# Razvoj web aplikacije za pretraživanje jeftinih avionskih letova

**Ilić, Gabriel**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Gabriel Ilić**

# RAZVOJ WEB APLIKACIJE ZA PRETRAŽIVANJE JEFTINIH AVIONSKIH LETOVA

**DIPLOMSKI RAD**

**Varaždin, 2021.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Gabriel Ilić**

**Matični broj: 0016104632**

**Studij: Baze podataka i baze znanja**

# RAZVOJ WEB APLIKACIJE ZA PRETRAŽIVANJE JEFTINIH AVIONSKIH LETOVA

**DIPLOMSKI RAD**

**Mentor:**

Prof. dr. sc. Danijel Radošević

**Varaždin, rujan 2021.**

*Gabriel Ilić*

**Izjava o izvornosti**

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

*Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi*

_____

**Sažetak**

Cilj ovog diplomskog rada je prikazati kako izgleda cjelokupni proces razvoja web aplikacije za pretraživanje jeftinih avionskih letova po nizu različitih kriterija od početne faze istraživanja i planiranja, kroz razvoj sve do faze testiranja, analize kvalitete programskog koda i u konačnici kontinuirane integracije i isporuke gotovog programskog rješenja. Aplikacija će biti razvijena u ASP.NET Core razvojnom okruženju korištenjem novog Microsoftovog razvojnog okvira za klijentsko web programiranje pod nazivom Blazor, dok će se podaci o letovima dohvaćati s Amadeus web servisa putem API-ja temeljenog na REST tehnologiji. Naglasak u ovom radu će biti stavljen na prakse dobrog programiranja i pisanja čistog koda, kao i na važnost softverske arhitekture. Obradit će se poglavlja vezana uz značenje arhitekture u razvoju softvera, usporedba arhitekturalnih principa kroz povijest, te detaljno objašnjenje dizajna usmjerenog na domenu poslovanja kroz teorijsku podlogu i praktičnu primjenu u sklopu razvoja aplikacije za pretraživanje jeftinih letova.

**Ključne riječi:** čista arhitektura; slučajevi korištenja; port; adapter; jedinični test; Amadeus; REST API; ASP.NET Core; Blazor; Jenkins; Octopus Deploy.

# Sadržaj

# 1. Uvod

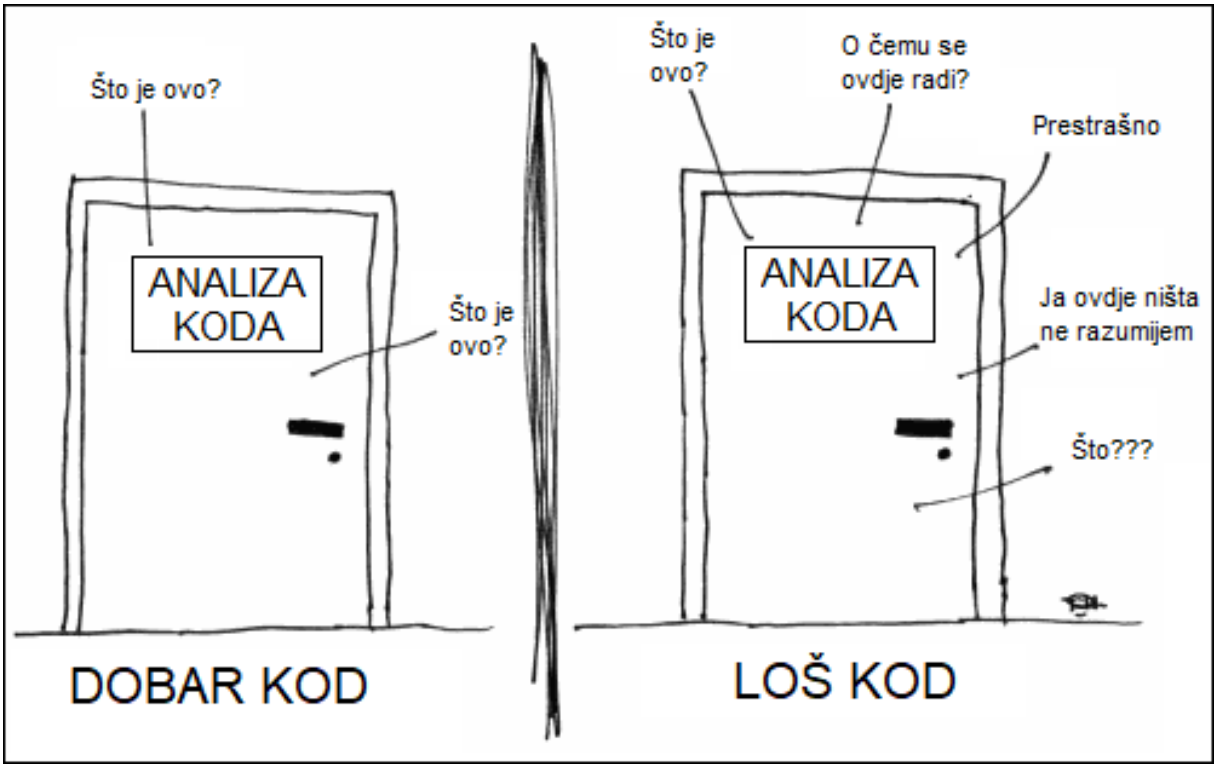
U današnjem svijetu ubrzanog digitalnog razvoja, računalne aplikacije su došle do te razine da trenutno igraju središnju ulogu u samoj svrsi i načinu poslovanja velikog broja poduzeća, pa ih u skladu s tim i tvrtke proizvode u dosad neviđenom broju. One više nisu neki popratni sadržaj kojeg bi bilo dobro imati kako bi se unaprijedilo poslovanje. Štoviše, one su u ovom modernom dobu *središnji stup* poslovanja, bez kojih bi bilo izrazito teško, gotovo i nemoguće ostvariti poslovne ciljeve. Računalne aplikacije danas pokrivaju široki spektar poslovanja suvremenih organizacija – od interakcije s kupcima, potencijalnim klijentima i partnerima sve do donošenja poslovnih odluka. Zapravo, čak i u ovom teškom razdoblju pandemije COVID-19, kada se brojne djelatnosti i industrije zaustave, softverska industrija je jedna od rijetkih koja ne samo da preživljava, nego i napreduje.

Medicinski uređaji i bolnički sustavi, automobili, zrakoplovi, bankomati i bankarski sustavi, SOS sustavi, a u konačnici i cijela kritična infrastruktura ovise o računalnom softveru za učinkovito i djelotvorno funkcioniranje. Međutim, kako bi se zadovoljili kriteriji djelotvornosti i učinkovitosti, postoje određeni preduvjeti koje softver mora ispuniti, od kojih svakako treba istaknuti aspekte kvalitete i sigurnosti. U skladu s tim se nameću određena pitanja, kao npr. što znači kvalitetan softver, kada se može reći da je softver dovoljno siguran ili koje kriterije treba zadovoljavati softver kako bi se smatrao kvalitetnim, odnosno sigurnim. Na ova pitanja nije uvijek jednostavno dati odgovor, a zapravo se tiču domene softverske arhitekture. Vrlo lako se stoga može zaključiti da arhitekturalni pristup osmišljavanju, dizajnu i razvoju programskog proizvoda igra neizmjerno važnu ulogu u kontekstu postizanja kvalitete i sigurnosti. Pored toga, ne smije se zanemariti ni uloga poslovne domene u procesu razvoja računalnog softvera – uspješan softver treba zadovoljiti potrebe klijenta i pružiti podršku u donošenju poslovnih odluka.

Međutim, posljednjih godina je softverska arhitektura pala u sjenu modernih razvojnih okvira i alata koji automatiziraju brojne koncepte najboljih praksi razvoja softvera. Objektno-orijentirani dizajn, principi pisanja modularnog i održivog koda, uzorci dizajna, razvoj programa koji se mogu skalirati, koji nisu zakrpani „pod maskom“ nestabilnog i nekvalitetnog koda, procesi analize koda, kontinuirana integracija i isporuka programskog proizvoda i brojni drugi koncepti dobrog razvoja su sve više integrirani u suvremene automatizacijske sustave – paralelno s tim uloga softverskog inženjera sve više blijedi. Sve češće se može čuti kako se kao odgovor na postavljeno pitanje o arhitekturi korištenoj u razvoju nekog softverskog sustava spominje razvojni okvir ili neki drugi tehnološki detalj, što je **krivo**. Treba naglasiti da tehnološki detalji nemaju nikakve poveznice s primijenjenim arhitekturalnim pristupom.

Ideja ovog diplomskog rada se temelji na razvoju web aplikacije za pretraživanje jeftinih avionskih letova u skladu s najboljim razvojnim praksama. Upravo na primjeru ovakve aplikacije će biti prikazan trenutno aktualan arhitekturalni trend usmjeren na domenu poslovanja pod nazivom „čista arhitektura“ (eng. *clean architecture*). U početku će biti opisana tradicionalna troslojna arhitektura i njeni brojni nedostatci, zatim će detaljno biti prikazan put od samih početaka domenski-usmjerenih arhitektura sve do čiste arhitekture, nakon čega slijedi opis izvedbe navedene arhitekture u kontekstu razvoja aplikacije za pretraživanje jeftinih avionskih letova, korišteni tehnološki detalji, uloga i važnost testova u kontekstu razvoja softvera te prikaz implementacije testova u spomenutoj aplikaciji, a u konačnici i sam proces kontinuirane integracije i isporuke kao i prikaz rada aplikacije.

Kao završni dio uvodnog poglavlja na slici 1 je istaknuta jedna šaljiva ilustracija koja na zanimljiv način prikazuje problem, prije svega kvalitete koda, a usto i same arhitekture softvera.




Slika 1. Šaljiva usporedba analize dobrog i lošeg koda (Prema: [1])

# 2. Uloga softverske arhitekture

Na samom početku ovog poglavlja vrijedi istaknuti jednu prilično zanimljivu misao: „Primarni zadatak arhitekta je pobrinuti se da kuća bude upotrebljiva, a ne da je napravljena od cigle“. Ovu misao je jednom prilikom izjavio Robert Cecil Martin (poznat i kao Ujak Bob), jedan od najutjecajnijih softverskih inženjera i predavača u području dizajna čistog programskog koda i arhitekture. Cilj ovog poglavlja je zapravo kroz nekakav povijesni okvir razvoja arhitekturalnih principa objasniti i približiti ulogu softverske arhitekture u razvoju složenih poslovnih aplikacija.

Pitanje koje se nerijetko može čuti u krugovima razvojnih programera je zašto bi se programer trebao brinuti o arhitekturi softvera. Naravno, ovo pitanje je opravdano i tiče se relativno kompleksnih područja unutar domene softverskog razvoja. Kako bi se došlo do odgovora, potrebno je granu programskog inženjerstva gledati kroz nešto širu sliku i razmotriti neke od ključnih čimbenika, kao što su široka dostupnost velikog broja razvojnih okvira i programskih jezika, ili pak sama svrha, odnosno uloga programskih jezika visoke razine. U skladu s tim, razvojni programer treba obratiti pozornost na određene kriterije, kao npr. da li je moguće izmijeniti neki dio projekta, a da se pritom ostatak sustava ne uruši, koliko je razumljiva struktura koda u projektu i u konačnici na koji način opisati, odnosno predočiti arhitekturu projekta.

Sva navedena promišljanja i posezanja za odgovorima na spomenuta pitanja vode prema sljedećem zaključku – raznorazni programski jezici visoke razine, brojni razvojni okviri (eng. *framework*), sintaktički šećeri (eng. *syntactic sugar*), jednolinijska rješenja u programskom kodu (eng. *one-liner*) i mnogi drugi koncepti i alati prvenstveno služe programeru kako bi pisao kvalitetniji, stabilniji i razumljiviji programski kod. Računalo razumije samo strojni kod, njemu kvalitetan i održiv kod napisan u programskom jeziku visoke razine ne predstavlja apsolutno nikakvu ulogu, budući da će ono bez problema pokrenuti softver čak i da je napisan najgorim programerskim praksama. U takvim situacijama jedino ispaštaju programeri, pa bi upravo zbog toga trebali pratiti suvremene trendove i pristupe razvoju softvera, kako bi proizveli fleksibilna, modularna i održiva programska rješenja, budući da je **čist kod** ujedno i **strukturiran kod**, a strukturiran kod je odraz **čiste arhitekture**. Ono što svakako treba izbjeći je pisanje tzv. „špageti koda“. Slika 2 prikazuje jedan od primjera špageti koda, a odnosi se na ulančavanje *callback* funkcija u programskom jeziku JavaScript, pa se taj fenomen ujedno naziva i „pakao *callback*-a“ (eng. *callback hell*).

```
http.get('https://api.github.com/users', function (users) {
  /* Prikaži sve korisnike */
  console.log(users);
  http.get('https://api.github.com/repos/javascript/contributors?q=contributions',
    function (contributors) {
      /* Prikaži sve suradnike */
      console.log(contributors);
      http.get('https://api.github.com/users/John', function (userData) {
        /* Prikaži korisnika s korisničkim imenom 'John' */
        console.log(userData);
      });
    });
});
```

Slika 2. *Callback hell* u programskom jeziku JavaScript (izrada autora)

Međutim, ponekad postoje situacije u kojima moramo zadovoljiti kratke vremenske rokove i ispuniti funkcionalne zahtjeve programskog proizvoda, zbog čega se silna pravila i koncepti pisanja čistog koda mogu činiti kao nepotrebno opterećenje. Pojedinci će ovakve situacije čak i iskoristiti kao izgovor za razvoj nekvalitetnog projekta, ali činjenica je da i u takvim situacijama ipak treba težiti tomu da projekt bude koliko je moguće dobro strukturiran, razumljiv i proširiv. Iznimke od ovih pravila se odnose na razvoj vrlo jednostavnih aplikacija koje nemaju specificirane slučajeve korištenja (eng. *use case*), skripte s jednostavnim naredbama, ugrađene (eng. *embedded*) aplikacije koje se temelje na visokim performansama ili scenariji u kojima programeri početnici u svojoj karijeri pristupajući razvoju softvera zbog nedostatka profesionalnog iskustva na krivi način realiziraju koncepte dobrog arhitekturalnog dizajna. U većini drugih slučajeva razumijevanje i pravilna primjena najboljih arhitekturalnih koncepata će rezultirati stvaranjem boljih aplikacija, a pojedinca će učiniti i **vrjednijim** razvojnim inženjerom. Također, istraživanje i proučavanje suvremenih arhitekturalnih trendova pomaže u razmišljanju i stvaranju ideja u kontekstu pristupa razvoju softvera i, što je još važnije, omogućava donošenje kritičnih odluka o arhitekturi softvera. Dobra softverska arhitektura poput čvrstih temelja čini aplikaciju fleksibilnom, jednostavnom za testiranje i smanjuje rizik programerskih pogrešaka, kao i pada cijelog sustava.

Kako bi se u nastavku istaknula prednost čiste arhitekture u odnosu na ostale arhitekturalne principe, potrebno je prvo razumjeti kako su se softverske arhitekture razvijale kroz određeno vremensko razdoblje i shvatiti koji su zapravo nedostatci nekih ranijih pristupa koji su bili na vrlo dobrom glasu i koji su kroz duže razdoblje bili prvi izbor većine razvojnih inženjera. U principu, potrebno je shvatiti zašto je došlo vrijeme za promjenu, pa će stoga u nastavku biti objašnjene dvije glavne grupe arhitekturalnih pristupa, a to su:

- arhitekture usmjerene na bazu podataka
- arhitekture usmjerene na domenu poslovanja.

## 2.1. Arhitekture usmjerene na bazu podataka

Kao prvu grupu u sklopu razumijevanja uloge softverske arhitekture treba istaknuti grupu arhitektura usmjerenih na bazu podataka. U povijesnom kontekstu, ovdje se zapravo radi o prvim poznatim varijantama arhitekturalnih pristupa razvoju softvera. Ovakvi oblici arhitektura su stekli veliku slavu u krugovima programerskih zajednica i još uvijek su u širokoj upotrebi prilikom razvoja softverskih sustava, kao i kod održavanja starih složenih sustava koje bi bilo dosta teško refaktorirati. Jedan od najčešće korištenih arhitekturalnih pristupa iz ove skupine je **troslojna arhitektura** koja je još uvijek toliko popularna, da je i neizostavni dio gradiva većine kolegija na fakultetima. Troslojna arhitektura je prisutna u programerskoj zajednici već dugi niz godina i pokazala se kao pouzdana i skalabilna. Najčešći prvi korak prilikom razvoja aplikacije koja se temelji na ovom arhitekturalnom pristupu je razvoj relacijskog modela podataka za određenu poslovnu domenu, što za sobom potеže i kreiranje baze podataka, relacija, okidača (eng. *trigger*), spremljenih procedura (eng. *stored procedure*) i svih ostalih elemenata u sklopu same baze podataka [2]. Tri sloja koja obuhvaća ova arhitektura su:

- Prezentacijski sloj – usmjeren prema razvoju korisničkog sučelja i prezentaciji sadržaja na određenoj platformi (web, mobilna aplikacija, stolna aplikacija itd.).
- Sloj poslovne logike – usmjeren prema implementaciji aplikacijske i poslovne logike, ovisi o podatkovnom sloju, ali ne smije imati izravnu ovisnost (eng. *dependency*) prema prezentacijskom sloju.
- Podatkovni sloj – sloj koji pretvara nadolazeće zahtjeve sa sloja poslovne logike u upite ili naredbe namijenjene sustavu za pohranu podataka, a u idealnom slučaju bi trebao biti implementiran u obliku apstraktnog sučelja bez ikakvih poslovnih pravila ili aplikacijske logike [2].

Važno je napomenuti kako se ovdje govori o logičkim arhitekturalnim slojevima. U kontekstu softverskog razvoja također postoje i fizički slojevi za organizaciju programskog koda, komponenti i modula. Logički slojevi se odnose na logičko razdvajanje odgovornosti, što znači da se programski kod raspoređuje po slojevima u skladu s odgovornostima i definiraju se komunikacijski protokoli između slojeva u obliku apstraktnih sučelja. Na taj način je vrlo jednostavno pojedine komponente zamijeniti drugima bez nastanka neželjenih posljedica. Također, svi logički slojevi se mogu izvršavati na jednom fizičkom uređaju, iako se u nekim složenijim situacijama u slučaju troslojne arhitekture prezentacijski sloj može izvoditi i na više

različitih strojeva. S druge strane, fizički slojevi predstavljaju implementacijske jedinice i nekad su takvi slojevi zahtijevali poseban fizički uređaj po pojedinom sloju, premda danas postoje moderna rješenja koja na temelju virtualizacije omogućavaju izvođenje više fizičkih slojeva na jednom uređaju (najpoznatiji primjer tomu je Docker) [2].

Troslojna arhitektura je tip arhitekture usmjerene na bazu podataka, pa stoga i ne čudi da je u takvom pristupu podatkovni sloj središnji i najvažniji sloj cijele arhitekture. Već je i ranije spomenuto kako se odmah na početku razvoja temeljenog na ovom pristupu u primarno žarište stavlja baza podataka, pa je tako u srpnju 2018. na konferenciji *Laracon* Ujak Bob na šaljivi način istaknuo kako se pobornici troslojne arhitekture klanjaju bazi podataka i nazvao ju je *božanstvom* troslojne arhitekture.

