sučelje je *TokenService*. Unutar BAL sloja, kreirati će se ta klasa, te će se konkretizirati metoda *CreateToken*.

```
public class TokenService : ITokenService
{
    private readonly SymmetricSecurityKey _key;
    public TokenService(IConfiguration config)
    {
        _key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(config["TokenKey"]));
```

```
        }
        public string CreateToken(User user)
        {
            var claims = new List<Claim>
            {
                new Claim(JwtRegisteredClaimNames.UniqueName, user.Id),
                new Claim(JwtRegisteredClaimNames.GivenName, user.Name),
                new Claim(JwtRegisteredClaimNames.FamilyName,
user.Surname),
                new Claim(JwtRegisteredClaimNames.NameId,user.Username),
                new Claim(JwtRegisteredClaimNames.Email,user.Email),
            };
            var credentials = new SigningCredentials(_key,
SecurityAlgorithms.HmacSha512Signature);
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(claims),
                Expires = DateTime.Now.AddDays(7),
                SigningCredentials = credentials,
            };
            var tokenHandler = new JwtSecurityTokenHandler();
            var token = tokenHandler.CreateToken(tokenDescriptor);
            return tokenHandler.WriteToken(token);
        }
    }
```

Metoda *CreateToken* koju implementira klasa *TokenService* služi za generiranje tokena koji će se slati kao odgovor http zahtjeva. Token će biti kriptiran asimetričnim ključem (koji je spremljen u konfiguracijsku datoteku) te će se u njega spremiti podaci o korisniku (npr. id, ime, prezime, email, korisničko ime itd.). Sama metoda će se pozivati unutar servisa za prijavu i registraciju. Korisnička aplikacija će tada moći trajno spremiti generirani token za nekog korisnika, te ga slati prilikom svakog slijedećeg zahtjeva ukoliko je to potrebno.

8.1.2. Middleware za autentifikaciju i autorizaciju

Idući korak je definirati korištenje autentifikacije i autorizacije na razini cijelog API servisa, a to je potrebno učiniti u *Program.cs* datoteci, ali je prije toga još potrebno instalirati nuget paket Microsoft.AspNetCore.Authentication.JwtBearer.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationSc
heme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["To
kenKey"])),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

```
app.UseAuthentication();
app.UseAuthorization();
```

U kodu iznad, definiran je način validiranja JWT tokena. Kroz parametar „*options*“, servisu je dano doznanja da se na pristupnim točkama servisa, gdje je potrebna autentifikacija odnosno autorizacija, validira token, i to na način da se prilikom dekripcije ključa koristi isti asimetrični ključ koji je korišten i prilikom njegovog kriptiranja. Validiranje izdavatelja ključa nije potrebno jer je u ovom slučaju izdavatelj uvijek sam servis.

Prethodno definirano sučelje koje će se prosljeđivati servisima, potrebno je dodati u DI kontejner.

```
builder.Services.AddScoped<ITokenService,TokenService>();
```

8.1.3. DTO klasa korisnika

Nakon ovih koraka, servisu je omogućeno korištenje validacije JWT tokena, ali potrebno je definirati mjesta gdje će se token generirati i validirati. Kao što je prethodno spomenuto, korisnik će kroz korisničku aplikaciju, do ključa doći kroz uspješnu prijavu i registraciju (prilikom uspješne registracije, korisnika će se automatski prijaviti u sustav).

Iz tog je razloga, kao odgovor na zahtjeve prijave i registracije, potrebno postaviti tip podatka koji će unutar sebe sadržavati i generirani token. Klasa *User* koja već postoji nema polje za spremanje tokena pa će se kreirati nova VM klasa koja će biti vrlo slična postojećoj.

Unutar sloja domene, kreirana je nova mapa *ViewModels* i unutar nje je dodana klasa *UserVM*.

```
public class UserVM
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public string Email { get; set; }
    public string Username { get; set; }
    public string Token { get; set; }
}
```

Ta klasa će se vraćati kao odgovor prilikom prijave i registracije, ali je isto tako, kao ulaz u te pristupne točke, potrebno poslati VM specifičan za prijavu odnosno registraciju. Prilikom prijave, iz korisničke će aplikacije pristizati *LoginVM* klasa s podacima o unesenom korisničkom imenu ili emailu (prijava je moguća preko korisničkog imena ili email-a), a prilikom registracije *RegisterVM* (koji će sadržavati podatke iz registracijske forme za kreiranje novog korisnika u bazi).

Za kraj je potrebno postaviti „[*Authorize*]“ dekorator na pristupnoj točki za brisanje korisničkog računa, jer je preduvjet za brisanje računa da je korisnik prijavljen (odnosno, za tog korisnika je generiran token). Na tom mjestu unutar kontrolera će se, prilikom zaprimanja Http zahtjeva, provjeriti token poslan u zaglavlju zahtjeva, pa će se potom validirati. Ukoliko je token validan, API servis će nastaviti s obradom zahtjeva (brisanjem korisničkog računa) a u protivnom će se korisniku javiti poruka pogreške, odnosno da korisnik nije autentificiran.

9. Prijava i registracija na strani klijenta

Idući dio je implementacija funkcionalnosti prijave i registracije na strani korisničke aplikacije, a započeti će se s prijavom.

9.1. Funkcionalnost prijave

9.1.1. Reaktivne forme

Za prijavu, kao i za registraciju, potrebno je omogućiti korisniku unos podataka kroz element forme. *Angular* sadrži dva različita modula za forme. Prvi modul su forme vođene predloškom (eng. *Template Driven*). To je jednostavnija varijanta formi koje se mogu jednostavno i brzo implementirati, a baziraju se na direktivama u samoj *html* datoteci kako bi se upravljalo odgovarajućim modelima u sloju logike. Glavni nedostatak ovakvih formi je nedostatak potencijala za skaliranje same forme. Druga varijanta formi se može ostvariti korištenjem modula za reaktivne forme. Reaktivne forme se instanciraju eksplicitno u klasi komponente (pripadajućoj .ts datoteci) te mogu puno bolje odgovoriti na zahtjeve skaliranja, testiranja i ponovnog korištenja. Iako u ovom slučaju ne postoje takvi zahtjevi, svejedno će se koristiti ovakav pristup, jer se smatra kao generalno bolje rješenje za izradu i upravljanje formama u angularu. Još jedan od razloga korištenja ovog modula je taj što se s ovakvim formama puno lakše može implementirati validacija polja za unos.

Za početak je potrebno uvesti modul „*ReactiveFormsModule*“ iz datoteke @angular/forms koja je dio osnovnog angular paketa, i ovime je omogućeno korištenje ovog modula u komponentama aplikacije.

Kako je spomenuto, kod reaktivnih se formi u angularu sama forma instancira eksplicitno u .ts datoteci. Sada je bitno spomenuti i *Lifecycle Hook* pojam. Prema službenoj dokumentaciji angulara, kuke životnog ciklusa (eng. *Lifecycle Hooks*) su metode koje klasa komponente može implementirati kako bi izvršila određeni zadatak u određenoj fazi životnog vijeka komponente. Svaka komponenta svoj život započinje prilikom poziva komponente (generiranja komponente) i završava ga uništavanjem komponente (npr. prilikom redirekcije na neku drugu „stranicu“, odnosno komponentu aplikacije). Instanciranje forme za prijavu potrebno je izvršiti na samom početku životnog vijeka komponente za prijavu. Idealno rješenje za taj problem je korištenje *OnInit* kuke koja se izvršava na samom početku životnog ciklusa komponente, odmah nakon pozivanja konstruktora.

```
export class LoginComponent implements OnInit {
  loginForm: FormGroup = new FormGroup({});
```

```
  constructor() {}
  ngOnInit(): void {
    this.initializeForm();
  }
  initializeForm() {
    this.loginForm = new FormGroup({
      usernameOrEmail: new FormControl('', [
        Validators.required,
        Validators.minLength(6),
        Validators.maxLength(40),
      ]),
      password: new FormControl('', [
        Validators.required,
        Validators.minLength(6),
        Validators.maxLength(20),
      ]),
    });
  }
}
```

Kako bi komponenta mogla izvršiti određeni zadatak u *OnInit* fazi, potrebno je implementirati sučelje *OnInit* i definirati metodu *OnInit*. Konkretan zadatak je definiran u metodi *initializeForm()* kroz koju se stvara instanca nove grupe forme (definirano na razini cijele forme), a potom su definirane kontrole te grupe (na razini elementa za unos). Isto tako, moguće je definirati inicijalnu vrijednost svake kontrole (elementa za unos) kao i jedan ili više validatora za tu kontrolu. Konkretno su definirane dvije kontrole, jedna za unos korisničkog imena ili email-a i jedna za unos lozinke, te su im postavljena ograničenja za obaveznost unosa te minimalnu i maksimalnu duljinu vrijednosti.

9.1.2. HTML prijave

Preostali dio posla je potrebno učiniti u .html datoteci komponente.

```
    <form [formGroup]="loginForm" (ngSubmit)="login()"
autocomplete="off">
      <div class="form-floating">
        <input formControlName="usernameOrEmail" [class.is-invalid]="
            loginForm.get('usernameOrEmail')?.errors &&
            loginForm.get('usernameOrEmail')?.touched
          " type="text" class="form-control" id="floatingInput"
placeholder="name@example.com" />
        <label for="floatingInput">Username or Email</label>
        <div class="invalid-feedback"
*ngIf="loginForm.get('usernameOrEmail')?.hasError('required')">
Please enter a username </div>
        <div class="invalid-feedback"
*ngIf="loginForm.get('usernameOrEmail')?.hasError('minlength')">
Username must contain at least 6 characters </div>
        <div class="invalid-feedback"
*ngIf="loginForm.get('usernameOrEmail')?.hasError('maxLength')">
Username may contain a maximum of 40 characters </div>
      </div>
      <div class="form-floating">
        <input formControlName="password" [class.is-invalid]="
```

```
            loginForm.get('password')?.errors &&
            loginForm.get('password')?.touched
          " type="password" class="form-control"
id="floatingPassword" placeholder="Password" />
        <label for="floatingPassword">Password</label>
        <div class="invalid-feedback"
*ngIf="loginForm.get('password')?.hasError('required')"> Please enter
a password </div>
        <div class="invalid-feedback"
*ngIf="loginForm.get('password')?.hasError('minlength')"> Password
must contain at least 6 characters </div>
        <div class="invalid-feedback"
*ngIf="loginForm.get('password')?.hasError('maxLength')"> Password
may contain a maximum of 20 characters </div>
      </div>
      <div class="form-check text-start my-3"></div>
      <button class="btn btn-primary w-100 py-2"
[disabled]="loginForm.invalid">Log in</button>
      <p class="mt-5 mb-3 text-body-secondary footer"> This
application is part of graduation thesis. <br /> &copy;2023 Tomislav
Periša </p>
    </form>
```

Kao što je moguće primjetiti, elementu forme pridružena je prethodno instancirana direktiva *FormGroup*, a elementima za unos u formi je pridružena odgovarajuća instanca *FormControl* klase korištenjem atributa *formControlName*. Sada je moguće definirati i validaciju samih polja za unos korištenjem direktive za atribut, putem koje je moguće dodati novu css klasu elementu ovisno o trenutnom statusu validacije elementa za unos kojeg nadzire *FormControl* klasa. Za svaki od elemenata za unos, dodani su i elementi koji sadržavaju pomoćnu poruku za korisnika ukoliko trenutno stanje elementa za unos nije validno, kako bi korisnik znao što točno nije u redu s njegovim unosom. Još jedna sitnica koju je potrebno uočiti je da se na atribut *ngSubmit* ,vidljivom na elementu forme, dodala metoda za prijavu, *login()*. Idući korak je implementacija te metode.

9.1.3. Korisnički servis

Ukoliko je forma za prijavu validna, gumb za prijavu je omogućen i korisnik se može prijaviti u aplikaciju. U tom trenutku, potrebno je spremiti podatke koje je korisnik aplikacije unio u formu za prijavu te ih proslijediti na određenu pristupnu točku API servisa. Pošto je korisnik jedan od najvažnijih entiteta u aplikaciji, ima smisla svu logiku vezanu uz praćenje statusa korisnika smjestiti u zaseban servis. *Angular* servisi imaju dvije glavne prednosti pored logike implementirane unutar same klase neke komponente. Prva je ta da se podaci koji su defnirani unutar servisa, čuvaju puno duže nego u slučaju komponenti. Ranije je već pojašnjen pojam životnog vijeka komponente. Prilikom uništavanja komponente, uništeni su i svi pripadajući podaci koje ta komponenta sadrži. U slučaju servisa, stvar je ista ali je zato životni ciklus servisa vezan uz životni ciklus čitave aplikacije, a ne uz jedan manji dio koji se često

mijenja. Servisi se u angularu instanciraju prilikom njihovog prvog poziva, a uništavaju kada se cijela aplikacija ugasi. Druga prednost je ponovna upotrebljivost servisa. Servisi imaju mogućnost „ubacivanja“ u bilo koju drugu komponentu ili modul. To znači da se kod definiran u servisu nemora duplicirati, već se isti servis može prosljediti u više komponenti i tim komponenatama se na taj način dopušta korištenje svih resursa tog servisa.

