PRIMJENA PARADIGME CLEAN ARHITEKTURE ZA RAZVOJ REST API-JA ZA APLIKACIJU SVEUČILIŠNE KNJIŽNICE

Tikvica, Mato

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I RAČUNARSTVO

MATO TIKVICA

PRIMJENA PARADIGME *CLEAN* ARHITEKTURE ZA RAZVOJ REST API-JA ZA APLIKACIJU SVEUČILIŠNE KNJIŽNICE

DIPLOMSKI RAD

Dubrovnik, srpanj, 2024.

SVEUČILIŠTE U DUBROVNIKU

ODJEL ZA ELEKTROTEHNIKU I
RAČUNARSTVO

PRIMJENA PARADIGME *CLEAN* ARHITEKTURE ZA RAZVOJ REST API-JA ZA APLIKACIJU SVEUČILIŠNE KNJIŽNICE

DIPLOMSKI RAD

Studij: Primijenjeno/poslovno računarstvo

Kolegij: Diplomski rad

Mentor: prof. dr. sc. Mario Miličević

Komentorica: Ana Kešelj Dilberović, mag. ing. comp.

Student: Mato Tikvica, JMBAG: 0035223178

Dubrovnik, srpanj, 2024.

Sažetak

Clean Arhitektura je oblik arhitekture koji naglašava jasno odvajanje odgovornosti s ciljem stvaranja modularnog i robusnog sustava jednostavnog za održavanje i skaliranje. Ovaj diplomski rad istražuje primjenu *Clean* Arhitekture u razvoju REST API-ja za sustav sveučilišne knjižnice. U radu su opisane motivacija, prednosti i izazovi *Clean* Arhitekture. Također, opisana su i načela koja su važna za primjenu *Clean* Arhitekture nad programom. Pokazana je praktična primjena arhitekture dodavanjem nove funkcionalnosti u programu, kao i razni testovi za osiguravanje pouzdanosti i integriteta programa. Implementirane su i razne tehnologije poput CI/CD cjevovoda kako bi se što bolje i jednostavnije osigurala dosljednost programa.

Ključne riječi: Clean Arhitektura, .NET, REST API

Abstract

Clean Architecture is a form of architecture that emphasizes clear separation of concerns with the goal of creating a modular and robust system that is easy to maintain and scale. This thesis explores the application of *Clean* Architecture in the development of a REST API for a university library system. Motivation, advantages, and disadvantages of *Clean* Architecture are also described. Additionally, it outlines the principles that are important for applying *Clean* Architecture to a program. The practical application of the architecture is demonstrated by adding new functionality to the program, as well as various tests to ensure the reliability and integrity of the program. Various technologies, such as CI/CD pipelines, are also implemented to ensure consistency.

Key words: Clean Architecture, .NET, REST API

SADRŽAJ

1 UVOD

Clean Arhitektura (engl. *Clean Architecture*) je koncept koji je predstavio Robert C. Martin. Ova arhitektura naglašava razdvajanje odgovornosti, stvaranje sustava neovisnog o okvirima (engl. *framework*), korisničkom sučelju, bazama podataka i vanjskim servisima. Cilj ove arhitekture je proizvesti program koji je jednostavan za održavanje, testiranje i skaliranje [1].

Tradicionalni sustavi često dovode do čvrsto povezanih komponenti što cijeli razvoj i održavanje aplikacije čini jako teškim. Rastom sustava raste složenost što za rezultat daje sporiji razvoj i veći rizik od novih grešaka.

U ovom diplomskom radu se istražuje primjena *Clean* Ahitektura u razvoju REST API-ja za sustav sveučilišne knjižnice. Pridržavajući se načela *Clean* arhitekture, cilj je postići modularnu, testiranu i skalabilnu aplikaciju.

1.1 Svrha i ciljevi

Primarni cilj diplomskog rada je implementacija *Clean* Arhitekture za razvoj REST API-ja sveučilišne knjižnice. Ciljevi implementacije su:

- Strukturiranje koda u različite slojeve. Svaki sloj sa svojim odgovornostima kako bi se poboljšalo odvajanje odgovornosti svakog sloja i smanjila međusobna ovisnost.
- Osigurati da se svaki sloj može neovisno testirati, što omogućuje pouzdanije i lakše procese testiranja.
- Razviti bazu koda koju je lako razumjeti, modificirati ili proširiti, čime se smanjuju troškovi i napor potrebni za buduće održavanje.
- Upotrijebiti suvremene prakse i tehnologije razvoja programske podrške kako bi se stvorila robusna aplikaciju za budućnost.

1.2 Opseg rada

Rad se fokusira na razvoj REST API-ja za upravljanje sveučilišne knjižnice, uključujući evidenciju knjiga, studenata i obradu posudba knjiga. U diplomskom radu će se implementirati osnovne funkcije

kao što su dodavanje i upravljanje knjigama i obradama posudbe i povrata knjiga.

1.3 Struktura rada

Ovaj diplomski rad sastoji se od nekoliko poglavlja. Prvo poglavlje objašnjava svrhu, ciljeve i opseg rada, te daje pregled strukture rada. Drugo poglavlje prikazuje motivaciju za primjenu *Clean* Arhitekture. Treće poglavlje uvodi pojam arhitekture, opisuje *Clean* Arhitekturu, pravila prema kojima se ta arhitektura treba graditi te izazove ove arhitekture. Četvrto poglavlje, Korištene tehnologije, opisuje tehnologije korištene u razvoju REST API-ja. Peto poglavlje detaljno opisuje arhitekturu sustava, uključujući slojeve domene, aplikacije, infrastrukture i prezentacije, te objašnjava obrasce pomoću kojih je građena arhitektura. Šesto poglavlje prikazuje praktične primjene i prednosti *Clean* Arhitekture kroz dodavanje nove funkcionalnosti. Sedmo poglavlje obuhvaća jedinične, integracijske i arhitekturne testove zajedno s primjerima za svaki od testova. Osmo poglavlje opisuje plan implementacije CI/CD-a, proces grananja i kontejnerizaciju s Dockerom, te prikazuje CI/CD cjevovode. Deveto poglavlje govori o implementaciji predmemoriranja s Redisom. Zaključno poglavlje sažima glavne zaključke rada.

2 Motivacija

Ako se nije vodilo puno brige prije samog razvoja programske podrške i ako arhitektura nije pomno dizajnirana, za posljedicu će imati negativne efekte na razvoj i održavanje cijelog sustava. Tradicionalne monolitne arhitekture u kojima je cijela programska podrška razvijena i implementirana kao jedinstvena cjelina, dovodi do čvrsto povezanih komponenti. U takvim vrstama arhitekture, cijeli sustav je u međusobnoj ovisnosti i svaka promjena u kodu može ugroziti cijeli sustav.

Prema istraživanju, 69% razvijatelja izvijestilo je kako tehnički dug usporava razvoj novih funkcionalnosti programskih rješenja. Tehnički dug je ono što se događa nakon donošenja odluka o razvoju ili implementaciji programskog rješenja koje daju prednost brzini nad kvalitetom. Slično kao financijski dug, tehnički dug zahtijeva otplatu, ali može biti puno kompliciraniji i dugotrajniji od mjesečnih "plaćanja" [2].

Produktivnost razvijatelja također značajno opada u sustavima s lošom arhitekturom. Razvijatelji troše više vremena na razumijevanje i prilagodbu postojećeg koda, što smanjuje vrijeme dostupno za razvoj novih značajki. Studije pokazuju da tehnički dug može smanjiti dnevnu produktivnost razvijatelja za čak 23%. [2]

Cilj ovog diplomskog rada je istražiti primjenu *Clean* Arhitekture u razvoju REST API-ja za sustav sveučilišne knjižnice. Pridržavajući se načela *Clean* Arhitekture, cilj je postići modularnu, testiranu i skalabilnu aplikaciju koja će biti jednostavna za održavanje i nadogradnju. Ovaj rad će pokazati kako pravilno strukturirana arhitektura može značajno poboljšati kvalitetu i učinkovitost razvoja programske podrške, posebno u kontekstu upravljanja sveučilišnom knjižnicom.

3 Arhitektura

3.1 Uvod u arhitekturu

"Programska arhitektura" često izaziva različita tumačenja i definicije. Programska arhitektura obuhvaća strukturu programskog sustava visoke razine, disciplinu stvaranja takvih struktura i dokumentaciju tih struktura. Međutim, unutar domene programske podrške postoji značajna rasprava o preciznoj definiciji arhitekture. [3]

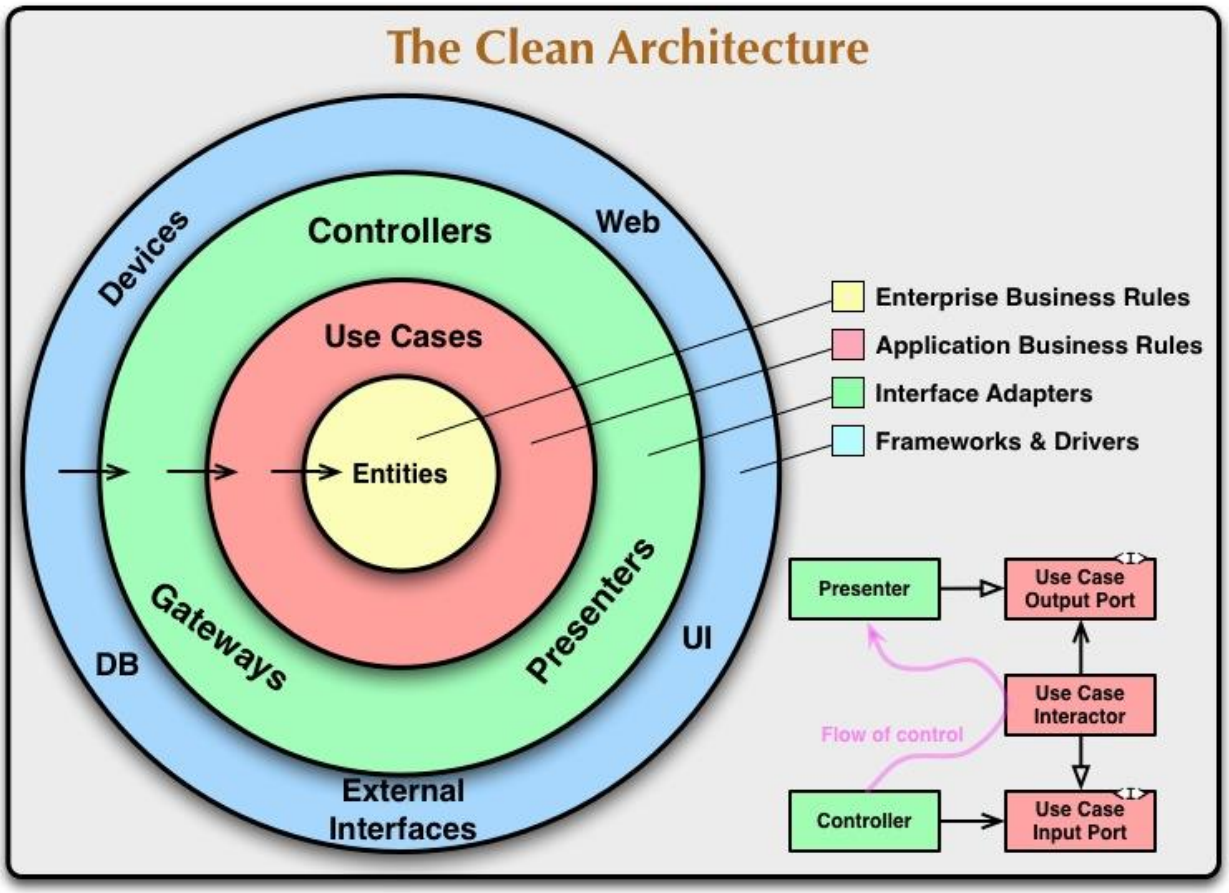
3.1.1 Fowlerova i Johnsonova perspektiva

