Arhitektura modernih web aplikacija na ASP.NET Core platformi

Grgić, Martina

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

ARHITEKTURA MODERNIH WEB APLIKACIJA NA ASP.NET CORE PLATFORMI

Diplomski rad

Martina Grgić

Osijek, 2021. godina.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 21.09.2021.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Martina Grgić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-955R, 05.10.2017.
OIB studenta:	61121560386
Mentor:	Izv. prof. dr. sc. Krešimir Nenadić
Sumentor:	Dr. sc. Krešimir Romić
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Doc. dr. sc. Tomislav Galba
Član Povjerenstva 1:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 2:	Dr. sc. Hrvoje Leventić
Naslov diplomskog rada:	Arhitektura modernih web aplikacija na ASP.NET Core platformi
Znanstvena grana rada:	**Informacijski sustavi (zn. polje računarstvo)**
Zadatak diplomskog rada:	Opisati glavne arhitekturalne principe koje koriste moderne arhitekture za izradu web aplikacija. Objasniti razlike između najpoznatijih serverskih arhitektura, te principe tehnologija koje se koriste na klijentskoj strani web aplikacija. Kroz jednostavan primjer ASP.NET Core aplikacije, te uz pomoć Azure platforme prikazati neke od opisanih arhitektura i principa.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	21.09.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 11.10.2021.

Ime i prezime studenta:	Martina Grgić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-955R, 05.10.2017.
Turnitin podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Arhitektura modernih web aplikacija na ASP.NET Core platformi**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Krešimir Nenadić

i sumentora Dr. sc. Krešimir Romić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija. Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

UVOD

Tema rada obuhvaća veliko područje u arhitekturi modernog softvera, kroz rad spomenuti su i pojašnjeni neki od najvažnijih principa i načela objektno orijentiranog programiranja te slojevite arhitekture aplikacija. Kroz jednostavan primjer web aplikacije u .NET Core tehnologiji prikazana su osnovna načela i slojevita arhitektura softvera.

REST API poglavlje govori o protokolima i načinu komunikacije između klijenta i servera te raznih servisa koje se koriste u određenim arhitekturama. U sljedećem poglavlju opisuje se preteča višeslojnih aplikacija; monolitne aplikacije te važnost i uloga slojeva. Opisane su derivacije slojevite arhitekture: heksagonalna, *onion* i *clean* arhitektura te sažete njihove razlike. U posebnom poglavlju objašnjene su razlike između tradicionalnih klijentskih web aplikacija (engl. MPA) te modernijih SPA aplikacija. Unutar rada opisan je problem koji zahtijeva izradu višeslojne aplikacije te postupak na koji način se može razviti arhitektura za nju u ASP.NET Core tehnologiji.

PREGLED STRUČNJAKA ZA ARHITEKTURU SOFTVERA

Najpoznatija i najkorištenija arhitektura softvera je slojevita arhitektura koja se koristi od 90tih godina prošlog stoljeća. Većinu arhitektura moguće je primijeniti u više objektno orijentiranim programskim jezicima. Detaljni pregled arhitekture u Microsoft tehnologijama opisali su Dino Esposito i Andrea Saltarello [1].

Iz slojevite arhitekture godinama i prateći razvoj softverskih tehnologija slojevita arhitektura se razvijala u kompleksnije. Heksagonalnu arhitekturu opisao je Alistair Cockburn 2005 godine [2], *Onion* arhitekturu predstavio je Jeffrey Palermo 2008 godine [3] i *Clean* arhitekturu Robert Martin poznat pod nazivom *„Uncle Bob“* [4,5].

Osim spomenutih stručnjaka, poznatih prema arhitekturama koje predlažu postoje brojni softver arhitekti poznati po svojim radom na velikim aplikacijama kao što je Werner Vogles CTO Amazona. Poduzeća poput Netflix su zbog rasta usluga svoje monolitne aplikacije bili primorani refaktorirati kako bi koristili modernije arhitekture [6]. Također, brojni stručnjaci u Microsoft i drugim tehnologijama koji svojim blogovima, člancima i knjigama prenose znanje: Martin Fowler [7,8], Scott Hanselman [9], Udi Dahan [10] i mnogi drugi.

IT stručnjaci poznati prema radu na arhitekturi softvera 2002. godine osnovali su udrugu IASA (engl. *International Association for Software Architects*) [11] čiji su članovi iz preko 50 država s brojkom od više od 70 000 ljudi. Udruga nudi trening, literaturu i certificiranje na temu softver arhitekture.

REST API

REST (engl. *Representational State Transfer*) je o arhitekturalni stil za distribuirane hipermedijske sustave, a prvi ga je predstavio Roy Thomas Fielding 2000. godine u svojoj disertaciji *„Architectural Styles and the Design of Network-based Software Architectures"* [12]. On je definirao i 6 ograničenja koja moraju biti zadovoljena kako bi aplikacijsko programsko sučelje bilo „*RESTful*". U ovom poglavlju biti će objašnjeno značenje aplikacijskog programskog sučelja, ograničenja REST arhitekture te komunikacija između klijenta i poslužitelja putem HTTP protokola (engl. *Hyper Text Trasfer Protocol)*.

3.1. API – Aplikacijsko programsko sučelje

Aplikacijsko programsko sučelje (engl. *Application programming interface,* API) opisuje pravila i specifikacije programske aplikacije kako bi se ona mogla koristiti u drugim aplikacijama ili na klijentu. Ona sadržava popis metoda koji se koriste kao i strukture podataka koje koriste te metode.

3.2. Ograničenja REST arhitekture

Klijent poslužitelj ograničenje reprezentira odvajanje korisničkog sučelja od problema spremišta podataka, poboljšava se prenosivost korisničkog sučelja na više platformi i poboljšava se skalabilnost pojednostavljenjem poslužiteljskih komponenti. Kod razvoja *web* aplikacija klijent je korisnik poslužiteljske aplikacije, konkretno to može biti internetski preglednik koji traži od poslužitelja resurse koji su mu potrebni u nekom obliku, stranice, podatke i slično. Poslužitelj u ovom primjeru je računalo koje je zaduženo za slanje resursa, čeka na zahtjev klijenta i vraća odgovor.

Komunikacija između klijenta i poslužitelja treba biti bez prenošenja stanja. Svaki zahtjev od klijenta do poslužitelja mora sadržavati sve potrebne informacije za razumijevanje zahtjeva i ne može se iskoristiti bilo koji pohranjeni kontekst na poslužitelju.

Ograničenja predmemorije zahtijevaju da podaci unutar odgovora na zahtjev budu označeni s obzirom mogu li biti upisani u predmemoriju ili ne. U pravilu klijent smije zapisati u predmemoriju svaki odgovor osim ako nije naznačeno suprotno.

Potrebno je definirati jedinstveno sučelje. Kako bi se osiguralo jedinstveno sučelje REST je definiran kroz 4 ograničenja sučelja: identifikacija resursa, manipulacija resursa putem prikaza, samo-opisne poruke i hipermedija kao prikaz stanja unutar aplikacije.
Potrebno je arhitekturu sustava prilagoditi slojevitom stilu. Takva arhitektura se sastoji od više komponenti. Na takav način klijent ne može znati je li spojen na krajnji server ili na server-posrednik.

Posljednje ograničenje je jedino opcionalno a to je kod na zahtjev. REST omogućuje proširenje funkcionalnost klijenta preuzimanjem i izvršavanjem koda u obliku minijaturnih programa (engl. *Applet*) ili skripti. To pojednostavljuje klijente smanjenjem broja značajki koje je potrebno unaprijed implementirati.

3.3. Komunikacija između klijenta i poslužitelja

U REST arhitekturi klijenti šalju zahtjeve za dohvaćanjem ili mijenjanjem resursa, a poslužitelji šalju odgovore na te zahtjeve. Zahtjev i odgovor sastoje se od izravnih podataka ali i meta-podataka koji govore o načinu i obliku samih podataka.

REST zahtijeva da klijent pošalje zahtjev poslužitelju kako bi dohvatio ili izmijenio podatke na poslužitelju. Za razmjenu upita i odgovora koristi se HTTP protokol [13,14]. Zahtjev se obično sastoji od: HTTP glagola koji definira koju vrstu operacije treba izvesti, zaglavlja koje omogućuje klijentu da proslijedi informacije o zahtjevu, putanja do resursa, te opcionalno tijelo poruke koje sadrži podatke.

Postoje 4 osnovna HTTP glagola koja se koriste u upitima za interakciju s resursima u REST arhitekturi: GET koji se koristi kod dohvaćanja jednog resursa prema jedinstvenom identifikatoru ili kod dohvaćanja kolekcije resursa prema filteru, POST kod kreiranja novog resursa, PUT kod izmjene resursa te DELETE za brisanje resursa. Ovi HTTP glagoli pomažu u realizaciji CRUD operacija. CRUD (engl. *Create, Read, Update, Delete*) u prijevodu: kreiraj, pročitaj, izmjeni i obriši operacije koje se koriste pri kreiranju aplikacijskog programskog sučelja. Svaki model, tj. tip resursa bi trebao biti opisan pomoću spomenute 4 operacije kako bi bio kompletan.

U zaglavlju zahtjeva klijent, između ostalog, šalje tip sadržaja koji može primiti s poslužitelja. To se zove *Accept* (prihvati) polje i to osigurava da poslužitelj ne šalje podatke koje klijent ne može

obraditi. Na primjer tekstualni dokument koji sadržava HTML bio bi specificiran s tipom: *text/html*. Ako tekstualni dokument sadržava CSS tada bi zahtjev u Accept polju imao *text/css*.

Zahtjevi moraju sadržavati putanju do resursa za koji se izvršava zadana operacija. Prema konvencijama prvi dio putanje treba biti ime resursa u množini. Na taj način ugniježđene putanje su jednostavne za čitanje i razumljive. Primjer takvog puta je: *website-domain-name.com/authors/123/books/456* gdje se može lako pretpostaviti da se pristupa knjizi s ID-jem (engl. *identifier*) 456 koja pripada autoru sa ID-jem 123. Moguće je imati više jednakih putanja ali označenih drugim HTTP glagolom te će te operacije raditi drugačije. Tako bi putanja *website-domain-name.com/books/111* označena HTTP glagolom DELETE brisala resurs knjige sa ID-jem 111, a jednaka putanja ali označena GET glagolom dohvaća resurs knjige.

Nakon što klijent pošalje zahtjev, server nakon njegove obrade dohvaća potrebne podatke i sastavlja odgovor ako je potreban. Ako server vraća odgovor klijentu u obliku podataka, mora navesti *content-type* polje u zaglavlju odgovora. *Content-type* polje informira klijenta na vrstu podataka koje server šalje u tijelu odgovora. *Content-type* treba biti jedan od *Accept* tipova koje je klijent zatražio.