Kako bi se kod održao što čitkijim, kreirati će se nova mapa *_services* unutar mape *app* (koja predstavlja korijensku mapu aplikacije), te će se unutar te mape kreirati novi servis korištenjem naredbe „ng generate service _services/UserService“. Usporedno će se kreirati i mapa *_models* sa entitetima prijave (login.ts) i korisnika (user.ts). Te datoteke zapravo predstavljaju entitete forme za prijavu i korisnika.

```
export interface Login{
    usernameOrEmail: string;
    password: string;
}

export interface User{
    email: string,
    name: string,
    surname: string,
    token: string,
    username: string
}
```

Prethodno definirani servis će sadržavati sljedeću logiku:

```
export class UserService {
  baseUrl: string = 'https://localhost:7013/api/User/';
  private currentUser = new BehaviorSubject<User | null>(null);
  currentUser$ = this.currentUser.asObservable();
  constructor(private http: HttpClient, private router: Router) {}

  login(model: Login) {
    return this.http.post<User>(this.baseUrl + 'login', model).pipe(
      map((response: User) => {
        const user = response;
        if (user) {
          localStorage.setItem('user', JSON.stringify(user));
          this.currentUser.next(user);
        }
      })
    );
  }

  getUserNameAndSurname(): String {
    let user: string | null = localStorage.getItem('user');
    if (!user) return '';
    let UserObject = JSON.parse(user);
    return UserObject.name + ' ' + UserObject.surname;
  }

  setCurrentUser(user: User) {
    this.currentUser.next(user);
```

```
  }

  logout() {
    localStorage.removeItem('user');
    this.currentUser.next(null);
    this.router.navigateByUrl('/');
  }
}
```

UserService servis ima definirane 2 osnovne metode kojima je implementirana logika za funkcionalnost prijave a to su prijava i odjava. Metoda *login()* kao ulazni parametar prihvaća objekt tipa *Login* (podaci s forme za prijavu) i šalje te podatke na API točku za prijavu. Tu mogućnost (komunikacija s API servisom) pruža modul *HttpClient* kojeg je potrebno uvesti na razini cijele aplikacije i u sam servis, kako bi servis mogao koristiti modul. Zatim se poziva *post()* metoda i na nju se nadovezuje metoda *pipe()* koja služi za tranformaciju ulaza (ulaz je u ovom slučaju vrijednost koju vraća API servis kao odgovor na poslani zahtjev) i *map()* metoda iz RXJS modula kojom se „uočljiva“ vrijednost koju vraća *post()* metoda, transformira i emitira dalje, također kao „uočljiva“. „Uočljiva“ (eng. *Observable*) vrijednost je dio uzorka dizajna kod kojega objekt (zvan subjekt) bilježi listu pretplatnika (drugih objekata ili varijabli) te ih obaviještava o svojoj vrijednosti i stanju kad god se njegovo stanje ili vrijednost promijene. Na taj način se odvija asinkrona komunikacija između različitih dijelova aplikacije. U ovom slučaju, potrebno je obavijestiti komponentu prijave da je korisnik uspješno ulogiran kako bi ga se moglo pravovremeno preusmjeriti na drugu komponentu. Isto tako, vrijednost koja je transformirana metodom *map()*, a koja predstavlja objekt korisnika kojega je vratio servis kao odgovor na zahtjev, sprema se u lokalni spremnik web preglednika, kako bi se mogla dohvatiti po potrebi i pročitati.

9.1.4. Korištenje servisa unutar komponente

Jednom kada je servis implementiran, u klasi komponente za prijavu će se dodati *login()* metoda, koja će se okinuti kada korisnik pritisne gumb za prijavu.

```
login() {
  var loginModel: Login = <Login>{
    usernameOrEmail: this.loginForm.get('usernameOrEmail')?.value,
    password: this.loginForm.get('password')?.value,
  };
  this.userService.login(loginModel).subscribe({
    next: (response) => {
      this.router.navigateByUrl('/home');
    },
    error: (error) => console.log(error),
  });
}
```

Ta metoda će pokupiti podatke iz forme te ih prosljediti metodi za prijavu iz servisa *UserService*. Prilikom te radnje, metoda *login()*, iz komponente za prijavu, će se pretplatiti na metodu *login()* iz servisa kako bi mogla pratiti njezino stanje. Jednom kada se korisnika uspješno prijavi u sustav, metoda *login()* iz komponente za prijavu će se obavijestiti o uspjehu, te će se korisnika preusmjeriti na početnu stranicu.

U njoj je definirana glavna struktura aplikacije koja se sastoji od navigacijske trake i glavnog dijela koji se zamjenjuje odgovarajućom komponentom ovisno o trenutnoj web adresi na kojoj se korisnik nalazi. Ako korisnik nije prijavljen u sustav, odnosno ako „uočljiva“ varijabla *currentUser$* iz servisa *UserService* ima vrijednost *null*, tada se navigacijska traka neće uopće prikazivati. Kako bi korisnik mogao ući u glavni dio aplikacije i koristiti sve funkcionalnosti preko navigacijske trake, mora se prvo uspješno prijaviti. Jednom kada se korisnik uspješno prijavi, ovaj dio aplikacije će se o tome obavijestiti, te će strukturalna direktiva **ngIf* biti ispunjena, pa će i sama navigacijska traka biti prikazana. Jedna zanimljivost ove strukturalne direktive je taj što ukoliko uvjet nije zadovoljen, pripadni element se neće uopće generirati (DOM komponente neće postojati na stranici).

9.2. Funkcionalnost registracije

Za registracijsku komponentu primijenjena je ista tehnika za kreiranje i rad s formama te za njihovu validaciju. Jedina razlika je u tome što registracijska forma sadrži 5 ulaznih polja od kojih je jedna i email adresa koju korisnik mora unijeti, pa se koristi i validator za email adresu.

10. Middleware za iznimke

10.1. Rukovanje iznimkama na strani API servisa

Kako bi se olakšao rad s potencijalnim iznimkama u radu servisa, implementirati će se prilagođeni međuprogram (eng. *Middleware*) za rukovanje iznimkama. Kao prvi korak potrebno je definirati klasu koja će predstavljati izlaznu poruku koja će se slati kao odgovor na http zahtjev. Unutar slojava domene kreirati će se nova mapa i nazvati *Errors*, te će se unutar nje kreirati nova klasa *CustomException*.

```
    public class CustomException
    {
        public CustomException(int statusCode, string message, string
details)
        {
            StatusCode = statusCode;
            Message = message;
```

```
            Details = details;
        }

        public int StatusCode { get; set; }
        public string Message { get; set; }
        public string Details { get; set; }

    }
```

Ta klasa će sadržavati tri bitne informacije o iznimci. Prva je statusni kod iznimke, koja će uvijek biti jedna od mogućih http statusnih kodova. Druga je poruka (sažeta i ljudski lako čitljiva poruka), i treća koja će biti opširna poruka s puno više informacija o tome gdje je točno iznimka nastala.

Zatim je potrebno kreirati novu klasu koja će predstavljati međuprogram. Ta klasa će se kreirati unutar prezentacijskog sloja, unutar novo kreirane mape *Middlewares* i nazvati će se *ExceptionMiddleware*.

```
    private readonly RequestDelegate _next;
    private readonly ILogger<ExceptionMiddleware> _logger;
    private readonly IHostEnvironment _env;

    public ExceptionMiddleware(RequestDelegate next,
ILogger<ExceptionMiddleware> logger, IHostEnvironment env)
    {
        this._next = next;
        this._logger = logger;
        this._env = env;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, ex.Message);
            context.Response.ContentType = "application/json";
            context.Response.StatusCode =
(int)HttpStatusCode.InternalServerError;
            var response = _env.IsDevelopment()
                ? new CustomException(context.Response.StatusCode,
ex.Message, ex.StackTrace?.ToString())
                : new CustomException(context.Response.StatusCode,
ex.Message, "Internal Server Error");
            var options = new JsonSerializerOptions {
PropertyNamingPolicy = JsonNamingPolicy.CamelCase };
            var json = JsonSerializer.Serialize(response, options);
            await context.Response.WriteAsync(json);
        }
    }
```

Ta klasa će zahtjevati parametar *RequestDelegate* (kako bi mogla prosljediti obradu idućem dijelu aplikacije), te klasu koja će odrađivati posao zapisivanja „dnevnika“ (eng. *Log*) u kojemu će biti zapisane sve iznimke koje se dogode, kao i informaciju o trenutnom okruženju kroz parametar *IhostEnvironment* (koji može biti razvoj ili produkcija).

U metodi *InvokeAsync* (koju klasa mora implementirati ukoliko se želi smatrati valjanim međuprogramom), ukoliko se ne naiđe niti na jednu iznimku, klasa će prosljediti trenutni kontekst idućem posredniku u aplikaciji. Ukoliko se naiđe na iznimku, iznimka će se spremiti u dnevnik, te će se stvoriti instanca prethodno kreirane klase *CustomException* i njoj će se proslijediti informacije o iznimci. Kao odgovor na zahtjev, vratiti će se podaci json formata.

10.2. Rukovanje iznimkama na strani klijentske aplikacije

Kako bi se korisniku mogla prikazati poruka o grešci ili iznimci, iskoristiti će se paket *ngx-toast* preko kojega se na vrlo jednostavan način može prikazati tost poruka na zaslonu. Kako bi se mogao koristiti, prvo ga je potrebno instalirati naredbom npm install ngx-toastr@16.2 (zbog verzije angulara koja je 15). Nakon instalacije, potrebno je dodati *toastr* css datoteku pod *style* sekciju unutar *angular.json* datoteke. Posljednji korak je uvesti *ToastrModule* u *app.module.ts* datoteku kako bi postao dostupan cijeloj aplikaciji.

Toastr će poslužiti kod slučajeva kada API servis, kao odgovor klijentskoj aplikaciji, vrati poruku greške. U tom slučaju se ta poruka mora „presresti“ i prosljediti toster modulu kako bi ju prikazao na zaslonu. Angular ima sposobnost kreiranja i korištenja tzv. „presretača“ (eng. *Interceptor*).

Unutar *app* mape će se kreirati nova mapa *_interceptors* i unutar nje će se korištenjem naredbe *ng generate interceptor _interceptors/error* kreirati nova datoteka. Presretač će biti zadužen za presretanje grešaka koje dođu s API servisa i ispisivati će poruku na zaslon preko prethodno dodanog *ToastrModule* modula.

```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  constructor(private toastr: ToastrService) {}

  intercept(
    request: HttpRequest<unknown>,
    next: HttpHandler
  ): Observable<HttpEvent<unknown>> {
    return next.handle(request).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error) {
          this.toastr.error(error.error.message);
        }
```

```
          throw error;
        })
      );
    }
}
```

10.3. Zaštitar navigacijskih ruta

Jedna od posljednjih stavki koja je preostala za implementirati u ovom dijelu rada na projektu je postavljanje zaštite kod preusmjeravanja na različite stranice odnosno komponente. Aplikacija trenutno prikazuje nekoliko komponenti samo u slučaju ako je korisnik prijavljen u sustav. U slučaju kada nije, direktna poveznica na te komponente ne postoji, no to korisnika neće spriječiti da direktno ne pristupi nekoj od tih komponenti preko URL-a. Kako bi se to spriječilo, može se koristiti tzv. „čuvar“ (eng. *Route guard*), odnosno funkcionalnost koja je dio samog angulara.

Za početak će se kreirati nova mapa unutar *app* mape i nazvati *_guards*, a potom će se unutar nje, koristeći naredbu *ng generate guard _guards/authentication* kreirati novi čuvar.

```
export class AuthenticationGuard implements CanActivate {
  userService: UserService = inject(UserService);
  router: Router = inject(Router);
  canActivate():
    | Observable<boolean | UrlTree>
    | Promise<boolean | UrlTree>
    | boolean
    | UrlTree {
    return this.userService.currentUser$.pipe(
      map((user) => {
        if (user) {
          return true;
        } else {
          this.router.navigateByUrl('');
          return false;
        }
      })
    );
  }
}
```

Klasa *AuthenticationGuard* mora pogledati vrijednost varijable *currentUser$* koja se nalazi u servisu *UserService* kako bi znala koji je status prijave korisnika, odnosno je li prijavljen ili nije. Ukoliko je, vraća se vrijednost true (dopušta se navigacija), a ukoliko nije vraća se vrijednost false, odnosno navigacija se zabranjuje i korisnika se prusmjerava na početnu stranicu. Još preostaje definirati putanje koje će čuvar „čuvati“, a to je moguće unutar *app-routing.module.ts* datoteke. Za putanje na koje se želi postaviti čuvara, potrebno je definirati *canActivate* atribut s vrijednošću pripadajućeg čuvara, u ovom slučaju [*AuthenticationGuard*].

Potrebno je dodati još jednog čuvara koji će čuvati komponente za prijavu, registraciju i naslovnu stranicu, ukoliko je korisnik prijavljen. Dodati će se još jedan čuvar i nazvati *NoAuthentication*, i taj će čuvar dopuštati navigaciju ukoliko korisnik nije prijavljen, a ukoliko je, korisnika će se preusmjeriti na glavnu stranicu i ispisati će mu se poruka koristeći *ToastrModule* kako bi mu se dala validna informacija o pogrešci kod navigacije.