Martin Fowler, poznati autor i razvijatelj programske podrške, daje pogled na arhitekturu oblikovanu razmjenom e-pošte s Ralphom Johnsonom. U toj razmjeni, Johnson je doveo u pitanje uobičajene definicije arhitekture koje se fokusiraju na "temeljnu organizaciju sustava" ili "način na koji su komponente najviše razine spojene zajedno." Tvrdio je da ne postoji objektivan način definiranja što je temeljna ili visoka razina. Umjesto toga, Johnson je predložio da je bolji pogled na arhitekturu zajedničko razumijevanje dizajna sustava koje imaju stručni programeri.

Drugi uobičajeni stil definicije arhitekture jest da ona uključuje "odluke o dizajnu koje je potrebno donijeti u ranoj fazi projekta." Johnson je također kritizirao ovo, sugerirajući kako se točnije opisuje kao odluke koje želite donijeti ispravne na početku projekta.

Johnson je zaključio da je “Arhitektura bitna stvar. Što god to bilo”. Iako ovo na prvu može zvučati trivijalno, nosi značajnu dubinu. To znači kako je bit razmišljanja o arhitekturi odlučivanje o tome što je važno i zatim usmjeravanje energije na održavanje tih važnih elemenata u dobrom stanju. Kako bi razvijatelj postao arhitekt, mora biti sposoban prepoznati koji su elementi važni i razumjeti koji će elementi vjerojatno uzrokovati ozbiljne probleme ako se njima ne upravlja pravilno [3].

3.2 *Clean* Arhitektura



Slika 1 Shema Clean Arhitekture [1]

Clean Arhitektura (engl. *Clean Architecture*) je arhitektura koju je predstavio Robert C. Martin, također poznat kao "Uncle Bob", u svojoj knjizi "Clean Architecture: A Craftsman's Guide to Software Structure and Design". Naglašava razdvajanje odgovornosti u programskim sustavima i ima za cilj stvoriti kod koji je modularan, testiran i održiv tijekom vremena.

Glavni cilj je odvajanje odgovornosti (engl. *separation of concerns*). Odvajanje odgovornosti se postiže dijeleći program na slojeve.

Svaka od ovih arhitektura proizvodi sustave koji su:

- Neovisni o radnom okviru (engl. *framework*). Arhitektura ne ovisi o postojanju neke biblioteke bogate značajkama. To omogućuje korištenje radnog okvira kao alata za izradu arhitekture, umjesto prilagođavanja sustava ograničenim mogućnostima radnih okvira.
- Podložni testiranju. Poslovna logika se može testirati bez korisničkog sučelja, baze podataka, web poslužitelja ili bilo kojeg drugog vanjskog elementa.
- Neovisni o korisničkom sučelju (engl. *User interface, UI*). UI se može lako promijeniti, bez promjene ostatka sustava. Npr. web sučelje može se zamijeniti nekim drugim sučeljem bez mijenjanja poslovnih pravila.
- Neovisni o bazi podataka. Jednako kao i kod korisničkog sučelja, baza podataka se može isto mijenjati bez mijenjanja ostatka sustava, primjerice SQL Server za se može zamijeniti za Mongo. Poslovna pravila nisu vezana za bazu podataka.
- Neovisno o bilo kojem vanjskoj usluzi ili sustavu. Poslovna pravila ne znaju ništa o vanjskim komponentama.

Koncentrični krugovi (Slika 1) predstavljaju drugačija područja programske podrške. Unutarnji krugovi predstavljaju pravila, dok vanjski predstavljaju mehanizme.

Najvažnije pravilo je pravilo ovisnosti. Ovisnosti izvornog koda mogu usmjeravati jedino prema unutra. Unutarnji krug ne zna ništa o vanjskom krugu.

Primjenom ovih pravila, postiže se sustav podložan testiranju sustav koji je otporan na bilo kakve vanjske promjene. [1]

3.3 Pravilo ovisnosti (engl. *Dependency rule)*

Pravilo ovisnosti ključno je za *Clean* Arhitekturu. Martin navodi kako ovisnosti izvornog koda moraju biti usmjerene prema unutra. Unutarnji krugovi, koji sadrže logiku visoke razine, ne bi trebali ovisiti o vanjskim krugovima, koji sadrže detalje niske razine (Slika 1) [1].

3.4 Entiteti

Entiteti obuhvaćaju poslovna pravila na razini cijelog sustava. Entitet može biti objekt s metodama,

ili može biti skup podatkovnih struktura i funkcija. Nije važno sve dok se entiteti mogu koristiti u različitim aplikacijama unutar sustava.

Ako je samo jedna aplikacija u pitanju, ti entiteti su poslovni objekti aplikacije. Oni obuhvaćaju najopćenitija i najviša pravila. Najmanje je vjerojatno da će se mijenjati kada se nešto vanjsko promijeni. Nikakva operativna promjena bilo koje specifične aplikacije ne bi trebala utjecati na sloj entiteta. [1]

3.5 Slučaji upotrebe

Programsko rješenje u ovom sloju sadrži poslovna pravila specifična za aplikaciju. Učahuruje i implementira sve slučaje korištenja sustava. Ovi slučaji upotrebe orkestriraju tok podataka do i od entiteta i usmjeravaju te entitete na korištenje svojih poslovnih pravila za kako bi postigli ciljeve slučaja upotrebe.

Nije za očekivat da će promjene u ovom sloju utjecati na entitete. Također nije za očekivat da će na ovaj sloj utjecati promjene vanjskih usluga kao što su baza podataka, korisničko sučelje ili bilo koji od uobičajenih okvira. Ovaj sloj je izoliran od takvih usluga.

Međutim, za očekivati je da će promjene u radu aplikacije utjecati na slučaje upotrebe, a time i na programsku podršku u ovom sloju. Ako se pojedinosti slučaja upotrebe promijene, to će sigurno utjecati na neki kod u ovom sloju. [1]

3.6 Adapterski sloj (engl. *Interface Adapters*)

Programska podrška u ovom sloju je skup adaptera koji pretvaraju podatke iz formata koji je najprikladniji za slučajeve upotrebe i entitete u format koji je najprikladniji za neku vanjsku uslugu poput baze podataka ili weba.

Slično tome, u ovom sloju se podaci pretvaraju iz oblika najprikladnijeg za entitete i slučajeve upotrebe u oblik najprikladniji za bilo koji okvir za pohranu koji se koristi, npr. baza podataka. Nijedan kod unutar ovog kruga ne bi trebao znati išta o bazi podataka. Ako je baza podataka SQL baza, cijeli SQL bi trebao biti ograničen na ovaj sloj, a posebno na dijelove ovog sloja koji se odnose na bazu podataka. [1]

3.7 Okviri i upravljački programi (engl. *Frameworks and Drivers*)

Sloj koji se nalazi najviše prema vanka je općenito se sastoji od radnih okvira i alata kao što su baza podataka, web okvir, itd. Općenito se ne piše mnogo koda u ovom sloju, osim veznog koda koji komunicira sa sljedećim unutarnjim slojem.

Ovaj sloj je mjesto gdje se nalaze svi detalji kao što je baza. Te usluge se drže na vanjskoj strani kako bi nanijele što manje štete programskoj podršci. [1]

3.8 SOLID načela

U programiranju, SOLID je mnemonički akronim za pet principa dizajna koji imaju za cilj učiniti objektno orijentirane dizajne razumljivijim, fleksibilnijim i lakšim za održavanje. Iako se SOLID principi primjenjuju na bilo koji objektno orijentirani dizajn, oni također mogu tvoriti temeljnu filozofiju za metodologije kao što su agilni razvoj ili prilagodljivi razvoj softvera. Ova načela postavljaju temelje za dobro strukturiran programski kod, pomažući razvijateljima programske potpore izbjegavanje uobičajenih problema kao što su visoka međuzavisnost i niska modularnost. [4]

SOLID predstavljaju sljedeća načela:

- Načelo jedne odgovornosti (engl. *Single Responsibility Principle*) kaže kako modul treba imati samo jedan razlog za promjenu, tj. trebao biti odgovoran samo jednoj grupi korisnika ili kako ih Martin naziva, aktera. Ovo osigurava da su moduli kohezivni, fokusirani i lakši za održavanje, smanjujući rizik da promjene utječu na više odgovornosti.

- Načelo otvorenosti/zatvorenosti (engl. *Open/Closed Principle*) kaže kako klase trebaju biti otvorene za proširenje, ali zatvorene za izmjene. Ponašanje modula se može proširiti bez izmjene postojećeg izvornog koda, što omogućava fleksibilnost i smanjuje rizik od unošenja grešaka.

- Načelo Liskovljeve zamjene (engl. *Liskov Substitution Principle*) kaže kako objekti nad-klase trebaju biti zamjenjivi objektima pod-klase bez utjecaja na ispravnost programa. Ovo osigurava da izvedene klase proširuju osnovnu klasu bez mijenjanja njenog očekivanog ponašanja.

- Načelo razdvajanja sučelja (engl. *Interface Segregation Principle*) kaže da nijedan akter ne bi trebao biti prisiljen ovisiti o sučeljima koja ne koristi. Ovo potiče stvaranje manjih,

specifičnijih sučelja koja su prilagođena klijentima, poboljšavajući modularnost i smanjujući utjecaj promjena.

- Načelo inverzije ovisnosti (engl. *Dependency Inversion Principle*) kaže kako moduli visoke razine ne bi trebali ovisiti o modulima niske razine već bi oba trebala ovisiti o apstrakcijama. Ovo odvaja module visoke i niske razine, potičući fleksibilniju i lakše održivu arhitekturu.