Server uz sam odgovor i informacije o odgovoru šalje status kod koji informira klijenta o uspješnosti operacije koju je zatražio. Status kod opisan je pomoću troznamenkastog broja i kratkog opisa. Statusi kao što je 1** (npr. *101* status – izmjena protokola) su informacijski, npr. označavaju da je zahtjev uspješno primljen i da će se izvođenje nastaviti, 2** statusi označavaju uspješno izvršavanje zahtjeva, 3** opisuju preusmjeravanje tj. da se moraju dogoditi dodatne operacije kako bi se zahtjev izvršio, 4** označava grešku klijenta, zahtjev nije dobro formatiran, dogodila se sintaksna pogreška, zahtjev nije moguće izvršiti, 5** je greška na serveru. Neki od najčešće korištenih statusa su: *200 OK* – sve je prošlo dobro i ukoliko se radi o GET zahtjevu možemo očekivati zahtijevane podatke u odgovoru, *400 BadRequest* – server ne želi ili ne može izvršiti zahtjev zbog klijentske greške, *401 Unauthorized* – autentikacija korisnika nije provedena ili nije uspješno izvršena, *403 Forbiden* – autenticirani korisnik nema dovoljno pravo da izvrši zahtjev, *404 NotFound* – traženi resursi nisu pronađeni i *500 InternalServerError* – dogodio se nepredviđeni izuzetak.

ARHITEKTURALNI PRINCIPI

Objektno orijentirano programiranje je način programiranja u kojem se koriste klase i objekti kako bi se riješio problem. Klasa je način definiranja svojstava (atributa) i mogućnosti (metoda). S obzirom da se klasa ne može koristi direktno, pristup atributima i metodama dobiva se korištenjem instance klase tj. stvaranjem objekta te klase.

Objektno orijentirano programiranje temelji se na 4 bazna načela (engl. *4 pillars of OOP, Object Oriented Programming)*[15]. Jedno od osnovnih načela je apstrakcija koja se koristi pomoću apstraktnih klasa i sučelja kako bi definirali više klasa koje se minimalno razlikuju i tako izbjegli redudanciju. Enkapsulacijom ili učahurivanjem klasa skriva neke atribute i metode od ostalih klasa, na taj način se postiže slaba povezanost objekata (engl. *loose coupling*). Slabom povezanošću objekti postaju neovisniji i interne promjene jednog objekta ne utječu na rad drugog. Nasljeđivanje se koristi u slučajevima kada se definira objekt koji za podskup ima već postojeći objekt. Polimorfizam ili višeobličje je preopterećivanje metoda, definiranje nekoliko metoda istog imena s različitim parametrima. Sva ostala načela i principi temelje se na ovim osnovnim pravilima.

Projektirati i dizajnirati softver treba s obzirom na njegovu održivost. Kako bi softver bio održiv najčešće se dizajnira u skladu s nekoliko načela: razdvajanje interesa (engl. *Separation of Concerns, SoC*), inverzija ovisnosti (engl. *Dependency Inversion Principle, DIP*), načelo jedinstvene odgovornosti (engl. *Single Responsibility Principle, SRP*), načelo ne-ponavljanja (engl. *Don't Repeat Yourself, DRY*), uzorak ograničenog konteksta (engl. *Bounded contexts*). Prateći ta načela aplikacija će imati slabo povezane komponente te će one međusobno komunicirati putem sučelja ili sustava za razmjenu poruka. Spomenuta načela nalaze se u akronimu SOLID; SRP, OCP (engl. *Open/Closed Principle*), LSP (engl. *Liskov Substitution Principle*), ISP (engl. *Interface Segragation Principle*), DIP [16]. Neki od ovih načela objašnjeni su u popratnom tekstu jer su bitni za razumijevanje arhitektura.

Separation of Concerns, SoC, uveo je Edsger W. Dijkstra, 1974 u svom radu: *„On the Role of Scientific Thought.“*. Spomenuti članak kao i druge moguće je pronaći u knjizi „*Selected Writings on Computing“* [17]. SoC govori o razbijanju sustava u različite i nepreklapajuće značajke. Svaka značajka koja postoji u sustavu predstavlja aspekt sustava. SoC sugerira da je potrebno usredotočiti se na jedan određeni aspekt. Odnosno podijeliti sustav prema samoodrživim modulima gdje iz perspektive tog modula bilo koji drugi problem, tj. modul nije važan.

Single Responsibility Principle, SRP, kao pojam uveo je Robert C. Martin [18]. Prema ovom principu svaka klasa unutar aplikacije treba imati odgovornost nad jednim dijelom funkcionalnosti te aplikacije. Robert C. Martin rekao je „Klasa treba imati samo jedan razlog za promjenu“.

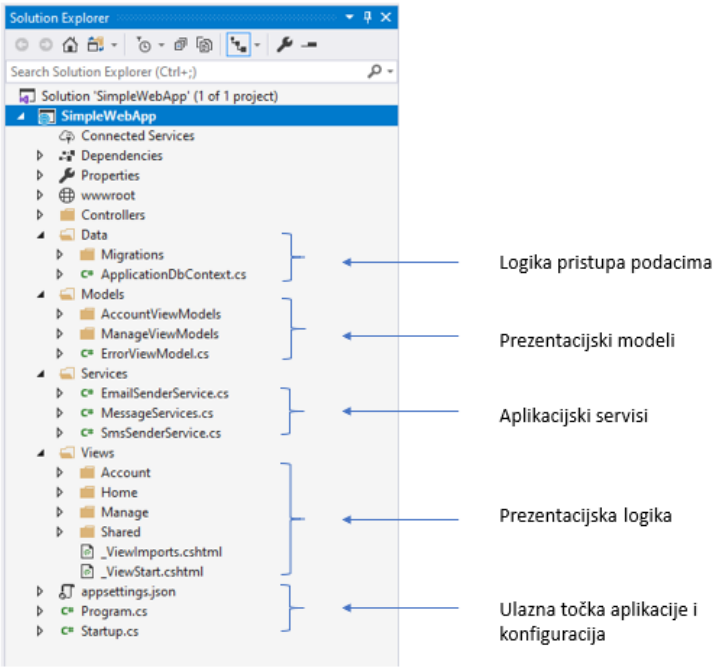
Višeslojna arhitektura jedan je od arhitekturnih obrazaca koji prati glavna načela dizajna softvera, koristi se za izradu srednjih i velikih, složenih aplikacija. Postoje razne interpretacije višeslojne arhitekture zbog prilagođavanja same arhitekture zahtjevima projekta, mogućnosti jezika u kojem je pisana aplikacija a mogu se podijeliti u tri kategorije: slojevita, *onion*, heksagonalna. U slojevitoj arhitekturi podaci se kreću kroz jednu definiranu razinu obrade na drugu. Onion arhitektura ima slojeve definirane od jezgre do infrastrukture i kod može ovisiti o slojevima koji su centralniji, ali kod ne može ovisiti o slojevima koji su dalje od jezgre. Heksagonalna arhitektura (engl. *Hexagonal* ili *Ports and Adapters*) je definirana adapterima koji služe za ulazne i izlazne interakcije sa aplikacijom.

4.1. Monolitne i jednostavne aplikacije

Monolitna aplikacija je u potpunosti samostalna. Može komunicirati s drugim uslugama ili pohranama podataka, ali srž ponašanja se izvodi unutar vlastitog procesa, a cjelokupna aplikacija je obično rapoređena kao jedna cjelina. Ako se takva aplikacija treba skalirati, obično se cijela aplikacija duplicira na više poslužitelja ili virtualnih strojeva.

Najmanji mogući broj projekata jedne aplikacije je jedan. U takvoj arhitekturi cijela logika aplikacije sadržana je u jednom projektu koji je sastavljen u jedan *assembly* te se prilikom podizanja aplikacije u produkciju prenosi kao jedna cjelina.

Kreiranjem primjera aplikacije kroz Visual Studio dobivamo jednostavnu monolitnu aplikaciju koja je smještena u jedan projekt (Slika 4.1.).



Slika 4.1. Prikaz strukture monolitne aplikacije

Kako bi se i dalje poštivao *SoC* princip koriste se mape te tako osiguralo odvajanje glavnih elemenata aplikacije. Prema ovoj podijeli jedna mapa sadržava logiku za pristup podacima što su u ovom slučaju Migracije i *DbContext* s definiranim entitetima, modelima, za manipulaciju bazom podataka, odnosno potrebni dijelovi za korištenje *Entity Frameworka* kao objektno-relacijski pridruživać (engl. *Object-relational mapping*, ORM). Prezentacijski modeli odvojeni su u zasebnu mapu, oni služe kao opis podataka koji se prikazuje na korisničkom sučelju. U nekim literaturama i arhitekturama koristi se i dodatna vrsta modela DTO,(engl. *Data Transfer Logic*). Takav model koristi se isključivo za prijenos podataka između servisa i procesa. Entiteti (engl. *Entity Model*) pisuju tablice u bazi podataka, *ViewModeli* opisuju podatke koje se nalaze u sučelju, a ukoliko postoji višestruka manipulacija podataka u biznis procesu koriste se i DTO modeli. U prikazu monolitne aplikacije postoji još i mapa za prezentacijsku logiku gdje su smješteni pogledi odnosno .*cshtml* dokumenti koji opisuju izgled *web* stranice ili dijela stranice. Aplikacijski servisi nalaze u zasebnoj mapi, a oni služe za upravljanje poslovnom logikom i manipulacijom podataka iz spremišta kao bi bili pravilno prikazani na sučelju, koristimo ih kao poveznicu između *Entity* modela i *ViewModela* u manje zahtjevnim aplikacijama.

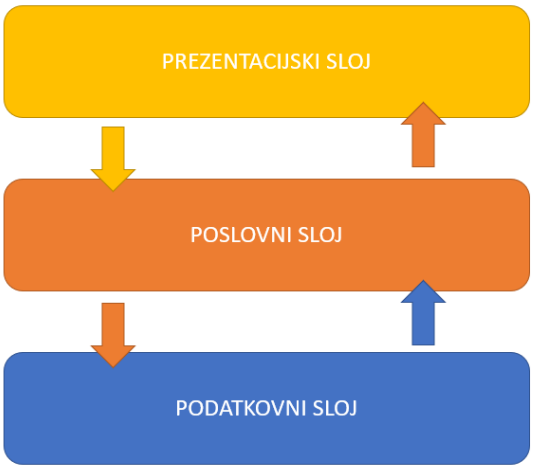
Ovakva vrsta arhitekture pogodna je isključivo za vrlo jednostavne aplikacije, dodavanjem tablica, modela i logike ovakav način raspodijele koda je neodrživ. Za veće aplikacije, pogotovo na EAS (engl. *enterprise software*) nivou arhitektura se postavlja na više projekata i odvojenih servisa koje možemo smatrati slojevima aplikacije (*multi-layer*).