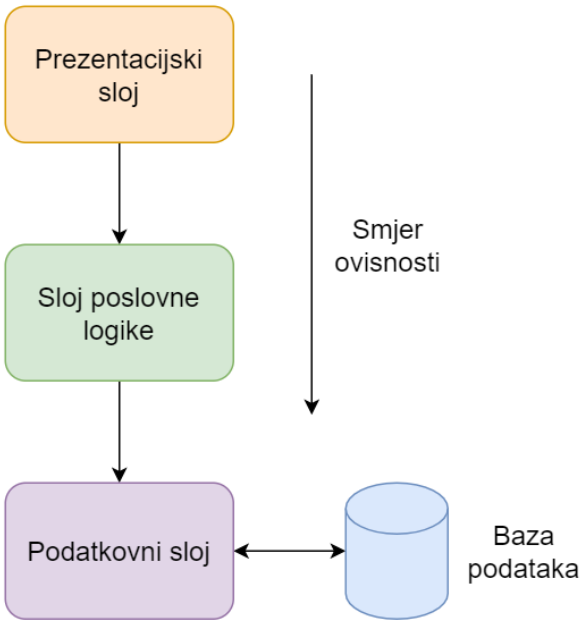
Treba istaknuti da ovakav pristup razvoju softvera nije nužno loš, budući da se temelji na razdvajanju odgovornosti na tri arhitekturalna sloja kako bi se postigla održivost i fleksibilnost programskog proizvoda. Međutim, često se dogodi da poslovna pravila „procure“ u podatkovni sloj i elementi unutar programskog koda postanu toliko čvrsto povezani (eng. *tight coupled*) i zapetljani da je aplikaciju dosta teško, gotovo i nemoguće održavati ili proširiti, pa se u najgorem slučaju aplikacijski kod treba ispočetka napisati. Slika 3 prikazuje okvirni primjer takve situacije i ne treba uopće naglašavati da takav scenarij treba izbjeći po svaku cijenu.

```
namespace OrdersDemo
{
    public class Order
    {
        public string GetCustomersByOrderId(int orderId)
        {
            string connString = @"Data Source=localhost\SQLEXPRESS;Initial Catalog=OrdersDemo;User ID=sa;Password=demo";
            SqlConnection connection = new SqlConnection(connString);
            connection.Open();

            string sql = @"SELECT c.CustomerName, o.OrderID
                           FROM Customers c
                           LEFT JOIN Orders o
                           ON c.CustomerID = o.CustomerID
                           WHERE o.OrderID = @OrderID
                           ORDER BY c.CustomerName";

            string output;

            SqlCommand command = new SqlCommand(sql, connection);
            command.Parameters.AddWithValue("@OrderID", orderId);

            SqlDataReader dataReader = command.ExecuteReader();

            while (dataReader.Read())
            {
                output = output + dataReader.GetValue(0) + " - " + dataReader.GetValue(1) + "\n";
            }

            dataReader.Close();
            command.Dispose();
            connection.Close();

            return output;
        }
    }
}
```

Slika 3. Primjer lošeg koda napisanog u programskom jeziku C# (izrada autora)

Iz navedenog primjera se jasno vide brojni problemi u kodu. Iako ovisnosti idu iz sloja poslovne logike u smjeru podatkovnog sloja, uopće ne postoji sučelje za pristup podacima, nego im se izravno pristupa, tako da u konačnici ovaj isječak koda nije niti u skladu s troslojnom arhitekturom.

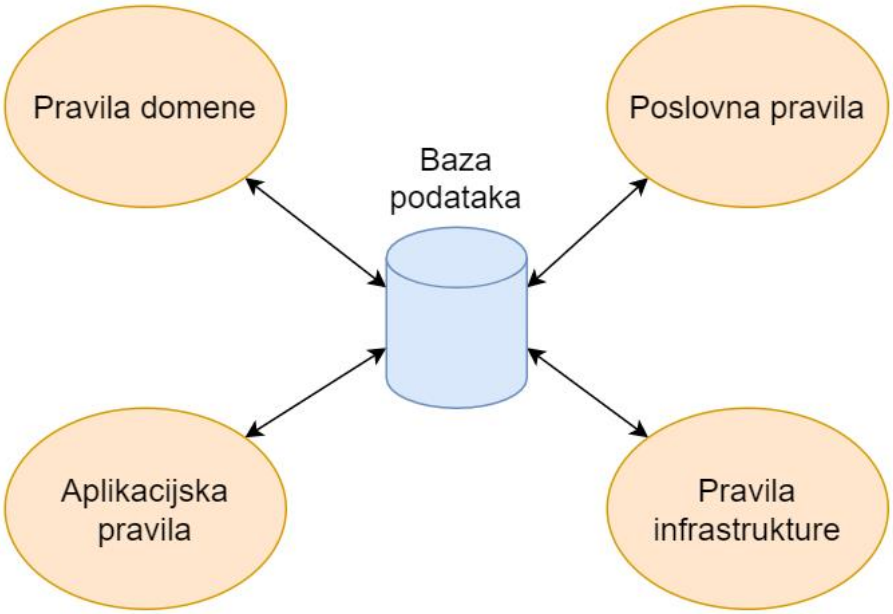
Kada je riječ o troslojnoj ili bilo kojoj drugoj softverskoj arhitekturi, vrlo je važno dobro poznavati smjerove ovisnosti između pojedinih arhitekturalnih slojeva i strogo ih se pridržavati. Ovisnost o nekom sloju se u programskom kodu realizira uvoženjem modula ili klasa tog sloja u sloj koji o njemu ovisi (u programskom jeziku C# se za uvoženje ovisnosti koristi ključna riječ **using**, u Javi je to **import**, u PHP-u **use** itd.) [2]. Slika 4 prikazuje dijagram smjera ovisnosti između slojeva troslojne arhitekture.




Slika 4. Dijagram smjera ovisnosti između slojeva troslojne arhitekture (prema: [2])

Pored već spomenutog problema lomljivih granica između sloja poslovne logike i podatkovnog sloja, nerijetko se događa da se iste klase modela koriste kroz više slojeva, umjesto da se mapiraju u odgovarajuće klase po pojedinom sloju, pa zbog toga prezentacijski sloj neizravno ovisi o podatkovnom sloju i na taj način se zapravo krši propisani smjer ovisnosti troslojne arhitekture. Ovaj problem se najčešće pojavljuje prilikom korištenja sustava za objektno-relacijsko mapiranje (eng. *object-relational mapping*, *ORM*). Posljednji problem koji će ovdje biti istaknut je prevelika ovisnost o podatkovnom sloju. Generalno, problem cijele ove skupine arhitektura je ovisnost o podatkovnom sloju, odnosno o bazi podataka, što se u praksi i nije pokazalo kao idealno. Na slici 5 je prikazan primjer ovog problema, a radi se o tome da,

čak i kada bi postojalo više različitih slojeva (npr. sloj aplikacijskih pravila, infrastrukturnih servisa itd.), opet bi svi usko bili povezani s bazom podataka.




Slika 5. Dijagram prikaza ovisnosti višeslojne arhitekture o bazi podataka (prema: [2])

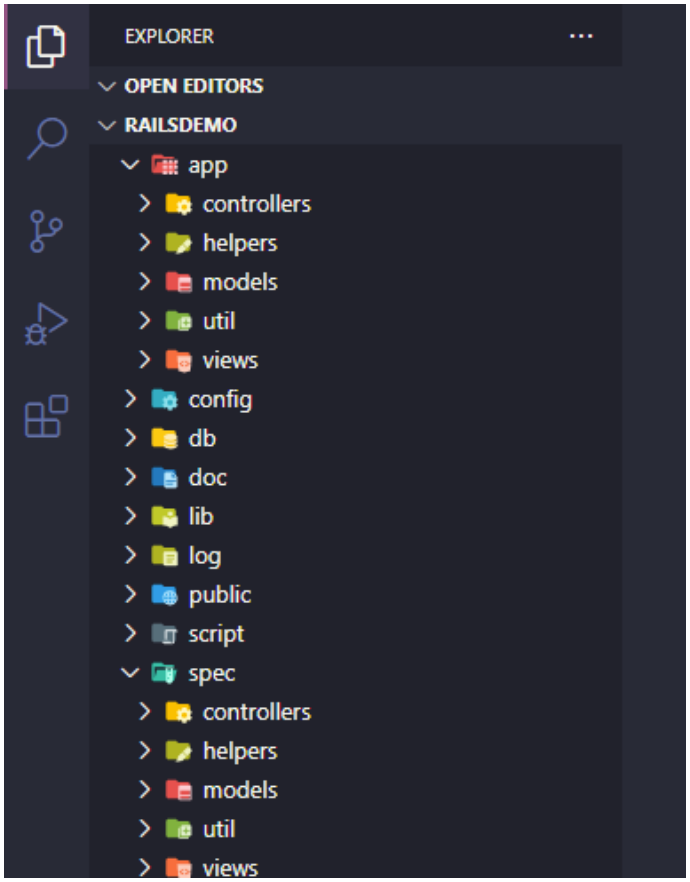
Kako bi se riješili navedeni problemi, razvijeni su moderni arhitekturalni trendovi koji, za razliku od baze podataka, u središte stavljaju domenu poslovanja.

## 2.2. Arhitekture usmjerene na domenu poslovanja

Poslovni svijet se konstantno mijenja, tehnologija se brzo razvija i danas je više nego ikad važno ostati konkurentan na tržištu i vrlo brzo odgovarati na nove izazove i svladavati nove prepreke u poslovanju. Klijenti imaju sve složenije zahtjeve, vremenski rokovi su sve kraći, a potražnja za programerima sve veća, zbog čega tvrtke moraju pravilno reagirati pod silnim pritiskom konkurencije i zahtjevima tržišta. Budući da su i očekivanja vezana uz softver sve veća, drastično se povećava i pritisak na programere, od kojih se očekuje da budu produktivniji više nego ikad i da u što kraćem roku zadovolje sve nove funkcionalne zahtjeve. Zbog navedenih razloga, kao i zbog nedostataka klasičnih višeslojnih arhitektura usmjerenih na baze podataka, pojavili su se novi arhitekturalni trendovi i u zadnje vrijeme postaju sve popularniji u krugovima softverskih inženjera. Središnji stup takvih pristupa arhitekturi je postao sloj poslovnih pravila i generalno ovakve arhitekture su razvijene s ciljem rješavanja poslovnih problema. Do ideje ovakvih pristupa arhitekturi se zapravo došlo kroz niz uzročno-posljedičnih

veza – osoba se želi zaposliti kao programer, kako bi razvila softver koji automatizira svakodnevne zadatke; tvrtka želi automatizirati svoj poslovni proces, pa će stoga zaposliti programera; programer se zatim treba voditi najboljim praksama razvoja softvera, kako bi bio produktivan i zadržao svoje radno mjesto; u konačnici programer radi na razvoju kvalitetnog softvera, kako bi automatizirao poslovni proces tvrtke i riješio poslovne probleme određene domene poslovanja.

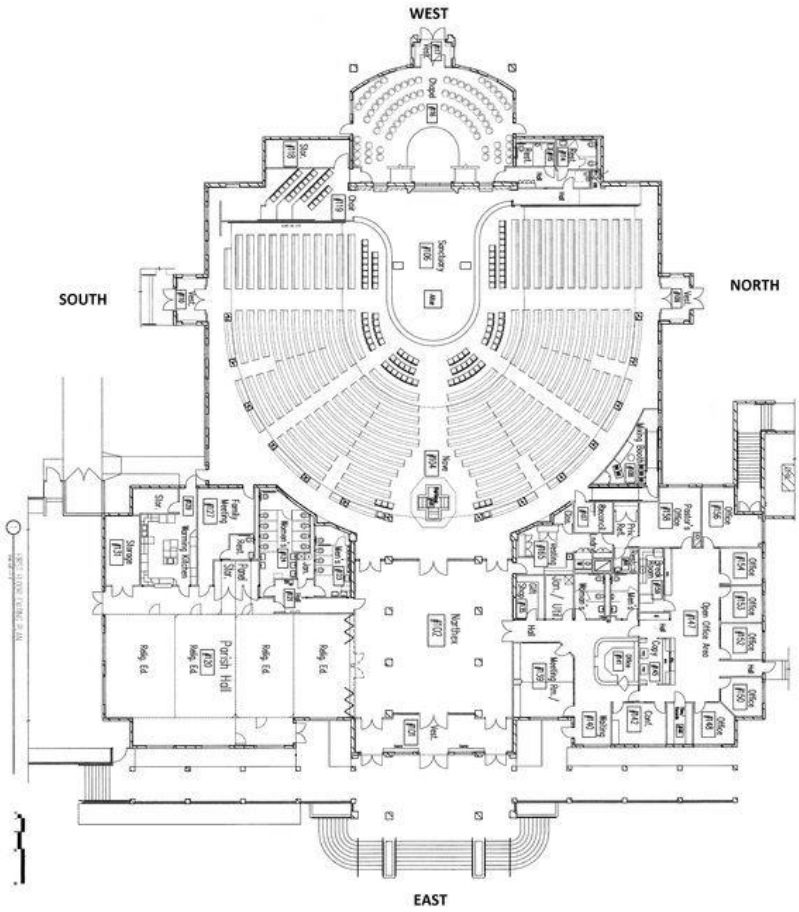
U cijelom nizu navedenih uzročno-posljedičnih veza se nigdje ne spominje nikakav razvojni okvir, alat, baza podataka niti nešto slično. Naravno da se radi o vrlo važnim stavkama koje se koriste u razvoju softvera, međutim poanta arhitektura usmjerenih na domenu poslovanja je rješavanje problema vezanih uz **poslovnu domenu**, dok su svi alati i razvojni okviri zapravo samo **tehnološki detalji**, odnosno mehanizmi koji programeru pomažu kako bi riješio zadane probleme. Zanimljiv primjer prikazuje slika 6. Radi se o strukturi direktorija jednog projekta razvijenog korištenjem Ruby on Rails razvojnog okvira. Vrlo važno pitanje koje se može postaviti je zašto je prva stvar koja upada u oči prilikom pogleda na strukturu direktorija tog projekta razvojni okvir, odnosno Ruby on Rails. Uopće se ne vidi koja je svrha aplikacije. Štoviše, razvojni okvir u ovom primjeru dominira.




Slika 6. Struktura direktorija Ruby on Rails projekta (izrada autora)

Sredinom devedesetih godina prošloga stoljeća veliki se naglasak stavljao na dizajn i arhitekturu programskog koda. Tada su zapravo rasprave oko dizajna i arhitekture bile na vrhuncu i mnogi utjecajni razvojni inženjeri kao što su Ivar Jacobson i Grady Booch su iznosili svoje ideje po tim pitanjima. Programeri su bili usredotočeni na postizanje kvalitetne arhitekture i dizajna softverskih rješenja, sve dok nisu web tehnologije krenule dominirati u informatičkom svijetu i potisnule ideju dizajna i arhitekture u zaborav. Primjer za to je upravo i prikazan na slici 6 – tehnologija dominira, a dizajnu i arhitekturi nema ni traga ni glasa. Programerska struka je u tom i brojnim sličnim primjerima prepoznala ključni problem, posljedica čega je nastanak modernih pristupa razvoju i prije svega arhitekturi programskih proizvoda. U tom kontekstu se web tehnologije, baze podataka, razvojni okviri, brojni alati i sučelja smatraju tehnološkim detaljima, odnosno ulazno-izlaznim uređajima, o kojima programski kod ne bi smio ovisiti.

Razmišljajući o izgledu softverske arhitekture mogu se povući i određene poveznice s arhitekturama građevina, koje su inače dosta precizne i jasno definiraju izgled građevine za koju su namijenjene. Primjer prikazuje slika 7 na kojoj se vrlo jasno vidi da prikazuje arhitekturu crkve.

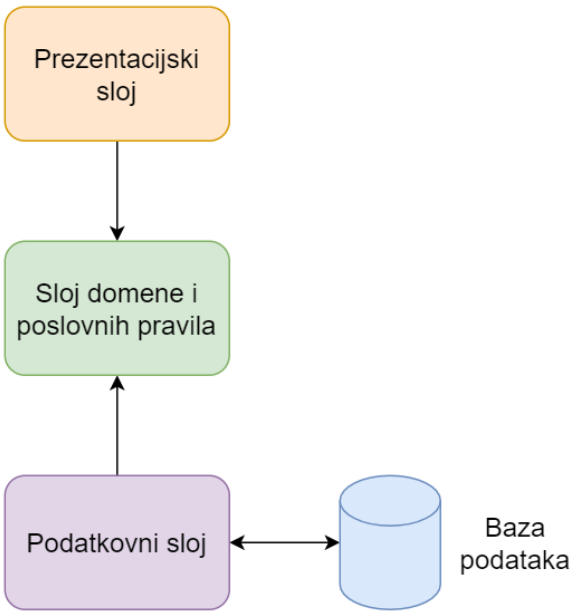



Slika 7. Arhitektura crkve (izvor: [3])

Izgled arhitekture prikazane na slici 7 „vrišti“ svojom namjenom, za razliku od namjene strukture direktorija Ruby on Rails projekta prikazane na slici 6. Upravo na tome se temelje arhitekture usmjerene na domenu poslovanja – arhitektura sustava se ne odnosi na razvojne okvire koje taj sustav koristi, nego na njegovu namjenu. U skladu s tim i ranije prikazana struktura direktorija bi trebala predočiti namjenu pripadajućeg projekta, kao i slučajeve korištenja samog sustava. Softversku arhitekturu koja jasno prikazuje namjenu i slučajeve korištenja pripadajućeg softvera Ujak Bob naziva „vrištećom“ arhitekturom (eng. *screaming architecture*) [4].

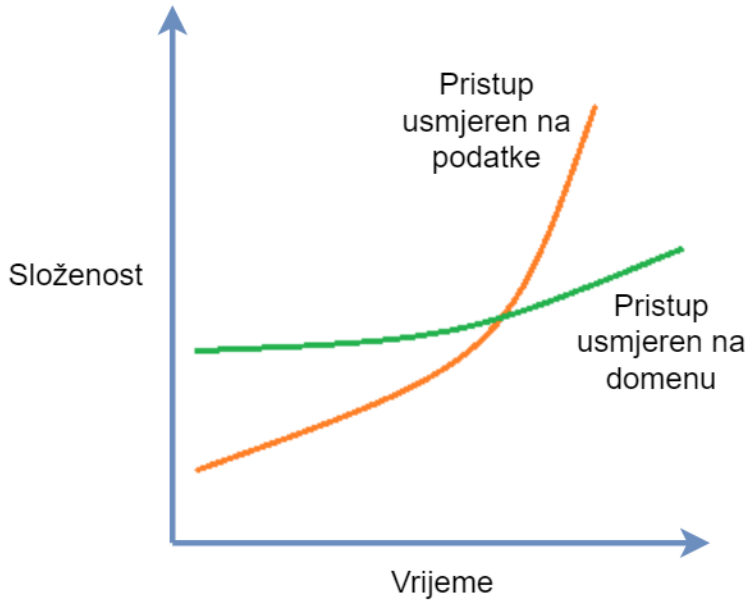
Poanta cijele ove priče je da se prilikom pristupa razvoju softvera trebaju jasno definirati pravila domene i slučajevi korištenja i izdvojiti u poseban arhitekturalni sloj, umjesto ispreplitanja s drugim slojevima. Poslovna pravila se ne smiju miješati s korisničkim sučeljem, razvojnim okvirima, bazom podataka i drugim tehnološkim detaljima. Poštujući ova pravila, vrlo je jednostavno na temelju neke web aplikacije razviti stolnu ili mobilnu aplikaciju iste namjene, budući da nije uopće potrebno dirati sloj domene poslovanja, nego samo prilagoditi korisničko sučelje, odnosno tehnološki detalj. Samim tim se štedi vrijeme i povećava produktivnost, a to su ključne karakteristike za opstanak na konkurentnom tržištu.