3.9 Izazovi *Clean* Arhitekture

Clean Arhitektura također dolazi sa sljedećim izazovima:

- Odvajanje projekta u više slojeva i apstrakcija može uvesti složenost, posebno u manjim projektima. Zahtijeva pažljivo planiranje i dobro razumijevanje principa arhitekture.
- Postavljanje projekta temeljenog na *Clean* Arhitekturi može uključivati više rada u kontekstu postavljanja projekta u usporedbi s jednostavnijim arhitekturama. To uključuje definiranje slojeva, stvaranje sučelja i slično.
- Razvijatelji koji nisu upoznati s *Clean* Arhitekturom će trebati više vremena da se prilagode novoj strukturi, što može usporiti početni razvoj programskog rješenja.

4 KORIŠTENE TEHNOLOGIJE

4.1 API

API (engl. *Application Programming Interface)* definira pravila za komunikaciju s drugim programskim sustavima. Razvijatelji izlažu API-je kako bi druge aplikacije mogle programski komunicirati s njihovim sustavima. Na primjer, aplikacija za evidenciju radnog vremena može izložiti API koji obrađuje radne sate zaposlenika i vraća broj odrađenih sati. [5]

4.2 REST

REST (Representational State Transfer) je stil arhitekture za dizajn mrežnih aplikacija. Predstavio ga je Roy Fielding 2000. godine u svojoj doktorskoj disertaciji. REST nameće određena ograničenja na način kako API treba raditi, uključujući:

- Svaki zahtjev sadrži sve informacije potrebne za njegovo razumijevanje i obradu.
- Klijentski i serverski dijelovi sustava su odvojeni i mogu se razvijati neovisno.
- Odgovori mogu biti označeni kao predmemorirani ili ne predmemorirani kako bi se poboljšala učinkovitost.
- API koristi konzistentno sučelje za sve resurse. [6]

4.2.1 RESTful API

RESTful API je sučelje koje koriste dva računalna sustava za sigurnu razmjenu informacija putem interneta. Većina poslovnih aplikacija mora komunicirati s drugim internim i vanjskim aplikacijama kako bi obavila razne zadatke, poput generiranja mjesečnih platnih lista dijeljenjem podataka između internih računovodstvenih sustava i bankovnih sustava kupaca. [5]

RESTful API koristi HTTP zahtjeve za pristup i korištenje podataka za izvođenje CRUD (*Create, Read, Update, Delete*) operacija na resursima.

HTTP metode su:

- *GET*: dohvaćanje podataka (npr. dohvaćanje pojedinosti o knjizi, popis svih knjiga);
- *POST*: Stvaranje novih podataka (npr. dodavanje knjige, posudba knjige);

- *PUT*: ažuriranje postojećih podataka (npr. ažuriranje pojedinosti o knjizi);
- *DELETE*: Uklanjanje podataka (npr. brisanje knjige, otkazivanje rezervacije).

Svaki resurs treba imati jedinstveni URL(engl. *Uniform Resource Locator*) koji predstavlja stanje resursa. Format zahtjeva je obično u JSON formatu koji sadrži podatke potrebne za obradu, a format odgovor je isto najčešće u JSON formatu, ali može biti i XML format. Osim JSON-a, kao odgovor će doći i statusni kod. HTTP status kodovi su standardni odgovori koje serveri koriste kako bi komunicirali s klijentima o rezultatima njihovih zahtjeva, grupirani u pet kategorija: informativni (100-199), uspješni (200-299), preusmjeravajući (300-399), klijentske greške (400-499) i greške servera (500-599). Ovi kodovi pomažu u razumijevanju rezultata interakcije između klijenata i servera, označavajući stanje zahtjeva, bilo da je uspješno obrađen, zahtijeva dodatne akcije, sadrži klijentsku grešku ili se suočava s problemima na serveru. Primjeri će se detaljnije objasniti u poglavlju 0.

4.3 .NET

.NET je besplatna razvojna platforma otvorenog koda za više platformi za izradu programskih rješenja. Može pokrenuti programska rješenja razvijena u različitim programskim jezicima, a C# (korišten u radu) je najpopularniji, u kojem je .NET i napisan.

.NET uključuje sljedeće komponente:

- Vrijeme izvođenja izvršava kod aplikacije.
- Biblioteke pružaju funkcije kao što je raščlanjivanje JSON-a.
- Prevoditelj (engl. *compiler*) prevodi izvorni kod C# (i drugih jezika) u izvršni kod (vrijeme izvođenja).
- Paket za razvoj programa (engl. *Software development kit, SDK*) i drugi alati omogućuju izradu i nadzor aplikacija s modernim tijekovima rada koji se odnose na suvremene metode i prakse koje se koriste u razvoju programske podrške kao što je primjerice praksa kontinuirane integracije (CI).
- Skupovi aplikacija poput ASP.NET Core i Windows Forms, koji omogućuju pisanje aplikacija.

Vrijeme izvođenja, biblioteke i jezici su temelj .NET. Komponente više razine, poput .NET alata i drugih aplikacija, poput ASP.NET Core, nadograđuju se na ove komponente [7].

4.4 Entity Framework

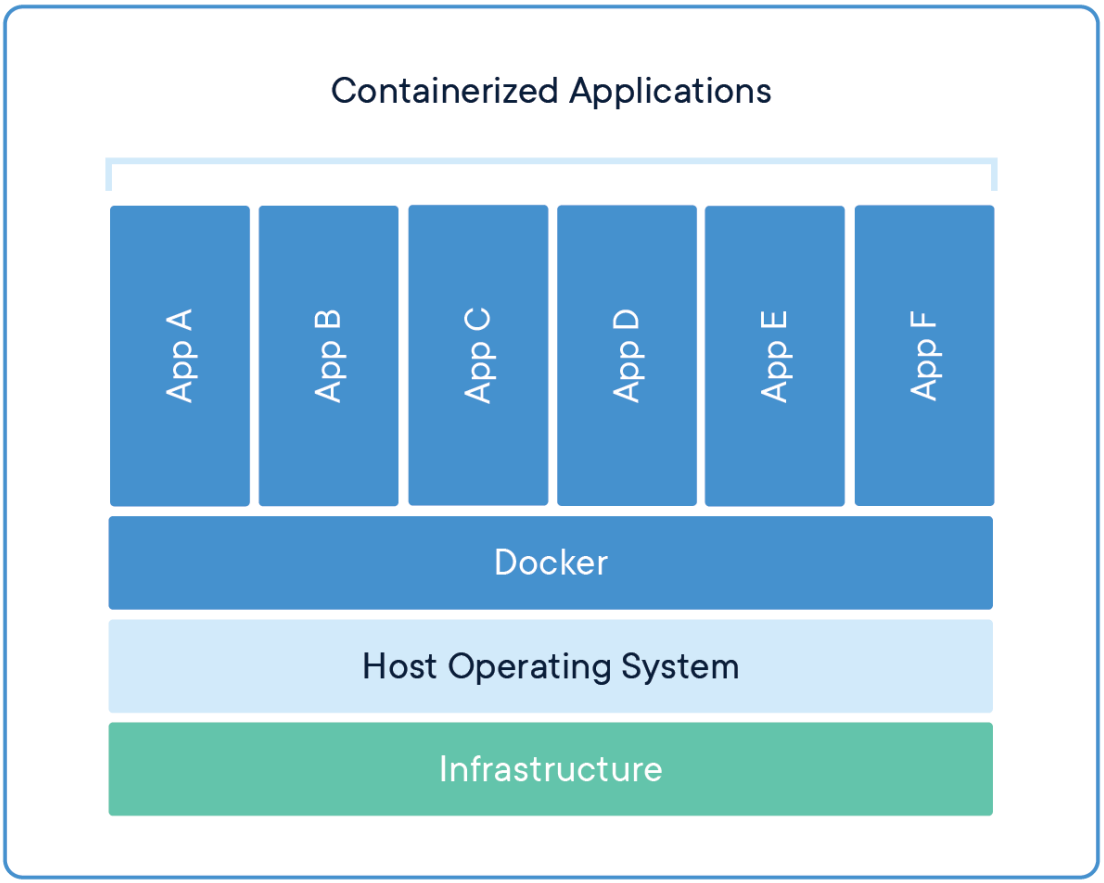
Entity Framework moderni je ORM (*Object-relational mapping*) koji omogućuje izgradnju čistog, prenosivog sloja pristupa podacima visoke razine s .NET-om (C#) u različitim bazama podataka, uključujući SQL bazu podataka, SQLite, MySQL, PostgreSQL i Azure Cosmos DB.[8] Razvijatelj ne mora direktno pisati SQL kod, već sve upite može pisati pomoću C# koda.

4.5 SQL server

Microsoft SQL Server je licencirani sustav za upravljanje relacijskim bazama podataka koji je razvio Microsoft. Kao poslužitelj baze podataka, to je programsko rješenje s primarnom funkcijom pohranjivanja i dohvaćanja podataka prema zahtjevu drugih programskih aplikacija koje se mogu izvoditi ili na istom računalu ili na drugom računalu preko mreže. Microsoft prodaje najmanje desetak različitih izdanja Microsoft SQL Servera, namijenjenih različitim korisnicima i za radna opterećenja u rasponu od malih aplikacija s jednim strojem do velikih aplikacija okrenutih prema Internetu s mnogo istodobnih korisnika. [9]

4.6 Docker

Docker je otvorena platforma za razvoj, isporuku i pokretanje aplikacija. Docker omogućuje odvajanje aplikacije od infrastrukture kako bi se mogao isporučiti programsko rješenje. S Dockerom se može upravljati svojom infrastrukturom na isti način na koji se upravlja aplikacijom. Iskorištavanjem prednosti Dockerovih metodologija za slanje, testiranje i implementaciju koda, može se značajno smanjiti kašnjenje između pisanja koda i njegovog pokretanja u proizvodnji [10].



Slika 2 Docker kontejner [10]

4.7 Redis

Redis je sustav za pohranu strukture podataka otvorenog koda, licenciran za *Berkeley Software Distribution* (BSD) u memoriji koja se koristi kao baza podataka, predmemorija, broker poruka i mehanizam za podatkovne tokove(engl. *streaming*).

Redis pruža mogućnost uporabe različitih struktura podataka kao što su nizovi, raspršeno adresiranje, liste, skupovi, sortirani skupove, bitmape, geoprostorni indeksi i podatkovni tokovi. Redis ima ugrađenu replikaciju, Lua skriptiranje, izbacivanje LRU-a, transakcije i različite razine postojanosti na disku.

Na ovim tipovima se mogu izvoditi atomske operacije, poput dodavanja nizu, povećanje vrijednosti u hash-u, guranje elementa na listu, presjek, unija i razlika računskog skupa ili dobivanje člana s najvišim rangom u sortiranom skupu [11].

4.8 Git

Git je distribuirani sustav kontrole verzija koji prati promjene u bilo kojem skupu datoteka, obično se koristi za koordinaciju rada među razvijateljima koji zajednički razvijaju kod tijekom razvoja programske podrške.

Ciljevi Git-a uključuju brzinu, integritet podataka i podršku za distribuirane i nelinearne tijekove rada [12].