4.2. Slojevita arhitektura

Velike aplikacije koje se bave organizacijom unutar poduzeća, škole ili slično zahtijevaju više entiteta, modela, servisa i koda općenito. Kako bi se i dalje mogao pratiti SoC princip dijelovi koda se, prema odgovornosti, dijele u više projekata koje zovemo slojevima. Na takav način osiguravamo održivost i neometan rast aplikacije. Slojevita arhitektura uz prednost organizacije koda nudi i mogućnost ponovne upotrebe određenog dijela koda i na taj način se prati DRY princip. Ovakvom arhitekturom omogućava se i lakša zamjenu funkcionalnosti unutar aplikacije. Na primjer, aplikacija može inicijalno koristiti SQL (engl. *structure query language*) bazu kao spremište podataka, a naknadno dođe do potrebe korištenja *cloud* načina ili nešto drugo. Ukoliko aplikacija ima pravilno postavljenu enkapsulaciju dovoljno je zamijeniti jedan sloj (SQL sa *cloud*-om) te ispraviti komunikaciju s tim slojem. Osim moguće zamjene implementacije određenog sloja, slojevi aplikacije se mogu zamijeniti sa lažnim slojevima kako bi olakšali testiranje određenog dijela aplikacije. Na ovaj način jednostavnije je i brže pisati testove te je brže dobiti rezultate nego na klasičan način, testiranja cijele aplikacije kroz cijelu strukturu (od UI do spremišta podataka).

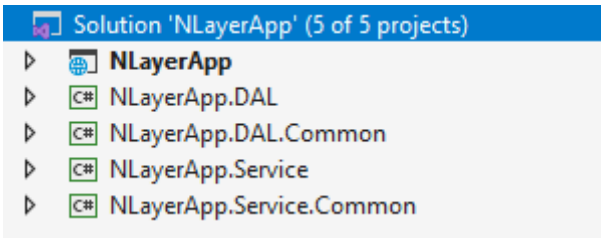
Slojevita arhitektura se može postići na više različitih načina. Na neke zahtjeve odgovara tradicionalna N-sloj arhitektura (engl. *N-Layer*) koja je opisana u ovom poglavlju, dok je za druge zahtjeve potrebno koristiti i složenije arhitekture, neke od njih su opisane u nastavku rada.

N-sloj arhitektura najčešće uključuje ove slojeve: prezentacijski sloj, sloj domene tj. poslovni sloj i podatkovni sloj (Slika 4.2.). Slojevi u ovoj arhitekturi predstavljaju fizički i logički odvojene komponente aplikacije. Ako je arhitektura postavljena ispravno svaki sloj komunicira samo susjednim slojem, na primjer podatkovni sloj komunicira samo sa poslovnim. U nekim literaturama odvaja se još jedan dodatan sloj; servis sloj, koji preuzima dio odgovornosti prezentacijskog sloja i odgovara poslovnom sloju [1].



Slika 4.2. Prikaz slojeva 3-slojne arhitekture

Razdvajanje slojeva u kodu postiže se kreiranjem više projekata prema slici 4.3.



Slika 4.3. Prikaz strukture višeslojne arhitekture

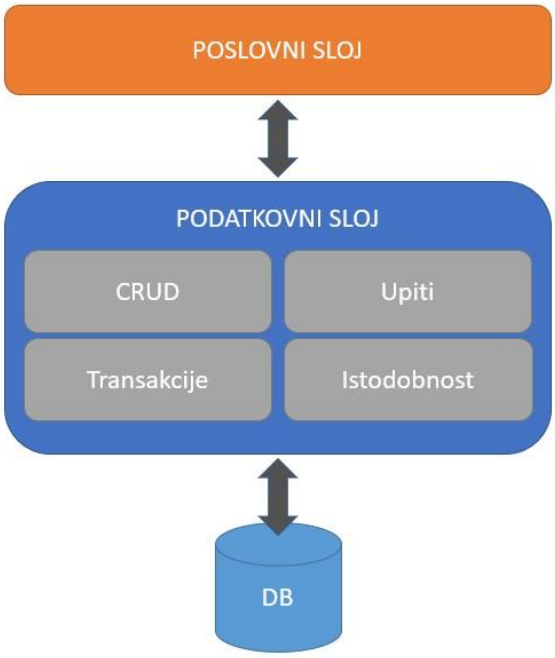
NLayerApp je projekt koji sadržava ulaznu točku aplikacije i prezentacijski sloj. *NLayerApp.DAL* je podatkovni sloj, te *NLayerApp.Service* poslovni. *DAL* i *Service* imaju i dodatne *„Common“* projekte u kojima su smještena sučelja. Slojevi međusobno komuniciraju preko sučelja u kojima se nalaze definicije metoda dostupnih na korištenje. Svi projekti imaju vlastiti *assembly* te prateći ovisnosti slojeva poslovni sloj ima referencu na *DAL.Common* dok prezentacijski sloj ima referencu samo na *Service.Common*.

4.2.1. Podatkovni sloj – DAL

Podatkovni sloj (engl. *Data Access Layer, DAL*), u literaturi još pod nazivom „*Data Persistence Layer“*, bavi se isključivo upravljanjem podataka, izvršava osnovne operacije nad podacima (CRUD) nad zadanom bazom podataka.

Podatkovni sloj upravlja spremištem podataka (ili više njih). Prema zahtjevima klijenta ili sustava definira se spremište podataka (npr. relacijska SQL baza podataka ili više različitih izvora kao dokumenti i/ili *cloud*). Podaci o konfiguraciji baza najčešće se spremaju u konfiguracijski

dokument kojem pristup smije imati samo DAL sloj. Gledajući izvana poslovni sloj promatra podatkovni kao crnu kutiju (engl. *black box*) sa adapterima (sučeljima) za pristup podacima. (Slika 4.4.).



Slika 4.4. Prikaz poslovnog i podatkovnog sloja

Najvažnija odgovornost podatkovnog sloja je čuvanje podataka u fizičkoj pohrani te isporuka istih kroz CRUD usluge. Također DAL je odgovoran za posluživanje podataka koje sadrži kroz zahtjevne upite, mora biti u mogućnosti pružiti transakcijsku semantiku te pravilno rukovati istodobnošću (engl. *concurrency*).

CRUD usluge su metode koje se koriste za upisivanje podataka u bazu i dohvaćanje istih. CRUD usluge rade sa postojanim (engl. *persistent*) i prolaznim (engl. *transient*) objektima. Postojani objekt je instanca klase koja reprezentira informaciju dohvaćenu iz baze podataka. Dohvaćanjem objekta knjige iz baze podataka koristi se postojani objekt, kreiranjem novog objekta u memoriji s ciljem spremanja istog u bazu koristi se prolazni objekt.

Najčešće uz obično čitanje iz baze (*R – Read*) potrebno je implementirati zahtjevnije upite (engl. *query*). Upit može sadržavati više parametara kako bi objekt bio uspješno dohvaćen, moguće je da aplikacija zahtijeva dohvaćanje svih knjiga određenog autora u zadnjih 10 godina ili slično. Za zahtjevnije aplikacije upiti mogu biti vrlo kompliciranih te se najčešće koristi *repository pattern*.

Repository pattern je zapravo zamjena za CRUD usluge i upite, u kodu to bi predstavljale klase koje izvode određene CRUD operacije nad specifičnim tipom. Za knjige to bi bio *BookRepository* a za autore *AuthorRepository*. Prateći ovaj princip svaka od tih klasa imala bi minimalno 4 metode: kreiraj, izmjeni, obriši i pročitaj, te dodatnu pretraži metodu koja dohvaća objekte uz određene parametre [20].

Podatkovni sloj treba osigurati siguran i optimalan način spremanja, izmjene i čitanja podataka. U većim aplikacijama dolazi do velikih količina zahtijeva na bazu podataka, kako bi se to optimiziralo smanjuje se broj zahtijeva. Svaki upit na bazu može se promatrati kao jedna transakcija u kojoj se otvara konekcija na bazu, izvrši željena radnja i potom zatvori konekcija. Svaka baza podataka ima limitirani broj mogućih aktivnih konekcija. Kako bi se broj konekcija smanjio potrebno je zahtjeve grupirati te ih izvršiti unutar što manje transakcija a to se postiže koristeći *UoW patterna* (engl. *Unit of work*) [20].

UoW je pojednostavljeno jedan objekt koji sadržava više upita na bazu poredanih redoslijedom izvođenja. Također sadržava i metode za izvršavanje (engl. *commit*) i povratak baze podataka na prethodno stanje (engl. *rollback*).

4.2.2. Poslovni sloj – BLL

Poslovni sloj (engl. *Business Logic Layer, BLL*) sadržava metode koje opisuju cijelu poslovnu logiku aplikacije, koristi DAL kako bi dohvatio potrebne podatke i spremio promjene. Unutar BLL sloja nalaze se poslovni objekti koji opisuju podatke, poslovna pravila koja izražavaju sve politike i zahtjeve korisnika, usluge za implementaciju autonomnih funkcionalnosti i tijekove rada koji definiraju način prenošenja dokumenata i podataka iz jednog modula u drugi i njihov prijelaz između slojeva. Sigurnost podataka u BLL sloju postiže se uvođenjem uloga (engl. *role-based*) kako bi se ograničio pristup samo ovlaštenim korisnicima.

Poslovni sloj implementira se prema dizajnerskom uzorku za organizirane logike domene. Svaki od uzoraka ima različite ciljeve i prigodan je za određen stupanj složenosti. Najčešća su 4 uzorka: transakcijska skripta (engl. *Transaction Script, TS*), tablični modul (engl. *Table Module, TM*) koji pripadaju proceduralnim uzorcima, te objektno orijentirani uzorci: aktivni zapis (engl. *Active Record, AR)* i model domene (engl. *Domain Model, DM*).

Uzorak transakcijske skripte definirao je Martin Fowler [8]. TS je proceduralni uzorak koji je od 4 navedena najjednostavniji. U fokusu su operacije korisnika, za svaku operaciju koju korisnik može izvesti implementira se metoda ili set akcija odnosno transakcijska skripta. Ovaj uzorak koristi se za brzi razvoj aplikacija (engl. *Rapid application development, RAD*) te ukoliko je prije same izrade projekta dostupna detaljna dokumentacija te projekt nije podložan kasnijim izmjenama.

Uzorak tabličnog modula je dodatak na TS, također proceduralan uzorak sa fokusom na operacijama korisnika. Za razliku od TS-a, TM zahtjeva kreiranje poslovne komponente (klase) za svaku od tablica u bazi podataka.

Za razliku od TM-a koji ima tablični pogled (engl. *table-based view*), uzorak aktivnog zapisa, AR, ima pogled na redak (engl. *row-based view*). AR modeli reprezentiraju redak tablice, svaki model sadržava svojstva (engl. *properties*) koji su preslika stupaca tablice te metode za validaciju i manipulaciju tih svojstava uz CRUD operacije.

Model domene kao fokus ima domenu problema. DM kao i AR je objektno orijentirani te je glavna značajka na modelu, no za razliku od AR model je apstraktan i opisuje poslovni model sa svojstvima i metodama, a ne redak tablice. DM je i najzahtjevniji model za osmisliti. Korištenjem DM uzorka i najzahtjevnije poslovne logike mogu uspješno biti implementirane i održavane.