11. Funkcionalnost postavki

Kako bi korisnik mogao obrisati svoj račun, morati će prosljediti svoj token koji je za njega kreiran prilikom prijave ili registracije. Taj token će se dakle morati prosljediti servisu kako bi servis iz tokena mogao pročitati identifikator korisnika preko kojega se onda može izvršiti brisanje instance u bazi podataka.

Čitanje podataka iz tokena se na strani servisa može učiniti preko „*User*“ atributa iz *System.Security.Claims* paketa. Taj atribut će sadržavati sve tvrdnje iz prosljeđenog tokena i na taj način se može dohvatiti identifikator korisnika, koji se onda prosljeđuje servisu kao „id“ oznaka.

```
var userId = User.FindFirst(ClaimTypes.Name)?.Value;
```

Na strani klijentske aplikacije, korisnik će imati pristup gumbu za brisanje svog korisničkog računa u komponenti *Settings* (postavke). Jednom kada klikne na gumb pozvati će se metoda za brisanje računa.

```
  deleteAccount() {
    this.userService.deleteAccount()?.subscribe({
      next: (_) => {
        this.toastr.success('Your account has been successfuly
deleted.');
        this.userService.logout();
        this.router.navigateByUrl('');
      }
    });
  }
```

Ta metoda će pak pozivati drugu metodu iz korisničkog servisa koja će poslati http zahtjev API servisu.

```
  deleteAccount() {
    let token = this.getToken();
    let tokenString = 'Bearer ' + token;
    if (!token) return;
    const headersObj = new HttpHeaders({
      'Authorization': tokenString
    });
    const options = { headers: headersObj };
    return this.http.delete(this.baseUrl + 'deleteAccount', options);
  }
```

Ta metoda će iz lokalnog spremnika web preglednika pročitati podatke o korisniku, među kojima će biti i token. Taj token će se onda prosljediti kao parametar zaglavlja http zahtjeva kako bi se mogao pročitati na strani servisa. Jednom kada je zahtjev uspješno proveden, o statusu će se obavijstiti metoda u komponenti postavki, te će se korisnika odjaviti iz sustava i preusmjeriti na naslovnu stranicu, a potom će ga se obavijestiti tost porukom da je brisanje uspješno izvršeno.

12. Funkcionalnost zadataka

12.1. Definiranje modela

Implementacija funkcionalnosti za upravljanje osobnim zadacima će započeti s radom na API servisu. Prvi korak je dodavanje klase modela koji će predstavljati jedan zadatak. Uz glavni model kreirati će se i DTO model zbog unosa novog zadatka (taj model će sadržavati sve atribute glavnog modela osim atributa id).

Glavni model:

```
        public class Event
        {
            [BsonId]
            [BsonRepresentation(BsonType.ObjectId)]
            [Required]
            public string Id { get; set; } =
  ObjectId.GenerateNewId().ToString();
            [Required]
            public string Title { get; set; }
            [Required]
            public DateTime Date { get; set; }
            public string Description { get; set; } = null;
            public bool Status { get; set; } = false;
        }
```

DTO:

```
public class EventVM
{
    [Required]
    public string Title { get; set; }
    [Required]
    public DateTime Date { get; set; }
    public string Description { get; set; } = null;
    public bool Status { get; set; } = false;
}
```

Isto tako, već kreirana klasa *User* koja predstavlja korisnika mora se modificirati kako bi mogla sadržavati listu zadataka.

```
public List<Event> Events { get; set; }
```

Prilikom kreiranja novog korisnika (zahtjev za registraciju), atribut *Events* će biti inicijaliziran kao prazna lista kako bi ga *MongoDb Driver* mogao ispravno mapirati u model baze podataka.

12.2. Repozitorij zadataka

Idući korak je izrada nove klase koja će služiti za pristup repozitoriju zadataka.

```
    public class EventRepository
    {
        private readonly MongoDBService _mongoDBService;

        public EventRepository(MongoDBService mongoDBService)
        {
            _mongoDBService = mongoDBService;
        }

        public async Task AddEventForAUser(string id, Event newEvent)
        {
            FilterDefinition<User> filter = Builders<User>.Filter.Eq("Id",
id);
            UpdateDefinition<User> update =
Builders<User>.Update.AddToSet("Events", newEvent);
            var result = await
_mongoDBService.userCollection.UpdateOneAsync(filter, update);
            if (result.ModifiedCount == 1) return;
            throw new ApplicationException("Internal server error while
trying to add event for a user!");
        }

        public async Task<List<Event>> GetEventsForAUser(string userId)
        {
            var user = await
_mongoDBService.userCollection.AsQueryable().Where(x => x.Id ==
userId).FirstOrDefaultAsync();
            return user.Events;
        }

        public async Task DeleteAnEventForAUser(string userId, string
eventId)
        {
            var pull = Builders<User>.Update.PullFilter(x => x.Events, a =>
a.Id == eventId);
            var filter =
Builders<User>.Filter.And(Builders<User>.Filter.Eq(a => a.Id, userId),
Builders<User>.Filter.ElemMatch(q => q.Events, t => t.Id == eventId));
            var result = await
_mongoDBService.userCollection.UpdateOneAsync(filter, pull);
            return;
        }
        public async Task UpdateAnEventForAUser(string userId, Event
newEvent)
        {
            var filter = Builders<User>.Filter.Eq(x => x.Id, userId) &
Builders<User>.Filter.ElemMatch(x => x.Events, Builders<Event>.Filter.Eq(x
=> x.Id, newEvent.Id));
```

```
            var update = Builders<User>.Update.Set(x =>
x.Events.FirstMatchingElement(), newEvent);
            await _mongoDBService.userCollection.UpdateOneAsync(filter,
update);
            return;
        }
    }
```

EventRepository je zadužen za dodavanje, izmjenu, brisanje i dohvaćanje podataka o zadacima korisnika u bazi podataka. Za dodavanje zadatka, prvo se filtrira kolekcija *User* po id-u korisnika koji je zatražio dodavanje novog zadatka, i potom se u atribut *Events* dodaje novi zadatak. Nakon što se asinkrono vrati rezultat iz baze podataka o statusu izvršene naredbe, provjerava se broj modificiranih zapisa u bazi, te ukoliko je broj 1 (znači da je uspješno dodan novi zapis), kao izlazni parametar se vraća id oznaka novo unesenog zadatka. Za dohvaćanje zadataka, filtrira se kolekcija *User* i dohvaća se jedan korisnik zajedno sa svim zadacima i na kraju se kao izlazni parametar vraća lista zadataka. Brisanje se izvršava preko *PullFilter* metode *MongoDb driver*-a, koja filtrira korisnike kako bi došla do željenog korisnika a potom filtrira i sve zadatke za tog korisnika kako bi došla do željenog zadatka koji se mora obrisati. Izmjena odnosno ažuriranje podataka nekog zadatka se izvršava preko *Update.Set* metode koja prvi pronađeni element u polju mijenja sa prosljeđenim novim zadatkom (prije toga je potrebno definirati i način filtriranja slično kao i kod brisanja zadatka). Na kraju se definicija filtriranja i ažuriranja prosljeđuju metodi *UpdateOneAsync*.

12.3. Servis zadataka

Idući korak je definirati klasu servisa u sloju poslovne logike.

```
    public class EventService
    {
        private readonly EventRepository _eventRepository;
        private readonly UserRepository _userRepository;

        public EventService(EventRepository eventRepository, UserRepository
userRepository)
        {
            _eventRepository = eventRepository;
            _userRepository = userRepository;
        }

        public async Task<List<Event>> GetEventsForAUser(string id)
        {
            if (await UserExists(id))
            {
                return await _eventRepository.GetEventsForAUser(id);
            }
            throw new ArgumentException("User does not exist!");
        }
```

```
        public async Task AddEventForAUser(string id, EventVM newEvent)
        {
            if (await UserExists(id))
            {
                var eventMapped = new Event
                {
                    Date = newEvent.Date,
                    Description = newEvent.Description,
                    Status = newEvent.Status,
                    Title = newEvent.Title
                };
                await _eventRepository.AddEventForAUser(id, eventMapped);
                return;
            }
            throw new ArgumentException("User does not exist!");
        }

        public async Task DeleteAnEventForAUser(string userId, string
eventId)
        {
            if (await UserExists(userId))
            {
                var user = await _userRepository.GetUserById(userId);
                if (user.Events.Exists(x => x.Id == eventId)){
                    await _eventRepository.DeleteAnEventForAUser(userId,
eventId);
                    return;
                }
                throw new ArgumentException("Event does not exist!");
            }
            throw new ArgumentException("User does not exist!");
        }

        public async Task UpdateAnEventForAUser(string userId, Event
updatedEvent)
        {
            if (await UserExists(userId))
            {
                var user = await _userRepository.GetUserById(userId);
                if (user.Events.Exists(x => x.Id == updatedEvent.Id)){
                    await _eventRepository.UpdateAnEventForAUser(userId,
updatedEvent);
                    return;
                }
                throw new ArgumentException("Event does not exist!");
            }
            throw new ArgumentException("User does not exist!");
        }

        private async Task<bool> UserExists(string userId)
        {
            var user = await _userRepository.GetUserById(userId);
            if (user != null) return true;
            return false;
        }
    }
```

Ovaj servis je zadužen za modifikaciju ulaznih podataka koji dolaze iz kontrolera (ukoliko je to potrebno, npr. u metodi za unos novog zadatka potrebno je *UserVM* klasu

pretvoriti u *User* klasu koji predstavlja entitet u bazi podataka) te pozivanje odgovarajuće metode iz repozitorija. Na kraju se rezultat (ukoliko ga ima) vraća nazad prema kontroleru.

12.4. Kontroler za zadatke

Na poslijetku je potrebno definirati i sam kontroler s pristupnim točkama.

```
    [Route("api/[controller]")]
    [Authorize]
    [ApiController]
    public class EventController : ControllerBase
    {
        private readonly EventService _eventService;
        public EventController(EventService eventService)
        {
            _eventService = eventService;
        }

        [HttpGet]
        public async Task<ActionResult<List<Event>>> GetEvents()
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            var eventList = await _eventService.GetEventsForAUser(userId);
            return Ok(eventList);
        }

        [HttpPost]
        public async Task<IActionResult> AddEvent([FromBody] EventVM
newEvent)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _eventService.AddEventForAUser(userId, newEvent);
            return Ok();
        }

        [HttpDelete]
        public async Task<IActionResult> DeleteEvent(string eventId)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _eventService.DeleteAnEventForAUser(userId,eventId);
            return Ok();
        }

        [HttpPut]
        public async Task<IActionResult> UpdateEvent([FromBody] Event
newEvent)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _eventService.UpdateAnEventForAUser(userId, newEvent);
            return Ok();
        }
    }
```

Kontroler ima četiri točke na koje se zahtjevi mogu zaprimiti, i svaka od njih u svojoj metodi zove odgovarajuću metodu iz pripadnog servisa i prosljeđuje im ulazne podatke

zahtjeva. Treba uočiti da je dekorator [*Authorize*] definiran na razini cijele kontroler klase, jer se na taj način ne mora definirati taj isti dekorator za svaku pristupnu točku posebno (ukoliko je definiran na razini cijelog kontrolera, onda se odnosi na sve metode tog kontrolera).

Kako bi aplikacija mogla pravilno i pravovremeno instancirati repozitorij i servis, potrebno ih je dodati u DI kontejner u *Program.cs* datoteci.

```
builder.Services.AddScoped<EventRepository>();
builder.Services.AddScoped<EventService>();
```

12.5. Funkcionalnost zadataka na strani klijenta

12.5.1. Definiranje modela

Na klijentskoj aplikaciji prvo će se izraditi dva modela koji će predstavljati entitete zadataka (razlika između ta dva modela je u polju id, kojeg DTO model neće sadržavati).