4.9 Github

GitHub je razvojna platforma koja razvijateljima programske podrške omogućuje stvaranje, pohranu, upravljanje i dijeljenje svog koda. Koristi program Git, pružajući kontrolu distribuirane verzije Git-a te kontrolu pristupa, praćenje greški, zahtjeve programskih značajki, upravljanje zadacima, kontinuiranu integraciju i wiki-je za svaki projekt. Podružnica je Microsofta od 2018. sa sjedištem u Kaliforniji.

Od siječnja 2023. GitHub ima više od 100 milijuna programera i više od 420 milijuna spremišta, uključujući najmanje 28 milijuna javnih spremišta. To je najveći svjetski poslužitelj izvornog koda od lipnja 2023 [13].

4.10 CI/CD

Continuous integration and continuous delivery/deployment (CI/CD), koji označava kontinuiranu integraciju i kontinuiranu isporuku/uvođenje, ima za cilj pojednostaviti i ubrzati životni ciklus razvoja programske podrške.

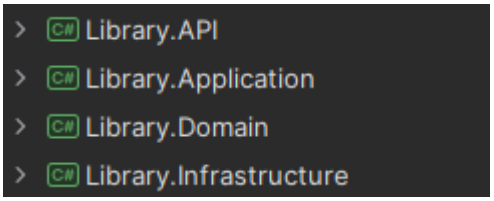
Kontinuirana integracija (CI) odnosi se na praksu automatske i česte integracije promjena koda u zajednički repozitorij izvornog koda. Kontinuirana isporuka i/ili implementacija (Slika 3) je proces koji se sastoji od dva dijela, a odnosi se na integraciju, testiranje i isporuku promjena koda. Kontinuirana isporuka zaustavlja se bez automatske proizvodne implementacije, dok kontinuirana implementacija automatski pušta ažuriranja u proizvodno okruženje. [14]

Slika 3 CI/CD cjevovod [14]

5 Dizajn sustava za upravljanje sveučilišnom knjižnicom

5.1 Arhitektura sustava

Principi *Clean* arhitekture opisani su kroz četiri razine: domena, aplikacija, infrastruktura i API (Slika 4). U poglavlju će se opisati odgovornost svakog sloja, interakcije između slojeva i odluke o dizajnu koje su donesene kako bi se osigurao sustav koji se može skalirati, održavati i testirati.



Slika 4 Struktura projekta u .NET-u

5.1.1 Sloj domene

Sloj domene predstavlja temelj aplikacije, koja sadrži poslovna pravila, entitete i logiku domene. Obuhvaća temeljnu poslovnu logiku i pravila koja upravljaju sustavom upravljanja knjižnice.

Entiteti su osnovni poslovni objekti kao što su knjiga, student ili posudba. Oni sadržavaju stanje i ponašanje relevantno za domenu.

U projektu su korišteni i objekti vrijednosti (engl. *value objects*). Objekt vrijednosti je objekt koji predstavlja jednostavan entitet čija se jednakost ne temelji na njegovom identitetu, već na njegovim svojstvima. Drugim riječima, dva objekta vrijednosti su jednaka ako imaju iste vrijednosti za sva svoja svojstva, bez obzira na to jesu li to dva različita objekta u memoriji. Primjeri vrijednosnih objekata su objekti koji predstavljaju iznos novca ili datumski raspon [15].

U programu su definirani kao zapisi (engl. *records*) (Primjer koda 1) jer objekti vrijednosti bi trebali bit nepromjenjivi i jednaki ako dva imaju iste vrijednosti.

```
public sealed record LoanDateRange
{
    private LoanDateRange(DateOnly loanedDate, DateOnly dueDate)
    {
        LoanedDate = loanedDate;
        DueDate = dueDate;
    }
```

```
    public DateOnly LoanedDate { get; init; }
    public DateOnly? ReturnedDate { get; set; }
    public DateOnly DueDate { get; init; }

    public static LoanDateRange Create(DateTime date)
    {
        var loanedDateTime = DateOnly.FromDateTime(date);

        var dueDateTime = loanedDateTime.AddDays(14);

        return new LoanDateRange(loanedDateTime, dueDateTime);
    }

    public bool IsOverdue()
    {
        if (ReturnedDate is not null)
        {
            return DueDate < ReturnedDate;
        }
        return false;
    }

    public bool Contains(DateOnly date)
    {
        return LoanedDate <= date && date <= ReturnedDate;
    }
}
```

Primjer koda 1 Objekti vrijednosti (engl. value objects)

5.1.2 Aplikacijski sloj

Aplikacijski sloj sadrži poslovna pravila, tj. logiku programskog rješenja. Definira servise ili kako je u radu korištena biblioteka MediatR (detaljnije objašnjen u potpoglavlju 5.3), rukovatelje koji upravljaju interakcijama između sloja domene i sloja infrastrukture. Ovaj sloj implementira usluge koje koordiniraju operacijama poput posuđivanja knjige, dodavanja knjiga i slično. Primjeri upravljače naredbi (engl. *command handlers*), koristeći MediatR za odvajanje pošiljatelja i primatelja naredbi (Primjer koda 9).

5.1.3 Infrastrukturni sloj

Infrastrukturni sloj upravlja interakcijama s vanjskim resursima i uslugama poput baza podataka i vanjskih API-ja. Ovaj sloj implementira repozitorije i druge komponente koje omogućuju pristup

podacima i resursima izvan programskog sustava. Infrastrukturni sloj osigurava apstrakciju nad podacima i vanjskim servisima, omogućujući neovisnost poslovne logike o tehničkim detaljima pohrane podataka.

5.1.4 Prezentacijski sloj (API sloj)

Prezentacijski sloj (poznat i kao API sloj) služi kao ulazna točka za interakciju vanjskih klijenata sa sustavom. Izlaže krajnje točke RESTful API-ja za CRUD operacije i druge funkcionalnosti, u interakciji s aplikacijskim i infrastrukturnim slojevima. API sloj obrađuje dolazne HTTP zahtjeve, provjerava ih, poziva odgovarajuće aplikacijske servise i vraća HTTP odgovore.

```
[HttpGet]
public async Task<IActionResult> GetBooks([FromQuery] int page = 1, [FromQuery] int pageSize = 10,
CancellationToken cancellationToken = default)
{
    var query = new GetBooksQuery(page, pageSize);
    var result = await _sender.Send(query, cancellationToken);

    return Ok(result);
}
```

Primjer koda 2 Metoda Book kontrolera

5.2 Repository pattern

Repository pattern pruža način za enkapsulaciju. Odnosno učahurivanje (engl. *encapsulation*) logike pristupa podacima, odvajajući je od poslovne logike. Ovaj obrazac apstrahira pojedinosti pohrane i dohvaćanja podataka (Primjer koda 3).

Neke od prednosti su:

- Osigurava da sloj domene ne ovisi o specifičnostima pohrane podataka skrivanjem logike pristupa podacima.
- Omogućuje lakše testiranje jedinica dopuštajući korištenje *mock* repozitorija (7.1.4).
- Odvaja logiku domene od sloja pristupa podacima, koja za rezultat ima fleksibilniju bazu koda koja se lakše održava.

```
using Microsoft.EntityFrameworkCore;
```

```
namespace Library.Infrastructure.Repositories;
internal abstract class Repository<T> where T : class
{
    protected readonly LibraryDbContext _dbContext;
    protected readonly DbSet<T> _dbSet;
    protected Repository(LibraryDbContext dbContext)
    {
        _dbContext = dbContext;
        _dbSet = dbContext.Set<T>();
    }

    public async Task<T?> GetByIdAsync(Guid id, CancellationToken cancellationToken = default)
    {
        return await _dbSet.FindAsync(id, cancellationToken);
    }

    public async Task<List<T>> GetAllAsync()
    {
        return await _dbSet.AsNoTracking().ToListAsync();
    }

    public void Add(T entity)
    {
        _dbSet.Add(entity);
    }

    public void Update(T entity)
    {
        _dbSet.Update(entity);
    }

    public void Delete(T entity)
    {
        _dbSet.Remove(entity);
    }
}
```

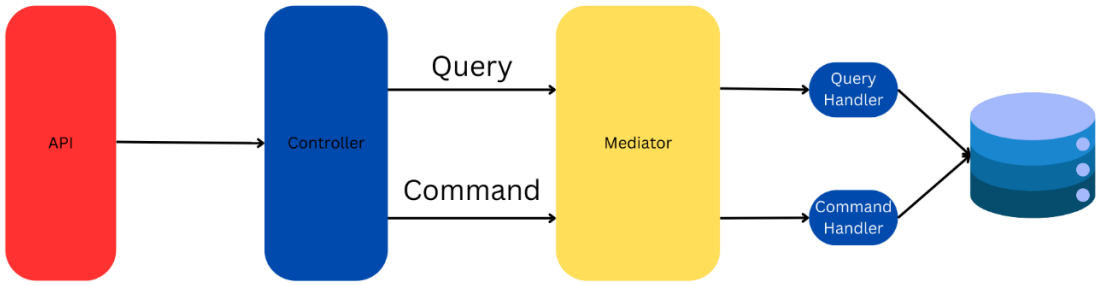
Primjer koda 3 Implementacija Repository pattern-a

5.3 MediatR

MediatR [16] je biblioteka koja implementira medijator obrazac. Medijator obrazac (engl. *Mediator pattern*) je obrazac dizajna koji se koristi za smanjenje izravnih ovisnosti između objekata i za omogućavanje komunikacije između njih preko jednog središnjeg objekta poznatog kao medijator.

Ovaj obrazac centralizira komunikaciju između različitih komponenti, čime se smanjuje kompleksnost i poboljšava upravljivost programskog sustava.

CQRS (*Command Query Responsibility Segregation*) je još jedan obrazac koji koristi MediatR, a glavna mu je zadaća odvojit operacije upita (čitanje podataka) od komandi (unos podataka) (Slika 5). Ovim pristupom se postiže implementacija svake funkcionalnosti u zasebnoj klasi (Primjer koda 9) i također smanjuje ovisnosti među komponentama jer se naredbe šalju kroz posrednika odnosno koji usmjerava zahtjev odgovarajućem rukovatelju (engl. *handler*) (Primjer koda 11).



Slika 5 CQRS

5.4 Shema baze podataka

Slika 6 Shema baze podataka

Baza podataka sustava sveučilišne knjižnice opisana je preko sljedećih tablica i atributa.

Tablica Author sadrži detalje o autorima, uključujući njihovo ime, prezime. Tablica Book pohranjuje informacije o knjigama, uključujući naslov, ISBN, broj stranica, godinu izdanja, te odnose s odjelom, izdavačem i lokacijom. BookCopy tablica sadrži informacije o fizičkim kopijama knjiga, uključujući dostupnost i status rezervacije.

Tablica Department sadrži pojedinosti o sveučilišnim odjelima, uključujući naziv odjela. Employee tablica sadrži informacije o zaposlenicima i članovima osoblja koji upravljaju knjižnicom, uključujući njihova imena, prezimena, pozicije i kontaktne informacije. Tablica Fine bilježi novčane kazne povezane s posudbama, uključujući iznos kazne, status plaćanja i datum izdavanja.