Poslovni sloj sadržava i aplikacijsku logiku ili servisni sloj (API). API je adapter aplikacije na koji se spajaju vanjske komponente, recimo *web* aplikacije, mobilne, desktop aplikacije, ili dodatni vanjski servisi. U današnje vrijeme najčešće se koristi REST paradigma za kreiranje API sloja.

4.2.3. Prezentacijski sloj - UI

Prezentacijski sloj (engl. *Presentation Layer*), poznat kao korisničko sučelje (engl. *Front-End User Interface*) koje može biti izrađen za više platformi (mobilnu, desktop, *web* i sl.).

Uloga prezentacijskog sloja je ispravno i na razumljiv način prikazati podatke, prikupiti informacije od korisnika pomoću formi, početna validacija, prikazati rezultat upita korisniku itd. U većim aplikacijama koriste se arhitekturni principi primjereni za prezentacijski sloj te tehnologiju u kojoj se piše. Prezentacijski sloj može biti prilagođen za *web*, mobilni uređaj,

desktop, bilo koji uređaj koji ima mogućnost korištenja multimedije kao televizor, pametni hladnjak itd.

4.3. *Dependency Injection*

Većina spomenutih arhitektura zahtjeva slabo povezani kod, a *Dependency Injection (DI)* je način kako to postići. *Dependency Injection* je kolekcija softverskih dizajnerskih načela i obrazaca koji omogućavaju izradu slabo povezanog kod. Ideja iza *Dependency Injectiona* prati *Dependency Inversion* načelo [19] koje govori o tome da moduli višeg nivoa ne trebaju ovisiti o modulima nižeg nivoa, nego obje vrste modula trebaju ovisiti o apstrakciji. Odnosno, apstrakcije ne trebaju ovisiti o detaljima nego detalji o apstrakcijama. Kada govorimo o apstrakcijama u kontekstu objektno orijentiranih programskih jezika mislimo na *interface, „Program to an interface not an implementation" (GoF)* [21].

Obrasci koji se koriste za implementaciju DI-ja su: *Constructor Injection, Property Injection, Method Injection* i *Ambient Context*.

Najčešće korišten i najvažniji obrazac je *constructor injection*. *Constructor Injection* koristi *Inversion of Control, IoC* obrazac. Umjesto kreiranja ovisnosti unutar klase kontrola se invertira na način da sada klasa očekuje od treće strane potrebnu ovisnost.

Prema sljedećem primjeru koda s lijeve strane vidimo usku povezanost: *BookController* mora znati za postojanje i implementaciju *BookService*-a i kreira tu povezanost pomoću ključne riječi *new*. Na desnoj strani je primjer korištenja *Dependency Injectiona* na način da se *injecta* implementacija sučelja *IBookService* te se instanciranje prave implementacije prepušta DI sustavu.

```
private readonly BookService bookService;
public BookController()
{
        bookService = new BookService();
}
```

```
private readonly IBookService bookService;
public BookController(IBookService bookService)
{
        this.bookService = bookService;
}
```

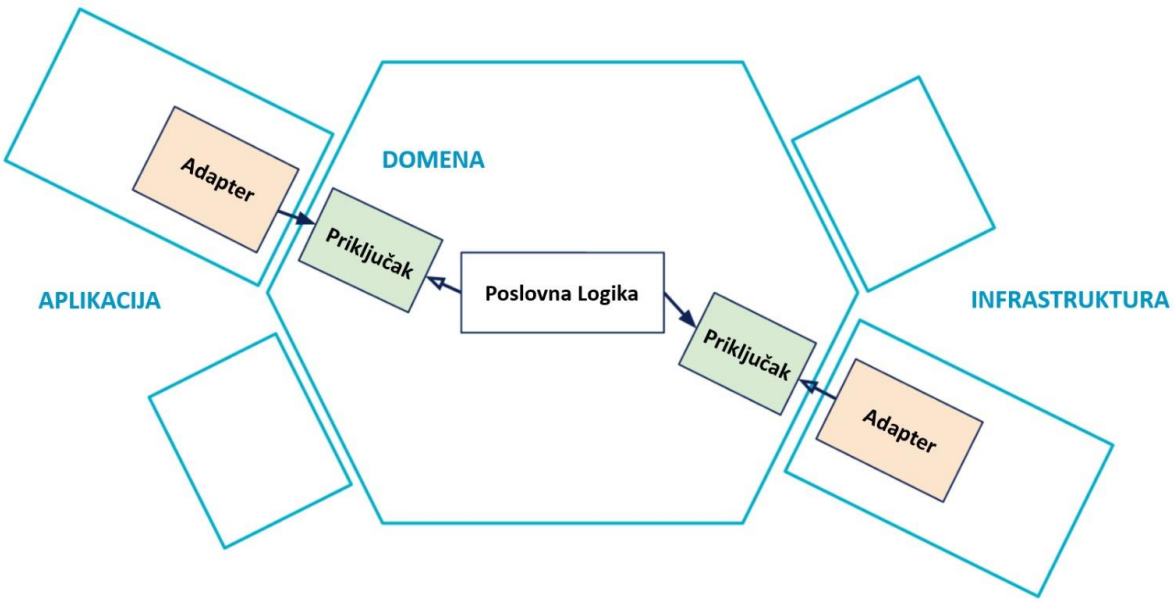
Korištenjem DI-ja odnosno prateći IoC obrazac postiže se slaba povezanost koda koja omogućava jednostavnije testiranje; moguće je testirati rad *Controller* klase na način da se u DI ubaci implementacija *mock BookService*-a umjesto stvarne implementacije.

4.4. Heksagonalna arhitektura

Hexagonal architecture ili *Ports and Adapters* [2] se može opisati kao: „Ukoliko primijenite načelo inverzije ovisnosti na slojevitu arhitekturu dobit ćete portove i adaptere“ [22].

Izvorna namjera heksagonalne arhitekture je stvaranje aplikacije koja će raditi jednako bez obzira na to pokreće li ju korisnik, neka druga aplikacija, automatizirani testovi te da radi neovisno o ostalim vanjskim uređajima i bazama podataka o kojima ovisi.

Heksagonalnu arhitekturu možemo podijeli u 3 sloja (Slika 4.5.): aplikacija (lijeva strana, korisnička strana), domena (središte, poslovna logika) i infrastruktura (desna strana, serverska strana). Uloge ovih slojeva slične su ulogama slojevite arhitekture opisane u prethodnom poglavlju. Promjena koja se uvodi u ovoj arhitekturi je shvaćanje baze podataka kao jedan od adaptera, a ne kao zadnji sloj o kojem ovisi sustav.



Slika 4.5. Prikaz heksagonalne arhitekture

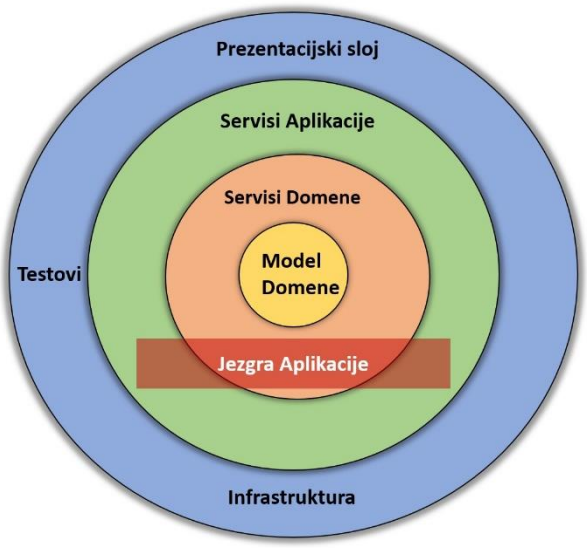
Cockburn nije definirao broj adaptera, iz definicije arhitekture adaptera može biti jednako koliko i metoda tako da svaki *use-case* ima vlastiti port, s druge strane moguće je koristiti samo dva porta, jedan s lijeve strane (iz literature: „*driving* side“) i jedan s desne (iz literature: „*driven* side“).

Driving side su adapteri koji pokreću aplikaciju, to može biti UI; korisnički upiti ili test klase koji oponašaju te upite. Adapter ovisi o portu koji je implementiran u središtu, adapter ne zna tko

odgovara na njegove upite, jedino što zna su metode koje su dostupne kroz sučelje. *Driven side* su adapteri za pohranu kao baza podataka ili test baza podataka, adapteri za pohranu u dokumente i slično. Adapteri u ovom slučaju implementiraju sučelja koja se nalaze u središtu kao port te središte promatra kao veći nivo (engl. *higher level*).

4.5. *Onion* arhitektura

Jeffrey Palermo, 2008. [3]. *Onion* arhitektura ovisi o *Dependency Injectionu* jednako kao i heksagonalna arhitektura. Sastoji se od 4 sloja: UI (engl. *User Interface* ili *Web*), *Service Interface*, *Repository* i *Domain Entities*. (Slika 4.6.)



Slika 4.6. *Onion* arhitektura

Domain Entities sloj, odnosno sloj entiteta domene je središnji dio arhitekture. Sadrži sve objekte domene aplikacije koji nemaju ovisnost na neki drugi sloj aplikacije.

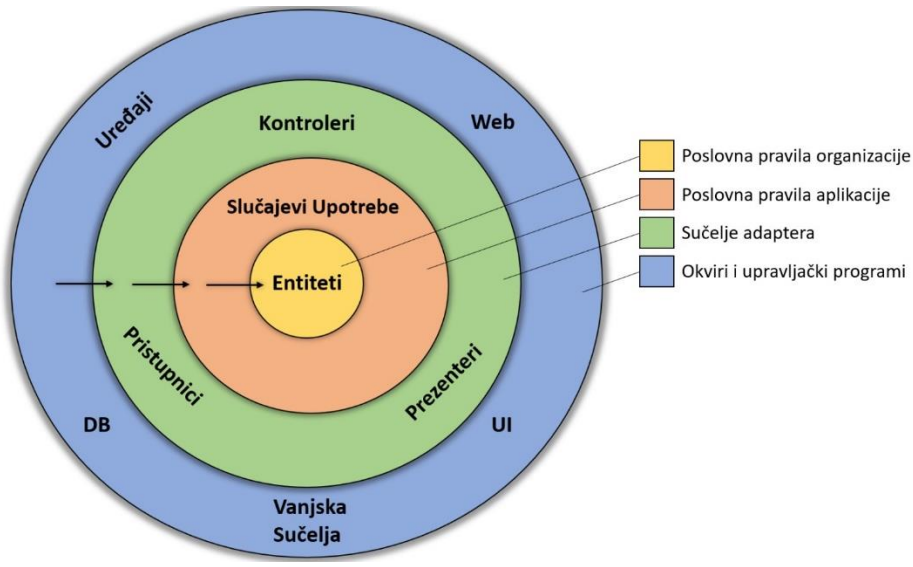
Repository sloj je namijenjen kreiranju apstrakcije između *Domain Entities* sloja i *Business* sloja. To je obrazac za pristup podacima iz *Domain Entities* sloja, podaci se mapiraju u poslovne entitete, te u suprotnom smjeru iz poslovnog entiteta na *Domain* entitete odnosno izvor podataka.