Iako se većinom u praksi softverske arhitekture ne koriste u svom izvornom obliku zbog specifičnih funkcionalnih zahtjeva, nekakav izvorni oblik arhitekturalnih principa usmjerenih na domenu poslovanja je prikazan na slici 8. Osnovna ideja je da sloj domene ne smije ovisiti o detaljima.




Slika 8. Dijagram ovisnosti između slojeva arhitektura usmjerenih na domenu poslovanja (prema: [2])

Za kraj ovog poglavlja će još biti navedena usporedba u kontekstu kompleksnosti implementacije sustava temeljenog na arhitekturi usmjerenoj na bazu podataka u odnosu na pristup usmjeren na domenu poslovanja. Zbog jednostavne i jezgrovite programske paradigme koju predlaže, pristup usmjeren na bazu podataka je često lakše izvediv u početnim fazama razvoja sustava. Međutim, što je sustav složeniji, to je pristup usmjeren na podatke sve teže i teže izvediv, a naposlijetku dovodi do toga da je u jednom trenutku gotovo nemoguće uvesti neku novu funkcionalnost u sustav razumnim tempom. S druge strane, pristup usmjeren na domenu poslovanja uvodi dodatnu dozu kompleksnosti u ranim fazama razvoja, ali se s vremenom drastično isplati. U jednom trenutku prilikom razvoja, sustav temeljen na pristupu usmjerenom na domenu postaje lakše održavati i razvijati u odnosu na sustav temeljen na pristupu usmjerenom na podatke. Glavni razlog tomu je činjenica da je problemska domena važnija od podataka koje sustav proizvodi, pa u skladu s tim i napori utrošeni prilikom modeliranja domene nude bolji povrat ulaganja [5]. Slika 9 prikazuje graf složenosti u kontekstu izvedivosti navedenih pristupa.




Slika 9. Graf složenosti u kontekstu izvedivosti pristupa usmjerenog na podatke i pristupa usmjerenog na domenu (prema: [5])

# 3. Dizajn upravljan domenom

Kao što je već ranije spomenuto, sredinom devedesetih godina prošloga stoljeća značajno je porastao interes za područje softverskog dizajna i arhitekture. Ivar Jacobson je još 1992. istaknuo važnost slučajeva korištenja u softverskoj arhitekturi [6], međutim jedan od prvih arhitekata koji je definirao principe dizajna upravljanog domenom 2004. godine je bio Eric Evans [7]. Tada nije nužno govorio o arhitekturi, nego više o skupu principa i savjeta koje je potrebno uzeti u obzir prilikom razvoja sustava koji se bave složenim poslovnim domenama. Radi se o konceptima softverskog dizajna koji je **upravljan problemima domene**, a ne detaljima. Eric Evans je, definirajući spomenute principe, postavio temelj za budući razvoj i nadogradnju softverskih arhitektura usmjerenih na domenu poslovanja [2]. Budući da se radi o iznimno važnim konceptima na kojima se temelje i sve ostale arhitekture iz ove skupine, u nastavku će svaki od njih biti ukratko objašnjen.

Na bilo kojem većem projektu postoji više klasa modela. Ukoliko se kombinira programski kod temeljen na različitim modelima, softver postaje podložan pogreškama, nepouzdan i teško razumljiv. Također, komunikacija između članova tima postaje zbunjujuća, budući da nije jasno u kojem kontekstu treba primijeniti određenu klasu modela. Primjer bi bio situacija u kojoj postoji klasa modela s ogromnim brojem svojstava, a zbog istog naziva se koristi u kontekstu potpuno različitih entiteta problemske domene. Stoga je potrebno izričito definirati kontekst unutar kojeg se model primjenjuje, postaviti granice u smislu organizacije tima i fizičkih manifestacija poput programskog koda i sheme baza podataka. Model treba dosljedno čuvati unutar okvira definiranih granica. **Ograničeni kontekst** (eng. *bounded context*) je koncept dizajna upravljanog domenom koji se odnosi na ograničavanje primjenjivosti određenog modela izvan granica konteksta kojem pripada taj model [7]. Podjelom aplikacijske domene na ograničene kontekste aplikacija postaje još jednostavnija za održavanje, a samim tim se i podvrgava dobrim praksama labavog spajanja (eng. *loose coupling*) i ponovne upotrebljivosti (engl. *reusability*).

Komunikacija između članova tima temeljena na modelima poslovne domene nije nužno ograničena samo na UML dijagrame. Kako bi se model što učinkovitije iskoristio, mora podržavati bilo koji komunikacijski medij. Model domene povećava korisnost pisane tekstualne dokumentacije, kao i neformalnih dijagrama i razgovora u sklopu agilnih procesa. Također poboljšava komunikaciju putem samog programskog koda i testova napisanih za taj kod. Međutim, projekt je suočen s ozbiljnim problemima kada je jezik korišten za komunikaciju između timova nejasan. Primjer je situacija kada domenski stručnjaci koriste svoj žargon, dok članovi tima imaju vlastiti jezik prilagođen za raspravu o domeni u smislu dizajna. Rješenje je zajednički jezik za koji, uz svjestan trud tima, model domene može pružiti oslonac, povezujući

komunikaciju tima s implementacijom softvera. **Sveprisutan jezik** (eng. *ubiquitous language*) je koncept dizajna upravljanog domenom koji uključuje pojmove za raspravu o eksplicitno navedenim pravilima u modelu domene [7]. Obogaćen je nazivima uzoraka koje tim obično primjenjuje na model domene, pomaže u eliminiranju kontradikcija tijekom životnog ciklusa projekta i predstavlja poveznicu između programera i domenskih stručnjaka (dionika).

Modeliranje objekta teži prema tome da se u središte pozornosti stavljaju njegovi atributi. Međutim, neki objekti nisu definirani prvenstveno svojim atributima, nego predstavljaju oblik identiteta koji prolazi kroz vrijeme i često kroz različite reprezentacije. Ponekad se takav objekt mora uskladiti s drugim objektom, iako im se atributi razlikuju, a s druge strane se mora razlikovati od drugog objekta, iako mogu imati iste atribute. Pojam identiteta je vrlo važan, samim tim što pogrešan identitet može dovesti do oštećenja podataka. **Entitet** (eng. *entity*) je koncept dizajna upravljanog domenom koji predstavlja objekt definiran prvenstveno svojim identitetom [7]. Entiteti su poslovni modeli koji imaju svoj životni ciklus, poslovnu vrijednost i identitet. Imaju svoje određeno stanje, mogu se mijenjati i sadrže jedinstveni identifikator. Primjer bi bile dvije narudžbe s istim proizvodima, ali s poslovnog gledišta potpuno su različite.

Praćenje identiteta entiteta je bitno, ali dodavanje identiteta drugim objektima može naštetiti performansama sustava i zapetljati model uzrokujući da svi objekti isto izgledaju. Softverski dizajn predstavlja konstantnu bitku sa složenošću. Potrebno je napraviti određene razlike između entiteta i ostalih objekata. Kao posljedica tomu, postavlja se pitanje što napraviti s ostalim objektima koji nisu entiteti, ali imaju neke svoje karakteristike i svoj značaj za model. Takvi objekti prvenstveno opisuju stvari. **Vrijednosni objekt** (eng. *value object*) je koncept dizajna upravljanog domenom koji predstavlja objekt čija je svrha predočiti opisni aspekt domene bez konceptualnog identiteta [7]. Radi se o nepromjenjivim objektima koji se najčešće koriste kao atributi i uspoređuju se po jednakosti na temelju svojih svojstava. Mogu imati i određena ograničenja u skladu s pravilima domene, a najčešći primjeri takvih objekata su datum, cijena, valuta, koordinata, težina itd.

Minimalistički dizajn asocijacija između klasa pojednostavljuje iteraciju kroz povezane elemente i donekle ograničava eksploziju njihovih međusobnih odnosa. Međutim, većina poslovnih domena je toliko uzajamno povezana da na kraju iteracije često prolaze dugačkim i dubokim stazama kroz reference objekata. U konačnici, ovakav splet veza odražava stvarnost svijeta koja rijetko definira oštre granice. To je jedan od značajnih problema u dizajnu softvera. Teško je jamčiti dosljednost promjena objekata u modelu sa složenim asocijacijama. Potrebno je održavati svojstva nepromjenjivosti koja se odnose na usko povezane skupine objekata, a ne samo na diskretne objekte. Zapravo, pronalaženje uravnoteženog rješenja za ovakve probleme zahtijeva dublje razumijevanje domene protežući se na čimbenike poput učestalosti promjena između instanci određenih klasa. Potrebno je pronaći model koji točke s visokim

sadržajem čini labavijima, a točke sa strogim svojstvima nepromjenjivosti čvršćima. **Agregat** (eng. *aggregate*) je koncept dizajna upravljanog domenom koji predstavlja skupinu povezanih objekata tretiranih kao cjelovitu jedinicu u svrhu izmjene podataka. Svaki agregat sadrži **korijen** i **granicu** [7]. Granica definira što se sve nalazi unutar agregata, a korijen predstavlja jedan specifični entitet sadržan u agregatu. To je jedini član agregata kojeg vanjski objekti smiju referencirati, iako objekti unutar granice smiju jedni druge referencirati. Svi entiteti osim korijena imaju lokalni identitet, ali taj identitet se treba moći razlikovati samo unutar agregata, budući da ga nijedan vanjski objekt ne može vidjeti izvan konteksta korijenskog entiteta. Primjer agregata bi mogao biti automobil, pri čemu bi entiteti unutar granica tog agregata bili guma, kotač, vrata itd. a korijen agregata bi bio entitet koji predstavlja automobil.

U konačnici, glavna ideja je predstaviti softversku arhitekturu na način da što je moguće bliže oslikava problemsku domenu poduzeća. Samim tim će se održati stabilna komunikacija s domenskim stručnjacima, a softverski sustav će biti održiv i spreman za promjene pravila domene. U nastavku će kronološkim redoslijedom biti opisane najznačajnije arhitekture usmjerene na domenu poslovanja.

## 3.1. Heksagonalna arhitektura

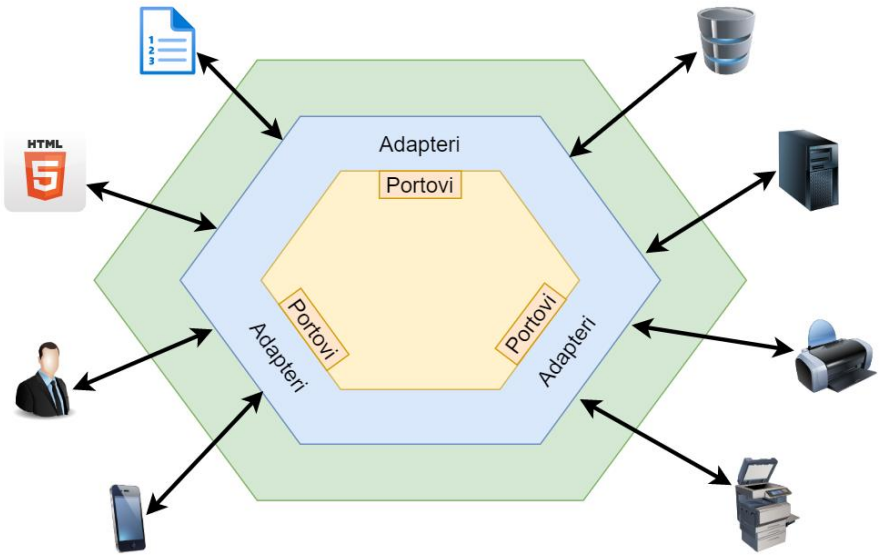
Jedna od prvih arhitektura kod koje je sloj domene smješten u samoj jezgri je heksagonalna arhitektura, odnosno arhitektura portova i adaptera. Predstavio ju je 2005. Alistair Cockburn u jednom od svojih članaka istaknuvši da sloj domene treba biti odvojen od vanjskog svijeta (razvojnih okvira, korisničkih sučelja i drugih tehnoloških detalja) [8]. Stoga se može zaključiti kako u ovoj arhitekturi postoje dva glavna sloja: vanjski i unutarnji sloj. Unutarnji sloj sadrži svu poslovnu logiku, pravila domene, objekte domene, kontekste i agregate. Nasuprot tomu, vanjski sloj predstavlja sloj tehnoloških detalja, alata, građevnih blokova i razvojnih okvira koji pružaju neophodnu funkcionalnost za unutarnji sloj. Oba sloja su međusobno povezana putem portova i adaptera [2]. Unutarnji sloj je predstavljen u obliku šesterokuta (po čemu je i sama arhitektura dobila naziv). Važno je naglasiti da Alistair Cockburn nije napisao ništa vezano uz način strukturiranja koda unutar šesterokuta, nego je samo promovirao ideju o portovima i adapterima [8].

Kako bi se spriječilo curenje tehnoloških detalja u aplikacijsku jezgru, koristi se koncept **portova** i **adaptera** u ulozi posrednika između vanjskog i unutarnjeg sloja. Navedeni koncept se također koristi i kako bi se spriječilo curenje poslovnih pravila prema vanjskom sloju, održavajući aplikacijsku jezgru stabilnom. Adapteri se u ovoj arhitekturi koriste u obliku komunikacijskih premosnica između servisa koji su potrebni domenskoj jezgri, dok portovi

predstavljaju sučelja, izlažući samo određene funkcionalnosti i značajke aplikacije pa se na taj način podvrgavaju *interface segregation* principu dizajna.

Vanjski sloj se sastoji od pokretača, odnosno primarnih sudionika koji predstavljaju vanjske aktere i stupaju u interakciju s aplikacijom radi dobivanja odgovora ili izvršavanja neke naredbe (to su korisnici aplikacije) i pokrenutih (sporednih) sudionika koje pokreće aplikacija (npr. baza podataka ili vanjski poslužitelj) [8]. Također je važno naglasiti da ovisnosti idu iz vanjskog prema unutarnjem sloju, što je zapravo i glavna ideja arhitekture usmjerene na domenu. Svi tehnološki detalji zadovoljavaju potrebe jezgre aplikacije samo onim značajkama koje su joj potrebne i koje su najčešće definirane u obliku sučelja unutar sloja domene.

Portovi su smješteni između sudionika i jezgre aplikacije, s tim da pripadaju unutarnjem sloju, a nalaze se na samom rubu sloja i obrađuju interakcije u skladu s određenom svrhom. Oni u principu predstavljaju sučelja koja aplikacija koristi kako bi ostvarila komunikaciju s vanjskim sudionicima. Pružaju način pristupa vanjskim servisima kako bi aplikacija ostvarila određeni cilj. Također, važno je istaknuti da portovi nisu svjesni tehnoloških detalja, oni samo pružaju sučelje, a adapteri koji ih implementiraju izvršavaju pozadinsku logiku. Adapteri su softverske komponente koje omogućavaju interakciju između tehnoloških detalja i portova unutarnjeg sloja. Nalaze se izvan razine portova i koriste sučelja tih portova pretvarajući specifičan tehnološki zahtjev u zahtjev koji nije svjestan tehnologije. Ovakav pristup dizajnu je u skladu sa SOLID principima dizajna programskog koda i omogućava vrlo jednostavno pisanje jediničnih testova kao i izmjenu tehnologije bez urušavanja aplikacijske domene [8]. Slika 10 prikazuje dijagram heksagonalne arhitekture.

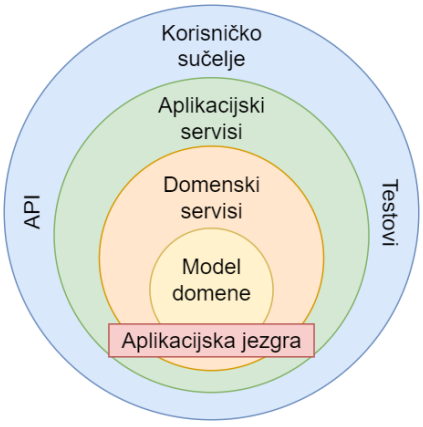



Slika 10. Dijagram heksagonalne arhitekture (prema: [2])

## 3.2. Onion arhitektura

Jeffrey Palermo je 2008. godine pokušavajući riješiti uobičajene probleme održavanja aplikacijskih sustava te naglašavajući princip razdvajanja odgovornosti (eng. *separation of concerns*) predstavio arhitekturu koja se temelji na principima dizajna upravljanog domenom (eng. *domain-driven design*, *DDD*) i *dependency inversion* principa, a poznata je pod nazivom onion arhitektura [9]. Sadrži više slojeva u odnosu na heksagonalnu arhitekturu, a osnovna pretpostavka joj je da uvodi kontrolu u pogledu povezivanja slojeva. Temeljno pravilo je da sav programski kod u vanjskim slojevima može ovisiti o središnjim slojevima, ali programski kod u jezgri ne može ovisiti o slojevima dalje od jezgre, odnosno ovisnosti idu izvana prema unutra. U principu, kronološki gledano svaka sljedeća arhitektura iz ove skupine sadrži više slojeva i detaljnija je, ali glavne ideje i principi ostaju isti. Dodatni slojevi su uvedeni iz razloga kako bi se osiguralo bolje razdvajanje odgovornosti između dijelova sustava.

Kao što je bio slučaj kod heksagonalne arhitekture, tako je i ovdje sloj domene u samoj jezgri i predstavlja stanja i ponašanja koja modeliraju sliku poslovnih pravila organizacije. Nakon njega se nalazi sloj koji generalno obuhvaća sučelja korištena od strane domene za komunikaciju prema vanjskom svijetu. Aplikacijski sloj je dodatni posrednik između jezgre domene i vanjskog svijeta. Najčešće sadrži slučajeve korištenja (eng. *use case*) koji se odnose na samu aplikaciju i mogu se predočiti u obliku obuhvatnih jedinica koje prikupljaju sve potrebne ulazne podatke, usmjeravaju ih prema domenskim servisima i naposljetku vraćaju rezultat pozivatelju. U situacijama kada značajan proces ili transformacija u domeni ne spada pod prirodnu odgovornost entiteta ili vrijednosnog objekta, modelu se dodaje operacija u obliku samostalnog sučelja deklariranog kao domenski servis i smještenog u sloj servisa domene. Sučelje se treba definirati u skladu s jezikom modela, a naziv operacije treba biti dio sveprisutnog jezika organizacije. Slika 11 prikazuje dijagram onion arhitekture.




Slika 11. Dijagram onion arhitekture (prema: [9])

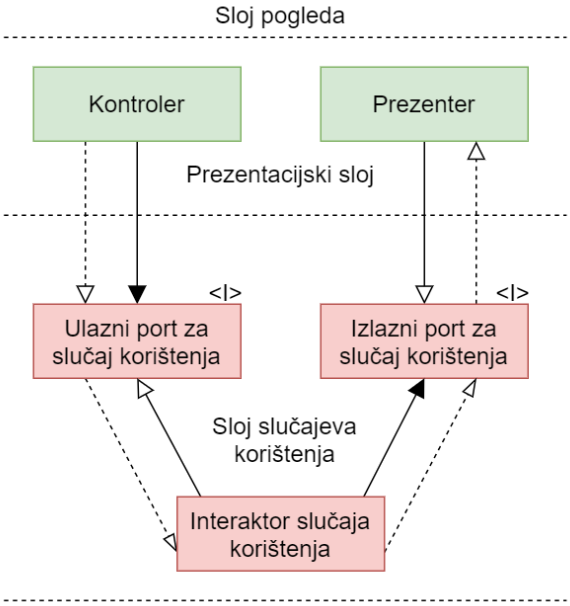
## 3.3. Čista arhitektura

Slijedeći principe heksagonalne i onion arhitekture, Robert Cecil Martin (Ujak Bob) je pokušavajući napraviti korak naprijed u smislu izoliranog, održivog, jednostavnog za testiranje, skalabilnog i čistog koda 2012. godine predstavio svoju verziju arhitekture usmjerene na domenu poslovanja i nazvao ju je čistom arhitekturom [10]. Ujak Bob je inače vrlo utjecajan na području softverskog dizajna i arhitekture. Napisao je niz sjajnih knjiga o agilnim praksama, čistom kodu, objektno-orijentiranom dizajnu i drugim sličnim temama, a njegova nedavno izdana knjiga pod nazivom „Clean Architecture: A Craftsman's Guide to Software Structure and Design“ [4] je privukla mnogo pozornosti u programerskoj zajednici i pokrenula ogromnu zanimaciju za pojmom čiste arhitekture. Neke od ključnih pretpostavki koje je istaknuo Ujak Bob vezano uz čistu arhitekturu su neovisnost o razvojnom okviru, neovisnost o tehnološkim detaljima (baza podataka, korisničko sučelje itd.), pravilo koje nalaže da ovisnosti idu iz vanjskog sloja prema unutarnjem, uloga entiteta i slučajeva korištenja, nužnost korištenja adaptera i pretvarača za mapiranje modela u oblik prikladan određenom kontekstu, važnost *dependency inversion* principa dizajna te pojam granica između slojeva i tehnike prosljeđivanja podataka među slojevima. S obzirom na to da je trenutno tema čiste arhitekture postala vrlo popularna, postoje ogledni primjeri za skoro svaki programski jezik koji implementiraju koncepte takve arhitekture. Za iOS sustave je čak razvijen poseban oblik čiste arhitekture pod nazivom VIPER arhitektura [2].