Tablica Loan upravlja posudbama primjeraka knjiga studentima, prati status posudbe, datum dospijeća, datum posudbe i datum vraćanja. Location tablica opisuje detalje lokacije knjižnice na sveučilištu, uključujući naziv lokacije. Tablica Publisher sadrži naziv izdavača.

Reservation tablica upravlja rezervacijama knjiga koje su napravili studenti, uključujući datum rezervacije i status rezervacije. Tablica Student sadrži podatke o studentima, uključujući njihovu akademsku godinu, odjel i e-poštu.

5.5 RESTful API dizajn

Neke od krajnjih točaka sveučilišne knjižnice su:

- Knjige
    - GET /books: Vraća knjige određene paginacijom.
        - Parametri:
            - page (opcionalno, zadano: 1) – broj stranice
            - pageSize (opcionalno, zadano: 10) – broj knjiga po stranici
        - Odgovori:
            - 200: Uspješno dohvaćene knjige

- POST /books: Dodavanje nove knjige.

  Tijelo zahtjeva:

```
{
  "isbn": "string (optional)"
}
```

Primjer koda 4 Tijelo zahtjeva za POST /books krajnju točku u JSON-u

  - Odgovori:

    200: Uspješno dodana knjiga.

- Kopije knjiga
  - POST /bookcopies: Dodavanje nove kopije.
    - Tijelo zahtjeva:

```
{
  "bookId": "string (UUID)",
  "ammount": "integer"
}
```

Primjer koda 5 Tijelo zahtjeva za POST /bookcopies krajnju točku u JSON-u

  - Odgovori:
    - 200: Uspješno dodana kopija.
- Posudbe
  - POST /loans: Posudba knjige.
    - Parametri:
      - bookId (potrebno) - Book ID (UUID)
      - studentId (potrebno) - Student ID (UUID)
    - Odgovori:

      200: Uspješno posuđena knjiga

- POST /loans/reservation/{id}: Posudba knjige preko rezervacije.
  - Parametri:
    - id (potrebno) - Reservation ID (UUID)
  - Odgovori:
    - 200: Uspješno posuđena kopija.
- PUT /loans/return/{id}: Vrati posuđenu knjigu.
  - Parametri:
    - id (potrebno) - Loan ID (UUID)
  - Odgovori:
    - 200: Uspješno vraćena knjiga.

6 PRAKTIČNE PREDNOSTI CLEAN ARHITEKTURE

U ovom poglavlju će se pokazati praktične prednosti *Clean* Arhitekture proširenjem postojećeg REST API-ja, ističući implementacije novih funkcionalnosti.

6.1 Proširenje postojećeg REST API-ja

Na primjeru proširenja postojećeg REST API-ja, prikazat će se prednosti *Clean* Arhitekture. Ovaj sustav već ima funkcije kao što su dodavanje, ažuriranje i brisanje knjiga. Proširit će se dodavanjem nove funkcije za rezervaciju knjiga.

6.2 Početno postavljanje

Novo proširenje znači izmjenu svakog od postojećih slojeva *Clean* Arhitekture:

- Sloj domene: Sadrži osnovnu poslovnu logiku i entitete.
- Aplikacijski sloj: Sadrži servise i upravlja tokom podataka od sloja domene do sloja infrastrukture.
- Infrastrukturni sloj: Sadrži implementacije za interakciju s vanjskim sustavima kao što su baze podataka, datotečni sustavi ili web usluge kao OpenLibrary API.
- Prezentacijski sloj: Sadrži API upravljače.

6.3 Dodavanje nove funkcionalnosti: Rezervacija knjige

Dodavanjem nove funkcionalnosti, tj. rezervacije knjige, potrebno je izmijeniti ili proširiti svaki sloj aplikacije bez diranja postojećih funkcionalnosti.

6.3.1 Domenski sloj

Prvo dodajemo novi entitet i povezanu poslovnu logiku za upravljanje rezervacijama. Sloj domene je mjesto gdje se nalazi temeljna poslovna logika. Stvaranjem entiteta *Reservation*, enkapsuliramo pravila i svojstva koja definiraju rezervaciju knjige.

```
using Library.Domain.Abstractions;

namespace Library.Domain.Reservations;
```

```
public sealed class Reservation : Entity
{
    private Reservation(Guid studentId, Guid bookCopyId, DateTime date)
    {
        StudentId = studentId;
        BookCopyId = bookCopyId;
        DateRange = ReservationDateRange.Create(date);
    }

    private Reservation() { }

    public Guid StudentId { get; }
    public Guid BookCopyId { get; }
    public ReservationDateRange DateRange { get; }
    public bool IsProcessed { get; private set; }

    public static Reservation Create(Guid studentId, Guid bookCopyId, DateTime date)
    {
        return new Reservation(studentId, bookCopyId, date);
    }

    public Result Expired()
    {
        return DateRange.EndDate < DateTime.Now ? Result.Success() :
Result.Failure(ReservationErrors.Expired);
    }

    public void ProcessReservation()
    {
        IsProcessed = true;
    }
}
```

Primjer koda 6 Dodavanje entiteta rezervacije

Definira se sučelje repozitorija za rukovanje podatkovnim operacijama povezanim s rezervacijama knjiga. Ova apstrakcija omogućuje odvajanje logike domene od implementacije pristupa podacima.

```
namespace Library.Domain.Reservations;
public interface IReservationRepository
{
    void Add(Reservation reservation);
    Task<Reservation?> GetActiveReservationOnBookAsync(Guid studentId, Guid bookId);
    Task<Reservation?> GetByIdAsync(Guid id, CancellationToken cancellationToken);
}
```

Primjer koda 7 Repozitorij rezervacija

6.3.2 Aplikacijski sloj

Zatim se dodava usluga kreiranja rezervacije knjige. Aplikacijski sloj koordinira procese aplikacije, dohvaća podatke iz domenskog sloja, obrađuje ih i zatim prosljeđuje prezentacijskom sloju. Prvo je potrebno napraviti naredbu za napraviti rezervaciju tj. naredbu *CreateReservationCommand* koja sadrži potrebne podatke za napravit rezervaciju (Primjer koda 8).

```
using Library.Application.Abstractions.Clock;
using Library.Application.Abstractions.Messaging;

namespace Library.Application.Reservations.CreateReservation;

public sealed record CreateReservationCommand(Guid StudentId, Guid BookId, IDateTimeProvider
DateTimeProvider) : ICommand<Guid>;
```

Primjer koda 8 CreateReservationCommand.cs

Potom se implementira rukovatelj za naredbu *CreateReservationCommand*. Rukovatelj sadrži logiku za izvršavanje procesa, kao što je stvaranje i spremanje rezervacije knjige.

```
using Library.Application.Abstractions.Clock;
using Library.Application.Abstractions.Messaging;
using Library.Domain.Abstractions;
using Library.Domain.BookCopies;
using Library.Domain.Fines;
using Library.Domain.Reservations;

namespace Library.Application.Reservations.CreateReservation;

internal class CreateReservationCommandHandler(
    IReservationRepository reservationRepository,
    IFineRepository fineRepository,
    IUnitOfWork unitOfWork,
    IDateTimeProvider dateTimeProvider,
    IBookCopyRepository bookCopyRepository) : ICommandHandler<CreateReservationCommand, Guid>
{
    private readonly IBookCopyRepository _bookCopyRepository = bookCopyRepository;
    private readonly IDateTimeProvider _dateTimeProvider = dateTimeProvider;
    private readonly IFineRepository _fineRepository = fineRepository;
    private readonly IReservationRepository _reservationRepository = reservationRepository;
```

```
    private readonly IUnitOfWork _unitOfWork = unitOfWork;

    public async Task<Result<Guid>> Handle(CreateReservationCommand request, CancellationToken
cancellationToken)
    {
        var fines = await _fineRepository.GetUnpaidFinesByStudent(request.StudentId,
cancellationToken);

        if (fines is not null && fines.Any())
        {
            return Result.Failure<Guid>(FineErrors.Fined);
        }

        var studentAlreadyHasReservationOnABook =
            await _reservationRepository.GetActiveReservationOnBookAsync(request.StudentId,
request.BookId);

        if (studentAlreadyHasReservationOnABook is not null)
        {
            return Result.Failure<Guid>(ReservationErrors.AlreadyReserved);
        }

        var bookCopy =
            await _bookCopyRepository.GetAvailableBookCopyForReservationAsync(request.BookId,
cancellationToken);

        if (bookCopy is null)
        {
            return Result.Failure<Guid>(ReservationErrors.NoAvailableCopies);
        }

        var reservation = Reservation.Create(request.StudentId, bookCopy.Id,
request.DateTimeProvider.UtcNow);

        _reservationRepository.Add(reservation);

        bookCopy.ProcessReservation();

        _bookCopyRepository.Update(bookCopy);

        await _unitOfWork.SaveChangesAsync(cancellationToken);

        return reservation.Id;
    }
}
```

Primjer koda 9 CreateReservationCommandHandler.cs

6.3.3 Infrastrukturni sloj

Implementira se sučelje repozitorija za pristup podacima. Infrastrukturni sloj bavi se i vanjskim ovisnostima poput baza podataka ili vanjskih API-ja. Implementacija sučelja repozitorija u ovom sloju osigurava da je logika pristupa podacima izolirana od ostatka aplikacije.

```
using Library.Domain.Reservations;
using Microsoft.EntityFrameworkCore;

namespace Library.Infrastructure.Repositories;
internal class ReservationRepository(LibraryDbContext dbContext) : Repository<Reservation>(dbContext),
IReservationRepository
{
    public Task<Reservation?> GetActiveReservationOnBookAsync(Guid studentId, Guid bookId)
    {
        return _dbSet
            .Where(reservation => reservation.DateRange.StartDate <= DateTime.Now)
            .Where(reservation => reservation.DateRange.EndDate >= DateTime.Now)
            .Where(reservation => reservation.StudentId == studentId)
            .Where(reservation => reservation.BookCopyId == bookId)
            .Where(reservation => !reservation.IsProcessed)
            .SingleOrDefaultAsync();
    }
}
```

Primjer koda 10 Implementacija repozitorija

6.3.4 Prezentacijski sloj

Konačno, dodaje se nova krajnja točku API-ja za obradu zahtjeva za rezervaciju. Prezentacijski sloj upravlja API upravljačima koji obrađuju dolazne HTTP zahtjeve i šalju odgovor pozivatelju. Odgovoran je za primanje unosa od korisnika, njihovu obradu kroz aplikacijski sloj i vraćanje odgovora korisniku.

```
using Library.Application.Abstractions.Clock;
using Library.Application.Reservations.CreateReservation;
using MediatR;
using Microsoft.AspNetCore.Mvc;

namespace Library.API.Controllers.Reservations;

[Route("reservations")]
```

```
[ApiController]
public class ReservationController(ISender sender, IDateTimeProvider dateTimeProvider) :
ControllerBase
{
    private readonly IDateTimeProvider _dateTimeProvider = dateTimeProvider;
    private readonly ISender _sender = sender;

    [HttpPost]
    public async Task<IActionResult> ReserveBook(CreateReservationCommand command, CancellationToken
cancellationToken)
    {
        var result = await _sender.Send(command, cancellationToken);

        return CreatedAtAction(nameof(ReserveBook), result.IsSuccess ? result.Value : result.Error);
    }
}
```

Primjer koda 11 API upravljač

6.4 Pokazane prednosti

6.4.1 Jednostavna implementacija novih funkcionalnosti

Svaki je sloj neovisan, što omogućuje implementaciju rezervacija bez mijenjanja postojećih funkcija povezanih s knjigama.