Service sloj sadrži sučelja koja se koriste za komunikaciju između korisničkog sloja i *repository* sloja. Zadaća ovog sloja je poslovna logika.

Sloj korisničkog sučelja je zadnji, vanjski sloj koji može biti *web* aplikacija, *web API*, *Unit Test* projekt, bilo koja druga aplikacija. I ona komunicira sa *service* slojem preko sučelja.

4.6. *Clean* arhitektura

Bit *clean* arhitekture je držanje detalja kao što je *web framework* i baza podataka u vanjskim slojevima i što dalje od poslovne logike koja ne smije „znati“ za te detalje [4,5]. Na taj način se održava ranije spomenuto načelo *Separation of Concerns*. Robert Martin u svom radu navodi da je ova arhitektura pokušaj integriranja svih ranije spomenutih arhitektura u jednu. Prikaz osnovnih slojeva i pojmova *clean* arhitekture nalazi se na slici 4.7.



Slika 4.7. *Clean* arhitektura

Kružnice predstavljaju različita područja softvera. Općenito što je dalje od centra to je softver na većem nivou. Vanjski krugovi su mehanizmi dok su unutarnji poslovna pravila. Najvažnije pravilo koje čini ovu arhitekturu je *The Dependency Rule*. To pravilo govori o tome da ovisnosti izvornog koda mogu dolaziti samo iznutra. Ništa u unutarnjem krugu ne smije znati o nečemu iz vanjskog kruga. Ništa iz vanjskog kruga ne smije utjecati na unutarnji krug.

Entiteti mogu biti objekti s metodama ili skup struktura podataka i funkcija. Entiteti su poslovni objekti aplikacije ili poduzeća. Oni obuhvaćaju najopćenitija pravila, te je najmanje vjerojatno da će se mijenjati ukoliko se nešto izvana promijeni. Na primjer, promjena navigacije stranice ili sigurnosti ne bi trebala utjecati na entitete.

Use Cases sloj sadrži specifična poslovna pravila. Ako se detalji oko uporabe aplikacije promijene očekivana je promjena u ovom sloju aplikacije. Ovaj sloj je izoliran od korisničkog sučelja i baze podataka i bilo kojih drugih eksternih uređaja.

Interface Adapters sadrži skup adaptera koji pretvaraju podatke iz formata koji je najpogodniji za use cases i entities sloj u format koji je najpogodniji za vanjsku upotrebu kao što je baza podataka ili *web*.

Frameworks and Drivers sloj je najudaljeniji sloj koji se sastoji od okvira i alata kao što su baza podataka, *web framework* i slično. Općenito ne sadržava mnogo koda, osim koda koji komunicira sa prvim slojem unutra. Ovom sloju pripadaju svi „detalji“ kao što su baza podataka npr. MySql ili PostgreSQL ili neka NoSql baza podataka, *web framework* je isto detalj itd.

4.7. Usporedba arhitektura

Onion, Clean i Heksagonalna arhitektura proizlaze iz slojevite arhitekture a rješavaju problem ovisnosti. Sve tri arhitekture su superiornije u odnosu na slojevitu arhitekturu jer sloj infrastrukture (DAL) stavljaju na vanjski položaj te je time lako zamjenjiv.

Svaka od arhitektura pruža drugačiji presjek na arhitekturu aplikacije te koriste drugačije pojmove koji opisuju iste stvari. Svaka od njih može jednako dobro riješiti problem ovisnosti. Mark Seemann u svom članku: *„Layers, Onions, Ports, Adapters: it's all the same“* čak govori da su te arhitekture zapravo jednake samo pod drugačijim nazivom [22].

Kao cilj svaka od arhitektura ima odvojiti ovisnosti pomoću sučelja. Time povećati broj i kvalitetu testova, olakšati raspodjelu posla developerima, održivost, ukoliko dođe o promjena da ih je lako uvesti itd.

SPA I TRADICIONALNE *WEB* APLIKACIJE (MPA)

Postoje dva osnovna pristupa kreiranja *web* aplikacija, to su tradicionalne *web* aplikacije odnosno MPA (engl. *Multi-page application*) koje većinu aplikacijske logike izvode na serverskoj strani, te SPA (engl. *Single-page application*) koje dijele serversku i klijentsku logiku, te razni hibridi.

5.1. Tradicionalne *web* aplikacije

Tradicionalne *web* aplikacije odrađuju većinu logike na serverskoj strani, te je serverska strana zaslužna za izgled i manipulaciju podataka na stranici. Za svaku stranicu preglednik treba napraviti upit prema serveru a on vraća odgovor u obliku HTML stranice. MPA možemo podijeliti na dinamičke i statičke stranice. Statičke stranice su najstariji, najjednostavniji oblik. Sav sadržaj je napravljen unaprijed, HTML stranice su pohranjene na serveru koji ih šalje na *web* preglednik kada dobije upit, koriste se za jednostavne *web* stranice gdje se sadržaj ne mijenja ili se mijenja vrlo rijetko, na primjer za osobni *web*. Dinamičke stranice koriste se za aplikacije koje zahtijevaju obradu korisničkog unosa koji utječe na izgled stranice, npr. filter kao korisnički unos koji utječe na izgled tablice s podacima. Stranice se kreiraj dinamički, na serveru, uključujući HTML, CSS i JavaScript skripte.

Tradicionalne *web* aplikacije koriste se kada aplikacija ima jednostavno sučelje, predviđena je isključivo za davanje informacija korisnicima te zahtjeva vrlo malo ili nimalo unosa korisnika. Velik broj *web* aplikacija primarno se koristi samo za čitanje (engl. *read-only*). Primjer takve aplikacije je tražilica (engl. *search engine*) koja se može sastojati od jedne ulazne točke s okvirom za tekst i stranicom sa prikazom rezultata pretraživanja. Korisnici mogu jednostavno poslati upit za koji je potrebno vrlo malo logike na strani klijenta. Slično, blogovi se uglavnom sastoje od sadržaja koji ne zahtjeva komplicirane radnje na klijentskoj strani. Također za aplikacije koje trebaju raditi u preglednicima koji nemaju podršku za *JavaScript* nije moguće korištenje SPA na klijentskoj strani.

5.2. SPA

Aplikacije s jednom stranicom (engl. *Single-Page Application*) generiraju sadržaj HTML-a unutar preglednika uz pomoć JavaScript-a. Na prvi upit preglednika prema serveru, server šalje odgovor u obliku jedne stranice (index.html), ona sadržava JavaScript kod koji služi za dinamičko

mijenjanje aplikacije, on se brine o prelasku na sljedeći dio aplikacije, dohvaćanju podataka sa servera, prikazu, mijenjanju podataka na stranici.

SPA aplikacije koriste se kada aplikacija mora imati bogato korisničko sučelje. Neke od prednosti ovakvih aplikacija: jednostavnija implementacija responzivnog dizajna, nije potrebno ponovno učitavati svaku stranicu, pojedini dijelovi stranice koji se ne mijenjaju učitavaju se samo jednom, a ne sa svakim upitom na server kao u MPA, moguće je potpuno odvojiti klijentsku stranu od serverske i tako olakšati proces izrade aplikacije, puštanje u rad aplikacije je jednostavno.

TEHNOLOGIJE

Slojevitu arhitekturu i ostale spomenute obrasce i načela moguće je primijeniti na sve objektno orijentirane programske jezike koji se koriste danas. U radu se koristi C# programski jezik odnosno ASP.NET Core programski okvir te pripadajuće biblioteke .NET programskog okvira. Integrirano razvojno okruženje koje se koristi za izradu aplikacije je Visual Studio 2019. U nastavku poglavlja opisane su tehnologije koje su se koristile u svrhu izrade primjera aplikacije.

6.1. ASP.NET Core

ASP.NET Core je razvojni okvir otvorenog koda, te je višeplatformski odnosno aplikacije napravljene nad ovim razvojnim okvirom moguće je razvijati i pokretati na Windows, Mac i Linux operacijskim sustavima [23]. Točnije ove platforme su podržane: Windows 7 SPI i više verzije, Windows Server 2008 RS SPI i više, Red Gat Enterprise Linux 7.2 i više, Fedora 23 i više, Debian 8.2 i više, Ubuntu 14.04 LTS/16.04 LTS i više, Linux Mint 17 i više, openSUSE 13.2 i više, CentOS 7.1 i više, Oracle Linux 7.1 i više, macOS X 10.11 i više. Moguće je da radi i sa drugim platformama, no ove platforme je Microsoft testirao.

ASP.NET Core pruža jednoliku mogućnost programiranja *Web* API i *Web* UI sustava. Također moguće je raditi desktop i mobilne aplikacije nad istom platformom.

6.2. *DI Container*

Postoji više *Dependency Injection Containera* koji se mogu koristiti u ASP.NET Core aplikaciji. Svi imaju istu ulogu, pomoći razvojnom programeru u poštivanju *Dependency Inversion* principa. DI kontejneri povezuju sučelja (engl. *Interface*) sa implementacijom tog sučelja. Najpoznatiji su: Ninject, Autofac, SimpleInjector ili ugrađeni. Ugrađene metode za korištenje DI nalaze se u zaglavlju *Microsoft.Extensions.DependencyInjection.*

DI zaglavlje nudi metode za registraciju implementacije koja je povezana na određeno sučelje. Svaka od registracija označena je rokom trajanja koje može biti: *Scoped* – nova instanca servisa će biti kreirana za svaki *scope* npr. *web* zahtjev i ista ta instanca će uvijek biti vraćena za isti zahtjev svaki put kada je zatražena, *Singleton* – instanca će biti kreirana i zadržana u memoriji i uvijek biti vraćena, *Transient* – nova instanca će biti kreirana svaki put kada je zatražena.

DI Container treba biti korišten unutar *Composition Roota* [19]. *Composition Root* je ulazna točka svake aplikacije u kojoj su moduli cijele aplikacije sastavljeni što znači da svaka aplikacija treba imati centralizirano mjesto gdje će se povezati implementacije klasa. Osim DI postavki *Composition Root* sadržava i sve potrebne konfiguracije aplikacije, npr. *ConnectionString* za spajanje na bazu podataka.

6.3. EF Core

Entity Framework Core (skraćeno: EF Core) je objektno-relacijski razvojni okvir (engl. *Object-relational mapping, ORM*) koji služi za preslikavanje (engl. *mapping*) relacijskih tablica u C# objekte [24].

Pristupanje podacima i mijenjanje istih razvojni programer može postići mijenjanjem C# objekata a EF se brine da promjene budu odražene i na bazi podataka. Kako bi imali pristup bazi podataka koristimo *Context* klasu koja služi kao poveznica aplikacije i spremišta podataka.

EF nudi dvije glavne opcije početka korištenja: kreiranje spremišta podataka (engl. *Database first*), te kreiranjem modela (engl. *Code first*). Ako se koristi postojeća baza podataka sa definiranim relacijskim tablicama i ključevima za povezivanje odabire se *Database first* način rada. Tada je potrebno definirati modele prema postojećoj strukturi baze, na taj način dolazi do ograničenja za *Domain* modele jer se moraju prilagoditi strukturi baze.