Jedan od najvažnijih koncepata koji se koristi gotovo u svim razvojnim okvirima je inverzija kontrole (eng. *inversion of control*), a u ovom kontekstu se primjenjuje kako bi se realiziralo pravilo ovisnosti u čistoj arhitekturi koje nalaže da ovisnosti idu izvana prema unutra. Slika 12 prikazuje **tijek kontrole** između komponenti čiste arhitekture. Ukoliko dođe do nekog događaja, kontroler će započeti njegovo obrađivanje. Pozvat će neku metodu objekta koji implementira sučelje predstavljeno kao ulazni port za slučaj korištenja, pa će tijek kontrole prijeći granicu između slojeva i nastaviti sa izvođenjem u sloju slučajeva korištenja. Implementacija slučaja korištenja procesira pristigli zahtjev orkestrirajući entitete ili neke druge objekte iz jezgre domene. Nakon toga implementacija slučaja korištenja poziva metodu objekta koji implementira sučelje predstavljeno kao izlazni port za slučaj korištenja te u konačnici prezenter dobiva rezultate od slučaja korištenja, transformira ih u prikladan oblik i nadalje prosljeđuje sloju pogleda [11].

Glavna ideja opisanog tijeka kontrole je da moduli više razine ne bi trebali ovisiti o tehnološkim detaljima, razvojnim okvirima, korisničkom sučelju itd. Kako se može i vidjeti na slici 12, slučaj korištenja surađuje samo sa sučeljima definiranim na istoj razini. Ovdje dolazi do izražaja **pravilo ovisnosti** predstavljeno inverzijom kontrole, *dependency injection*

uzorkom te *dependency inversion* principom dizajna. Implementaciju je moguće u bilo kojem trenutku zamijeniti bez utjecaja na jezgru aplikacije.

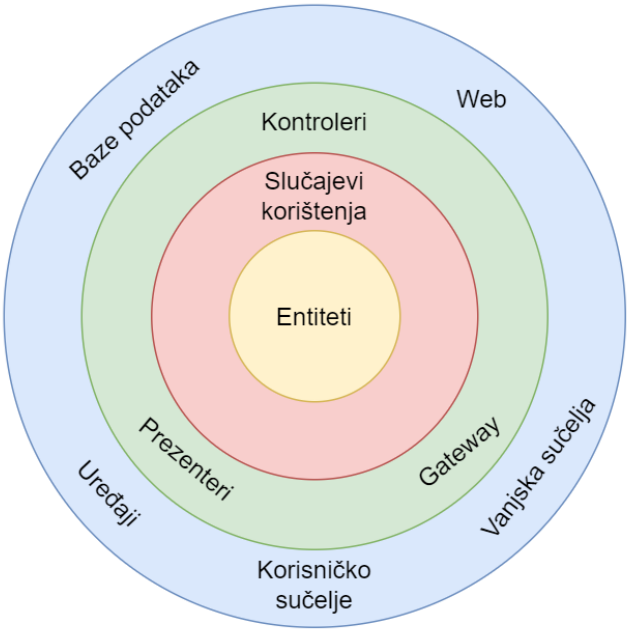



Slika 12. Tijek kontrole u čistoj arhitekturi (prema: [11])

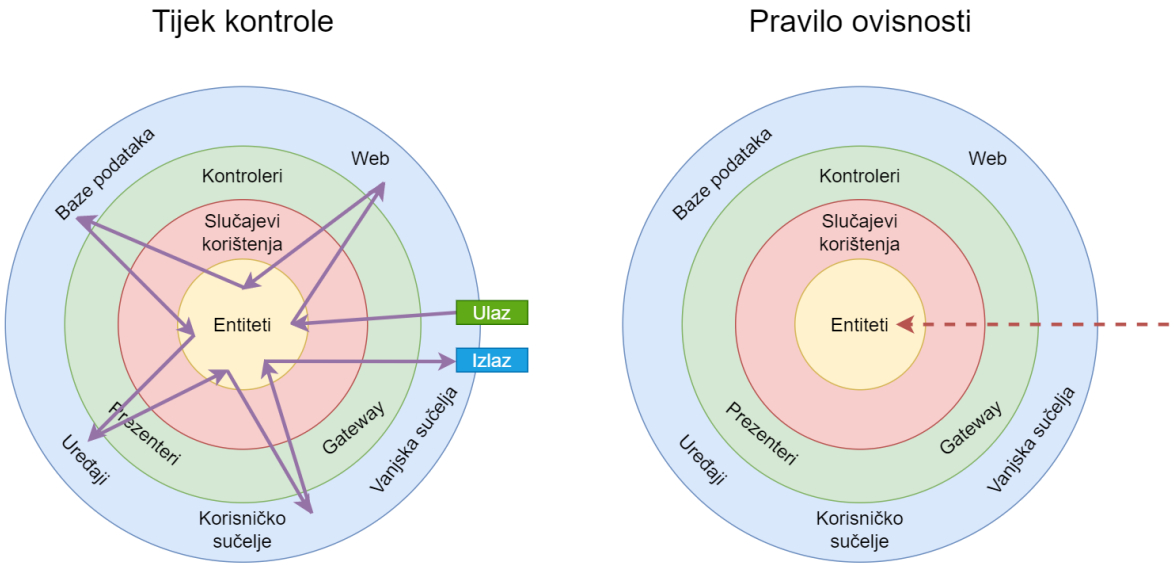
U samom središtu čiste arhitekture je sloj domene, odnosno poslovnih pravila poduzeća, nakon kojeg slijedi sloj poslovnih pravila aplikacije koji obuhvaća slučajeve korištenja. Sljedeći sloj bi bio sloj adaptera sučelja koji obuhvaća kontrolere, prezentere i *gateway*, a posljednji sloj (sloj infrastrukture) obuhvaća razvojne okvire i pokretačke programe. Slika 13 prikazuje dijagram čiste arhitekture, a slika 14 nadopunjuje taj dijagram prikazima tijeka kontrole i pravila ovisnosti između pojedinih slojeva.

Najvažniji sloj iz perspektive arhitekture je sloj domene, odnosno sami entiteti koji obuhvaćaju kritična poslovna pravila. To su pravila koja pružaju potporu poslovnom postojanju neovisno o prisutnosti softverskog sustava. Ovaj sloj bi trebao biti otporan na vanjske promjene. S tehničke strane entiteti, koji se nalaze unutar ovog sloja, predstavljaju objekte koji uglavnom sadrže metode i poslovnu logiku, a ne samo podatke. Vrlo je važno naglasiti da entiteti **nisu** objekti prijenosa podataka (eng. *data transfer object*, *DTO*) niti objekti pristupa podacima (eng. *data access object*, *DAO*) [11]. Sam dizajn sloja domene nije uvijek trivijalan zadatak. U slučaju razvoja poslovne aplikacije, vrlo često se na dizajn ovog sloja utroši velika količina vremena, a pogreške u dizajnu mogu rezultirati ozbiljnim posljedicama. Koncepti dizajna upravljanog domenom imaju za cilj olakšati razvoj velikih poslovnih aplikacija s mnoštvom složenih poslovnih pravila. Čak i ako se ne radi o razvoju poslovne aplikacije, ovi

koncepti mogu uvelike olakšati dizajniranje domene sustava. U kontekstu aplikacija koje se ne razvijaju u sklopu poduzeća, entiteti trebaju predstavljati poslovne objekte same aplikacije, odnosno učahuriti najopćenitija pravila na koja promjene vanjskih slojeva ne smiju utjecati.



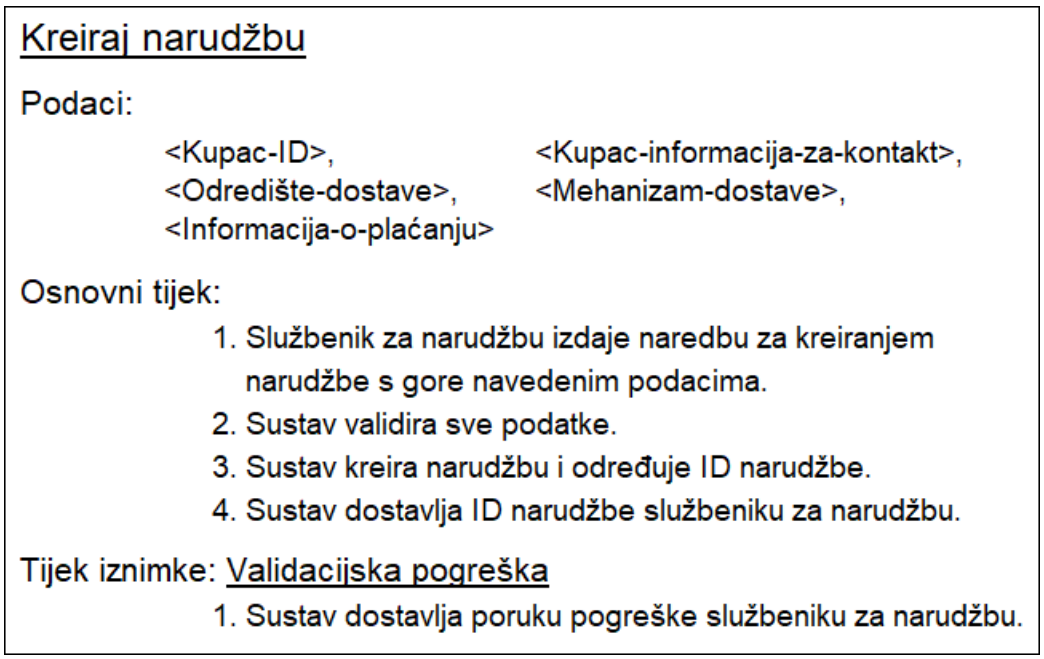

Slika 13. Dijagram čiste arhitekture (prema: [4])




Slika 14. Dijagram tijeka kontrole i pravila ovisnosti u čistoj arhitekturi (prema: [11])

Entiteti ne smiju pohranjivati stanje unutar baze podataka. Problem s velikim brojem ORM biblioteka je da nameću izravno mapiranje entiteta u relacije baze podataka i na taj način krše pravilo ovisnosti. Međutim, postoje razvojni okviri koji ovaj problem rješavaju primjenjujući inverziju kontrole.

Nakon sloja domene dolazi sloj slučajeva korištenja. Ovaj sloj je nešto kompleksniji za razumijevanje i vrlo često se vode rasprave vezane uz samu svrhu ovog sloja. Slučaj korištenja predstavlja popis aktivnosti i komunikacijskih koraka između sudionika i automatiziranog sustava koji su potrebni za postizanje cilja. U principu, skup slučajeva korištenja je način na koji korisnik vidi softverski sustav iz perspektive funkcionalnosti. U početnim fazama pristupa dizajniranju softverskog sustava često je neizostavan korak i izrada dijagrama slučajeva korištenja. Što su na tom dijagramu bolje pokrivene moguće operacije sustava, lakše je dizajnirati odgovarajuću arhitekturu i odabrati strategiju i metodologiju razvojnog procesa. S tehničkog gledišta slučajevi korištenja su najčešće zaduženi za orkestraciju entiteta. Softver u ovom sloju sadrži poslovna pravila specifična za samu aplikaciju. On sažima i implementira sve slučajeve korištenja u sustavu. Aplikacija u skladu s tim predstavlja automatizirani sustav, budući da se aplikacijski specifična pravila mogu promijeniti ukoliko dođe do promjene aplikacijskih zahtjeva. Slika 15 prikazuje primjer jednog slučaja korištenja vezanog uz kreiranje narudžbe.




Slika 15. Primjer slučaja korištenja vezanog uz kreiranje narudžbe (prema: [6])

U navedenom primjeru slučaj korištenja za kreiranje narudžbe je dio aplikativnog sustava za unos narudžbi. Sadrži skup ulaznih i izlaznih podataka pri čemu nisu spomenuti nikakvi detalji u smislu tipova podataka, budući da se takvi detalji niti ne bi smjeli nalaziti u slučaju korištenja. Zatim je naveden primarni tijek događaja koji je vrlo jednostavan i razumljiv. Na primjer, u drugom koraku se validiraju podaci, međutim nije navedeno na koji način se vrši validacija, što za slučaj korištenja nije niti bitno. Također, u trećem koraku se kreira narudžba i određuje ID narudžbe, vrlo vjerojatno putem baze podataka, ali u kontekstu slučajeva korištenja baza podataka je nepoznanica i ne smije se spominjati. Slična stvar je i sa četvrtim korakom, gdje se rezultat dostavlja vjerojatno putem web stranice, ali u ovom kontekstu to nije bitno.

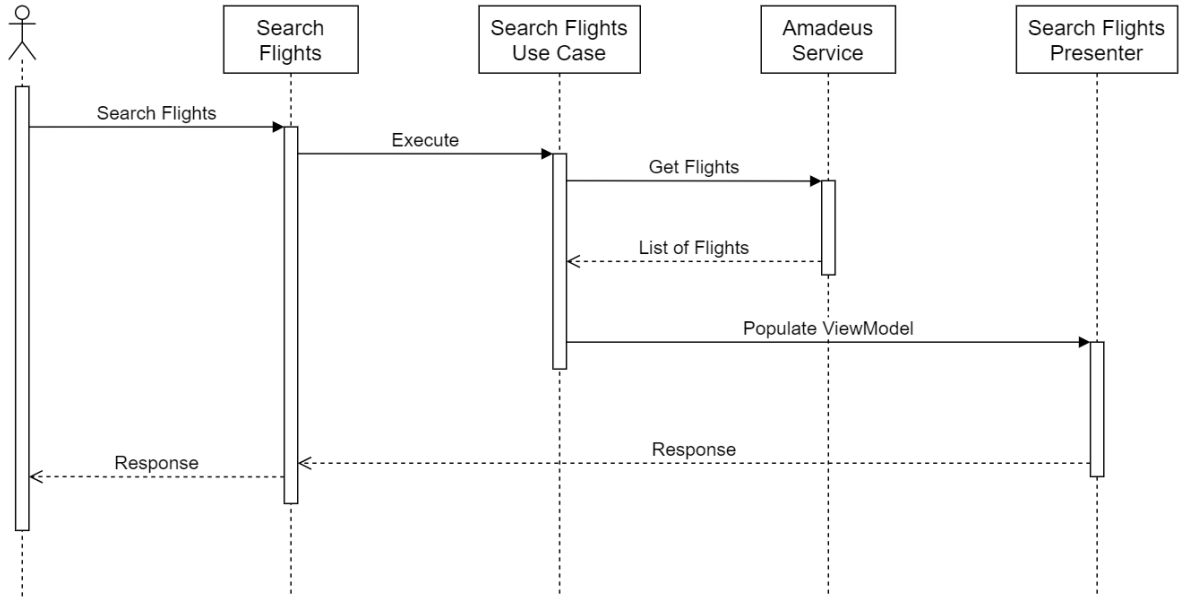
Sljedeći sloj u čistoj arhitekturi je sloj adaptera sučelja. Poanta ovog sloja je da se temelji na transformacijama podataka u prikladne strukture za unutarnji ili vanjski sloj prema kojem se ti podaci prosljeđuju. U principu, ovaj sloj služi kao poveznica između vanjskog sloja detalja i aplikacijskog sloja slučajeva korištenja. Na primjer, podaci koji se prosljeđuju prema sustavu za prikaz će se ovdje transformirati na način da im sva polja budu u obliku tekstualnih *string* tipova podataka. Podaci koji se razmjenjuju između sloja adaptera sučelja i okolnih slojeva se u principu mogu smatrati modelima. Ovaj sloj također obuhvaća koncept pod nazivom *gateway*, a radi se zapravo o apstrakciji iza koje se krije neka kompleksna implementacija (nešto slično *facade* uzorku dizajna) [11]. Primjeri bi bili skladište podataka (*repository* uzorak dizajna), API *gateway* itd.

Sloj koji se nalazi najdalje od jezgre domene je sloj infrastrukture i obuhvaća razvojne okvire i pokretačke programe. U ovaj sloj spadaju mrežni protokoli, korisnička sučelja, baze podataka, biblioteke i razvojni okviri te ulazno-izlazni uređaji. Glavna poanta je da ovaj sloj obuhvaća tehnološke detalje koji se vrlo često mijenjaju i nije stabilan kao ostali navedeni slojevi, pa je stoga ovaj sloj i puno teže testirati.

Nakon ovog vrlo detaljno teorijskog dijela u kojem su opisane arhitekture usmjerene na domenu poslovanja kao i sam dizajn softvera upravljan domenom, vrijeme je da se prikaže kako ove stvari funkcioniraju u praksi. U nastavku će biti objašnjen programski kod i struktura web aplikacije za pretraživanje jeftinih avionskih letova temeljene na dobrim softverskim praksama i čistoj arhitekturi. Aplikacija je razvijena pomoću ASP.NET Core razvojnog okruženja i Blazor okvira za klijentsko web programiranje o kojima će isto biti više riječi u sljedećem poglavlju.