Standardni model:

```
export interface Event{
    id: string;
    title: string;
    date: Date;
    description: string | null;
    status: boolean;
}
```

DTO model:

```
export interface EventDTO {
  title: string;
  date: Date;
  description: string | null;
  status: boolean;
}
```

12.5.2. Servis zadataka na strani klijenta

Idući korak je dodavanje *Angular* servisa koji će sadržavati logiku za slanje http zahtjeva prema API servisu.

```
export class EventService {
  baseUrl: string = 'https://localhost:7013/api/Event';
  constructor(private http: HttpClient, private userService:
UserService) { }

  getEvents() {
    let options = this.userService.getHeaderOptions();
    return this.http.get<Event[]>(this.baseUrl,options);
```

```
  }

  updateEvent(event: Event) {
    let options = this.userService.getHeaderOptions();
    return this.http.put(this.baseUrl, event,options);
  }

  deleteEvent(id: string) {
    let options = this.userService.getHeaderOptions();
    return this.http.delete(this.baseUrl+"?eventId="+id, options);
  }

  addEvent(event: EventDTO) {
    let options = this.userService.getHeaderOptions();
    return this.http.post(this.baseUrl, event, options);
  }
}
```

12.5.3. Komponenta zadataka

Taj servis će se koristiti u novoj komponenti koja će predstavljati zaslon za pregled i upravljanje zadacima korisnika.

```
export class EventsComponent implements OnInit {
  events: Event[] = [];
  isNew: boolean = false;
  isEdit: boolean = false;
  eventForm: FormGroup = new FormGroup({});
  eventForEdit: Event | null = null;

  filterStatusList = [
    { value: true, display: 'Completed' },
    { value: false, display: 'Pending' },
  ];
  statusFilter: boolean | null = null;

  constructor(private eventService: EventService) {}

  ngOnInit(): void {
    this.loadEvents();
    this.initializeForm();
  }

  loadEvents() {
    this.eventService.getEvents().subscribe({
      next: (events) => {
        this.events = events;
      },
    });
  }

  initializeForm() {
    this.eventForm = new FormGroup({
      title: new FormControl('', [Validators.required]),
      description: new FormControl('', []),
      date: new FormControl('', [Validators.required]),
      time: new FormControl('', [Validators.required]),
      status: new FormControl('', []),
```

```
  });
}

setInitialFormValues() {
  this.eventForm.setValue({
    title: this.eventForEdit?.title,
    description: this.eventForEdit?.description,
    date: formatDate(this.eventForEdit!.date, 'yyyy-MM-dd', 'en'),
    time: formatDate(this.eventForEdit!.date, 'HH:mm', 'en'),
    status: this.eventForEdit?.status,
  });
}

addNewEvent() {
  this.isNew = true;
  this.isEdit = false;
  this.initializeForm();
}

editEvent(event: Event) {
  this.isEdit = true;
  this.isNew = false;
  this.eventForEdit = event;
  this.initializeForm();
  this.setInitialFormValues();
}

closeForm() {
  this.isNew = false;
  this.isEdit = false;
  this.eventForEdit = null;
}

submitEdit() {
  let date = this.eventForm.get('date')?.value.split('-');
  let time = this.eventForm.get('time')?.value.split(':');
  let updatedEvent = <Event>{
    id: this.eventForEdit?.id,
    title: this.eventForm.get('title')?.value,
    description: this.eventForm.get('description')?.value,
    status: this.eventForm.get('status')?.value,
    date: new Date(date[0], date[1], date[2], time[0], time[1]),
  };
  this.eventService.updateEvent(updatedEvent).subscribe({
    next: () => {
      this.isEdit = false;
      this.eventForEdit = null;
      this.loadEvents();
    },
  });
}

submitAdd() {
  let date = this.eventForm.get('date')?.value.split('-');
  let time = this.eventForm.get('time')?.value.split(':');
  let statusF = this.eventForm.get('status')?.touched
    ? this.eventForm.get('status')?.value
    : false;
  let newEvent = <EventDTO>{
    title: this.eventForm.get('title')?.value,
    description: this.eventForm.get('description')?.value,
```

```
      status: statusF,
      date: new Date(date[0], date[1], date[2], time[0], time[1]),
    };
    this.eventService.addEvent(newEvent).subscribe({
      next: () => {
        this.isNew = false;
        this.loadEvents();
      },
    });
  }

  submitDelete() {
    this.eventService.deleteEvent(this.eventForEdit!.id).subscribe({
      next: () => {
        this.isEdit = false;
        this.eventForEdit = null;
        this.loadEvents();
      },
    });
  }

  manipulateResult() {
    let result = this.events.filter(
      (x) => Boolean(x.status) === Boolean(this.statusFilter)
    );
    this.events = result;
  }

  resetFilters() {
    this.loadEvents();
  }
}
```

Logika komponente sadrži veći broj metoda:

i. Metoda *loadEvents()*: zove servis zadataka kako bi dohvatila sve zadatke za nekog korisnika i rezultat sprema u lokalnu varijablu prema kojoj se puni sadržaj liste svih zadataka koju vidi korisnik
ii. Metoda *initializeForm()*: kreira novu instancu forme za dodavanje novog zadatka ili izmjenu postojećeg
iii. Metoda *setInitialFormValues()*: metoda koja se primarno koristi kako bi se novoj formi proslijedile inicijalne vrijednosti za određeni zadatak i zove se samo prilikom izmjene odnosno ažuriranja zadataka
iv. Metode *addNewEvent()* i *editEvent()*: služe za otvaranje forme za unos novog zadatka ili ažuriranje postojećeg na način da mijenjaju vrijednost lokalnih varijabli *isNew* i *isEdit* čija se vrijednost prati u .html datoteci komponente unutar strukturne direktive **ngIf* za prikaz forme
v. Metoda *closeForm()*: mijenja vrijednost lokalnih varijabli u *false* i forma se time zatvara

vi. Metoda *submitEdit()*: poziva se kod potvrde forme za ažuriranje kako bi se podaci proslijedili servisu zadataka i poslali API servisu na obradu; također sadrži i logiku za pretvorbu podataka iz forme (npr. za datum i vrijeme)

vii. Metoda *submitAdd()*: metoda slična prethodnoj a služi za proslijeđivanje podataka iz forme za unos novog zadatka servisu zadataka

viii. Metode *manipulateResult()* i *resetFilters()*: služe za obradu zahtjeva forme za filtriranje podataka po statusu i postavljanje na inicijalne vrijednosti

13. Funkcionalnost kalendara

Kao jedna od funkcionalnosti web aplikacije, korisniku će se omogućiti pregled njegovih zadataka u obliku kalendara.

13.1. Uvoženje vanjskog modula FullCalendar

Za tu potrebu iskoristiti će se vanjski modul zvan *FullCalendar* koji je besplatan i projekt je otvorenog koda kojeg razvija *FullCalendar LLC*.

Za tu funkcionalnost nije potrebna nikakva dorada na području API servisa, već je funkcionalnost potrebno implementirati na klijentskoj strani. Sam prikaz kalendara će biti zasebna komponenta do koje će se moći doći preko navigacijskog izbornika.

Za početak, potrebno je instalirati @fullcalendar/core, @fullcalendar/angular i @fullcalendar/timegrid node pakete.

Jednom kada su paketi instalirani, u *app.module.ts* datoteku potrebno je uvesti *FullCalendarModule* modul, te jednom kada je modul uvežen, moguće ga je koristiti u novokreiranoj *calendar* komponenti.

FullCalendar ima odličnu i bogatu dokumentaciju koja je korištena i za potrebe ovog rada. *FullCalendar* je zapravo komponenta, koju je potrebno pozvati u .html datoteci glavne komponente, i ona prima jedan ulazni parametar koji je podatkovnog tipa *CalendarOptions*. Taj parametar će se inicijalizirati i postaviti u logici komponente (u njezinoj .ts datoteci) te će se po završetku proslijediti komponenti u .html datoteci.

```
export class CalendarComponent implements OnInit {
  events: Event[] = [];
  calendarOptions: CalendarOptions | null = null;

  constructor(private eventService: EventService) {}
```

```
  ngOnInit(): void {
    let eventsFormatted: any[] = [];
    this.eventService.getEvents().subscribe({
      next: (events) => {
        this.events = events;
        this.events.forEach((element) => {
          eventsFormatted.push({
            title: element.title,
            date: element.date,
            backgroundColor: element.status ? '#18aa50' : '#dc3545',
            borderColor: element.status ? '#18aa50' : '#dc3545',
          });
        });
        this.calendarOptions = <CalendarOptions>{
          plugins: [timeGridPlugin],
          initialView: 'timeGridWeek',
          events: eventsFormatted,
          headerToolbar: {
            left: 'prev,next,today',
            center: 'title',
            right: 'timeGridWeek,timeGridDay',
          },
          slotDuration: '00:30',
          nowIndicator: true,
          allDaySlot: false,
          expandRows: true,
          height: 1000,
          dayHeaderFormat: { weekday: 'short' },
          navLinks: false,
          nowIndicatorClassNames: 'today-indicator'
        };
      },
    });
  }
}
```

CalendarOptions varijabla će se na početku inicijalizirati kao *null* vrijednost. Potom će se u metodi *ngOnInit* prvo pozvati servis za zadatke kako bi se dohvatili svi zadaci nekog korisnika. Jednom kada su podaci dohvaćeni, potrebno ih je mapirati u novi objekt s atributima:

- „*title*“: naslov zadatka koji će biti prikazan u kalendaru
- „*date*“: opisuje datum i vrijeme za kada je zadatak postavljen
- „*backgroundColor*“: specificira pozadinsku boju kartice zadatka u kalendaru (biti će zelen ukoliko je zadatak riješen, a u suprotnom crven)
- „*borderColor*“: specificira boju obruba kartice zadatka (postavljen isto kao i pozadinska boja)

Nakon toga je moguće inicijalizirati novu CalendarOptions varijablu u kojoj će biti opisane slijedeće postavke:

- „*plugins*“: dodatak *FullCalendar* modula koji određuje mogući tip kalendara; u ovom slučaju je iskorišten *timeGridPlugin* kako bi kalendar prikazivao zadatke po satima

- „*initialView*“: postavlja inicijalni tip kalendara koji je definiran u dodatku dodanom kroz prethodnu postavku
- „*events*“: jedna od glavnih postavki u koju se spremaju svi zadaci koje se želi prikazati u kalendaru; u ovom slučaju se koriste prethodno mapirani objekti
- „*headerToolbar*“: postavke za alatnu traku u zaglavlju kalendara
- „*slotDuration*“: frekvencija prikaza zadataka po satima (npr. ukoliko se proslijedi vrijednost '01:00', tada će se zadaci prikazivati po svakom satu)
- Ostali parametri: ostale navedene postavke služe za konfiguraciju nekih manjih detalja *FullCalendar* komponente

Sa definiranim postavkama ostaje još samo proslijediti ih kao ulaz u komponentu unutar .html datoteke.

```
<div class="main-screen darkback boundaries">
  <full-calendar *ngIf="this.calendarOptions"
[options]="this.calendarOptions" style="max-width: 89vw;max-height:
89vh;"></full-calendar>
</div>
```

14. Funkcionalnost „brzih“ bilješki

Iduća funkionalnost koja će se implementirati je mogućnost korisnika da vodi bilješke. Bilješke će biti podijeljene na dvije skupine. Prva skupina će biti tzv. „brze“ bilješke koje će služiti korisniku da na brzinu napiše nekakav tekst te ga pregledava i izmjenjuje po želji. Brze bilješke će efektivno biti vezane uz samo jedan „dokument“. Funkcionalnost nakon ove, će biti vezana uz ovu drugu skupinu bilješki, koje će biti realizirane kao dokumenti ili bilježnice koje će korisnik moći voditi. Te dvije funkcionalnosti će biti vrlo slične, a glavna razlika će biti u tome da će kroz brze bilješke korisnik održavati samo jedan dokument, a kroz bilježnice više njih.

14.1. Definiranje modela na strani API servisa

Na strani API servisa, dodati će se nova DTO klasa koja će sadržavati samo jedan atribut a to je sadržaj brzih bilješki.

```
public class QuickNoteVM
{
    public string Content { get; set; } = null;
}
```

Posljedično se odmah mora izmijeniti i klasa korisnika kako bi mogla sadržavati atribut sa sadržajem brzih bilješki.

```
public string QuickNotes { get; set; } = null;
```

14.2. Repozitorij brzih bilješki

Sada je potrebno implementirati standardne klase koje će predstavljati repozitorij, servis i kontroler za rad s brzim bilješkama.

```
    public class QuickNoteRepository
    {
        private readonly MongoDBService _mongoDBService;

        public QuickNoteRepository(MongoDBService mongoDBService)
        {
            _mongoDBService = mongoDBService;
        }

        public async Task<string> GetQuickNotes(string userId)
        {
            var user = await
_mongoDBService.userCollection.AsQueryable().Where(x => x.Id ==
userId).FirstOrDefaultAsync();
            return user.QuickNotes;
        }

        public async Task DeleteQuickNotes(string userId)
        {
            var filter = Builders<User>.Filter.Eq(x => x.Id, userId);
            var update = Builders<User>.Update.Set(x => x.QuickNotes, "");
            await _mongoDBService.userCollection.UpdateOneAsync(filter,
update);
            return;
        }
        public async Task UpdateQuickNotes(string userId, string
quickNotes)
        {
            var filter = Builders<User>.Filter.Eq(x => x.Id, userId);
            var update = Builders<User>.Update.Set(x => x.QuickNotes,
quickNotes);
            await _mongoDBService.userCollection.UpdateOneAsync(filter,
update);
            return;
        }
    }
```

Klasa koja predstavlja repozitorij ima samo tri metode. Metoda *GetQuickNotes* sadrži logiku za dohvaćanje vrijednosti brzih bilješki za određenog korisnika, metoda *DeleteQuickNotes* postavlja vrijednost atributa „*QuickNotes*“ kao praznu string vrijednost, i metoda *UpdateQuickNotes* sadrži logiku za ažuriranje vrijednosti atributa „*QuickNotes*“.