U novijim i većim aplikacijama preporuča se *Code first* način rada. Modeliraju se prvo *Domain* modeli i na taj način se prati osnovna objektno orijentirana paradigma gdje je naglasak na modelima. Fokus se postavlja na ponašanje aplikacije umjesto na podatke koji su generirani za nju. U teoriji moguće je transformirati objekte u rekorde u bazi podataka bez mijenjanja strukture aplikacije, i dalje je potrebno dizajnirati modele aplikacije tako da sadrže primarne i strane ključeve i svojstva potrebna za generiranje tablica.

PRIMJER APLIKACIJE

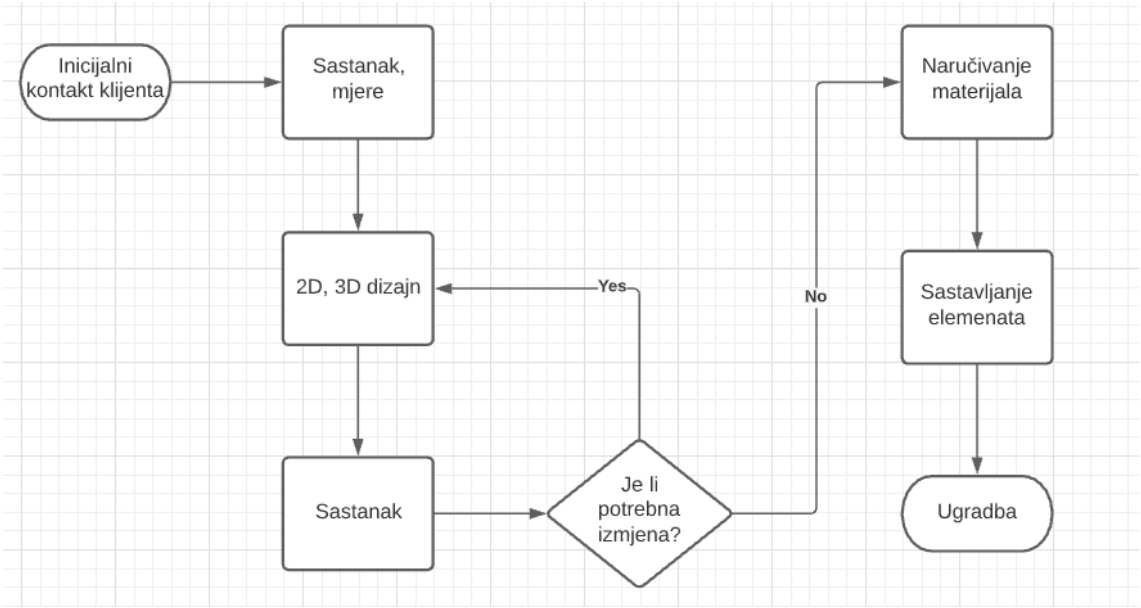
Kroz jednostavan primjer aplikacije prikazani su spomenuti principi. Slojevita arhitektura zajedno sa objektno orijentiranim načelima i ostalim principima ima smisla u većim i složenim aplikacijama koje se izrađuju i održavaju kroz veći period vremena (višegodišnji projekti) i za koje je bitna testabilnost pojedinog modula ili sloja.

Primjer aplikacije služi za razumijevanje spomenutih arhitektonskih principa te je većina funkcionalnosti zamišljena, odnosno implementacija nije provedena.

7.1. Opis problema

Biznis model koji se proučava u aplikaciji obuhvaća dizajn i sastavljanje namještaja po mjeri. Fiktivno poduzeće *FurnitureDesign* želi olakšati proces oglašavanja, dogovora s klijentom, plaćanja, organizacije djelatnika i nabavke potrebnih materijala kroz aplikaciju ili više njih.

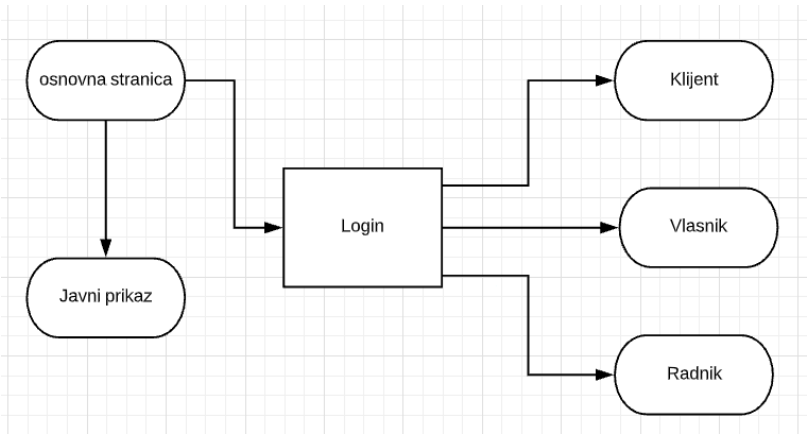
Slika 7.1 vizualizira tijek rada. Tijek rada poduzeća kreće kada klijent kontaktira poduzeće radi izrade npr. kuhinje. Osoba zadužena za inicijalni dogovor i prve mjere odgovara na poziv i dogovara sastanak. Na sastanku se definiraju mjere prostora u kojem će se kuhinja izgraditi te dogovaraju okvirni detalji za izradu; broj elemenata, ladica i njihovo pozicioniranje. Potom osoba zadužena za dizajn u adekvatnoj vanjskoj aplikaciji vizualno prikazuje model kuhinje. Nakon što je primjerak dostupan kontaktira se klijent. Ukoliko je klijent zadovoljan dizajnom predlaže se okvirna cijena materijala i rada. Naručuje se materijal; izrezani komadi iverice, okovi i ostalo. Nakon toga slijedi sastavljanje namještaja u radionici. Kao posljednji korak dogovara se termin ugradbe kuhinje i slijedi sama ugradnja i plaćanje.



Slika 7.1. Tijek rada poduzeća

7.2. Zahtjevi nad aplikacijom

Aplikacija mora sadržavati sljedeće funkcionalnosti za sadašnje i buduće klijente: prikaz gotovih projekata s galerijom fotografija i osnovnim informacijama, kontakt informacije poduzeća, formu za slanje upita budućih klijenata, login formu za postojeće klijente, poseban prikaz za klijente sa postojećim projektima i statusom gdje svaki od njih ima prikaz s detaljima; mjere, cijena, prikaz dizajna te informacije o sljedećem sastanku ili ugradbi (Slika 7.2).



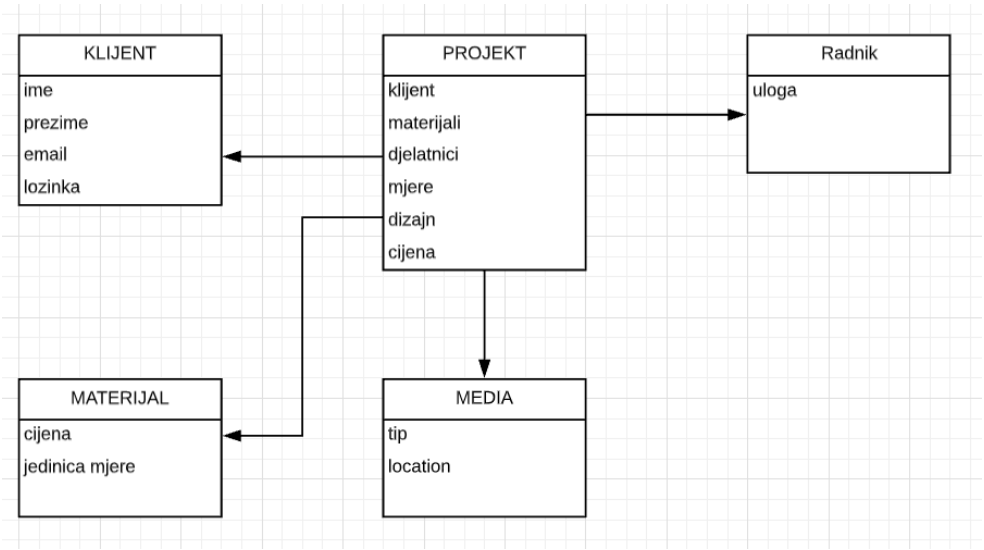
Slika 7.2. Ulazna točka aplikacije s mogućim putanjama

Za sam rad poduzeća potrebna je dodatna aplikacija ili dogradnja nad spomenutom gdje će radnici imati pristup kalendaru s informacijama o sastancima, dostupnosti materijala i slično te poseban prikaz za vlasnika poduzeća s detaljima naplate te informacijama o radnicima i klijentima.

Dodatak nad aplikacijom je sustav za provedbu inventure alata, strojeva i materijala.

7.3. Glavni model aplikacije

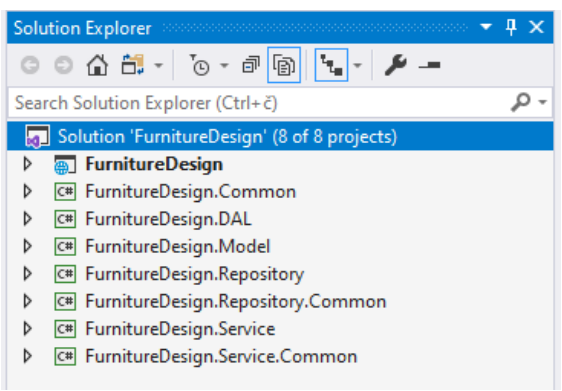
Glavni model aplikacije je sam projekt (Slika 7.3). Projekt pripada određenom klijentu, za njega se naručuje materijal, njemu pripadaju određeni djelatnici poduzeća sa svojim ulogama: radnik koji određuje mjerenja, radnik za dizajn, radnici za sastavljanje i ugradbu projekta. Projektu pripadaju određene mjere, sam dizajn sa galerijom slika i videa te cijenama materijala.



Slika 7.3. Pojednostavljeni prikaz modela aplikacije

7.4. Implementacija aplikacije

Prateći višeslojnu arhitekturu aplikacija sadržava više slojeva: DAL – za implementaciju *Entity* modela, *Repository* – za implementaciju metoda potrebnih za komunikaciju nad bazom podataka, Service – za implementaciju poslovne logike te API sloj koji će predstavljati završnu i početnu točku aplikacije i voditi komunikaciju s klijentima (npr. Angular kao *web framework*). Slika 7.4 prikazuje izgled *Solutiona* unutar *Visual Studia*.