# 4. Programski kod i struktura projekta

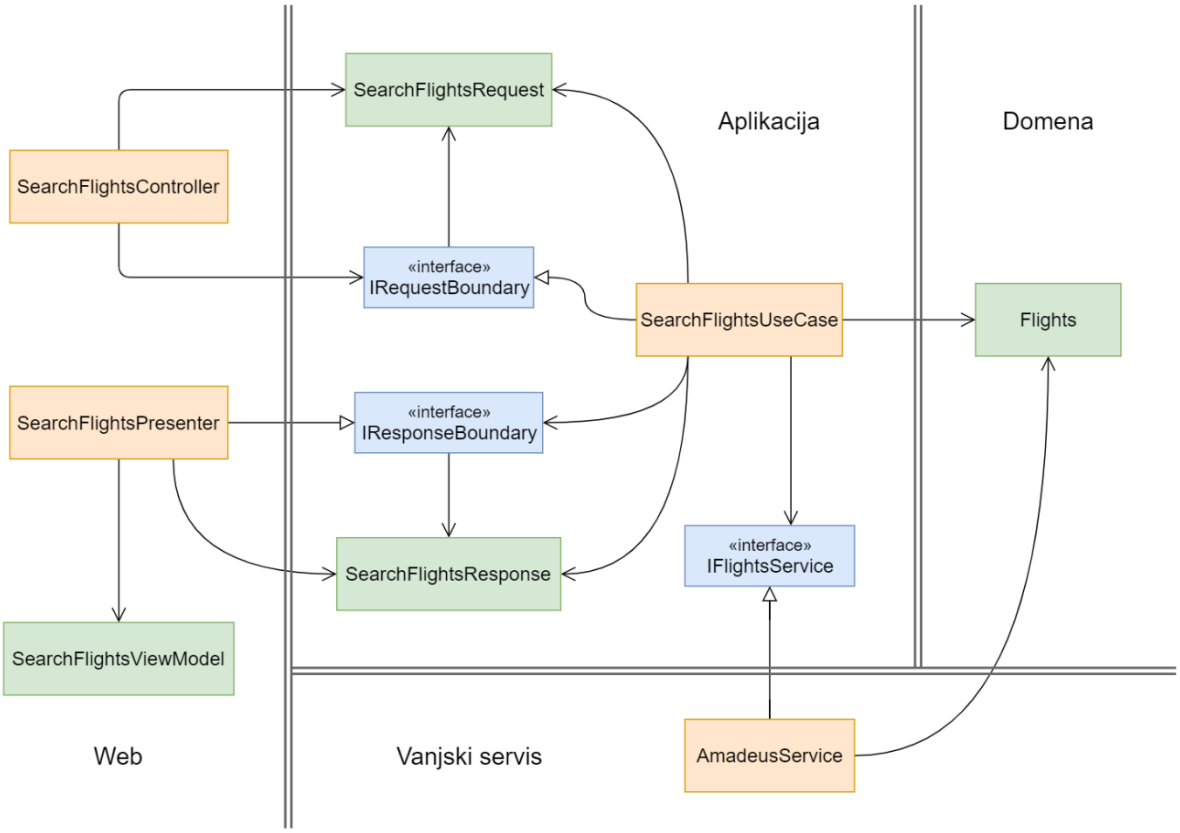
U ovom poglavlju će biti prikazana i objašnjena struktura web aplikacije za pretraživanje jeftinih avionskih letova kao i određeni isječci programskog koda. Projektno rješenje je temeljeno na principu čiste arhitekture i sastoji se od četiri modula koji oslikavaju četiri sloja navedene arhitekture. Poglavlje je koncipirano na način da će svaki modul biti zasebno objašnjen, a paralelno s tim će biti navedene i definirane tehnologije korištene za izvedbu svakog pojedinog modula. Isječci programskog koda prikazani u ovom poglavlju će biti vezani samo uz funkcionalnost pretraživanja avionskih letova, budući da se radi o osnovnoj funkcionalnosti cijelog sustava, a pored toga je i poanta zapravo objasniti kako u praktičnom smislu izgleda jedan tijek kontrole, odnosno korisnička priča za određenu funkcionalnost, bez da se poglavlje nepotrebno optereti suvišnim implementacijskim detaljima. Slika 16 prikazuje tijek kontrole između komponenti sustava za funkcionalnost pretraživanja letova.




Slika 16. Dijagram tijeka kontrole između komponenti aplikacije za funkcionalnost pretraživanja letova (izrada autora)

Korisnik inicira akciju pretraživanja jeftinih avionskih letova koja aktivira upravitelja događajima na strani korisničkog sučelja. On zatim kreira ulaznu poruku koja sadrži parametre po kojima korisnik želi pretražiti letove i prosljeđuje ju slučaju korištenja. Potom slučaj korištenja za pretraživanje letova kontaktira sučelje prema vanjskom Amadeus servisu tražeći listu entiteta letova. Nakon što zaprimi listu letova, šalje ju u obliku izlazne poruke prema

prezenteru. U konačnici prezenter transformira poruku u oblik prikladan za prikaz na korisničkom sučelju, šalje ju prema njemu te korisnik dobiva traženi sadržaj. Dijagram za navedeni tijek kontrole se može i dodatno proširiti na način da se u prikaz uključe slojevi čiste arhitekture, kao što se može vidjeti na slici 17.




Slika 17. Alternativni prikaz tijeka kontrole između komponenti aplikacije koji uključuje slojeve čiste arhitekture (izrada autora)

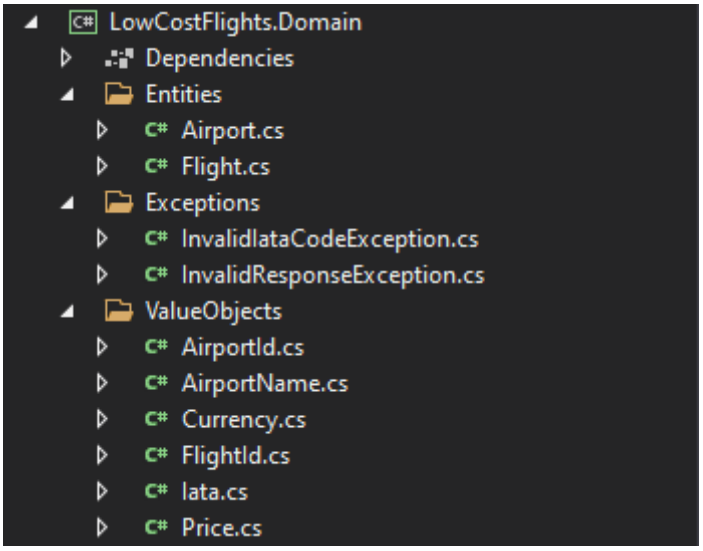
Strukturu projekta čine sljedeći moduli:

- modul domene
- modul aplikacije
- modul infrastrukture
- modul korisničkog sučelja.

U nastavku će detaljnije biti objašnjen svaki od navedenih modula, kao i jedinični testovi korišteni u projektu.

## 4.1. Modul domene

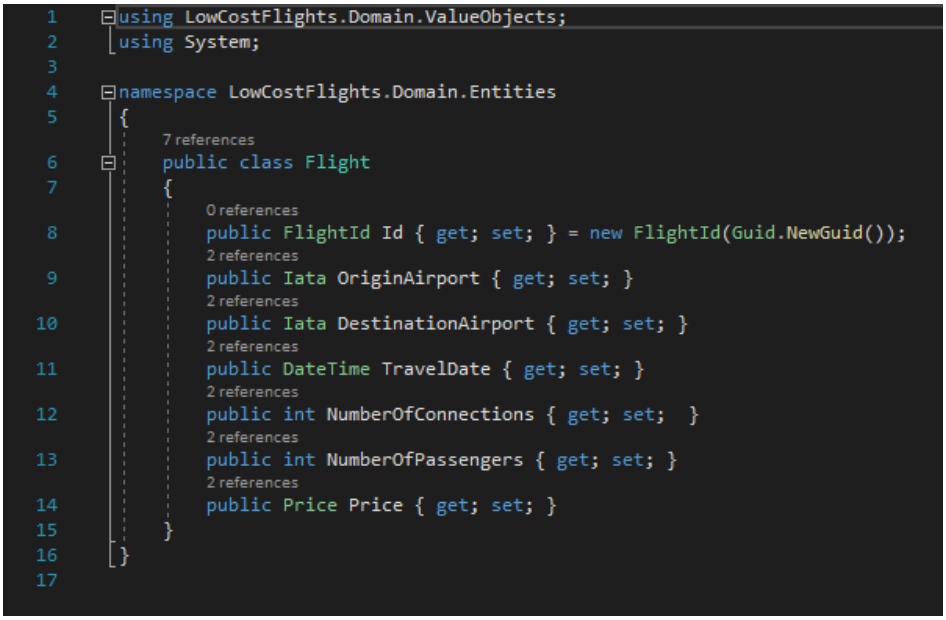
Modul domene obuhvaća kritična poslovna pravila i oslikava domenski sloj čiste arhitekture. Sadržava sve entitete koji igraju ulogu poslovnih objekata aplikacije za pretraživanje jeftinih avionskih letova, kao i prilagođene iznimke te vrijednosne objekte. Tu se nalaze entiteti *Flight* i *Airport*, iznimke *InvalidIataCodeException* i *InvalidResponseException* te vrijednosni objekti *FlightId*, *AirportId*, *AirportName*, *Currency*, *Iata* i *Price*. U slučaju kada bi se ova aplikacija koristila u nekom većem poduzeću, navedeni modul bi se mogao dijeliti između više aplikacija u obliku *Nuget* paketa, pa je stoga iznimno bitno da sadrži elemente koji su karakteristični za cjelokupno poslovanje. Slika 18 prikazuje strukturu direktorija modula domene.




Slika 18. Struktura direktorija modula domene (izrada autora)

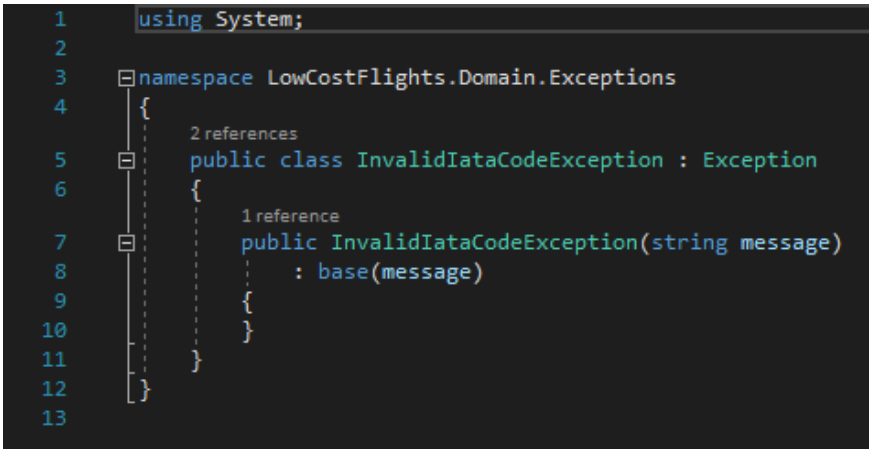
U nastavku će biti prikazani i objašnjeni isječci koda za entitet *Flight*, iznimku *InvalidIataCodeException* i vrijednosni objekt *Iata*. Slika 19 prikazuje programski kod za implementaciju entiteta *Flight*. Kod je dosta jednostavan i zapravo prikazuje klasu *Flight* s nizom svojstava. Ono što treba istaknuti jesu tipovi podataka tih svojstava. Može se vidjeti da se kao tipovi podataka zapravo ne koriste primitivni tipovi, nego vrijednosni objekti. To je jedna od značajnih karakteristika dizajna upravljanog domenom i smatra se generalno vrlo dobrom programerskom praksom prilikom dizajniranja sloja domene, budući da bi entiteti što vjerodostojnije trebali oslikavati poslovne objekte. Pretjerano korištenje primitivnih tipova se smatra *code smell*-om i često se takav pristup naziva primitivnom opsesijom (eng. *primitive obsession*) [12]. Gledajući u kod također treba istaknuti svojstvo *Id*. Takvo svojstvo je prisutno

u bilo kojem entitetu, budući da je već ranije bilo naglašeno kako svaki entitet u poslovnom svijetu ima svoj jedinstveni identitet po kojem se razlikuje od svih ostalih entiteta.




Slika 19. Programski kod za entitet *Flight* (izrada autora)

Na slici 20 je prikazan programski kod za iznimku *InvalidIataCodeException*. Radi se zapravo o klasičnom primjeru implementacije korisnički definirane iznimke. Klasa *InvalidIataCodeException* nasljeđuje klasu *Exception* i u svom konstruktoru prima parametar *message*, nakon čega poziva konstruktor roditeljske klase prosljeđujući mu primljeni parametar. U kontekstu poslovnih pravila, ova iznimka se poziva prilikom definiranja ili korištenja neispravnog formata IATA koda, međunarodnog koda za označavanje aerodroma.




Slika 20. Programski kod za iznimku *InvalidIataCodeException* (izrada autora)

Slika 21 prikazuje programski kod za vrijednosni objekt *Iata*. Ovaj kod je nešto složeniji od prethodna dva i prikazuje zapravo strukturu *Iata* koja implementira sučelje *IEquatable*. Ovo sučelje propisuje generaliziranu metodu *Equals* koja omogućava usporedbu tipova, odnosno klasa po jednakosti. Već je ranije bilo spomenuto kako je jedan od uvjeta vrijednosnih objekata taj da su usporedivi po jednakosti. Struktura također sadrži konstantu koja definira veličinu IATA koda te svojstvo *Code* koje omogućava dohvaćanje samog IATA koda. U implementaciji konstruktora se može vidjeti i primjer korištenja ranije spomenute iznimke koja se aktivira ukoliko proslijeđeni parametar *code* u konstruktoru ne zadovoljava odgovarajuću veličinu IATA koda. Metoda *Equals* strukture *Iata* je nadjačana implementacijom provjere uvjeta jednakosti pri čemu koristi ranije definiranu metodu *Equals* sučelja *IEquatable*, a uz nadjačanu metodu *Equals* se uvijek mora nadjačati i metoda *GetHashCode*, koja vraća *hash* kod strukture kako bi se mogao izvršiti algoritam provjere jednakosti instanci. Zatim su definirani operatori == i != za osiguravanje funkcionalnosti provjere jednakosti i nejednakosti instanci strukture, a nadjačana je i metoda *ToString* kako bi se osigurala mogućnost vraćanja tekstualne reprezentacije strukture.

```
using LowCostFlights.Domain.Exceptions;
using System;

namespace LowCostFlights.Domain.ValueObjects
{
    12 references
    public readonly struct Iata : IEquatable<Iata>
    {
        private const int CodeLength = 3;
        5 references
        public string Code { get; }

        2 references
        public Iata(string code)
        {
            if (code.Length != CodeLength)
            {
                throw new InvalidIataCodeException("IATA code format is not valid!");
            }

            Code = code;
        }

        2 references
        public bool Equals(Iata other)
        {
            return Code == other.Code;
        }

        0 references
        public override bool Equals(object obj)
        {
            return obj is Iata o && Equals(o);
        }

        0 references
        public override int GetHashCode()
        {
            return HashCode.Combine(Code);
        }

        1 reference
        public static bool operator ==(Iata left, Iata right) => left.Equals(right);

        0 references
        public static bool operator !=(Iata left, Iata right) => !(left == right);

        2 references
        public override string ToString()
        {
            return Code;
        }
    }
}
```

Slika 21. Programski kod za vrijednosni objekt *Iata* (izrada autora)

Svi ovi isječci koda kao i cjelokupno projektno rješenje je implementirano u ASP.NET Core razvojnom okruženju, pa će u nastavku biti nešto riječi na tu temu.

### 4.1.1. ASP.NET Core

ASP.NET Core[1] je višeplatformski razvojni okvir visokih performansi i otvorenog koda, a služi za razvoj modernih web aplikacija, aplikacija interneta stvari (eng. *Internet of Things*) kao i poslužiteljskih aplikacija za mobilne uređaje [13]. Nudi brojni niz mogućnosti, kao što su razvoj web korisničkih sučelja pomoću MVC, Razor Pages i Blazor razvojnih okvira, razvoj web poslužiteljskih aplikacija, podrška testiranju, integracija modernih razvojnih okvira i razvojnog tijeka rada, podrška za hosting servisa za pozivanje udaljenih procedura (eng. *Remote Procedure Call*, *RPC*), ugrađen *dependency injection*, jednostavan i modularan cjevovod HTTP zahtjeva (eng. *HTTP Request Pipeline*) visokih performansi, mogućnost hostanja na raznim poslužiteljima poput IIS-a, Kestrel-a, Nginx-a, Apache-a i drugih, podrška za Docker i još mnoge druge mogućnosti [13].

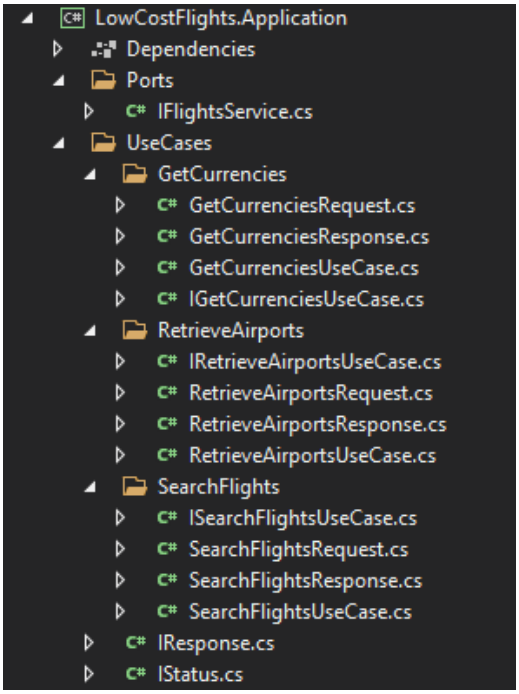
## 4.2. Modul aplikacije

Modul aplikacije oslikava aplikacijski sloj čiste arhitekture. Radi se o iznimno važnom sloju, budući da on sadržava poslovnu logiku aplikacije temeljenu na različitim slučajevima korištenja i funkcionalnim zahtjevima. Budući da ovaj sloj ovisi o sloju domene, modul aplikacije se referencira na domenski modul u strukturi projekta, pa je na taj način omogućen pristup elementima domene iz modula aplikacije. Ovaj modul će dobiti ulazne podatke iz prezentacijskog sloja i na njima će primijeniti pravila poslovne logike koristeći sučelje *IFlightsService* koje predstavlja port prema vanjskom Amadeus servisu. Važno je napomenuti kako ovaj modul uopće nema ideju o kakvom se vanjskom servisu radi, budući da ne smije znati ništa o tehnologiji, zbog čega i koristi apstrakciju (u principu se radi o konceptu portova i adaptera). Konkretna implementacija Amadeus servisa se nalazi u vanjskom sloju i prilikom izvršavanja aplikacije će biti „ubrizgana“ na mjesto apstrakcije putem *dependency injection* uzorka dizajna. Pored spomenutog *IFlightsService* porta, aplikacijski modul obuhvaća *SearchFlights*, *RetrieveAirports* i *GetCurrencies* slučajeve korištenja te još neka pomoćna sučelja. Slika 22 prikazuje strukturu direktorija modula aplikacije.

U nastavku će biti prikazani i objašnjeni isječci koda za slučaj korištenja *SearchFlightsUseCase* te za port *IFlightsService*. Slika 23 prikazuje programski kod za slučaj korištenja *SearchFlightsUseCase*. Klasa *SearchFlightsUseCase* implementira sučelje

[1] https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core

*ISearchFlightsUseCase* koje propisuje metodu *ExecuteAsync* te koristi port *IFlightsService* putem konstruktorskog ubrizgavanja tog porta *dependecy injection* uzorkom dizajna.




Slika 22. Struktura direktorija modula aplikacije (izrada autora)

```
using LowCostFlights.Application.Ports;
using LowCostFlights.Domain.Exceptions;
using System.Threading.Tasks;

namespace LowCostFlights.Application.UseCases.SearchFlights
{
    2 references
    public class SearchFlightsUseCase : ISearchFlightsUseCase
    {
        private readonly IFlightsService _service;

        0 references
        public SearchFlightsUseCase(IFlightsService service)
        {
            _service = service;
        }

        2 references
        public async Task<SearchFlightsResponse> ExecuteAsync(SearchFlightsRequest request)
        {
            SearchFlightsResponse response = await _service.GetFlightsAsync(request);

            if (response.Data != null)
            {
                return response;
            }

            throw new InvalidResponseException(response.StatusMessage);
        }
    }
}
```

Slika 23. Programski kod za slučaj korištenja *SearchFlightsUseCase* (izrada autora)

Metoda *ExecuteAsync* je asinkrona metoda koja kao parametar prima instancu klase *SearchFlightsRequest* te ju prosljeđuje kao argument metodi *GetFlightsAsync* ubrizganog porta i kao rezultat prima odgovor u obliku instance klase *SearchFlightsResponse*. Ukoliko je svojstvo *Data* primljenog odgovora definirano, metoda *ExecuteAsync* će vratiti primljeni odgovor svom pozivatelju, a u suprotnom će aktivirati iznimku *InvalidResponseException* definiranu u modulu domene.