S različitim slojevima možemo neovisno testirati poslovnu logiku i logiku pristupa podacima. Na primjer, jedinični testovi mogu se usredotočiti na pojedini servis, npr. *CreateReservationHandler* neovisno o bazi podataka.

Jasno razdvajanje odgovornosti (engl. *separation of concerns*) čini kod lakšim za razumijevanje, održavanje i proširenje. Dodavanje nove značajke poput rezervacija knjiga zahtijeva promjene u svakom sloju, ali te promjene ne ometaju postojeću funkcionalnost.

6.4.2 Jednostavna izmjena postojećih funkcionalnosti

Izmjene poslovnih pravila ili logike pristupa podacima mogu se izvršiti u njihovim odgovarajućim slojevima bez utjecaja na druge dijelove sustava. Ako želimo zamijeniti SQL Server s PostgreSQL bazom podataka promjene se mogu napraviti samo u infrastrukturnom sloju bez utjecaja na aplikacijski i domenski sloj. U primjeru je opisana konfiguracija SQL Servera (Primjer koda 12).

```
public static IServiceCollection ConfigureDatabase(this IServiceCollection services,
    IConfiguration configuration)
{
    var connectionString = configuration.GetConnectionString("Database") ??
        throw new ArgumentNullException(nameof(configuration));

    services.AddDbContext<LibraryDbContext>(options => options.UseSqlServer(connectionString));
}
```

Primjer koda 12 Konfiguracija SQL Servera

Kako bi se zamijenio SQL Server sa PostgreSQL potrebno je samo dodati metodu „UseNpgsql“ umjesto „UseSqlServer“ (Primjer koda 13).

```
public static IServiceCollection ConfigureDatabase(this IServiceCollection services,
    IConfiguration configuration)
{
    var connectionString = configuration.GetConnectionString("Database") ??
        throw new ArgumentNullException(nameof(configuration));

    services.AddDbContext<LibraryDbContext>(options => options.UseNpsql(connectionString));
}
```

Primjer koda 13 Konfiuracija PostgreSQL-a

Nakon toga je potrebno pokrenuti još migracije za novu bazu i cijeli posao je završen.

6.5 Zaključak

Slojeviti pristup *Clean* Arhitekture pruža značajne praktične prednosti pri proširenju i modificiranju funkcionalnosti. Demonstrirajući dodavanje značajke rezervacije knjiga, pokazano je kako *Clean* Arhitektura pojednostavljuje implementaciju novih funkcionalnosti i čini postojeće funkcionalnosti lakšim za izmjenu. Rezultat je čitljiva baza koda koja se lakše održava, testira i skalira.

7 TESTOVI

Testiranje u razvoju programske podrške je ključno za osiguravanje konačnog „pouzdanog“, učinkovitog proizvoda. Testovi pomažu u prepoznavanju i ispravljanju grešaka prije nego se programsko rješenje pusti u produkciju.

7.1 Jedinični testovi

Jedinični (engl. *Unit*) testovi su ključni dio procesa razvoja programske podrške, osobito unutar domenskog i aplikacijskog sloja *Clean* Arhitekture u kojima se nalazi poslovna logika. Pomažu osigurati ispravnost pojedinih komponenti prema očekivanju.

Jedinični testovi su granularni, tj. usmjereni su na testiranje najmanjih jedinica koda, kao što su npr. pojedinačne metode. U sustavu upravljanja knjižnicom, osigurava se ispravan rad metode entiteta knjiga ili posudba.

Kako bi testovi bili pouzdani i usmjereni, jedinične testove treba pisati zasebno. To često uključuje korištenje *mocking* biblioteka. Primjerice, kad se testira servis aplikacijskog sloja, može se koristiti „lažno“ spremište za simulaciju interakcija baze podataka.

Budući da su jedinični testovi usredotočeni na male dijelove koda, izvršavaju se brzo. Brzina omogućuje često izvođenje jediničnih testova, pružajući trenutnu povratnu informaciju o ispravnosti koda.

Jedinični testovi se obično izvode tijekom razvojnog procesa, integrirani su i u cjevovode kontinuirane integracije kako nove promjene ne bi utjecale na postojeću funkcionalnost programskog sustava.

7.1.1 Jedinično testiranje domenskog sloja

Jedinično testiranje u sloju domene usredotočeno je na provjeru funkcionalnosti poslovne logike i pravila sadržanih unutar entiteta domene i vrijednosnih objekata. Cilj je osiguranje očekivanog ponašanja modela domene u različitim uvjetima.

7.1.2 Primjer jediničnog testa domenskog sloja

U sustavu upravljanja knjižnicom, entitet *Book* učahuruje informacije o knjizi. Sljedeći testni slučaj

pokazuje kako jedinično testirati stvaranje entiteta knjige kako bi se osiguralo da su njegova svojstva ispravno postavljena (Primjer koda 14).

```
[Fact]
public void Create_Should_SetPropertyValues()
{
    var book = Book.CreateFromOpenLibrary(BookData.Title, BookData.Isbn, BookData.PublicationYear,
        BookData.NumberOfPages, BookData.PublisherId);
         book.Title.Should().Be(BookData.Title);
    book.Isbn.Should().Be(BookData.Isbn);
    book.PublicationYear.Should().Be(int.Parse(BookData.PublicationYear));
    book.NumberOfPages.Should().Be(BookData.NumberOfPages);
    book.PublisherId.Should().Be(BookData.PublisherId);
}
```

Primjer koda 14 Jedinični test domenskog sloja

Pisanjem detaljnih jediničnih testova za sloj domene osigurava se ispravno ponašanje ključne poslovne logike programskog sustava. Ova praksa povećava pouzdanost i robusnost aplikacije. U ovom primjeru je pokazano kako testirati postavljanje vrijednosti svojstva u entitetu *Book* koristeći xUnit [17] i FluentAssertions [18] biblioteke. Ista načela mogu se primijeniti na druge entitete u sloju domene kako bi se postigla sveobuhvatna pokrivenost testovima.

7.1.3 Jedinično testiranje aplikacijskog sloja

Jedinično testiranje u aplikacijskom sloju usmjereno je na provjeru ispravnosti logike ili poslovnih pravila servisa. Cilj je osigurati da logika aplikacije radi kako je predviđeno.

7.1.4 Primjer

U kontekstu sustava upravljanja knjižnicom, slučaj koji treba provjeru je neplaćene kazne studenta prije nego mu se dopusti posudba knjige. Kao i u sloju domene, sljedeći testni slučaj pokazuje kako jedinično testirati logiku aplikacijskog sloja koristeći xUnit i FluentAssertions (Primjer koda 15).

```
[Fact]
public async Task Handle_Should_ReturnFailure_WhenStudentHasFine()
{
    var fine = Fine.Create(Guid.NewGuid(), 10, false, _dateTimeProviderMock.UtcNow);

    _fineRepositoryMock.GetUnpaidFinesByStudent(Command.StudentId, Arg.Any<CancellationToken>())
        .Returns(new List<Fine> { fine });
```

```
    var result = await _handler.Handle(Command, default);

    result.Error.Should().Be(FineErrors.Fined);
}
```

Primjer koda 15 Jedinični test aplikacijskog sloja

Pisanjem detaljnih jediničnih testova za aplikacijski sloj osigurava se ispravno ponašanje poslovne logike. Ova praksa povećava pouzdanost i robusnost aplikacije. Ista načela mogu se primijeniti na druge servise u aplikacijskom sloju kako bi se postigla sveobuhvatna pokrivenost testom.

7.2 Integracijski testovi

Integracijsko testiranje je oblik testiranja programske podrške u kojem se više dijelova programskog sustava testira kao grupa.

Integracijsko testiranje je ključno za provjeru interakcija između različitih komponenti aplikacije. U ovom odjeljku dublje testovi će biti postavljeni i izvođeni pomoću *Docker* spremnika za SQL Server i Redis-a, koristeći klase *IntegrationTestWebAppFactory* i *BaseIntegrationTest*.

Za upravljanje Docker spremnicima, koristit će se biblioteka Testcontainers [19]. Ova biblioteka omogućuje jednostavnu izradu i pokretanje kontejnera, ali i jednostavno brisanje kontejnera prilikom završetka testiranja.

7.2.1 Postavljanje testova

Klasa *IntegrationTestWebAppFactory* odgovorna je za postavljanje okruženja za testiranje, uključujući SQL Server i Redis spremnike. Proširuje klasu *WebApplicationFactory* i implementira *IAsyncLifetime* za upravljanje životnim ciklusom spremnika (Primjer koda 16).

```
public class IntegrationTestWebAppFactory : WebApplicationFactory<Program>, IAsyncLifetime
{
    private readonly MsSqlContainer _dbContainer = new MsSqlBuilder()
        .WithPassword("Pass@word")
        .WithImage("mcr.microsoft.com/mssql/server")
        .WithEnvironment("ACCEPT_EULA", "Y")
        .WithPortBinding("1433")
        .Build();

    private readonly RedisContainer _redisContainer = new RedisBuilder()
        .WithImage("redis:latest")
        .Build();

    public async Task InitializeAsync()
    {
        await _dbContainer.StartAsync();
        await _redisContainer.StartAsync();
    }

    public new async Task DisposeAsync()
    {
        await _dbContainer.StopAsync();
        await _redisContainer.StopAsync();
    }

    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            services.RemoveAll(typeof(DbContextOptions<LibraryDbContext>));

            services.AddDbContext<LibraryDbContext>(options =>
                options.UseSqlServer(_dbContainer.GetConnectionString()));

            services.AddSingleton<ISqlConnectionFactory>(_ =>
                new SqlConnectionFactory(_dbContainer.GetConnectionString()));

            services.Configure<RedisCacheOptions>(options =>
                options.Configuration = _redisContainer.GetConnectionString());
        });
    }
}
```

Primjer koda 16 IntegrationWebTestWebAppFactory.cs

Ova klasa postavlja SQL Server i Redis spremnike te namješta web poslužitelj za korištenje tih

spremnika tijekom testiranja.