Slika 7.4. Struktura aplikacije

Struktura aplikacije sastoji se od sljedećih projekata:

- *FurnitureDesign.DAL* sadržava klase entiteta potrebnih za kreiranje i komunikaciju s bazom podataka te migracijske datoteke potrebne za kreiranje i editiranje tablica koje entiteti predstavljaju. Ne sadrži referencu na projekt a referenciran je samo u *Composition Root* projektu.
- *FurnitureDesign.Common* sadržava klase koje se koriste u cijeloj aplikaciji, to su filteri za dohvaćanje podataka, *enumi* npr. *WorkerRoleEnum* sa ulogama radnika itd. Ne sadrži referencu na projekt, a referenciran je u svakom.
- *FurnitureDesign.Model* sadržava modele koji se koriste u kroz cijelu aplikaciju, predstavljaju domjenske modele. Ne sadrži referencu na projekt, a referenciran je u svakom.
- *FurnitureDesign.Repository* sadržava klase za komunikaciju s entitetima odnosno bazom podataka, implementira *repository pattern*. Ima referencu na *DAL*, *Model* te *Common* projekt.
- *FurnitureDesign.Repository.Common* sadržava sučelja za *repository* klase. Ima referencu na *Model* i *Common* projekt.
- *FurnitureDesign.Service* sadržava poslovnu logiku aplikacije. Ima referencu na *Model*, *Common* te *Repository.Common* projekt.
- *FurnitureDesign.Service.Common* sadržava sučelja za *service* klase, ima referencu na *Model* projekt te *Common* projekt. Ovaj projekt koristi samo *FurnitureDesign* projekt.
- *FurnitureDesign* sadržava ulaznu točku aplikacije (*Composition Root*) te API kontrolere za komunikaciju preko HTTP protokola. Kao *Composition Root* ovaj projekt ima referencu na sve projekte. Komunikacija API kontrolera treba teći isključivo prema *FurnitureDesign.Service.Common*.

7.4.1. DAL projekt

U DAL projektu kreirani su entiteti: *Project, Media, Material, Client, Worker, Date*. Primjer klase entiteta prikazan je na slici 7.5.

```
namespace FurnitureDesign.DAL
{
    public class ProjectEntity
    {
        public Guid Id { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        public string Measurements { get; set; }

        public Guid ClientId { get; set; }

        public DateTime DateCreated { get; set; }

        public DateTime DateUpdated { get; set; }

        #region Navigation Properties
        public virtual ICollection<MaterialEntity> Materials { get; set; }
        public virtual ICollection<WorkerEntity> Workers { get; set; }
        public virtual ICollection<MediaEntity> Media { get; set; }
        public virtual ICollection<DateEntity> Dates { get; set; }
        #endregion
    }
}
```

Slika 7.5. *Project* entitet

Svaki entitet mora imati sljedeća svojstva: *Id* jer je to primarni ključ, *DateCreated* i *DateUpdated*; zbog lakšeg pretraživanja. Svako svojstvo ima svoj tip (*Guid, string, DateTime* itd.) te metode za dohvaćanje i postavljanje vrijednosti (*get* i *set*). Navigacijska svojstva služe za predstavljanje povezanosti entiteta. *ProjectEntity* biti će mapiran u relacijsku bazu podataka, točnije tablicu *Project*, svako svojstvo predstavlja stupac tablice, te objekt ove klase predstavlja redak u tablici. Navigacijsko svojstvo *Workers* predstavlja vezu 1:N, *Project:Worker*, odnosno svaki projekt može imati više radnika.

Za kreiranje baze podataka potrebna je *ApplicationDbContext* klasa (Slika 7.6). Ova klasa sadržava podatak o konekciji na bazu, te *DbSet* svojstva pomoću kojih navodimo tipove i imena tablica koje želimo kreirati. Ona se koristi unutar EF Core naredbi za kreiranje i editiranje tablica.

```
namespace FurnitureDesign.DAL
{
    public class ApplicationDbContext : ApiAuthorizationDbContext<ApplicationUser>
    {
        public ApplicationDbContext(
            DbContextOptions options,
            IOptions<OperationalStoreOptions> operationalStoreOptions) : base(options, operationalStoreOptions)
        {
        }

        public DbSet<MaterialEntity> Material { get; set; }
        public DbSet<ClientEntity> Client { get; set; }
        public DbSet<DateEntity> Date { get; set; }
        public DbSet<MediaEntity> Media { get; set; }
        public DbSet<ProjectEntity> Project { get; set; }
        public DbSet<WorkerEntity> Worker { get; set; }
    }
}
```

Slika 7.6. *ApplicationDbContext* klasa

7.4.2. Model projekt

Model projekt sadržava DTO klase, odnosno modele koji se koriste u poslovnom dijelu aplikacije te služe za prijenos podataka od *Repository* projekta do API projekta. DTO modeli mogu ali ne moraju biti preslika *Entity* modela. Primjer *ProjectDto* modela prikazan je na slici 7.7.

```
namespace FurnitureDesign.Model
{
    public class ProjectDto
    {
        public Guid Id { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        public string Measurements { get; set; }

        public Guid ClientId { get; set; }

        public DateTime DateCreated { get; set; }

        public DateTime DateUpdated { get; set; }

        public ClientDto Client { get; set; }

        public IEnumerable<MaterialDto> Materials { get; set; }

        public IEnumerable<WorkerDto> Workers { get; set; }

        public IEnumerable<MediaDto> Media { get; set; }

        public IEnumerable<DateDto> Dates { get; set; }
    }
}
```

Slika 7.7. *ProjectDto* klasa

ProjectDto nema navigacijska svojstva nego sadržava liste i objekte potrebnih modela kako bi se u cijelosti definirao *Project* model.

7.4.3. *Common* projekt

Common projekt sadržava sve klase koji se sigurno koriste svakom sloju aplikacije a nisu DTO model. To su najčešće filteri i *enumi* za opisivanje modela. Primjer filtera prikazan je na slici 7.8, a *enuma* na slici 7.9.

```
namespace FurnitureDesign.Common
{
    public class ProjectFilter: FilterBase
    {
        public Guid ClientId { get; set; }

        public Guid WorkerId { get; set; }
    }
}
```

```
namespace FurnitureDesign.Common
{
    public class FilterBase
    {
        public string SearchString { get; set; }

        public string SortOrder { get; set; }

        public int PageNumber { get; set; }

        public int PageSize { get; set; }
    }
}
```

Slika 7.8. ProjectFilter i FilterBase klase

FilterBase klasa je osnovna filter klasa koju nasljeđuju svi filteri jer je pretpostavljeno da će svaki filter imati npr svojstva za odabir broja stranice te veličine stranice, ovime se poštuje DRY princip a koristi se jedno od osnovnih načela objektno orijentiranog programiranja. *ProjectFilter* nasljeđuje *FilterBase* klasu i time njena svojstva.

```
namespace FurnitureDesign.Common
{
    public enum WorkerTypeEnum
    {
        Measurement, Design, CEO, Marketing
    }
}
```

Slika 7.9. WorkerTypeEnum klasa

7.4.4. *Repository* i *Repository.Common* projekti

Repository.Common projekt sastoji se od sučelja za *repository* klase. U svakom sučelju opisane su CRUD i dodatne metode potrebne aplikaciji za kreiranje, čitanje, editiranje i brisanje podataka. Svaki od modela ima svoj *repository*. Na slici 7.10 prikazan je primjer jednog sučelja.

```
namespace FurnitureDesign.Repository.Common
{
    public interface IProjectRepository
    {
        Task<IEnumerable<ProjectDto>> FindAsync(ProjectFilter filter = null);
        Task<ProjectDto> GetAsync(Guid Id);
        Task<ProjectDto> GetWithDetails(Guid id);
        Task<bool> CreateAsync(ProjectDto project, IUnitOfWork uow = null);
        Task<bool> UpdateAsync(ProjectDto project, IUnitOfWork uow = null);
        Task<bool> DeleteAsync(ProjectDto project, IUnitOfWork uow = null);
    }
}
```

Slika 7.10 *IProjectRepository* sučelje

Sučelje *IProjectRepository* sadrži metode za dohvaćanje projekta putem jedinstvenog ključa, kreiranje novog projekta, editiranje postojećeg, brisanje projekta te dohvaćanje liste projekata prema filteru. *Create*, *Update* te *Delete* metoda kao parametar imaju i *uow* (Slika 7.11.). Korištenje *UnitOfWorka* označava da je to jedan u nizu naredbi koje se trebaju istovremeno izvršiti nad bazom podataka. Na taj način se osigurava jedna transakcija za više operacija nad bazom. Ukoliko se koristi *uow* potrebno je kao posljednju naredbu izvršiti *Commit* kako bi naredbe bile provedene.

```
namespace FurnitureDesign.Repository.Common
{
    public interface IUnitOfWork: IDisposable
    {
        Task<int> CommitAsync();
    }
}
```

Slika 7.11. *IUnitOfWork* sučelje

Implementacija *Commit* naredbe koristi *ApplicationDbContext* ugrađenu naredbu za spremanje izmjena. (Slika 7.12.)

```
namespace FurnitureDesign.Repository
{
    public class UnitOfWork: IUnitOfWork
    {
        private ApplicationDbContext DbContext { get; set; }

        public UnitOfWork(ApplicationDbContext dbContext)
        {
            if (dbContext == null)
            {
                throw new ArgumentException("DbContext");
            }
            this.DbContext = dbContext;
        }

        public Task<int> CommitAsync()
        {
            return DbContext.SaveChangesAsync();
        }

        public void Dispose()
        {
            DbContext.Dispose();
        }
    }
}
```

Slika 7.12. *UnitOfWork* implementacija

ProjectRepository implementira *IProjectRepository* sučelje, korištenjem *Dependency Injectiona*, točnije *ConstructorInjectiona* u klasu se ubacuju implementacije *ApplicationDbContexta* te osnovne *Repository* klase (slika 7.13.).

```
public class ProjectRepository :  IProjectRepository
{
    private ApplicationDbContext DbContext { get; set; }

    private IRepository<ProjectEntity> Repository { get; set; }

    public ProjectRepository(ApplicationDbContext dbContext, IRepository<ProjectEntity> repository)
    {
        this.DbContext = dbContext;
        this.Repository = repository;
    }
```

Slika 7.13. *ProjectRepository* atributi i konstruktor

Slika 7.14. prikazuje implementaciju *Find* metode za pretraživanje tablice *Project* pomoću filtera. Koristeći ugrađene *ApplicationDbContext* metode moguće je dohvatiti i veze projekt tablice (npr. *Workers* i *Materials*). Iz primjera je vidljivo i mapiranje *entity* objekta u DTO objekt, što je bitno kako bi poštivali slabu povezanost slojeva. Sljedeći sloj koji koristi *Repository* klase ne smije znati za implementaciju baze, pa tako i modele koji se koriste unutar DAL projekta.

```
public async Task<IEnumerable<ProjectDto>> FindAsync(ProjectFilter filter = null)
{
    var entities = await this.DbContext.Project
         .Include(project => project.Workers)
         .Include(project => project.Materials)
         .Where(project => project.ClientId == filter.ClientId)
         .Skip((filter.PageNumber - 1)*filter.PageSize)
         .Take(filter.PageSize)
         .ToListAsync();
    var list = new List<ProjectDto>() { };
    foreach(var entity in entities)
    {
        list.Add(this.MapToProjectDto(entity));
    }
    return list;
}
```