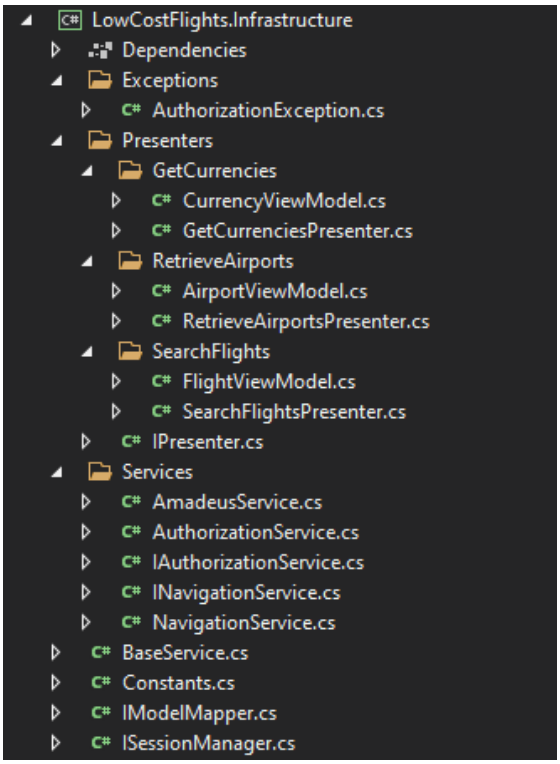
Slika 24 prikazuje programski kod za port *IFlightsService*. Sučelje *IFlightsService* propisuje metode koje implementira Amadeus servis u modulu infrastrukture, a radi se o dvije asinkrone metode *GetFlightsAsync* i *GetAirportsAsync* te jednoj sinkronoj metodi *GetCurrencies*. Sve navedene metode kao ulazni parametar primaju ulazni kriterij za dohvaćanje podataka s Amadeus servisa, a povratni tipovi su im tipovi dohvaćenog odgovora s navedenog servisa.

```
1  using LowCostFlights.Application.UseCases.GetCurrencies;
2  using LowCostFlights.Application.UseCases.RetrieveAirports;
3  using LowCostFlights.Application.UseCases.SearchFlights;
4  using System.Threading.Tasks;
5
6  namespace LowCostFlights.Application.Ports
7  {
       4 references
8      public interface IFlightsService
9      {
           2 references
10         Task<SearchFlightsResponse> GetFlightsAsync(SearchFlightsRequest requestValue);
           2 references
11         Task<RetrieveAirportsResponse> GetAirportsAsync(RetrieveAirportsRequest requestValue);
           2 references
12         GetCurrenciesResponse GetCurrencies(GetCurrenciesRequest requestValue);
13     }
14 }
15
```

Slika 24. Programski kod za port *IFlightsService* (izrada autora)

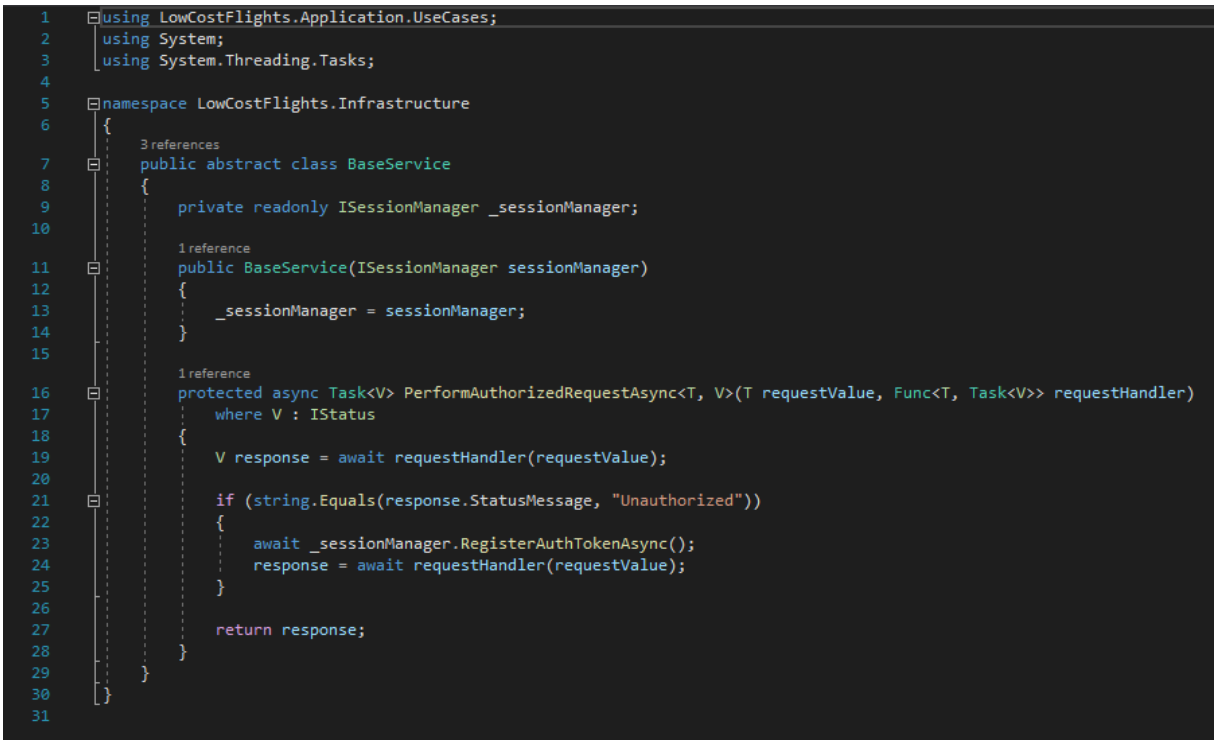
## 4.3. Modul infrastrukture

Modul infrastrukture oslikava infrastrukturni sloj čiste arhitekture i sadrži adaptere, odnosno implementacije portova definiranih u aplikacijskom modulu. Sadrži stvari koje nisu izravno povezane s domenom poslovanja, ali su potrebne kako bi aplikacija funkcionirala, kao što je to u ovom slučaju Amadeus servis. Samim tim, ovaj modul smije koristiti vanjske tehnološke detalje. Ovdje su također definirani i prezenteri koji transformiraju podatke primljene iz aplikacijskog modula i oblik prikladan za prikaz na korisničkom sučelju. Slika 25 prikazuje strukturu direktorija modula infrastrukture.




Slika 25. Struktura direktorija modula infrastrukture (izrada autora)

U nastavku će biti prikazani i objašnjeni isječci koda za apstraktni servis *BaseService*, zatim za servise *AmadeusService* i *AuthorizationService* te za prezenter *SearchFlightsPresenter*. Slika 26 prikazuje programski kod za apstraktni servis *BaseService*. Radi se o relativno složenoj implementaciji, a primarna svrha ovog servisa je pružiti ostalim servisima koji ga nasljeđuju podlogu za poziv autoriziranih zahtjeva prema vanjskom Amadeus API-ju, iz razloga što Amadeus API prihvaća samo zahtjeve autorizirane putem OAuth2 protokola. Glavni razlog tomu je sprječavanje zloupotrebe samog API-ja. *BaseService* koristi sučelje *ISessionManager* putem konstruktorskog *dependency injection*-a. Navedeno sučelje je implementirano u modulu korisničkog sučelja, a propisuje metode za pohranjivanje, registraciju i obnavljanje autorizacijskog tokena. Metoda *PerformAuthorizedRequestAsync* je generička metoda i kao ulazne parametre prima *requestValue* kriterij za dohvaćanje podataka s vanjskog API-ja i funkcijski delegat koji omogućava umetanje različitih implementacijskih logika unutar samog tijela metode. Ovdje se čak razmatrala opcija realizacije navedene situacije korištenjem *template method* uzorka dizajna, međutim situacija je u ovoj fazi još uvijek relativno jednostavna, pa je ta ideja izbačena u korist realizaciji putem funkcijskog delegata. Ukoliko izvršeni delegat vrati odgovor u obliku pogreške autorizacije, registrira se osvježeni autorizacijski token pozivom metode *RegisterAuthTokenAsync* sučelja *ISessionManager* te se ponavlja izvršavanje delegata. U konačnici se pozivatelju šalje odgovor vraćen od delegata.

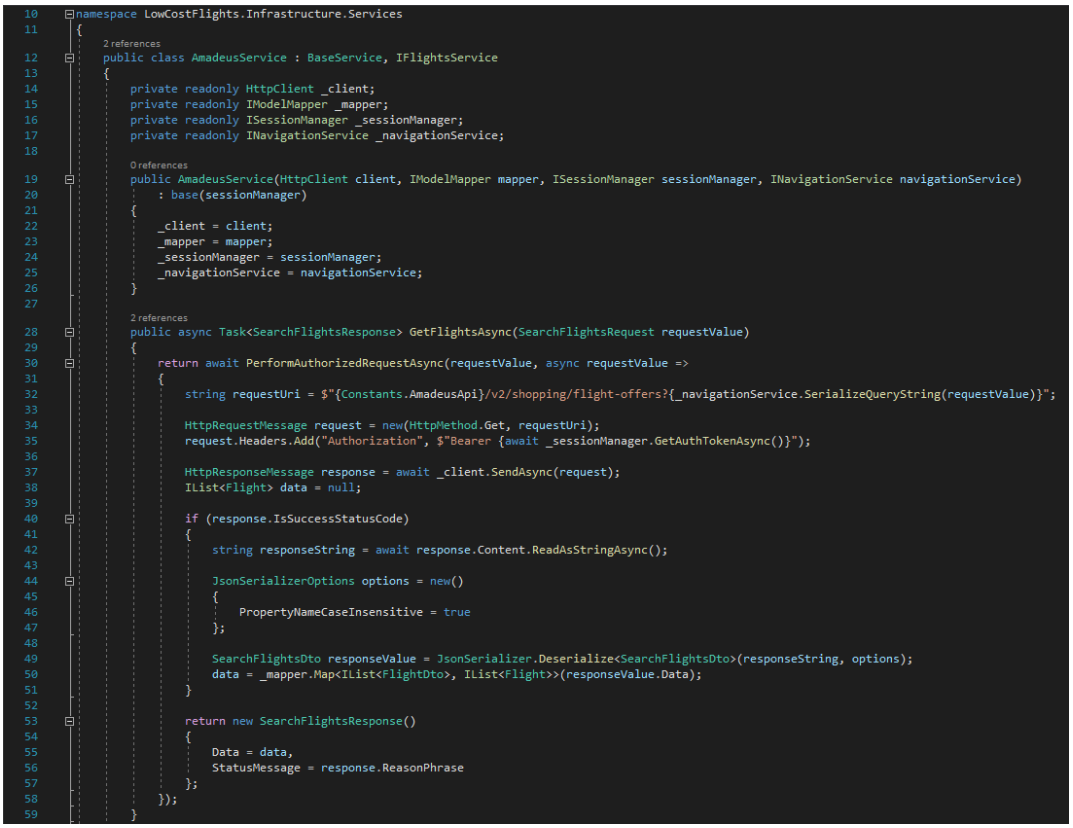


```
using LowCostFlights.Application.UseCases;
using System;
using System.Threading.Tasks;

namespace LowCostFlights.Infrastructure
{
    3 references
    public abstract class BaseService
    {
        private readonly ISessionManager _sessionManager;

        1 reference
        public BaseService(ISessionManager sessionManager)
        {
            _sessionManager = sessionManager;
        }

        1 reference
        protected async Task<V> PerformAuthorizedRequestAsync<T, V>(T requestValue, Func<T, Task<V>> requestHandler)
            where V : IStatus
        {
            V response = await requestHandler(requestValue);

            if (string.Equals(response.StatusMessage, "Unauthorized"))
            {
                await _sessionManager.RegisterAuthTokenAsync();
                response = await requestHandler(requestValue);
            }

            return response;
        }
    }
}
```

Slika 26. Programski kod za apstraktni servis *BaseService* (izrada autora)

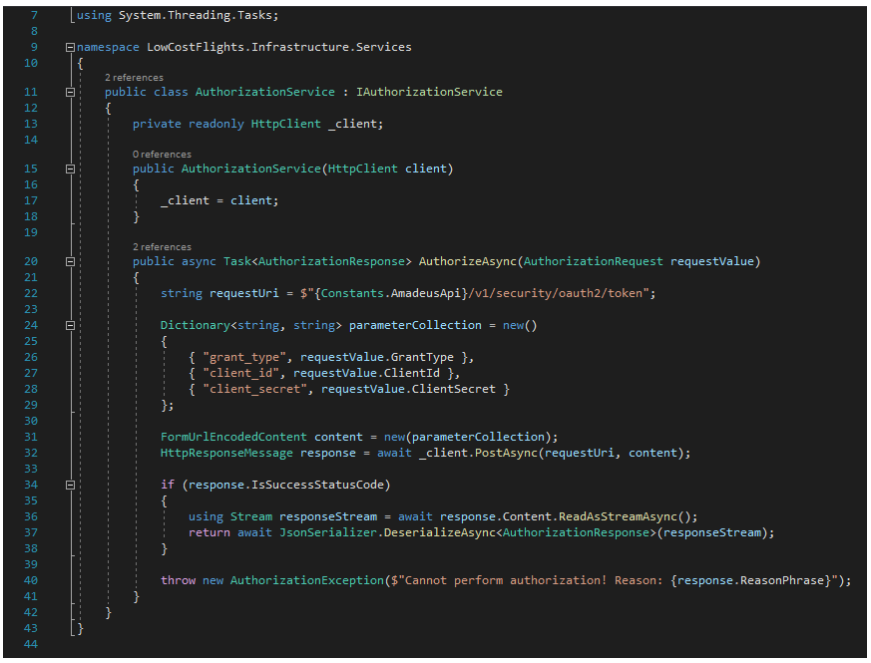
Slika 27 prikazuje programski kod za servis *AmadeusService*. Ovaj servis nasljeđuje klasu *BaseService* i implementira sučelje *IFlightsService*. Putem konstruktorskog *dependency injection*-a se ubacuje instanca klase *HttpClient* zaslužne za slanje i primanje HTTP zahtjeva te sučelja *IModelMapper*, *ISessionManager* i *INavigationService*. Također se poziva konstruktor roditeljske apstraktne klase *BaseService* kojoj se prosljeđuje sučelje *ISessionManager*. Metoda *GetFlightsAsync* je zaslužna za dohvaćanje liste letova s Amadeus API-ja. Ona poziva metodu *PerformAuthorizedRequest BaseService* klase, šalje joj *requestValue* kriterij pretraživanja letova i kao argument za funkcijski delegat definira anonimnu metodu koja zapravo sadrži logiku za dohvaćanje letova. Unutar te anonimne funkcije se definira URI zahtjeva s parametrima za pretraživanje te se formira *HttpRequestMessage* poruka za izvršavanje HTTP zahtjeva putem GET metode. U *header* se dodaje autorizacijski token, šalje se zahtjev i zaprima poruka u obliku instance *HttpResponseMessage* klase. Ukoliko je zahtjev bio uspješan, zaprimljeni odgovor će se deserijalizirati u instancu *SearchFlightsDto* klase i putem sučelja *IModelMapper* mapirati u oblik prikladan za vraćanje pozivatelju. U odgovor se također dodaje i poruka koja definira status izvršenog zahtjeva.

Slika 28 prikazuje programski kod za servis *AuthorizationService*. Ovaj servis implementira sučelje *IAuthorizationService* i putem konstruktorskog *dependency injection*-a mu se ubacuje instanca klase *HttpClient*.



```
namespace LowCostFlights.Infrastructure.Services
{
    2 references
    public class AmadeusService : BaseService, IFlightsService
    {
        private readonly HttpClient _client;
        private readonly IModelMapper _mapper;
        private readonly ISessionManager _sessionManager;
        private readonly INavigationService _navigationService;

        0 references
        public AmadeusService(HttpClient client, IModelMapper mapper, ISessionManager sessionManager, INavigationService navigationService)
            : base(sessionManager)
        {
            _client = client;
            _mapper = mapper;
            _sessionManager = sessionManager;
            _navigationService = navigationService;
        }

        2 references
        public async Task<SearchFlightsResponse> GetFlightsAsync(SearchFlightsRequest requestValue)
        {
            return await PerformAuthorizedRequestAsync(requestValue, async requestValue =>
            {
                string requestUri = $"{Constants.AmadeusApi}/v2/shopping/flight-offers?{_navigationService.SerializeQueryString(requestValue)}";

                HttpRequestMessage request = new(HttpMethod.Get, requestUri);
                request.Headers.Add("Authorization", $"Bearer {await _sessionManager.GetAuthTokenAsync()}");

                HttpResponseMessage response = await _client.SendAsync(request);
                IList<Flight> data = null;

                if (response.IsSuccessStatusCode)
                {
                    string responseString = await response.Content.ReadAsStringAsync();

                    JsonSerializerOptions options = new()
                    {
                        PropertyNameCaseInsensitive = true
                    };

                    SearchFlightsDto responseValue = JsonSerializer.Deserialize<SearchFlightsDto>(responseString, options);
                    data = _mapper.Map<IList<FlightDto>, IList<Flight>>(responseValue.Data);
                }

                return new SearchFlightsResponse()
                {
                    Data = data,
                    StatusMessage = response.ReasonPhrase
                };
            });
        }
```

Slika 27. Programski kod za servis *AmadeusService* (izrada autora)



```
using System.Threading.Tasks;

namespace LowCostFlights.Infrastructure.Services
{
    2 references
    public class AuthorizationService : IAuthorizationService
    {
        private readonly HttpClient _client;

        0 references
        public AuthorizationService(HttpClient client)
        {
            _client = client;
        }

        2 references
        public async Task<AuthorizationResponse> AuthorizeAsync(AuthorizationRequest requestValue)
        {
            string requestUri = $"{Constants.AmadeusApi}/v1/security/oauth2/token";

            Dictionary<string, string> parameterCollection = new()
            {
                { "grant_type", requestValue.GrantType },
                { "client_id", requestValue.ClientId },
                { "client_secret", requestValue.ClientSecret }
            };

            FormUrlEncodedContent content = new(parameterCollection);
            HttpResponseMessage response = await _client.PostAsync(requestUri, content);

            if (response.IsSuccessStatusCode)
            {
                using Stream responseStream = await response.Content.ReadAsStreamAsync();
                return await JsonSerializer.DeserializeAsync<AuthorizationResponse>(responseStream);
            }

            throw new AuthorizationException($"Cannot perform authorization! Reason: {response.ReasonPhrase}");
        }
    }
}
```

Slika 28. Programski kod za servis *AuthorizationService* (izrada autora)

Metoda *AuthorizeAsync* kao ulazni parametar prima *requestValue* autorizacijski zahtjev, nakon čega formira URI za dohvaćanje OAuth2 autorizacijskog tokena. Formira tijelo

HTTP POST poruke dodavajući parametre *grant_type*, *client_id* i *client_secret* i kao rezultat od Amadeus API-ja prima token. Ukoliko je autorizacijski zahtjev bio uspješan, odgovor primljen od Amadeus API-ja će se deserijalizirati u instancu objekta *AuthorizationResponse* i vratiti pozivatelju. U protivnom će se aktivirati iznimka *AuthorizationException*.