14.3. Servis brzih bilješki

```
    public class QuickNoteService
    {
        private readonly UserService _userService;
```

```
        private readonly QuickNoteRepository _quickNoteRepository;

        public QuickNoteService(QuickNoteRepository quickNoteRepository,
UserService userService)
        {
            _quickNoteRepository = quickNoteRepository;
            _userService = userService;
        }

        public async Task<QuickNoteVM> GetQuickNotes(string userId)
        {
            var userExists = await _userService.UserExistsById(userId);
            if (userExists.Item1)
            {
                var content = await
_quickNoteRepository.GetQuickNotes(userId);
                if(content!=null) return new QuickNoteVM { Content =
content };
                return new QuickNoteVM { Content = null };
            }
            throw new ArgumentException("User does not exist!");
        }

        public async Task DeleteQuickNotes(string userId)
        {
            var userExists = await _userService.UserExistsById(userId);
            if (userExists.Item1)
            {
                await _quickNoteRepository.DeleteQuickNotes(userId);
                return;
            }
            throw new ArgumentException("User does not exist!");
        }

        public async Task UpdateQuickEvents(string userId, QuickNoteVM
quickNotes)
        {
            var userExists = await _userService.UserExistsById(userId);
            if (userExists.Item1)
            {
                await _quickNoteRepository.UpdateQuickNotes(userId,
quickNotes.Content);
                return;
            }
            throw new ArgumentException("User does not exist!");
        }
    }
```

Servis za rad s brzim bilješkama koristi prethodno kreirani repozitorij kako bi mogao izvršavati naredbe nad bazom podataka, kao i već postojeći *UserService* kako bi mogao provjeravati postoji li korisnik, koji želi izvršiti neku naredbu, u bazi podataka.

14.4. Kontroler za brze bilješke

```
    [Authorize]
    [Route("api/[controller]")]
```

```
    [ApiController]
    public class QuickNoteController : ControllerBase
    {
        private readonly QuickNoteService _quickNoteService;
        public QuickNoteController(QuickNoteService quickNoteService)
        {
            _quickNoteService = quickNoteService;
        }

        [HttpGet]
        public async Task<ActionResult<QuickNoteVM>> GetQuickNotes()
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            var quickNotes = await _quickNoteService.GetQuickNotes(userId);
            return Ok(quickNotes);
        }

        [HttpDelete]
        public async Task<IActionResult> DeleteQuickNotes()
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _quickNoteService.DeleteQuickNotes(userId);
            return Ok();
        }

        [HttpPut]
        public async Task<IActionResult> UpdateEvent([FromBody] QuickNoteVM
quickNotes)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _quickNoteService.UpdateQuickEvents(userId, quickNotes);
            return Ok();
        }
    }
```

Posljednji zadatak za ovu funkcionalnost na API dijelu jest dodati novokreirani repozitorij i servis u DI kontejner.

```
builder.Services.AddScoped<QuickNoteRepository>();
builder.Services.AddScoped<QuickNoteService>();
```

14.5. Brze bilješke na strani klijenta

14.5.1. Definiranje modela

Na strani korisnika, za početak će se definirati novi DTO tip podatka koji će predstavljati sadržaj brzih bilješki koji će se slati prilikom ažuriranja te primati prilikom poziva pristupne točke GET.

```
export interface QuickNoteDTO{
    content: string | null;
}
```

14.5.2. Definiranje angular servisa

Nakon toga će se definirati *Angular* servis koji će sadržavati logiku za generiranje Http zahtjeva i njihovo slanje prema API servisu.

```
export class QuickNoteService {
  baseUrl: string = 'https://localhost:7013/api/QuickNote';
  constructor(private http: HttpClient, private userService:
UserService) {}

  getQuickNote() {
    let options = this.userService.getHeaderOptions();
    return this.http.get<QuickNoteDTO>(this.baseUrl, options);
  }

  updateQuickNote(quickNote: QuickNoteDTO) {
    let options = this.userService.getHeaderOptions();
    return this.http.put(this.baseUrl, quickNote, options);
  }

  deleteQuickNote() {
    let options = this.userService.getHeaderOptions();
    return this.http.delete(this.baseUrl, options);
  }
}
```

14.5.3. Komponenta brzih bilješki

Jednom kada je servis implementiran i omogućena razmjena podataka za brze bilješke s API servisom, preostalo je još samo dodati komponentu.

```
export class QuickNotesComponent implements OnInit {
  htmlContent: string | null = null;
  showEditor = false;
  editorConfig: AngularEditorConfig | null = null;

  constructor(
    private quickNoteService: QuickNoteService,
    private toastr: ToastrService
  ) {}

  ngOnInit(): void {
    this.getQuickNote();
  }

  setRTEOptions() {
    this.editorConfig = <AngularEditorConfig>{
      editable: true,
      spellcheck: false,
      height: 'auto',
      minHeight: '700px',
      maxHeight: 'auto',
      width: 'auto',
      enableToolbar: true,
      showToolbar: true,
      placeholder: 'Enter text here...',
```

```
      defaultParagraphSeparator: '',
      defaultFontName: 'Arial',
      fonts: [
        { class: 'arial', name: 'Arial' },
        { class: 'times-new-roman', name: 'Times New Roman' },
        { class: 'calibri', name: 'Calibri' },
        { class: 'comic-sans-ms', name: 'Comic Sans MS' },
      ],
      sanitize: true,
      toolbarPosition: 'top',
      toolbarHiddenButtons: [
        [
        ],
        [
          'customClasses',
          'insertImage',
          'insertVideo',
          'toggleEditorMode',
        ],
      ],
    };
  }

  getQuickNote() {
    this.quickNoteService.getQuickNote().subscribe({
      next: (response: QuickNoteDTO) => {
        if (response != null) this.htmlContent = response.content;
        else this.htmlContent = '';
        this.showEditor = true;
        this.setRTEOptions();
      },
    });
  }

  saveQuickNote() {
    let update: QuickNoteDTO = { content: this.htmlContent };
    this.quickNoteService.updateQuickNote(update).subscribe({
      next: () => {
        this.toastr.success('Quick note successfully saved.');
      },
    });
  }

  deleteQuickNote() {
    this.quickNoteService.deleteQuickNote().subscribe({
      next: () => (this.htmlContent = ''),
    });
  }
}
```

Za pisanje brzih bilješki koristiti će se open source modul zvan @kolkov/angular-editor. Kako bi se modul mogao koristiti, potrebno je instalirati paket @kolkov/angular-editor koristeći *Node package manager* koji je već ranije korišten. Nakon toga je unutar *app.module.ts* datoteke potrebno uvesti modul i time je korištenje paketa omogućeno u svim komponentama.

Slično kao i kod *FullCalendar* paketa, komponenti *angular-editor* potrebno je poslati opcije odnosno konfiguraciju samog uređivača. Definiranje opcija odrađeno je prilikom inicijalizacije same komponente metodom *setRTEOptions()*.

Preostale definirane metode služe za dohvaćanje podataka o brzim bilješkama (sadržaju) te brisanje i spremanje promjena (zadnje dvije spomenute metode se pozivaju klikom na gumbove koji se nalaze u zaglavlju stranice odnosno komponente).

15. Funkcionalnost bilježnica

Posljednja funkcionalnost koja je već spomenuta će biti vrlo slična brzim bilješkama, a razlikovati će se po tome što će korisnik moći imati više bilježnica u isto vrijeme koje će moći kreirati, uređivati i brisati.

15.1. Definiranje modela

Za početak će se dodati 3 nova entiteta.

```
public class Notebook
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    [Required]
    public string UserId { get; set; }
    [Required]
    public string Title { get; set; }
    public string Content { get; set; } = null;
}
```

Klasa *Notebook* će predstavljati model u bazi podataka. Taj model će biti spremljen u zasebnoj *Mongo DB* kolekciji.

```
public class NotebookVM
{
    [Required]
    public string Title { get; set; }
    public string Content { get; set; } = null;
}
```

Klasa *NotebookVM* će se koristiti prilikom kreiranja novog *Notebook* objekta i ona ne sadrži atribute *Id* (on se generira automatski u bazi podataka prema indeksu), i *UserId* (jer korisnik nema pristup svojoj identifikacijskoj oznaci jer je ona kriptirana i spremljena u token te ga prema tome može čitati samo API servis).

```
public class UpdateNotebookVM
{
    [Required]
```

```
        public string Id { get; set; }
        [Required]
        public string Title { get; set; }
        public string Content { get; set; } = null;
    }
```

Klasa *UpdateNotebookVM* će se slati prema API servisu kada korisnik želi ažurirati neku bilježnicu i on ne sadrži atribut *UserId* zbog istog razloga kao i kod klase *NotebookVM*. Idući korak je registracija nove kolekcije u *MongoDBService*, *MongoDBOptions* i *appsettings.json* datotekama.

```
        public string NotebookCollectionName { get; set; } = null;
```

Dodan je novi atribut klasi *MongoDBSettings*.

```
            notebookCollection = database.GetCollection<Notebook>(mongoDBSettings.Value.NotebookCollectionName);
```

U *MongoDBService* klasi je dodana nova kolekcija koja se čita iz *appsettings.json* datoteke pa je naziv kolekcije u bazi podataka potrebno dodati i tamo.

```
      "NotebookCollectionName": "Notebook"
```

15.2. Repozitorij bilježnica

Sada je još preostalo dodati repozitorij, servis i kontroler klase koje će izvršavati zadatke vezane uz upravljanje bilježnicama nekog korisnika.

```
    public class NotebookRepository
    {
        private readonly MongoDBService _mongoDBService;

        public NotebookRepository(MongoDBService mongoDBService)
        {
            _mongoDBService = mongoDBService;
        }

        public async Task AddNotebookForAUser(Notebook notebook)
        {
            await
_mongoDBService.notebookCollection.InsertOneAsync(notebook);
            return;
        }

        public async Task<List<Notebook>> GetNotebooksByAUserId(string
userId)
        {
            var notebooks = await
_mongoDBService.notebookCollection.AsQueryable().Where(x => x.UserId ==
userId).ToListAsync();
```

```
            if (notebooks.Count > 0) return notebooks;
            return null;
        }

        public async Task DeleteNotebook(string noteBookId)
        {
            FilterDefinition<Notebook> filter =
Builders<Notebook>.Filter.Eq("Id", noteBookId);
            var result = await
_mongoDBService.notebookCollection.DeleteOneAsync(filter);
            return;
        }

        public async Task DeleteAllNotebooksForAUser(string userId)
        {
            FilterDefinition<Notebook> filter =
Builders<Notebook>.Filter.Eq("UserId", userId);
            var result = await
_mongoDBService.notebookCollection.DeleteManyAsync(filter);
            return;
        }

        public async Task UpdateNotebook(Notebook notebook)
        {
            var filter = Builders<Notebook>.Filter.Eq(x => x.Id,
notebook.Id);
            await
_mongoDBService.notebookCollection.ReplaceOneAsync(filter, notebook);
            return;
        }

        public async Task<Notebook> GetANotebookById(string notebookId)
        {
            var notebook = await
_mongoDBService.notebookCollection.AsQueryable().Where(x => x.Id ==
notebookId).FirstOrDefaultAsync();
            if (notebook != null) return notebook;
            return null;
        }
    }
```

NotebookRepository klasa sadrži logiku vezanu uz CRUD (eng. *Create, read, update, delete*) operacije nad bilježnicama u njihovoj kolekciji te sadrži neke druge korisne i pomoćne metode poput metode za dohvaćanje bilježnice po njezinoj identifikacijskoj oznaci i brisanja svih bilježnica za nekog korisnika koja će se pozivati ako korisnik odluči obrisati svoj račun.

15.3. Servis bilježnica

```
    public class NotebookService
    {
        private readonly NotebookRepository _notebookRepository;
        private readonly UserService _userService;

        public NotebookService(NotebookRepository notebookRepository,
UserService userService)
```

```
        {
            _notebookRepository = notebookRepository;
            _userService = userService;
        }

        public async Task<List<Notebook>> GetNotebooksForAUser(string
userId)
        {
            var notebooks = await
_notebookRepository.GetNotebooksByAUserId(userId);
            if (notebooks!=null) return notebooks;
            return new List<Notebook>();
        }

        public async Task AddANotebookForAUser(string userId, NotebookVM
newNotebook)
        {
            var user = await _userService.UserExistsById(userId);
            if (user.Item1)
            {
                var notebookMapped = new Notebook
                {
                    UserId = userId,
                    Title = newNotebook.Title,
                    Content = newNotebook.Content
                };
                await
_notebookRepository.AddNotebookForAUser(notebookMapped);
                return;
            }
            throw new ArgumentException("User does not exist!");
        }

        public async Task DeleteANotebookForAUser(string notebookId)
        {
            await _notebookRepository.DeleteNotebook(notebookId);
            return;
        }

        public async Task UpdateANotebookForAUser(string
userId,UpdateNotebookVM updatedNotebook)
        {
            var notebookExists = await NotebookExists(updatedNotebook.Id);
            if (notebookExists.Item1)
            {
                var update = new Notebook
                {
                    Id = updatedNotebook.Id,
                    Content = updatedNotebook.Content,
                    Title = updatedNotebook.Title,
                    UserId = userId
                };
                await _notebookRepository.UpdateNotebook(update);
                return;
            }
            throw new ArgumentException("Notebook does not exist!");
        }

        public async Task<Tuple<bool, Notebook>> NotebookExists(string
notebookId)
        {
```

```
            var notebook = await
_notebookRepository.GetANotebookById(notebookId);
            if (notebook != null)
            {
                return new Tuple<bool, Notebook>(true, notebook);
            }
            return new Tuple<bool, Notebook>(false, null);
        }
    }
```

Servis za upravljanje bilježnicama sadrži repozitorij kao privatnu varijablu kojoj prosljeđuje određene zahtjeve za izvršavanje upita nad bazom podataka.