Slika 7.14. Implementacija *Find* metode

Primjer korištenja *uow* parametra vidljiv je na slici 7.15. Ukoliko je proslijeđen *uow* promijene se neće odmah spremiti nego se očekuje od pozivatelja da pozove *uow.CommitAsync* kao zadnju naredbu kako bi izvršio sve željene promjene u istoj transakciji. Kao parametar CreateAsync metoda prima DTO model jer vanjski sloj ne zna za postojanje *Entity* modela te ga je potrebno mapirati u *entity* model prije spremanja. Na sličan način implementiraju se i *Update* i *Delete* metode.

```
public async Task<bool> CreateAsync(ProjectDto project, IUnitOfWork uow = null)
{
    var entity = this.MapToProjectEntity(project);
    this.Repository.Add(entity);
    if (uow == null)
    {
        return await DbContext.SaveChangesAsync() > 0;
    }
    else return true;
}
```

Slika 7.15. Implementacija *Create* metode

7.4.5. *Service* i *Service.Common* projekti

Service kao poslovni sloj sadržava CRUD metode, slično kao i *repository* sloj, uz to unutar ovog sloja svaka poslovna logika treba biti implementirana. Na slici 7.16. vidljiva je implementacija *Create* metode za *Project*. Predviđeno je kreiranje pojedinih sastavnih dijelova projekta uz kreiranje samog projekta, te je zbog toga korišten *UoW pattern*, odnosno jedna transakcija koja će kreirati projekt i sve potrebne retke drugih tablica.

```
public async Task<bool> CreateAsync(ProjectDto project)
{
    if (project.Materials != null)
    {
        await this.MaterialRepository.CreateAsync(project.Materials, UnitOfWork);
    }
    if (project.Media != null)
    {
        await this.MediaRepository.CreateAsync(project.Media, UnitOfWork);
    }
    // ...
    await this.Repository.CreateAsync(project);
    return await UnitOfWork.CommitAsync() > 0;
}
```

Slika 7.16.

Slično kao i u *Repository.Common* sloju, *Service.Common* projekt sadržava sučelja za svaki od servisa koji su implementirani unutar *Service* projekta.

7.4.6. *Composition Root* projekt

Unutar *Composition Root* projekta osim ulazne točke aplikacije nalaze se API kontroleri koji putem HTTP protokola primaju zahtjeve i šalju odgovore. Primjer metode *Get* koja dohvaća projekt redak preko primarnog ključa vidljiv je na slici 7.17.

```
[HttpGet("{id}", Name = "Get")]
[Authorize("worker")]
public async Task<IActionResult> Get(Guid id)
{
    var dto = await this.Service.GetAsync(id);
    var response = this.MapToProject(dto);
    if (response != null)
    {
        return Ok(response);
    }
    else
    {
        return NotFound();
    }
}
```

Slika 7.17. Primjer *Get* metode

Prateći HTTP pravila objašnjena u prvom poglavlju potrebno je implementirati sve potrebne *endpointe* za korištenje aplikacije. DTO modeli trebaju se koristiti isključivo za poslovnu logiku te se iz tog razlika stvaraju *ViewModeli* (u ovom slučaju Project model) te prije nego se pozove servis mapira se *ViewModel* u DTO model, jer servis nema potrebe znati za postojanje *ViewModela*.

Osim implementacije *endpointa CompositionRoot* projekt, kao ulazna točka aplikacije sadržava i implementaciju *Dependency Injection Containera* te naznačene veze između implementacije i sučelja za svaki sloj (Slike 7.18. i 7.19.).

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services
        .AddDatabase(Configuration)
        .AddRepositories()
        .AddServices();
}
```

Slika 7.18. *ConfigureServices* metoda unutar *Startup* klase

```
public static class IServiceCollectionExtensions
{
    public static IServiceCollection AddDatabase(this IServiceCollection services, IConfiguration configuration)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
        {
            options.UseSqlServer(configuration.GetConnectionString("DefaultConnection"));
        });
        services.AddScoped<Func<ApplicationDbContext>>((provider) => () => provider.GetService<ApplicationDbContext>());
        services.AddScoped<IUnitOfWork, UnitOfWork>();
        return services;
    }

    public static IServiceCollection AddRepositories(this IServiceCollection services)
    {
        return services
            .AddScoped(typeof(IRepository<>), typeof(Repository<>))
            .AddScoped<IProjectRepository, ProjectRepository>();
        // ...
    }

    public static IServiceCollection AddServices(this IServiceCollection services)
    {
        return services.AddScoped<IProjectService, ProjectService>();
        // ...
    }

}
```

Slika 7.19. *IServiceCollectionExtensions* metode

ZAKLJUČAK

Svaka od spomenutih arhitektura kroz ovaj rad trebala bi ako je implementirana dobro donijeti sljedeće pogodnosti. Sustav koji je neovisan o razvojnom okviru, tj. arhitekturu koja ne ovisi o postojanju neke biblioteke te na taj način mogućnost korištenja okvira kao alata, umjesto izrade sustava unutar ograničenja razvojnog okvira. Testabilnost jer se poslovna pravila mogu testirati bez korisničkog sučelja, *web*-poslužitelja ili bilo kojeg drugog vanjskog elementa. Neovisnost o korisničkom sučelju koje se može lako promijeniti bez promjene ostatka sustava. Neovisnost o bazi podataka.

Izrada softvera na način opisan ovim arhitekturama ima smisla ako govorimo o srednjim do velikim sustavima za koje se očekuje duže vrijeme razvoja i održavanja. Često sustavi koji koriste DDD (engl. *Domain Driven Development*) ili TDD (engl. *Test Driven Development*) imaju upravo arhitekture opisane u ovom radu zbog dobre podloge *Agile* načinu rada kao i pogodnosti testabilnosti ovakvih sustava. Osnovni cilj APM-a (engl. *Agile Project Managment*) je imati kraće intervale razvoja sa puštanjem u rad projekta kako bi se već dogovoreni zahtjevi provjerili te dogovorili budući.

LITERATURA

[1] D. Esposito and A. Saltarello, "Architecting Microsoft® .NET Solutions for the Enterprise", Microsoft Press, 2008.

[2] A. Cockburn, "Hexagonal Architecture", https://alistair.cockburn.us/hexagonal-architecture/ (zadnji posjet 10.9.2021.)

[3] J. Palermo, "The Onion Architecture", https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (zadnji posjet 10.9.2021.)

[4] R. C. Martin, "The Clean Architecture", http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (zadnji posjet 10.9.2021.)

[5] R. C. Martin, "Clean Architecture: A Craftsman's Guide to Software Structure and Design", Prentice Hall, 2017

[6] Netflix Technology Blog, "Ready for Changes with Hexagonal Architecture", https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749 (zadnji posjet 12.9.2021.)

[7] M. Fowler, "Software Architecture Guide", https://martinfowler.com/architecture/ (zadnji posjet 11.9.2021)

[8] M. Fowler, "Patterns of Enterprise Application Architecture", Pearson Education, 2002.

[9] S. Hanselman, https://www.hanselman.com/ (zadnji posjet 11.9.2021)

[10] U. Dahan, "Fear those Tiers", https://udidahan.com/2007/03/20/fear-those-tiers/ (zadnji posjet 11.9.2021)

[11] https://iasaglobal.org/ (zadnji posjet 11.9.2021.)

[12] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", doktorska disertacija, p. 180, 2000.

[13] MDN Web Docs, "An overview of HTTP", https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (zadnji posjet 11.9.2021)

[14] RFC-2616, The Internet Society, "Hypertext Transfer Protocol", https://datatracker.ietf.org/doc/html/rfc2616 (zadnji posjet 11.9.2021)

[15] J. Purdum, "Beginning Object-Oriented Programming with C#", Indianapolis, IN: Wiley, 2013.

[16] B. Joshi, "Beginning SOLID Principles and Design Patterns for ASP.NET Developers

[17] E. W. Dijkstra, "Selected Writings on Computing", Springer New York, 1982.

[18] R. C. Martin, "Agile Software Development: Principles, Patterns, and Practices", Pearson Education, 2003.

[19] M. Seemann, "Dependency Injection in .Net", Manning, 2012.

[20] J. P. Smith, "Entity Framework Core in Action", Manning, 2018.
[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch aka Gang of Four (GoF), "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1994.
[22] M. Seemann, "Layers, Onions, Ports, Adapters: it's all the same", https://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/ (zadnji posjet 11.9.2021)
[23] R. Peres, "Modern Web Development with ASP.NET Core 3", Second Edition, Packt Publishing, 2020.
[24] R. Peres, "Entity Framework Core Cookbook", Second Edition, Packt Publishing, 2016.

SAŽETAK

Diplomski rad opisuje više modernih arhitektura softvera. Objašnjeni su različiti prikazi slojevite arhitekture: *clean, onion* i heksagonalna arhitektura. Radi razumijevanja teme obuhvaćeni su bitni principi i načela objektno orijentiranog programiranja. Kao najvažnije načelo slojevite arhitekture prikazan je primjer korištenja principa inverzije ovisnosti. Spomenuti principi i slojevita arhitektura pokazani su na primjeru jednostavne aplikacije izrađene u .NET Core tehnologiji. Kroz rad navedene su prednosti korištenja ove arhitekture te u kojim slučajevima ova arhitektura nije dobro rješenje. Za razvoj manjih aplikacija slojevita arhitektura nije pogodna, no za aplikacije srednjeg do velikog opsega vrijedi uložiti vrijeme na planiranje i implementaciju slojeva zbog veće održivosti, testabilnosti ovakvog sustava te lakše raspodijele posla u većem timu programera.

Ključne riječi: .NET Core, *clean*, heksagonalna arhitektura, *onion*, princip inverzije ovisnosti, slojevita arhitektura

ABSTRACT

Architecture of modern web applications in ASP.NET Core platform

The thesis describes several modern software architectures. Different representations of layered architecture are explained: clean, onion, and hexagonal architecture. In order to understand the topic, the essential principles and the principle of object-oriented programming are included. As the most important principle of layered architecture this paper presents an example of the dependency inversion principle. The mentioned principles and layered architecture are shown on the example of simple application made in .NET Core technology. The paper presents the advantages of using mentioned architectures and the cases in which these architectures are poor solution. Layered architecture is not suitable for the development of small applications, but for medium to large-scale applications it is worth investing time in planning and implementation of layers due to greater maintainability, testability and easier distrubution of work in larger team.

Key words: .NET Core, clean, dependency inversion principle, hexagonal, multi-layered architecture, onion

ŽIVOTOPIS

Martina Grgić rođena je 15.12.1995. godine u Osijeku, RH. Završila je srednju školu, III. Gimnazija Osijek, te je 2014. godine upisala preddiplomski sveučilišni studij Računarstvo, Sveučilišta Josipa Jurja Strossmayera u Osijeku koji završava 2017. godine. Iste godine upisuje diplomski studij na istom fakultetu, smjer programsko inženjerstvo gdje trenutno studira. Godine 2017. zaposlena je u Mono d.o.o. gdje je trenutno zaposlena.

Martina Grgić