Slika 29 prikazuje programski kod za prezenter *SearchFlightsPresenter*. Klasa *SearchFlightsPresenter* implementira generičko sučelje *IPresenter* koje propisuje svojstvo *ViewModel* i metodu *Populate*. Putem konstruktorskog *dependency injection*-a mu se ubacuje sučelje *IModelMapper* i koristi se u metodi *Populate* za mapiranje svojstva *Data* ulazne instance tipa *SearchFlightsResponse* u odgovarajući oblik koji se pohranjuje u svojstvo *ViewModel*.

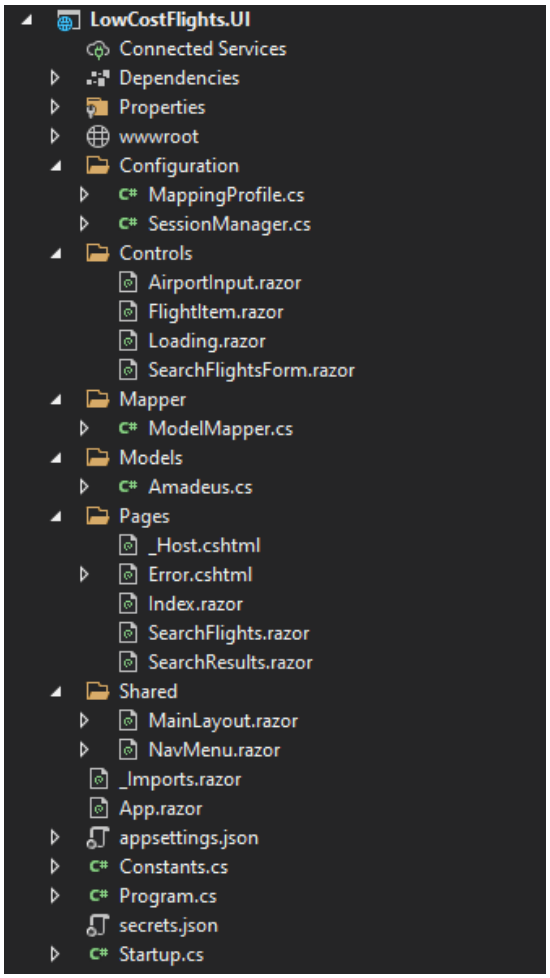
```
1   using LowCostFlights.Application.UseCases.SearchFlights;
2   using LowCostFlights.Domain.Entities;
3   using System.Collections.Generic;
4
5   namespace LowCostFlights.Infrastructure.Presenters.SearchFlights
6   {
        2 references
7       public class SearchFlightsPresenter : IPresenter<SearchFlightsResponse, IList<FlightViewModel>>
8       {
9           private readonly IModelMapper _mapper;
            3 references
10          public IList<FlightViewModel> ViewModel { get; set; }
11
            0 references
12          public SearchFlightsPresenter(IModelMapper mapper)
13          {
14              _mapper = mapper;
15          }
16
            2 references
17          public void Populate(SearchFlightsResponse response)
18          {
19              ViewModel = _mapper.Map<IList<Flight>, IList<FlightViewModel>>(response.Data);
20          }
21      }
22  }
23
```

Slika 29. Programski kod za prezenter *SearchFlightsPresenter* (izrada autora)

## 4.4. Modul korisničkog sučelja

Modul korisničkog sučelja oslikava sloj korisničkog sučelja čiste arhitekture. Radi se o najudaljenijem sloju koji sadrži najviše tehnoloških detalja i najviše se mijenja. Ovaj modul predstavlja ulaznu točku s aspekta korisnika za komunikaciju sa cijelim sustavom. Također, zaslužan je za umrežavanje svih ostalih modula te registriranje svih servisa u spremnik za inverziju kontrole (eng. *inversion of control*, *IoC*) kako bi se mogli ubacivati putem *dependency*

*injection* uzorka dizajna. Važno je napomenuti da u složenijim poslovnim sustavima može postojati i više prezentacijskih modula koji kontaktiraju aplikacijski modul za ulaz i izlaz podataka. Ovaj modul ne sadrži nikakvu poslovnu logiku, nego je zadužen za tehnološke detalje. Slika 30 prikazuje strukturu direktorija modula korisničkog sučelja.




Slika 30. Struktura direktorija modula korisničkog sučelja (izrada autora)

U nastavku će biti prikazani i objašnjeni isječci koda klase *SessionManager*, Blazor komponente *SearchResults* te metoda *ConfigureServices* konfiguracijske klase *Startup* za registraciju servisa u spremnik inverzije kontrole. Slika 31 prikazuje programski kod klase *SessionManager*. Ova klasa je specifična po tome što u neku ruku igra ulogu u procesu autorizacije aplikacije prema Amadeus API-ju. Implementira sučelje *ISessionManager* i putem konstruktorskog *dependency injection*-a joj se ubacuju sučelja *ISessionStorageService*, *IAuthorizationService* i *IOptions*. Ovdje treba posebnu pozornost obratiti na sučelje *IOptions* koje se koristi za implementaciju *options* uzorka dizajna specifičnog za ASP.NET Core

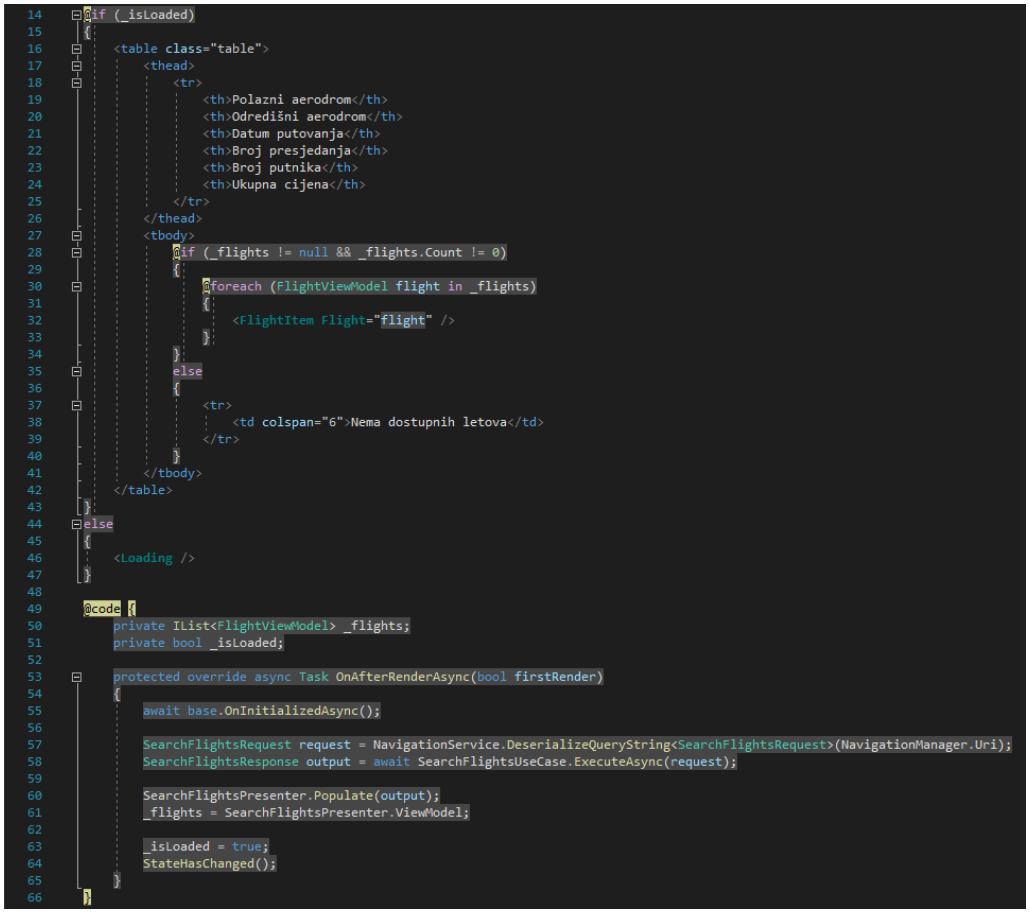
razvojno okruženje. Radi se o uzorku dizajna koji koristi klase kako bi osigurao strogo tipizirani pristup grupama povezanih postavki. Nadalje, ukoliko su konfiguracijske postavke izolirane u zasebne klase, aplikacija se podvrgava *interface segregation* principu dizajna, kao i razdvajanju odgovornosti. U slučaju klase *SessionManager*, koristi se klasa *Amadeus* za pristup postavkama definiranim u datoteci *secrets.json*. Radi se o vrlo osjetljivim podacima koji uključuju API ključ i API tajnu, pa se zbog toga nikako ne smiju nalaziti u programskom kodu. To je zapravo i razlog zašto su ti podaci definirani unutar *secrets.json* datoteke s postavkama. Kako bi se ostvarila dodatna razina sigurnosti, ovi podaci nikako ne smiju biti javno objavljeni, npr. u javnom GitHub repozitoriju. Čak se ne bi smjeli pojaviti u svom izvornom obliku niti u privatnom repozitoriju. Pitanje sigurnosti u ovoj situaciji se može riješiti određenim alatima, npr. osjetljivi podaci se mogu pohraniti u *Azure KeyVault* ili se može koristiti besplatni alat *gitcrypt* za kriptiranje osjetljivih podataka prilikom njihovog postavljanja u GitHub repozitorij. Metoda *GetAuthTokenAsync* klase *SessionManager* služi za dohvaćanje autorizacijskog tokena iz trenutne sesije, dok metoda *RegisterAuthTokenAsync* dohvaća autorizacijski token od Amadeus API-ja i pohranjuje ga u trenutnu sesiju.

```
9   namespace LowCostFlights.UI.Configuration
10  {
        2 references
11      public class SessionManager : ISessionManager
12      {
13          private readonly ISessionStorageService _sessionStorage;
14          private readonly IAuthorizationService _authorization;
15          private readonly Amadeus _amadeus;
16
            0 references
17          public SessionManager(ISessionStorageService sessionStorage, IAuthorizationService authorization, IOptions<Amadeus> options)
18          {
19              _sessionStorage = sessionStorage;
20              _authorization = authorization;
21              _amadeus = options.Value;
22          }
23
            2 references
24          public async Task<string> GetAuthTokenAsync()
25          {
26              bool containsKey = await _sessionStorage.ContainKeyAsync(Constants.TokenKey);
27              return containsKey ? await _sessionStorage.GetItemAsync<string>(Constants.TokenKey) : "";
28          }
29
            2 references
30          public async Task RegisterAuthTokenAsync()
31          {
32              AuthorizationRequest request = new()
33              {
34                  GrantType = "client_credentials",
35                  ClientId = _amadeus.ApiKey,
36                  ClientSecret = _amadeus.ApiSecret
37              };
38
39              AuthorizationResponse response = await _authorization.AuthorizeAsync(request);
40
41              await _sessionStorage.SetItemAsync(Constants.TokenKey, response.AccessToken);
42          }
43      }
44  }
```

Slika 31. Programski kod za klasu *SessionManager* (izrada autora)

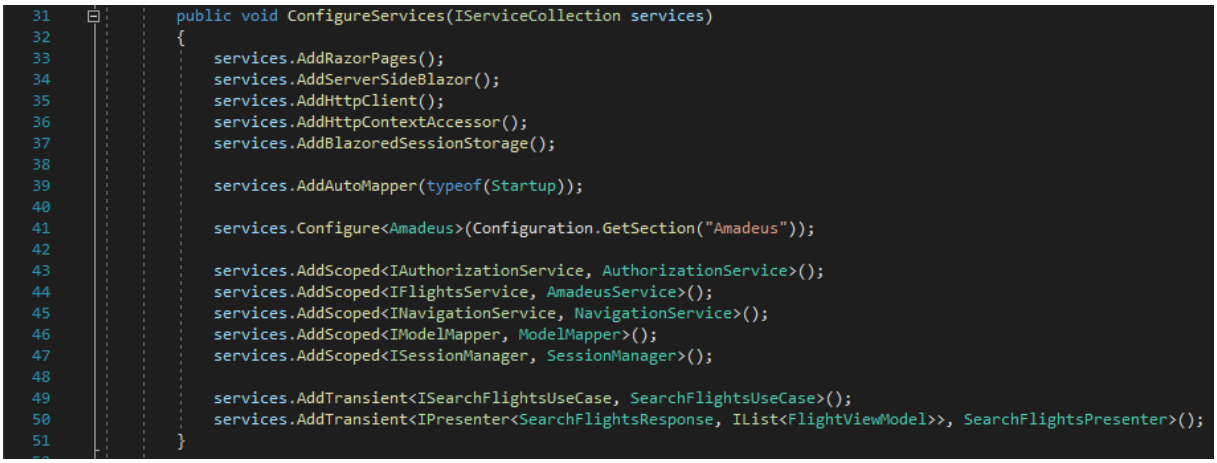
Slika 32 prikazuje programski kod Blazor komponente *SearchResults*. Blazor komponente se inače mogu podijeliti na tri segmenta: prezentacija, stanje i akcija. Prezentacija

se realizira putem *Razor* sintakse koja kombinira HTML jezik oznaka i C# programski jezik unutar istog koda. Stanje je predstavljeno atributima i svojstvima klase, a akcija putem metoda. U *SearchResults* komponenti, prezentacijski segment provjerava da li je komponenta učitana i, u slučaju potvrdnog odgovora, prikazuje tablicu s listom dohvaćenih avionskih letova. U ovom isječku koda se može vidjeti zapravo prednost *Razor* sintakse, gdje se usred HTML koda koriste *if* i *foreach* blokovi C# programskog koda za manipulaciju kolekcijom podataka. Svaki pojedini let se prikazuje putem *FlightItem* komponente djeteta. Segment stanja sadrži listu instanci *FlightViewModel* klase i atribut za provjeru stanja učitanosti komponente. Naposlijetku, segment akcije nakon renderiranja komponente deserijalizira parametre GET zahtjeva koji sadrže kriterije pretraživanja u instancu klase *SearchFlightsRequest* te ju prosljeđuje kao argument slučaju korištenja *SearchFlightsUseCase* za pretraživanje letova od kojeg u konačnici prima odgovor u obliku instance klase *SearchFlightsResponse*. Primljeni odgovor prosljeđuje kao argument metodi *Populate* prezentera *SearchFlightsPresenter* i na samom kraju koristi njegovo svojstvo *ViewModel* za popunjavanje atributa *_flights* koji se koristi u segmentu prezentacije za prikaz letova. Poziv metode *StateHasChanged* osigurava da se nakon primljenog sadržaja komponenta ponovno renderira.




Slika 32. Programski kod za Blazor komponentu *SearchFlights* (izrada autora)

Slika 33 prikazuje programski kod metode *ConfigureServices* konfiguracijske klase *Startup*. Ova metoda je zaslužna za registriranje svih servisa u spremnik za inverziju kontrole i tako zapravo umrežava sve ostale module. U isječku koda se može vidjeti kako se prvo registriraju određeni tehnički detalji, a potom i ostali servisi definirani u projektu. Ovdje treba istaknuti kako servisi mogu imati različiti životni vijek, pa se u skladu s tim koriste i različite metode za njihovo registriranje. Metoda *AddTransient* u sklopu *dependency injection*-a svaki put osigurava novu instancu servisa, *AddScoped* osigurava istu instancu unutar HTTP zahtjeva, a nove instance u različitim zahtjevima, dok metoda *AddSingleton* svaki put osigurava istu instancu servisa bez obzira na HTTP zahtjev.




Slika 33. Programski kod metode *ConfigureServices* konfiguracijske klase *Startup* (izrada autora)

### 4.4.1. Blazor

Blazor[2] je Microsoftov razvojni okvir koji služi za programiranje klijentskih web aplikacija [14]. Podržava dva modela hostinga:

- Blazor poslužitelj koji se izvršava na poslužiteljskoj strani i putem SignalR alata održava konstantnu vezu s klijentom u realnom vremenu
- Blazor WebAssembly koji se izvršava na klijentskoj strani u modernim web preglednicima putem WebAssembly otvorenog standarda za izvršavanje aplikacija napisanih u nekoliko različitih programskih jezika.

[2] https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

Blazor omogućava izgradnju zajedničkih komponenti i interaktivnih web sučelja korištenjem C# programskog jezika umjesto JavaScript-a. U travnju 2019. Microsoft je objavio da Blazor više nije u eksperimentalnoj fazi, nego se može koristiti u razvoju [14].

## 4.5. Jedinično testiranje

Kvaliteta softvera nije samo ograničena na kvalitetu koda i njegovu prilagodljivost promjenama. Testiranje koda također igra veliku ulogu u osiguravanju kvalitete softvera. Jedinično testiranje je disciplina pisanja koda koji testira drugi programski kod. Prilikom izvođenja svakog jediničnog testa izvještava se o uspjehu ili neuspjehu toga testa, često u obliku zelenih ili crvenih vizualnih pokazatelja. Ako svi jedinični testovi prolaze testiranje, produkcijski kod se smatra funkcionalnim, međutim ukoliko samo jedan test padne, smatra se da produkcijski kod ne zadovoljava postavljene zahtjeve. Poželjno je jedinične testove pisati što ranije u procesu razvoja, čak i prije pisanja produkcijskog koda. Takav pristup razvoju programskih proizvoda se naziva razvoj upravljan testovima (eng. *test driven development*, *TDD*). Proces pisanja jediničnih testova i paralelno poboljšavanje programskog koda osigurava konstantno povećavanje kvalitete koda i olakšava implementaciju novih funkcionalnosti [15].

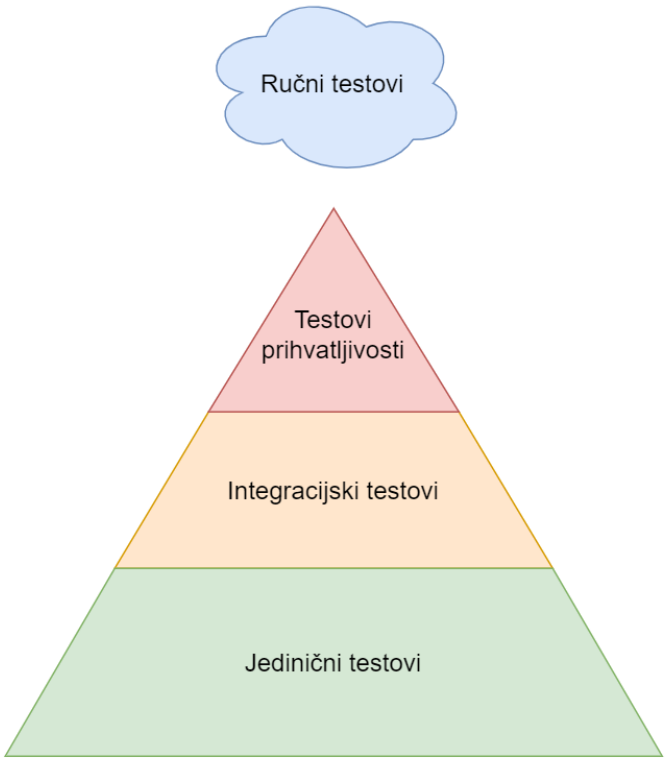
Svaki jedinični test se sastoji od tri ključna dijela:

- *arrange* – raspoređivanje preduvjeta testa
- *act* – izvođenje dijela koji se ispituje testom
- *assert* – tvrdnja da se dogodilo očekivano ponašanje.

Ova tri dijela čine uzorak **rasporedi, djeluj, potvrdi** (eng. *arrange, act, assert*, *AAA*) i svaki jedinični test bi ga trebao dosljedno pratiti [15].