Klasa *BaseIntegrationTest* je osnovna klasa za sve integracijske testove, tj. svaki test nasljeđuje tu klasu. Svaki od testova također ima pristup *LibraryDbContext* i *ISender* za slanje MediatR zahtjeva (Primjer koda 17).

```
using MediatR;
using Microsoft.Extensions.DependencyInjection;

namespace Library.Infrastructure.IntegrationTests.Infrastructure;

public abstract class BaseIntegrationTest : IClassFixture<IntegrationTestWebAppFactory>
{
    protected readonly LibraryDbContext DbContext;
    protected readonly ISender Sender;

    protected BaseIntegrationTest(IntegrationTestWebAppFactory factory)
    {
        var scope = factory.Services.CreateScope();

        Sender = scope.ServiceProvider.GetRequiredService<ISender>();
        DbContext = scope.ServiceProvider.GetRequiredService<LibraryDbContext>();
    }
}
```

Primjer koda 17 BaseIntegrationTest.cs

Ova klasa postavlja potrebni kontekst i ovisnosti potrebne za testove, osiguravajući da svaki test ima čisto i izolirano okruženje.

7.2.2 Primjer integracijskog testa

Integracijski test za dohvaćanje popisa učenika šalje zahtjev *GetStudentsQueryHandler-u* i provjerava odgovor funkcije.

```
using FluentAssertions;
using Library.Application.Students.GetStudentsQuery;
using Library.Infrastructure.IntegrationTests.Infrastructure;

namespace Library.Infrastructure.IntegrationTests.Students;

public class GetStudentsTests(IntegrationTestWebAppFactory factory) : BaseIntegrationTest(factory)
```

```
{
    [Fact]
    public async Task GetStudents_ShouldReturnStudents()
    {
        // Arrange
        var query = new GetStudentsQuery(1, 10);

        // Act
        var result = await Sender.Send(query);

        // Assert
        result.IsSuccess.Should().BeTrue();
        result.Value.Should().NotBeNull();
    }
}
```

Primjer koda 18 Integracijski test

U ovom se testu *GetStudentsQuery* šalje pomoću pošiljatelja iz MediatR-a, a rezultat se provjerava kako bi se osiguralo da sadrži očekivane podatke (Primjer koda 18).

7.2.3 Zaključak

Implementacijom integracijskih testova pomoću *Docker* spremnika i dobro strukturiranog okvira za testiranje, osigurava se ispravan rad svih komponenti programskog sustava . Ovaj pristup pruža robusnu, izoliranu okolinu koja oponaša produkcijsko okruženje, dopuštajući temeljito testiranje interakcija baze podataka, servisne logike i krajnjih točaka API-a.

7.3 Arhitekturni testovi

Testiranjem arhitekture osigurava se usklađenost sustava s utvrđenim smjernicama i načelima arhitekture.. To uključuje provjeru pravilnog odvajanja odgovornosti ispravno upravljanje ovisnostima i pridržavanje obrazaca dizajna. Za testove se koristila biblioteka NetArchTest koja provodi i provjerava valjanosti ovih ograničenja arhitekture unutar programskog sustava.

7.3.1 Postavljanje testova

Slično kao i u integracijskim testovima, postavljena je osnovna testna klasa pruža temeljne postavke za sve testove arhitekture. Klasa uključuje reference na glavne dijelove (engl. *assemblies*) programskog sustava (Primjer koda 19).

```
using Library.Application.Abstractions.Messaging;
using Library.Domain.Abstractions;
using Library.Infrastructure;
using System.Reflection;

namespace Library.ArchitectureTests.Infrastructure;

public abstract class BaseTest
{
    protected static readonly Assembly ApplicationAssembly = typeof(IBaseCommand).Assembly;
    protected static readonly Assembly DomainAssembly = typeof(Entity).Assembly;
    protected static readonly Assembly InfrastructureAssembly = typeof(LibraryDbContext).Assembly;
    protected static readonly Assembly PresentationAssembly = typeof(Program).Assembly;
}
```

Primjer koda 19 BaseTest.cs

Ova klasa postavlja sklopove (engl. *assemblies*) potrebne za testove arhitekture, koji će se koristiti za provjeru ovisnosti i drugih arhitektonskih pravila. Sklopovi su zbirke koda koje se prevode u jednu logičku cjelinu, obično u obliku .dll ili .exe datoteke.

7.3.2 Testiranje slojeva

Testovi slojeva osiguravaju odsutnost nedefiniranih ovisnosti među različitim slojevima programskog susatva (Primjer koda 20), odnosno sprječavaju ovisnosti koje nisu u skladu s onima definiranima u Clean arhitekturi (Slika 1).

```
 using FluentAssertions;
using Library.ArchitectureTests.Infrastructure;
using NetArchTest.Rules;

namespace Library.ArchitectureTests.Layers;
public class LayerTests : BaseTest
{
    [Fact]
    public void DomainLayer_ShouldNotHaveDependencyOn_AnyLayer()
    {
        var result = Types.InAssembly(DomainAssembly)
            .ShouldNot()
            .HaveDependencyOnAny()
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }
```

```
    [Fact]
    public void ApplicationLayer_ShouldNotHaveDependencyOn_InfrastructureLayer()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .ShouldNot()
            .HaveDependencyOn(InfrastructureAssembly.GetName().Name)
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void ApplicationLayer_ShouldNotHaveDependencyOn_PresentationLayer()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .ShouldNot()
            .HaveDependencyOn(PresentationAssembly.GetName().Name)
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void InfrastructureLayer_ShouldNotHaveDependencyOn_PresentationLayer()
    {
        var result = Types.InAssembly(InfrastructureAssembly)
            .ShouldNot()
            .HaveDependencyOn(PresentationAssembly.GetName().Name)
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }
}
```

Primjer koda 20 Test slojeva

Ovi testovi potvrđuju da svaki sloj ovisi samo o slojevima o kojima bi trebao ovisiti, osiguravajući odvajanje i pridržavanje načela *Clean* Arhitekture.

7.3.3 Testovi domenskog sloja

Testovi sloja domene osiguravaju usklađenost i praćenje pravila entiteta domene.

```
using FluentAssertions;
using Library.ArchitectureTests.Infrastructure;
using Library.Domain.Abstractions;
using NetArchTest.Rules;
using System.Reflection;

namespace Library.ArchitectureTests.Domain;

public class DomainTests : BaseTest
{
    [Fact]
    public void Entities_Should_BeSealed()
    {
        var result = Types.InAssembly(DomainAssembly)
            .That()
            .Inherit(typeof(Entity))
            .And()
            .AreNotAbstract()
            .Should()
            .BeSealed()
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void Entities_ShouldHave_PrivateParameterlessConstructor()
    {
        var entityTypes = Types.InAssembly(DomainAssembly)
            .That()
            .Inherit(typeof(Entity))
            .And()
            .AreNotAbstract()
            .GetTypes();

        var failingTypes = new List<Type>();
        foreach (var entityType in entityTypes)
        {
            var constructors = entityType.GetConstructors(BindingFlags.NonPublic |
BindingFlags.Instance);

            if (!constructors.Any(c => c.IsPrivate && c.GetParameters().Length == 0))
            {
                failingTypes.Add(entityType);
            }
        }
```

```
        failingTypes.Should().BeEmpty();
    }
}
```

Primjer koda 21 Testiranje sloja domene

Jedno od pravila od ove vrste testiranja jest da entiteti koji nasljeđuju klasu *Entity* moraju biti *sealed* (klasa s tom oznakom se ne može nasljeđivati), čime se osigurava integritet domene.

7.3.4 Testovi aplikacijskog sloja

Testovi aplikacijskog sloja osiguravaju pridržavanje naredbi (engl. *commands*) i upita (engl. *queries*), kao i validatora, definiranim konvencijama imenovanja i pravilima pristupačnosti (npr. validatori ne smiju biti *public* klase) (Primjer koda 22).

```
using FluentAssertions;
using FluentValidation;
using Library.Application.Abstractions.Messaging;
using Library.ArchitectureTests.Infrastructure;
using NetArchTest.Rules;

namespace Library.ArchitectureTests.Application;

public class ApplicationTests : BaseTest
{
    [Fact]
    public void CommandHandler_ShouldHave_NameEndingWith_CommandHandler()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .ImplementInterface(typeof(ICommandHandler<>))
            .Or()
            .ImplementInterface(typeof(ICommandHandler<,>))
            .Should()
            .HaveNameEndingWith("CommandHandler")
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void CommandHandler_Should_NotBePublic()
    {
```

```
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .ImplementInterface(typeof(ICommandHandler<>))
            .Or()
            .ImplementInterface(typeof(ICommandHandler<,>))
            .ShouldNot()
            .BePublic()
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void QueryHandler_ShouldHave_NameEndingWith_QueryHandler()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .ImplementInterface(typeof(IQueryHandler<,>))
            .Should()
            .HaveNameEndingWith("QueryHandler")
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void QueryHandler_Should_NotBePublic()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .ImplementInterface(typeof(IQueryHandler<,>))
            .ShouldNot()
            .BePublic()
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void Validator_ShouldHave_NameEndingWith_Validator()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .Inherit(typeof(AbstractValidator<>))
            .Should()
            .HaveNameEndingWith("Validator")
```

```
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }

    [Fact]
    public void Validator_Should_NotBePublic()
    {
        var result = Types.InAssembly(ApplicationAssembly)
            .That()
            .Inherit(typeof(AbstractValidator<>))
            .Should()
            .NotBePublic()
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }
}
```

Primjer koda 22 Testiranje aplikacijskog sloja

Ovi testovi provode konvencije i pravila pristupačnosti za rukovatelje naredbama, servisima, upitima i validatore, osiguravajući dosljednost i pravilno učahurivanje u sloju aplikacije.

7.3.5 Testovi prezentacijskog sloja

Testovi prezentacijskog sloja osiguravaju pridržavanje upravljača konvencijama imenovanja i nasljeđivanju od odgovarajućih osnovnih klasa (Primjer koda 23).

```
using FluentAssertions;
using Library.ArchitectureTests.Infrastructure;
using Microsoft.AspNetCore.Mvc;
using NetArchTest.Rules;

namespace Library.ArchitectureTests.Presentation;

public class PresentationTests : BaseTest
{
    [Fact]
    public void Controllers_ShouldHave_NameEndingWith_Controller()
    {
        var result = Types.InAssembly(PresentationAssembly)
            .That()
```

```
            .Inherit(typeof(ControllerBase))
            .Should()
            .HaveNameEndingWith("Controller")
            .GetResult();

        result.IsSuccessful.Should().BeTrue();
    }
}
```

Primjer koda 23 Testiranje prezentacijskog sloja

Ovaj test osigurava da svi upravljači slijede konvenciju imenovanja, promičući dosljednost u cijelom prezentacijskom sloju.

7.3.6 Zaključak poglavlja

Svaki od implementiranih testova osigurava robusnost i integritet aplikacije. Također, svaka vrsta ovih testova osigurava pravilno ponašanje aplikacije, sprječavanje, ali i uočavanje greški prije nego što se rješenje pusti vani na produkciju.