15.4. Kontroler za bilježnice

```
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class NotebookController : ControllerBase
    {
        private readonly NotebookService _notebookService;
        public NotebookController(NotebookService notebookService)
        {
            _notebookService = notebookService;
        }

        [HttpGet]
        public async Task<ActionResult<List<Notebook>>> GetNotebooks()
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            var notebooks = await
_notebookService.GetNotebooksForAUser(userId);
            return Ok(notebooks);
        }

        [HttpPost]
        public async Task<IActionResult> AddANotebook([FromBody] NotebookVM
notebook)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await _notebookService.AddANotebookForAUser(userId, notebook);
            return Ok();
        }

        [HttpDelete]
        public async Task<IActionResult> DeleteANotebook(string notebookID)
        {
            await _notebookService.DeleteANotebookForAUser(notebookID);
            return Ok();
        }

        [HttpPut]
        public async Task<IActionResult> UpdateANotebook([FromBody]
UpdateNotebookVM notebook)
        {
            var userId = User.FindFirst(ClaimTypes.Name)?.Value;
            await
_notebookService.UpdateANotebookForAUser(userId,notebook);
            return Ok();
```

```
    }
}
```

Posljednja dodana klasa je sam kontroler na koji će pristizati zahtjevi za bilježnice iz korisničke aplikacije te će se zadaci proslijeđivati pripadnom servisu. Za kraj je još preostalo dodati novokreirane klase repozitorija i servisa u DI kontejner kako bi ih servis mogao točno inicijalizirati i koristiti prema potrebi.

```
builder.Services.AddScoped<NotebookRepository>();
builder.Services.AddScoped<NotebookService>();
```

15.5. Bilježnice na strani klijenta

Na strani klijentske aplikacije dodana su tri sučelja koja su identična entitetima na strani API servisa.

15.5.1. Servis na strani klijenta

Nakon toga potrebno je dodati *Angular* servis koji će služiti za slanje http zahtjeva prema API servisu, slično kao i kod ostalih funkcionalnosti.

```
export class NotebookService {
  baseUrl: string = 'https://localhost:7013/api/Notebook';
  constructor(private http: HttpClient, private userService:
UserService) {}

  getNotebooks() {
    let options = this.userService.getHeaderOptions();
    return this.http.get<Notebook[]>(this.baseUrl, options);
  }

  updateNotebook(notebook: UpdateNotebookDTO) {
    let options = this.userService.getHeaderOptions();
    return this.http.put(this.baseUrl, notebook, options);
  }

  deleteNotebook(id: string) {
    let options = this.userService.getHeaderOptions();
    return this.http.delete(this.baseUrl + '?notebookID=' + id,
options);
  }

  addNotebook(notebook: NotebookDTO) {
    let options = this.userService.getHeaderOptions();
    return this.http.post(this.baseUrl, notebook, options);
  }
}
```

15.5.2. Komponenta za bilježnice

Završni dio je kreirati posljednju komponentu koja će biti korištena u aplikaciji, a služiti će korisniku za upravljanje vlastitim dokumentima odnosno bilježnicama.

```
export class NotebooksComponent implements OnInit {
  itemNotSelected: string =
    'list-group-item list-group-item-action d-flex gap-3 py-3 item
itemMargin';
  itemSelected: string =
    'list-group-item list-group-item-action d-flex gap-3 py-3 item
itemMargin itemSelected';
  notebooks: Notebook[] | null = null;
  selectedNotebook: Notebook | null = null;
  isEdit: boolean = false;
  isNew: boolean = false;
  notebookForm: FormGroup = new FormGroup({});
  editorConfig: AngularEditorConfig | null = null;
  listDisplayed: boolean = true;

  constructor(private notebookService: NotebookService) {}

  ngOnInit(): void {
    this.getNotebooks();
  }

  getNotebooks() {
    this.notebookService.getNotebooks().subscribe({
      next: (notebooks) => {
        this.notebooks = notebooks;
        console.log(notebooks);
      },
    });
  }

  initializeForm() {
    this.notebookForm = new FormGroup({
      title: new FormControl('', [Validators.required]),
      content: new FormControl('', []),
    });
  }

  setInitialFormValues() {
    this.notebookForm.setValue({
      title: this.selectedNotebook?.title,
      content: this.selectedNotebook?.content,
    });
  }

  selectNotebook(selected: Notebook) {
    this.selectedNotebook = selected;
    if (this.isEdit) {
      this.initializeForm();
      this.setRTEOptions();
      this.setInitialFormValues();
    } else this.isEdit = false;
    this.isNew = false;
```

```
  }
  addNotebook() {
    this.initializeForm();
    this.setRTEOptions();
    this.isEdit = false;
    this.isNew = true;
    this.selectedNotebook = null;
  }
  updateNotebook() {
    this.initializeForm();
    this.setRTEOptions();
    this.setInitialFormValues();
    this.isNew = false;
    this.isEdit = true;
  }

  deleteNotebook() {
    this.notebookService.deleteNotebook(this.selectedNotebook!.id).su
bscribe({
      next: () => {
        this.selectedNotebook = null;
        this.getNotebooks();
      },
    });
  }

  submitEdit() {
    let updatedNotebook = <UpdateNotebookDTO>{
      id: this.selectedNotebook?.id,
      title: this.notebookForm.get('title')?.value,
      content: this.notebookForm.get('content')?.value,
    };
    this.notebookService.updateNotebook(updatedNotebook).subscribe({
      next: () => {
        this.isEdit = false;
        this.selectedNotebook = null;
        this.getNotebooks();
      },
    });
  }
  submitAdd() {
    let newNotebook = <NotebookDTO>{
      title: this.notebookForm.get('title')?.value,
      content: this.notebookForm.get('content')?.value,
    };
    this.notebookService.addNotebook(newNotebook).subscribe({
      next: () => {
        this.isNew = false;
        this.getNotebooks();
      },
    });
  }

  closeForm() {
    this.isEdit = false;
    this.isNew = false;
  }

  changeListDisplay() {
    this.listDisplayed = !this.listDisplayed;
  }
```

```
}
```

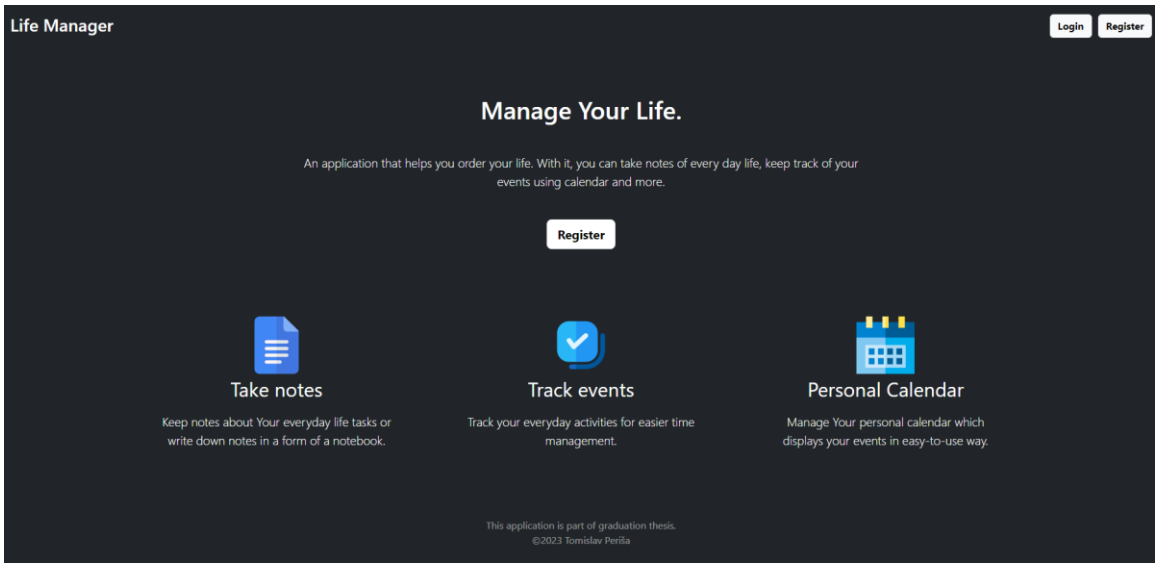
Metode koje koristi ova komponenta su:

- *getNotebooks()*: služi za dohvaćanje svih bilježnica nekog korisnika (poziv metode iz pripadnog servisa)
- *initializeForm()*: služi za kreiranje nove instance forme a poziva se prilikom poziva same forme
- *setInitialFormValues()*: postavlja zadane vrijednosti kontrolama forme kod ažuriranja podataka bilježnice
- *selectNotebook(Notebook)*: poziva se kada korisnik odabere jednu bilježnicu iz liste svojih bilježnica
- *addNotebook()*, *updateNotebook()*: služe za postavljanje zastavica komponente kako bi se dijelovi komponente dinamički prikazivali ili skrivali
- *deleteNotebook()*, *submitEdit()* i *submitAdd()*: pozivaju metode pripadnog servisa kako bi se željena akcija mogla provesti
- *closeForm()*: mijenja zastavice komponente kako bi se forma za uređivanje ili dodavanje nove bilježnice mogla sakriti
- *setRTEOptions()*: metoda identična istoimenoj metodi u komponenti brzih billješki
- *changeListDisplay()*: prikazuje ili skriva listu svih bilježnica kako bi uređivač bilježnice mogao zauzeti veći dio ekrana

Za ovu funkcionalnost iskorišten je već korišteni modul *angular-editor* kako bi korisnik mogao uređivati tekst s naprednijim opcijama.

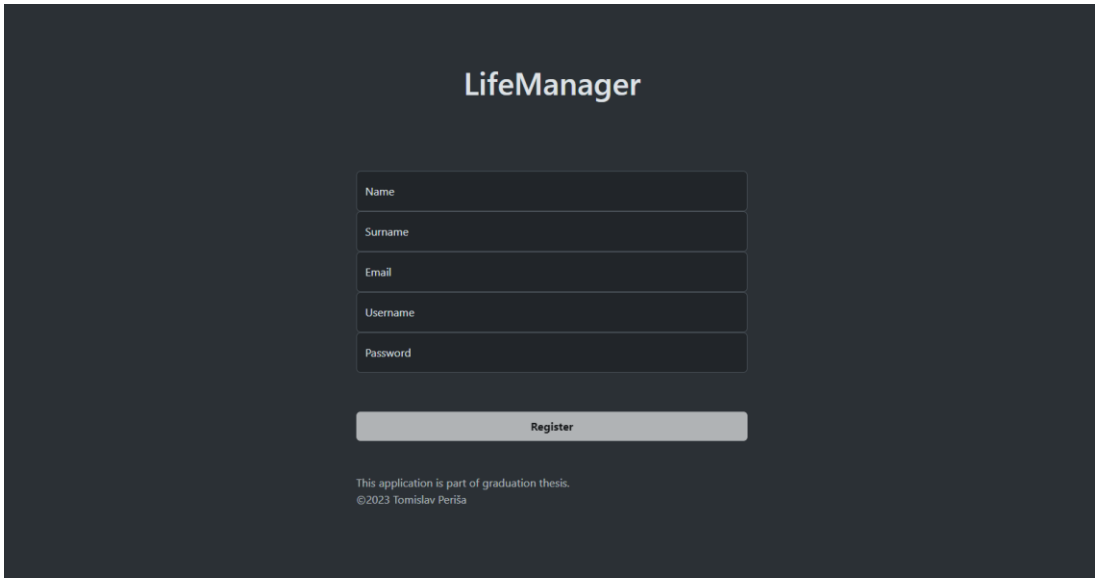
16. Prikaz aplikacije

O ovom poglavlju prikazati će se izgled same klijentske aplikacije i opisati njezin rad.