Jedinični testovi osiguravaju ispravnost rada poslovne logike, integracijski testovi provjeravaju rad aplikacije u cjelini, zajedno sa vanjskim komponentama poput baze podataka, dok arhitekturni testovi osiguravaju pridržavanje načela *Clean* Arhitekture.

8 KONTINUIRANA INTEGRACIJA I KONTINUIRANA IMPLEMENTACIJA (CI/CD)

CI/CD prakse u modernom razvoju programske podrške. Automatiziraju proces integracije promjena koda, izvođenje testova i implementaciju aplikacija, osiguravajući brzu i pouzdanu isporuku programske podrške. Ovo poglavlje bavi se postavljanjem i implementacijom CI/CD-a pomoću GitHub akcija, ističući prednosti i pružajući praktične primjere.

8.1 Plan implementacije

Plan je usmjeren na osiguravanje dosljedne implementacije u različitim okruženjima, uključujući razvoj, prikazivanje i proizvodnju. Korištenje Docker-a omogućuje smještanje programskog sustava i svih njegovih ovisnosti u kontejnere. To olakšava implementaciju i skaliranje aplikacije, jer Docker osigurava da programski sustav uvijek radi isto, bez obzira na okruženje u kojem se pokreće.

8.2 Git grananja

Grananje je ostvareno pomoću dvije grane: *develop* i *release*.

Grana `develop` služi kao glavna razvojna grana u kojoj se odvija razvoj svih značajki programskog sustava, ispravci grešaka i druga poboljšanja programskog sustava. Kada rade na novoj značajki ili ispravku greške, razvijatelji bi trebali stvoriti novu granu iz `develop`, implementirati promjene, a zatim stvoriti zahtjev za dohvat (engl. *pull request)* za spajanje promjena natrag u `develop`.

Grana *release* služi kao početna grana za produkcijsko okruženje. Nakon što su promjene u *develop* grani temeljito testirane i spremne za proizvodnju, one se spajaju u *release*. Nakon spajanja i konačne provjere, promjene se ručno postavljaju u proizvodno okruženje.

8.3 Kontejnerizacija s Dockerom

Docker se koristi kako bi se postigla dosljednost u različitim okruženjima i olakšala implementacija.. Definirani Dockerfile (Primjer koda 24) služi za izgradnju slike aplikacije i docker-compose.yml datoteku za orkestriranje usluga.

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 8080
```

```
EXPOSE 8081

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["Library.API/Library.API.csproj", "Library.API/"]
RUN dotnet restore "./Library.API/./Library.API.csproj"
COPY . .
WORKDIR "/src/Library.API"
RUN dotnet build "./Library.API.csproj" -c $BUILD_CONFIGURATION -o /app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./Library.API.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Library.API.dll"]
```

Primjer koda 24 Dockerfile

8.4 CI/CD cjevovod

Implementiran je CI/CD cjevovod koristeći *GitHub Actions* za automatizaciju procesa izgradnje, testiranja i implementacije. Ovaj cjevovod osigurava da je kod dosljedno izgrađen, testiran i implementiran, smanjujući ručne pogreške razvijatelja i ubrzavajući životni ciklus razvoja (Primjer koda 25). Procesi cjevovoda su sljedeći:

- Cjevovod pokreće proces izgradnje svaki put kada se promjene gurnu u glavnu granu. Sastavlja kod, rješava ovisnosti i proizvodi artefakt izgradnje (engl. *build artifact*). Artefakt izgradnje može uključivati binarne datoteke ili bilo koje druge izlazne datoteke koje su rezultat procesa izgradnje.

- Pokreću se automatizirani testovi kako bi se osigurala kvaliteta i pouzdanost našeg koda. To uključuje jedinične testove, integracijske testove, arhitekturne testove koji pokrivaju različite aspekte aplikacije.

- Nakon uspješnog testiranja, cjevovod implementira aplikaciju u ciljno okruženje, bilo da se radi o razvojnom, pripremnom ili proizvodnom okruženju.

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1

      - name: Log in to Docker Hub
        run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}"
--password-stdin

      - name: Build and push Docker image
        id: docker_build
        run: |
          docker build -t mtikvica/unidu-library:latest .
          docker push mtikvica/unidu-library:latest

      - name: Log in to Azure Container Registry
        run: |
          echo ${{ secrets.AZURE_PASSWORD }} | docker login ${{ secrets.AZURE_REGISTRY_LOGIN_SERVER }}
-u ${{ secrets.AZURE_USERNAME }} --password-stdin

      - name: Tag image for ACR
        run: |
          docker tag mtikvica/unidu-library:latest ${{ secrets.AZURE_REGISTRY_LOGIN_SERVER }}/unidu-
library:latest
          docker push ${{ secrets.AZURE_REGISTRY_LOGIN_SERVER }}/unidu-library:latest

      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v2
        with:
          app-name: your-azure-web-app-name
          images: ${{ secrets.AZURE_REGISTRY_LOGIN_SERVER }}/unidu-library:latest
```

Primjer koda 25 Generalni CI/CD cjevovod

9 Dodatne usluge programskog rješenja

9.1 Implementacija predmemoriranja s Redisom

Kako bi se poboljšale performanse aplikacije, implementirano je predmemoriranje pomoću Redisa. Pomoću biblioteke StackExchange.Redis [20] za interakciju s Redisom i predmemoriranje rezultata čestih upita baze podataka (Primjer koda 26).

```
using Library.Application.Abstractions.Caching;
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;

namespace Library.Infrastructure.Cache;
internal class CacheService(IDistributedCache cache) : ICacheService
{
    private readonly IDistributedCache _cache = cache;

    public async Task<T?> GetAsync<T>(string key, CancellationToken cancellationToken = default)
    {
        var bytes = await _cache.GetAsync(key, cancellationToken);

        return bytes is null ? default : JsonSerializer.Deserialize<T>(bytes);
    }

    public Task RemoveAsync(string key, CancellationToken cancellationToken = default)
    {
        return _cache.RemoveAsync(key, cancellationToken);
    }

    public Task SetAsync<T>(string key, T value, TimeSpan? expiration = null, CancellationToken
cancellationToken = default)
    {
        var bytes = JsonSerializer.SerializeToUtf8Bytes(value);

        return _cache.SetAsync(key, bytes, new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = expiration
        }, cancellationToken);
    }
}
```

Primjer koda 26 Cache servis

10 ZAKLJUČAK

U radu je prikazana primjena *Clean* Arhitekture za razvoj REST API-ja sveučilišne knjižnice. Svaki sloj arhitekture je analiziran i implementiran kako bi se pokazale ključne prednosti modularnosti, neovisnosti o tehnologijama i jednostavnog održavanja. Korištenje tehnologija poput .NET 8, Docker-a ili CI/CD cjevovoda doprinijelo je brzoj i efikasnoj izradi i testiranju aplikacije.

Kroz implementaciju načela *Clean* Arhitekture na razvoj REST API-ja sveučilišne knjižnice, omogućen je razvoj modularnog, lako održivog i skalabilnog sustava. Ovakav oblik arhitekture omogućuje programerima fokus na poslovnu logiku dok se tehnički ili vanjski dijelovi poput baza ili vanjskih servisa mogu mijenjati bez da se ometa poslovna logika.

Primjena paradigme *Clean* Arhitektura dokazala je svoju vrijednost u izradi kompleksnih sustava koji zahtijevaju dugoročnu održivost i spremnost za prilagodbu. Ova arhitektura pruža dobar temelj za daljnja istraživanja arhitektura i primjene ove paradigme u razvoju različitih vrsta programskih rješenja.

LITERATURA

[1] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Pearson, 2017.

[2] »cai.io,« [Mrežno]. Dostupno: https://www.cai.io/resources/articles/understanding-the-impact-of-technical-debt#fn:6. [Pokušaj pristupa Svibanj 2024].

[3] M. Fowler, »martinfowler.com,« 1 Kolovoz 2019. [Mrežno]. https://martinfowler.com/architecture/. [Pokušaj pristupa Svibanj 2024].

[4] »Wikipedia,« [Mrežno]. Dostupno: https://en.wikipedia.org/wiki/SOLID. [Pokušaj pristupa Lipanj 2024].

[5] Amazon, »amazon.com,« Amazon, [Mrežno]. https://aws.amazon.com/what-is/restful-api/. [Pokušaj pristupa Svibanj 2024].

[6] R. T. Fielding, »ics.uci.edu,« [Mrežno].https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Pokušaj pristupa Svibanj 2024].

[7] »Microsoft Learn,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/core/introduction?WT.mc_id=dotnet-35129-website.

[8] »Microsoft Learn,« [Mrežno]. Available: https://learn.microsoft.com/en-us/ef/. [Pokušaj pristupa Svibanj 2024].

[9] »Microsoft SQL server,« Wikimedia Foundation, [Mrežno]. https://en.wikipedia.org/wiki/Microsoft_SQL_Server. [Pokušaj pristupa Travanj 2024].

[10] »Docker Docs,« [Mrežno]. Available: https://docs.docker.com/get-started/overview/. [Pokušaj pristupa April 2024].

[11] »Redis,« [Mrežno]. Available: https://redis.io/docs/about/. [Pokušaj pristupa Travanj 2024].

[12] »Wikipedia - Git,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Git. [Pokušaj pristupa Travanj 2024].

[13] »Wikipedia - Github,« [Mrežno]. Available: https://en.wikipedia.org/wiki/GitHub. [Pokušaj pristupa Travanj 2024].

[14] Redhat, »What is CI/CD?,« Redhat, [Mrežno]. https://www.redhat.com/en/topics/devops/

what-is-ci-cd. [Pokušaj pristupa Travanj 2024].

[15] »Wikipedia - Value object,« [Mrežno]. https://en.wikipedia.org/wiki/Value_object. [Pokušaj pristupa Travanj 2024].

[16] J. Bogard, »Github,« [Mrežno]. https://github.com/jbogard/MediatR. [Pokušaj pristupa Svibanj 2024].

[17] Brad Wilson and James Newkirk, »Github,« [Mrežno]. https://github.com/xunit/xunit. [Pokušaj pristupa Svibanj 2024].

[18] [Mrežno]. https://github.com/fluentassertions/fluentassertions. [Pokušaj pristupa Svibanj 2024].

[19] »Testcontainers,« [Mrežno]. https://testcontainers.com/. [Pokušaj pristupa Svibanj 2024].

[20] »Github,« [Mrežno]. https://github.com/StackExchange/StackExchange.Redis. [Pokušaj pristupa Svibanj 2024].

11 PRILOZI

11.1 Popis slika

11.2 Popis primjera koda

IZJAVA

Izjavljujem pod punom moralnom odgovornošću da sam završni rad izradio samostalno, isključivo znanjem stečenim na studijima Sveučilišta u Dubrovniku, služeći se navedenim izvorima podataka i uz stručno vodstvo mentora prof.dr.sc. Marija Miličevića, i komentorice Ane Kešelj Dilberović, mag. ing. comp. kojima se još jednom srdačno zahvaljujem.

Mato Tikvica