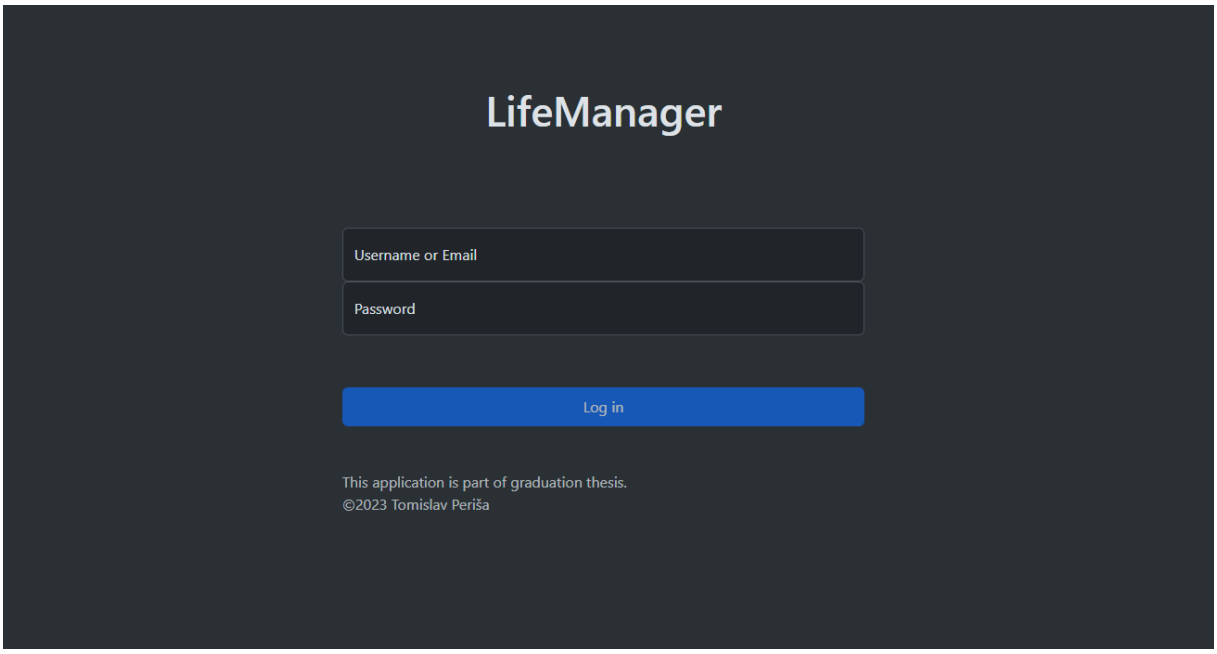
Slika 10. Naslovna stranica [Autorski rad]

Kada korisnik po prvi puta dođe na stranicu, preusmjerava se na naslovnu stranicu na kojoj se nalaze osnovne informacije i funkcionalnosti aplikacije. S naslovne stranice korisnik se može navigirati na stranicu za registraciju ili za prijavu, ali nikako nemože doći do navigacijske trake ili stranica koje sadržavaju funkcionalnosti.



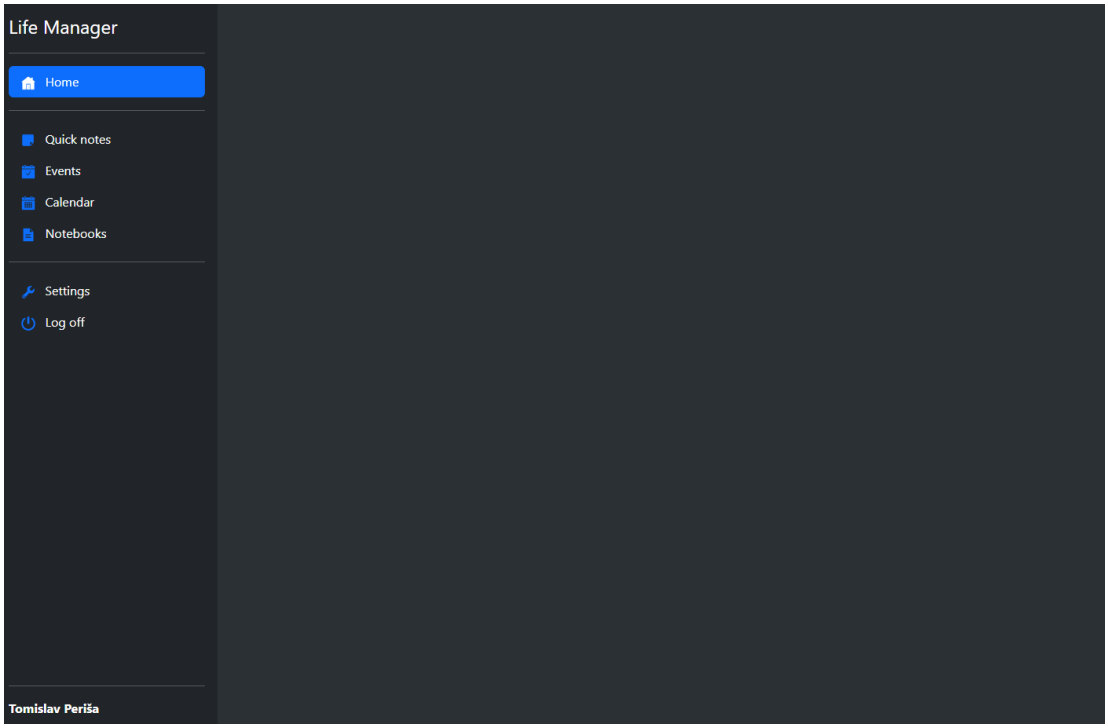
Slika 11. Registracijska stranica [Autorski rad]

Na registracijskoj se stranici (komponenti) nalazi reaktivna forma koja zahtjeva validan unos kroz sva polja. Jednom kada je korisnik unio sve potrebne informacije kroz formu, gumb u podnožju forme postaje aktivan i klikom na taj gumb, korisnik može stvoriti novi korisnički račun i prijaviti se u aplikaciju.



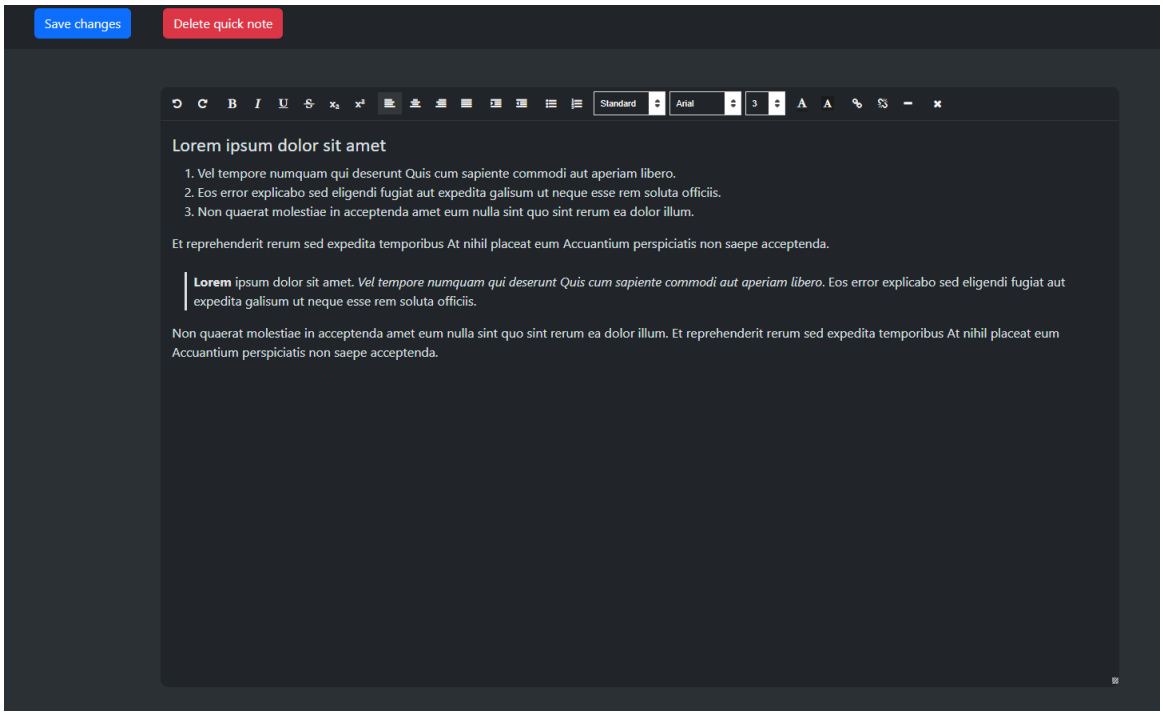
Slika 12. Stranica za prijavu [Autorski rad]

Na stranici za prijavu, nalazi se vrlo slična reaktivna forma ali ovoga puta ona služi za prijavu u aplikaciju. Ona zahtijeva unos korisničkog imena ili elektronske pošte i lozinke. Ukoliko su kredencijali validni, korisnika će se prijaviti u aplikaciju i preusmjeriti na glavnu stranicu s navigacijskim izbornikom.



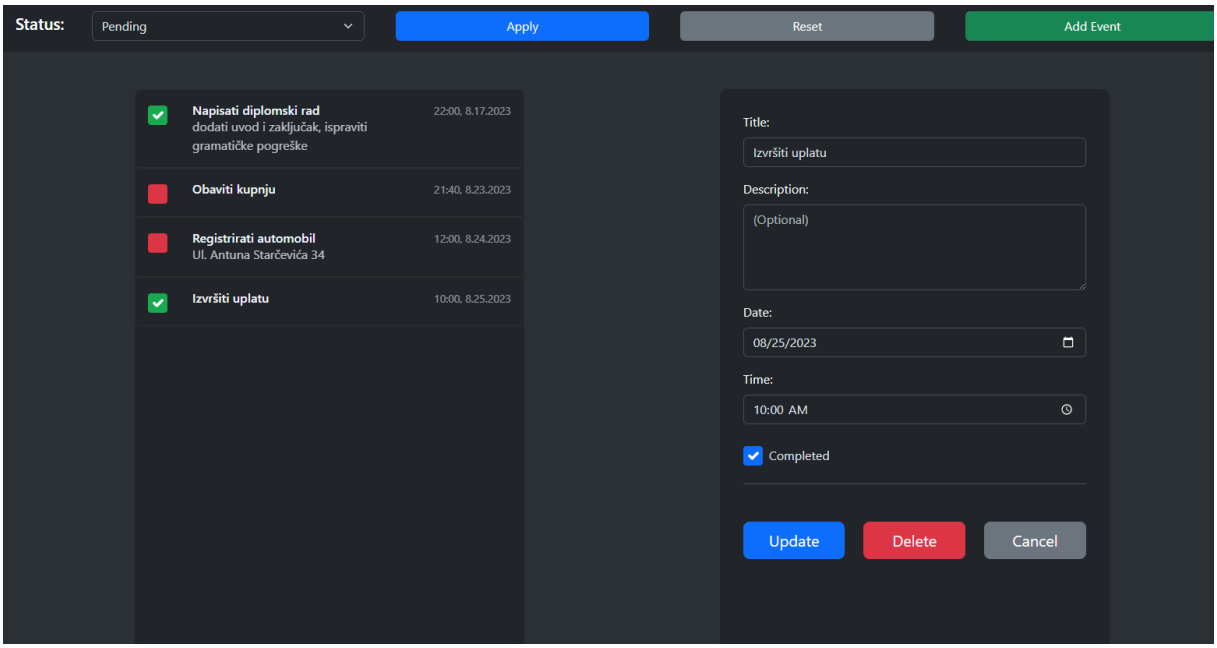
Slika 13. Glavna stranica [Autorski rad]

Jednom kada je korisnik uspješno prijavljen u sustav, na raspolaganju s lijeve strane ekrana biti će mu dostupan navigacijski izbornik sa svim mogućim funkcionalnostima. Klikom na jednu od funkcionalnosti korisnika će se preusmjeriti na odgovarajuću stranicu. Predzadnja opcija u izborniku je opcija za prikaz postavki, u kojoj se nalazi opcija za brisanje korisničkog računa. Posljednja opcija u izborniku je opcija za odjavu iz aplikacije kojom se korisnik može odjaviti iz aplikacije i u tom trenutku će ga se preumsjeriti nazad na naslovnu stranicu.



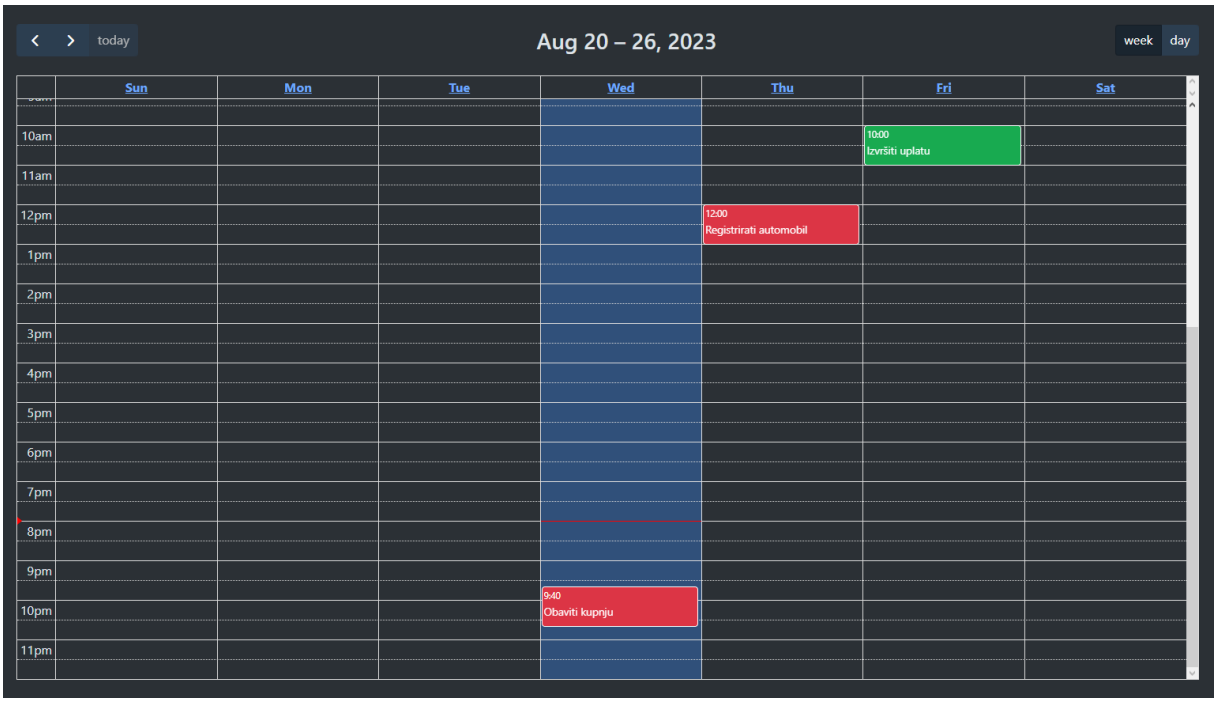
Slika 14. Brze bilješke [Autorski rad]

Ukoliko korisnik odabere brze bilješke, dobiti će prikaz svojih brzih bilješki koje može uređivati preko uređivača u zaglavlju elementa za prikaz teksta. Jednom kada je gotov s uređivanjem, svoje promjene može trajno spremiti klikom na gumb „*Save changes*“ u zaglavlju stranice. Druga opcija mu je brisanje brzih bilješki pritiskom na gumb „*Delete quick note*“.



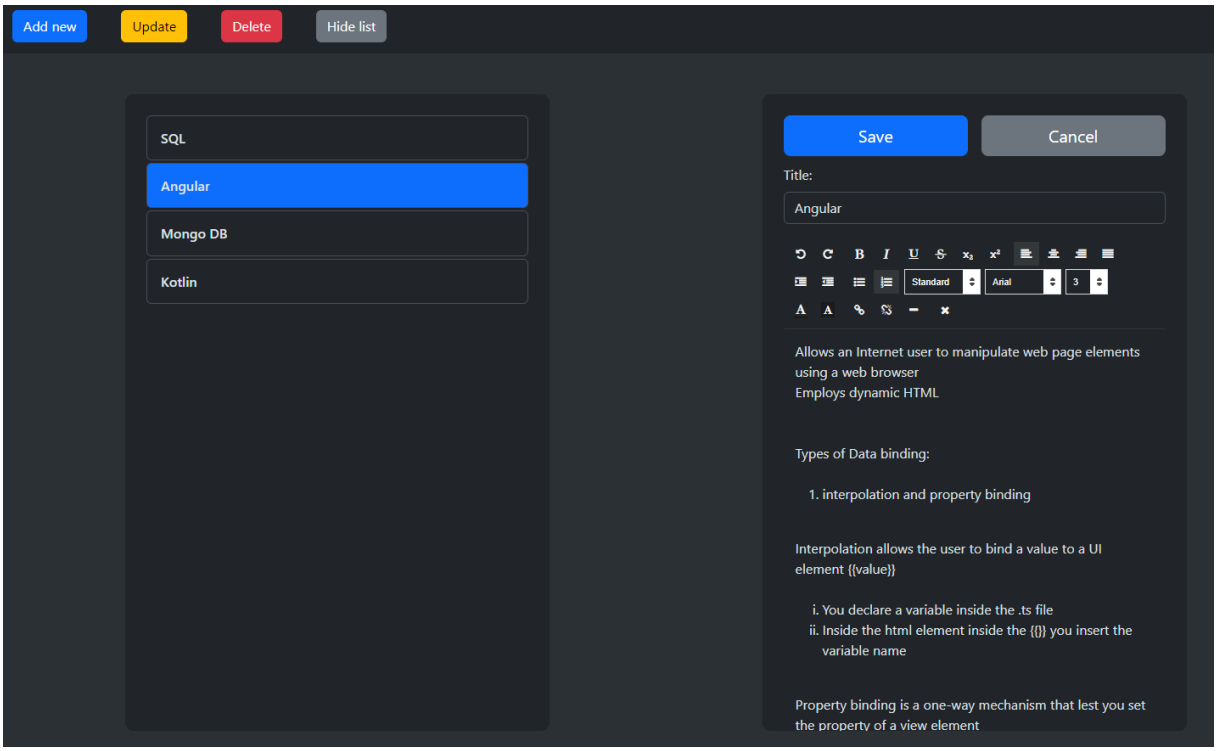
Slika 15. Stranica za zadatke [Autorski rad]

Na stranici sa zadacima, korisnik može pregledavati sve dodane zadatke uključujući njihov naslov, opis i status te datum za kada je određeni zadatak postavljen. Klikom na određeni zadatak u listi, korisniku se otvara forma kroz koju može uređivati postojeći zadatak, ili pak može dodati potpuno novi zadatak klikom na gumb u zaglavlju stranice. Isto tako, korisnik ima i mogućnost filtriranja svih postojećih zadataka po njihovom statusu.



Slika 16. Stranica s prikazom kalendara [Autorski rad]

Na stranici s prikazom kalendara, korisniku su prikazani svi njegovi zadaci koje je dodao, i raspoređeni su po datumima. Korisnik ima i dva različita tipa prikaza kalendara, od kojih jedan prikazuje kalendar po tjednima a drugi prikazuje sve zadatke za određeni dan. Isto tako, zadaci koji su označeni kao riješeni prikazani su zelenom bojom a ako nisu, onda su prikazani crvenom.



Slika 17. Stranica sa bilježnicama [Autorski rad]

Posljednja funkcionalnost koju korisnik može koristiti je stranica s popisom bilježnica. Korisnik kroz nju može stvarati nove bilježnice (dokumente), može ih pregledaviti, izmjenjivati i brisati po želji. Za uređivanje dokumenata koristi se identičan modul kao i kod brzih bilješki. Korisnik u ovom slučaju može i sakriti lijevi dio glavnog ekrana i proširiti dio za uređivanje dokumenta.

17. Zaključak

Tema ovog rada je bilo dublje upoznavanje s razvojem modernih web aplikacija i razvoj vlastite web aplikacije uz pomoć razvojnih okvira *Angular* i *ASP.NET Core*. Početak rada se bavio pregledom pojma web aplikacija te nekim najpopularnijim arhitekturama koji se u njihovoj izradi koriste, pa su tako spomenute monolitna, mikroservisna i višeslojna arhitektura, svaka sa svojim prednostima i nedostacima. Moram priznati da mi je ovaj dio rada puno pomogao da shvatim kako nije svejedno koja arhitektura je odabrana za izradu softvera, jer se u nekim slučajevima sam proces izrade može znatno otežati i produžiti ako je odabrana pogrešna arhitektura.

U nastavku rada fokus se prebacio na izradu web aplikacije za upravljanje osobnim podacima. U početku je bilo važno dobro postaviti skelet, kako klijentske aplikacije, tako i API servisa. To je uključivalo spajanje klijentske aplikacije i API servisa, te API servisa i MongoDB baze podataka. Uz to, na početku su implementirane i najvažnije funkcionalnosti najbitnijeg entiteta u aplikaciji, korisnika. Kroz sam početak sam već dobio dobar osjećaj kako Angular funkcionira kao razvojni okvir te kako ga treba koristiti, jer prije toga nisam imao nikakvih iskustava u radu s frontend okvirima.

Nakon što su kosturi bili završeni, krenulo se s implementacijom zasebnih funkcionalnosti zadataka, kalendara, brzih bilješki i bilježnica (dokumenata). Kako sam kroz ovaj rad imao prilike raditi s vanjskim paketima, vidio sam da je uvoz takvih vanjskih modula u Angularu poprilično jednostavno. Većina posla se svodi na poziv eksterne komponente nekog modula i generiranje početnih postavki kako bi se komponenta mogla prilagoditi po želji.

Smatram da je projekt generalno dobro odrađen s par iznimki. Na strani backenda, iako se koristi višeslojna arhitektura s odvojenim slojevima (svaki sloj je dll), kod definiranja načina instanciranja klasa iz tih slojeva (npr. servisa i repozitorija), koriste se konkretizacije (konkretne klase) a ne apstrakcije (sučelja). Ovakav način spajanja slojeva je dugotrajno vrlo ne efikasan jer ukoliko dođe do promjena zahtjeva okoline, kod se mora mijenjati na jako puno mjesta. Na strani klijentske aplikacije, najviše sam problema imao s dizajniranjem korisničkog sučelja i korištenjem css-a. Moram priznati da mi je Bootstrap okvir jako puno pomogao, s već gotovim dizajnom kojega sam modificirao kako bi dizajn odgovarao specifičnim zahtjevima aplikacije.

Na kraju moram reći kako sam kroz ovaj rad uspio ispuniti svoj cilj koji je bio upoznavanje s razvojem web aplikacije koristeći Angular i ASP.NET Core okvire. Angular se generalno smatra vrlo kompleksnim i zahtjevnim okvirom, ali iz osobnog iskustva mogu reći da je rad s Angularom poprilično jednostavan i zabavan. Većina modernih funkcionalnosti koje se očekuju od web aplikacije su vrlo jednostavne za implementirati kroz Angular module, poput

preusmjeravanja isl. Moram reći i kako je model baziran na komponentama vrlo jednostavan za shvatiti i koristiti, što je znatno olakšalo cjelokupni proces razvoja klijentske aplikacije.

Popis literature

[1] Google (bez dat.) *Build Your First Angular App* [Na internetu]. Dostupno: https://angular.io/tutorial/first-app [pristupano 02.06.2023.]

[2] Microsoft (bez dat.) *Introduction* [Na internetu]. Dostupno: https://learn.microsoft.com/en-us/training/modules/build-web-api-aspnet-core/1-introduction [pristupano 02.06.2023.]

[3] MongoDB (bez dat.) *Usage Examples* [Na internetu]. Dostupno: https://www.mongodb.com/docs/drivers/csharp/current/usage-examples/ [pristupano 02.06.2023.]

[4] N. Cummings, *Build an app with ASPNET Core and Angular from scratch*, 2023. [Na internetu]. Dostupno: https://www.udemy.com/course/build-an-app-with-aspnet-core-and-angular-from-scratch/ [pristupano 20.06.2023.]

[5] Bootstrap (bez dat.) *Get started with Bootstrap* [Na internetu]. Dostupno: https://getbootstrap.com/docs/5.3/getting-started/introduction/ [pristupano 25.06.2023.]

[6] FullCalendar (bez dat.) *Documentation* [Na internetu]. Dostupno: https://fullcalendar.io/docs [pristupano 13.08.2023.]

[7] Kolkov (2022.) *AngularEditor* [Na internetu]. Dostupno: https://www.npmjs.com/package/@kolkov/angular-editor?activeTab=readme [pristupano 17.08.2023.]

[8] Freepikcompany, *Flaticon* (2023.) [Web aplikacija]. Dostupno: https://www.flaticon.com/free-icons/web-application [pristupano 08.08.2023.]

[9] Icons8 LLC, *Icons8* (2023.) [Web aplikacija]. Dostupno: https://icons8.com/icons/set/web-application [pristupano 09.08.2023.]

[10] StackPath LLC (bez dat.) *WHAT IS A WEB APPLICATION* [Na internetu]. Dostupno: https://www.stackpath.com/edge-academy/what-is-a-web-application/ [pristupano 25.05.2023.]

[11] Amazon Web Services Inc. (2023.) *What Is A Web Application?* [Na internetu]. Dostupno: https://aws.amazon.com/what-is/web-application/ [pristupano 26.05.2023.]

[12] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), Lviv, Ukraine, 2020, pp. 150-153, doi: 10.1109/MEMSTECH49584.2020.9109514.

[13] Atlassian (bez dat.) *Microservices vs. Monolithic architecture* [Na internetu] Dostupno: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith [pristupano 27.05.2023.]

[14] Krishna Jaiswal, „Microservices Architecture“, 14.03.2022. [Na internetu]. Dostupno: https://blog.knoldus.com/microservices-architecture/ [pristupano 28.05.2023.]

[15] Kelsey Taylor, „5 Pros and Cons of Microservices Explained“ (bez dat.) [Na internetu]. Dostupno: https://www.hitechnectar.com/blogs/5-pros-and-cons-of-microservices-explained/ [pristupano 28.05.2023.]

[16] M. Richards, „*Software architecture patterns*“, 2015. [Na internetu]. Dostupno: http://103.62.146.201:8081/jspui/bitstream/1/5665/1/software-architecture-patterns.pdf [pristupano 28.05.2023.]

[17] Saurabh Singh, „*How to Choose the Best Software Architecture for Your Enterprise App*“, 28.01.2023. [Na internetu]. Dostupno: https://appinventiv.com/blog/choose-best-enterprise-architecture/ [pristupano 28.05.2023.]

[18] Shalu Sharma, „*Angular Features*“, 26.06.2023. [Na internetu] Dostupno: https://www.educba.com/angular-features/ [pristupano 02.07.2023.]

[19] Amazon Web Services Inc. „*What Is A RESTful API?*“ (bez dat.) [Na internetu] Dostupno: https://aws.amazon.com/what-is/restful-api/ [pristupano 02.07.2023.]

[20] Red Hat Inc. (08.05.2020.) *What is a REST API?* [Na internetu] Dostupno: https://www.redhat.com/en/topics/api/what-is-a-rest-api [pristupano 02.07.2023.]

Popis slika