Jedinično testiranje nije jedina vrsta testiranja koje osigurava kvalitetu. Jedinični testovi provjeravaju samo detalje najniže razine programskog koda. Piramida testiranja predstavlja popularan dijagram koji prikazuje razlike između testova na različitim razinama. Taj dijagram odgovara na pitanje koliko bi testova trebalo biti napisano po pojedinoj razini. Slika 34 prikazuje piramidu testova. Iznad najviše razine piramide se nalazi oblak neizvjesnosti i predstavlja ručne testove sustava. Takvi testovi se mogu izvesti u obliku pisanih kontrolnih popisa koji zahtijevaju interakciju stvarnog korisnika sa sustavom radi provjere ispravnosti sustava. Ručnih testova bi trebalo biti malo, jer prilikom svakog izvođenja zahtijevaju ručnu intervenciju, što je skupo i oduzima puno vremena. Ispod ručnih testova se nalaze testovi prihvatljivosti. Ovakvi testovi se mogu implementirati u paketu automatiziranog korisničkog sučelja kao što je npr. *Selenium*. Njihova namjena je repliciranje korisničkih interakcija i usklađivanje s očekivanjima. Trebalo bi biti samo nekoliko takvih testova, jer su veliki i teško ih je održavati.

Kako bi testovi prihvatljivosti bili relevantni, cijeli sustav bi trebao biti raspoređen u testno okruženje koje blisko imitira živo okruženje. Srednji sloj piramide sadrži integracijske testove na razini komponenti. Ovi testovi se mogu realizirati u obliku API testova u kojima se servis tretira kao crna kutija. Za različite unose podataka testovi potvrđuju da se vraćaju točni odgovori. Ovakvih testova najčešće ima više nego testova prihvatljivosti, ali ipak ne u tolikoj količini da postanu zanemareni i fragilni. Najniži sloj piramide je ujedno i najveći, a obuhvaća jedinične testove koji su ranije objašnjeni.




Slika 34. Piramida testova (prema: [15])

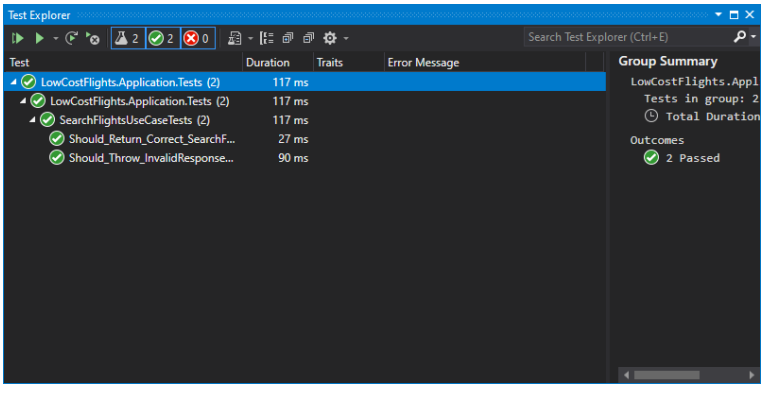
U nastavku slijedi prikaz jediničnih testova koji testiraju ispravnost funkcioniranja slučaja korištenja *SearchFlightsUseCase* definiranog u aplikacijskom modulu aplikacije za pretraživanje jeftinih avionskih letova. Testovi su napisani u xUnit.net alatu za jedinično testiranje. Slika 35 prikazuje jedinični test koji provjerava ispravnost metode *ExecuteAsync* u slučaju dobrog povratnog odgovora, a slika 36 u slučaju pogrešnog povratnog odgovora metode za dohvaćanje letova sučelja *IFlightsService*. Na slici 37 su prikazani rezultati izvođenja testova.

```
[Fact]
| 0 references
public async Task Should_Return_Correct_SearchFlightsResponse_Data()
{
    // Arrange
    Mock<IFlightsService> mock = new();

    mock.Setup(flightsService => flightsService.GetFlightsAsync(It.IsAny<SearchFlightsRequest>()))
        .ReturnsAsync(new SearchFlightsResponse()
        {
            Data = new List<Flight>()
        });

    SearchFlightsUseCase useCase = new (mock.Object);
    SearchFlightsResponse expectedResponse = new()
    {
        Data = new List<Flight>()
    };

    //Act
    SearchFlightsResponse actualResponse = await useCase.ExecuteAsync(It.IsAny<SearchFlightsRequest>());

    //Assert
    actualResponse.Should().BeEquivalentTo(expectedResponse);
}
```

Slika 35. Jedinični test za provjeravanje dobrog povratnog odgovora metode *ExecuteAsync* (izrada autora)

```
[Fact]
| 0 references
public async Task Should_Throw_InvalidResponseException_After_Receiving_Invalid_Data()
{
    //Arrange
    Mock<IFlightsService> mock = new();

    mock.Setup(flightsService => flightsService.GetFlightsAsync(It.IsAny<SearchFlightsRequest>()))
        .ReturnsAsync(new SearchFlightsResponse()
        {
            StatusMessage = "Error test message"
        });

    SearchFlightsUseCase useCase = new(mock.Object);

    //Act
    Func<Task> action = () => useCase.ExecuteAsync(It.IsAny<SearchFlightsRequest>());

    //Assert
    await action.Should().ThrowAsync<InvalidResponseException>()
        .WithMessage("Error test message");
}
```

Slika 36. Jedinični test za provjeravanje pogrešnog povratnog odgovora metode *ExecuteAsync* (izrada autora)




Slika 37. Rezultati izvođenja jediničnih testova (izrada autora)

# 5. Kontinuirana integracija i isporuka

Nakon razvoja aplikacije za pretraživanje jeftinih avionskih letova temeljene na čistoj arhitekturi i dizajnu upravljanom poslovnom domenom te pokrivanja projektnog rješenja testovima, slijedi proces kontinuirane integracije i isporuke aplikacije. Za te svrhe su korišteni alati Jenkins i Octopus Deploy. Slijedi kratak opis alata i prikaz njihovog korištenja.

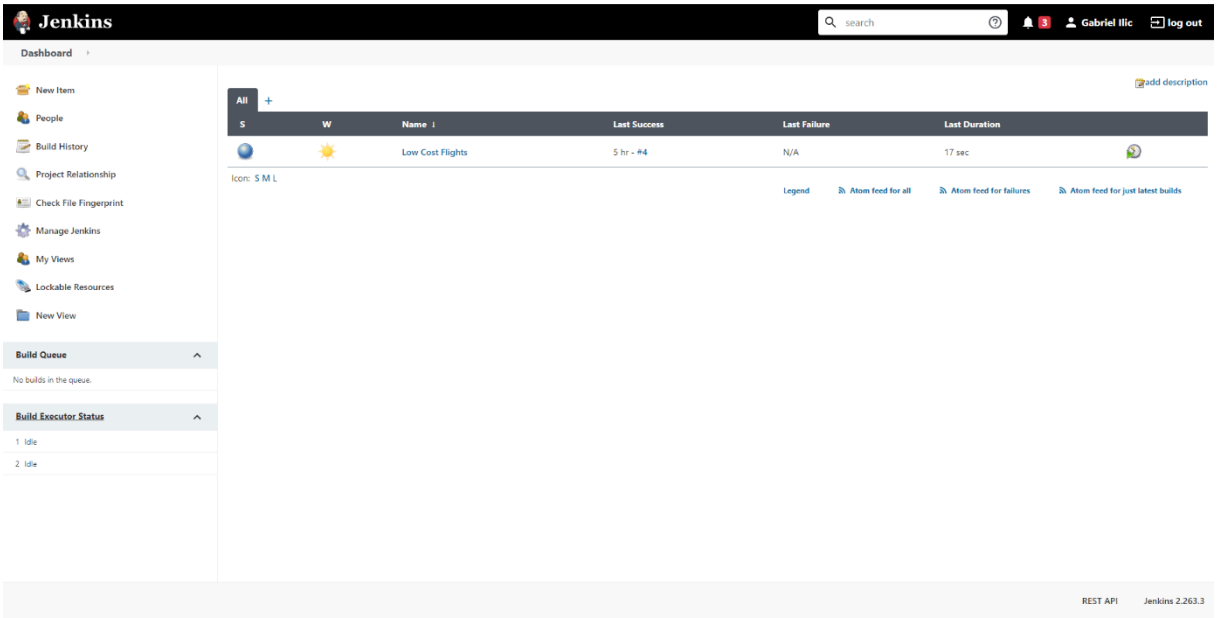
Jenkins[3] je poslužitelj za automatizaciju [16]. Pomaže organizacijama ubrzati proces razvoja softvera tako što ga automatizira. Kontrolira i upravlja procesima isporuke softvera tijekom cijelog njihovog životnog ciklusa, uključujući izgradnju, dokumentiranje, testiranje, pakiranje, izvođenje, implementaciju, statičku analizu koda i još mnogo toga. Jenkins također nudi mogućnost motrenja bilo kakvih promjena koda na GitHub-u, BitBucket-u ili GitLab-u i može se postaviti da u takvim situacijama automatski pokrene proces izgradnje softvera. Podržava i tehnologiju kontejnera kao što su Docker i Kubernetes, a u izgradnji cjevovoda isporuke (eng. *delivery pipeline*) se mogu definirati koraci koji će pokrenuti izvršavanje jediničnih testova [16]. Jenkins se također može povezati i s alatom SonarQube[4] koji služi za mjerenje kvalitete koda, pa se također može kao korak cjevovoda isporuke postaviti korak koji će odrediti da li softverski kod prolazi definiranu točku kvalitete. Za dodatnu sigurnost da kvalitetan kod ulazi u proces isporuke, može se i u ranijem procesu zabraniti slanje izmijenjenih inkremenata koda na GitHub repozitorij ukoliko ne prolaze kriterije kvalitete ili ne zadovoljavaju jedinične testove. Jedan od popularnih alata s kojim se to može postići je alat Husky[5]. Slika 38 prikazuje izgled sučelja alata Jenkins. Cjevovod isporuke za aplikaciju za pretraživanje jeftinih avionskih letova je sljedeći:

```
dotnet restore

dotnet clean

dotnet build --configuration Release

dotnet publish --configuration Release

octo.exe pack -id LowCostFlights -version 1.0.%BUILD_NUMBER% -include
bin\Release\net5.0\publish\

octo.exe push --server localhost:8087 --package
LowCostFlights.1.0.%BUILD_NUMBER%.nupkg --apiKey %OctopusAPIKey%
```

[3] https://www.jenkins.io/
[4] https://www.sonarqube.org/
[5] https://github.com/typicode/husky

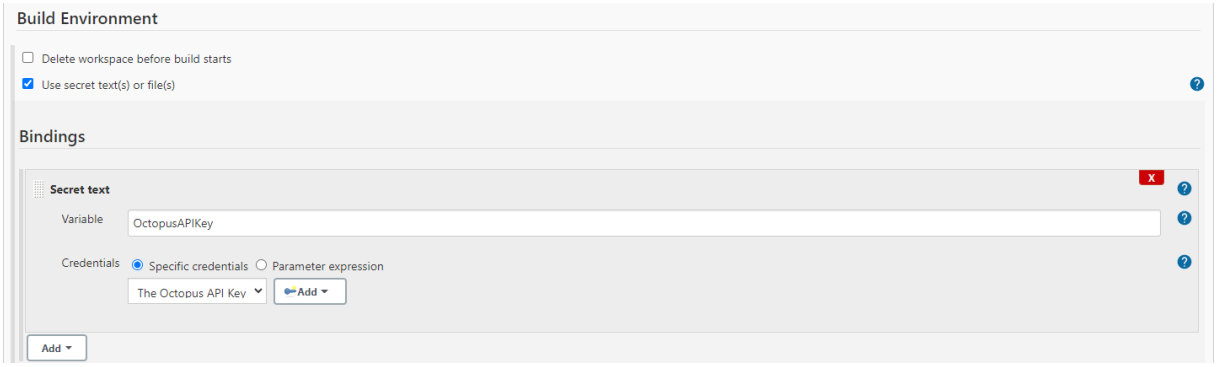
Zadnje dvije naredbe se odnose na integraciju s alatom Octopus Deploy, pri čemu se koristi API ključ za autorizaciju prilikom komunikacije s Octopus-om. On se definira unutar postavki okoline izgradnje, kao što prikazuje slika 39.

Octopus Deploy[6] je automatizirani alat za implementaciju koji se može integrirati s većinom procesa izgradnje koda za implementaciju i konfiguraciju aplikacija [17]. Na taj način DevOps tim postiže maksimalnu učinkovitost izbjegavajući repetitivan ručni rad i česte promjene u konfiguraciji aplikacije prebacujući ju iz jedne okoline u drugu. Octopus Deploy se sastoji od dva dijela. Prvi dio je Octopus poslužitelj na koji se dodaje i konfigurira aplikacija ili servis za implementaciju. Drugi dio je Octopus Tentacles koji se instalira na skupu uređaja na kojima bi se trebale postaviti aplikacije konfigurirane na Octopus poslužitelju. Oba navedena dijela se instaliraju u obliku servisa [17]. Na slici 40 se može vidjeti projektna grupa u Octopus Deploy-u kojoj pripada aplikacija za pretraživanje jeftinih avionskih letova. Definirane su tri okoline na koje se može implementirati aplikacija: okolina za razvoj, testiranje i produkciju. Slika 41 prikazuje kronološki popis isporuka aplikacije po verzijama na pojedine okoline.

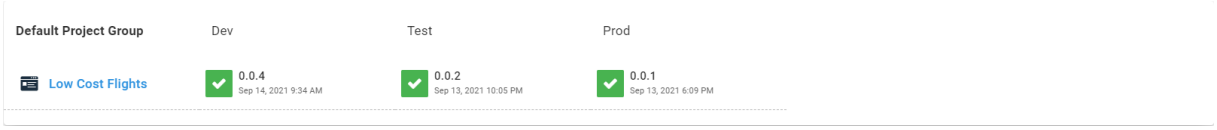



Slika 38. Sučelje alata Jenkins (izrada autora)

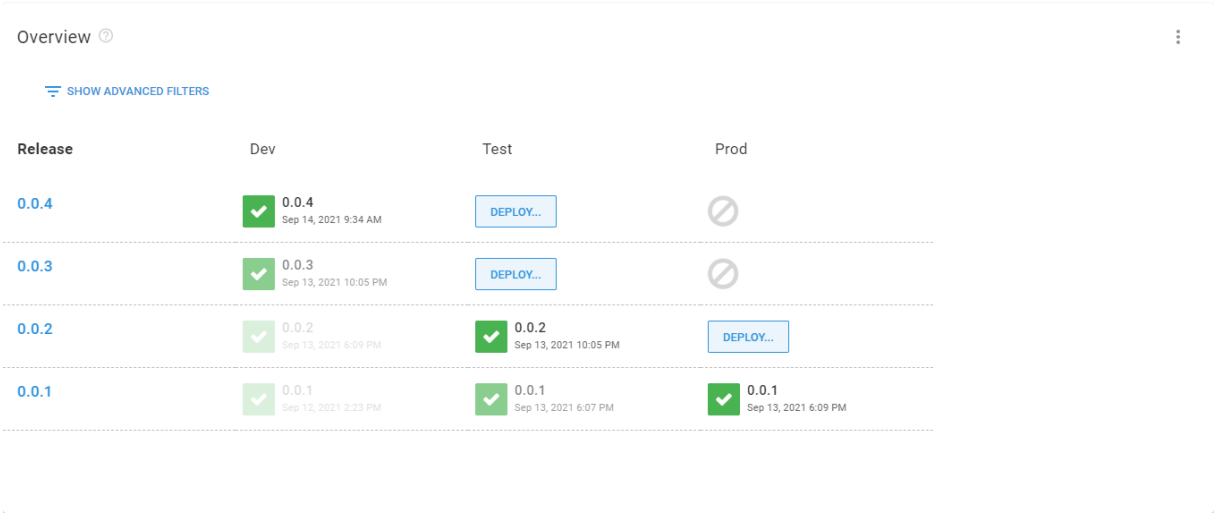
[6] https://octopus.com/




Slika 39. Postavke okoline izgradnje u alatu Jenkins (izrada autora)



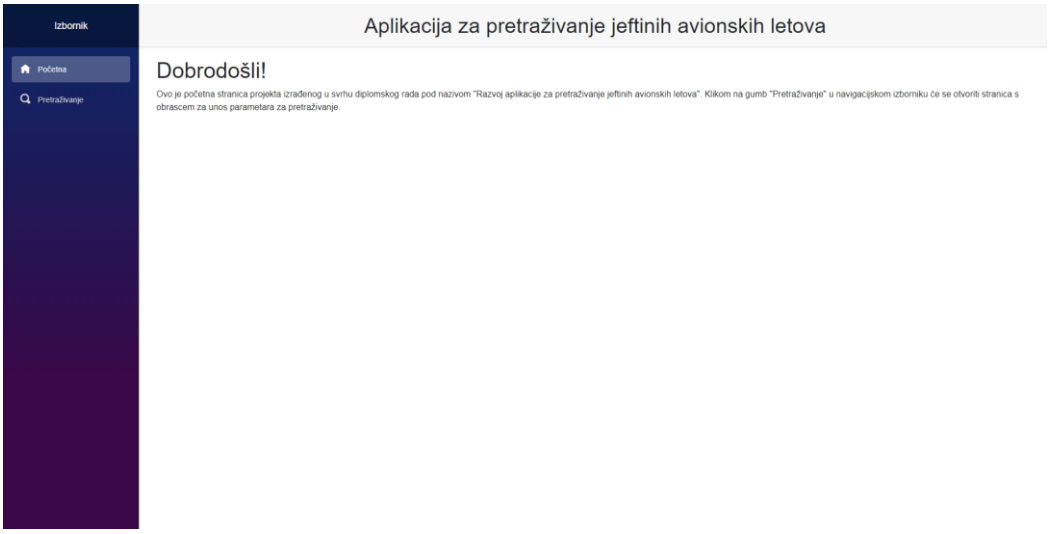

Slika 40. Projektna grupa kojoj pripada aplikacija u Octopus Deploy-u (izrada autora)



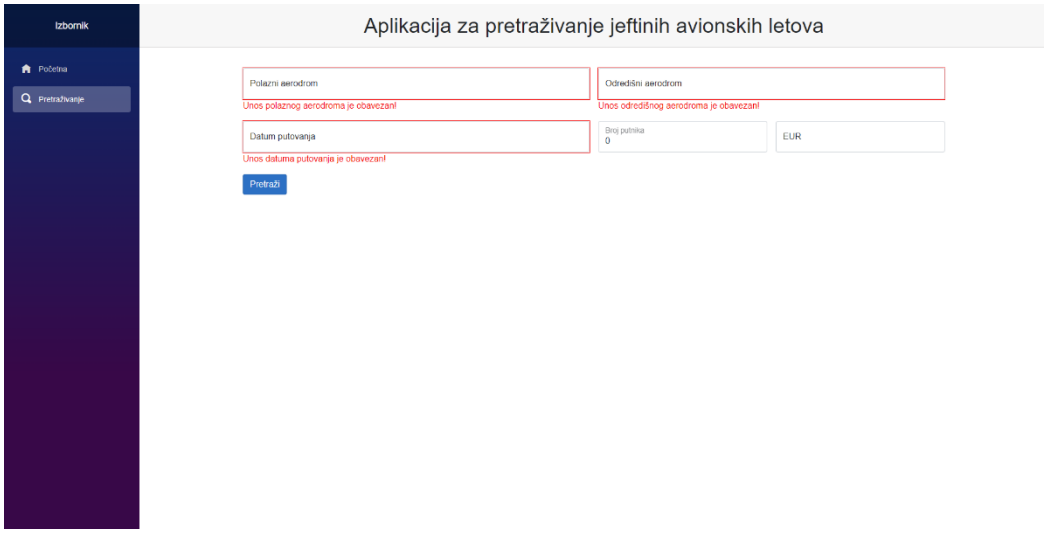

Slika 41. Kronološki popis isporuka aplikacije po verzijama na pojedine okoline u Octopus Deploy-u (izrada autora)

# 6. Prikaz rada aplikacije

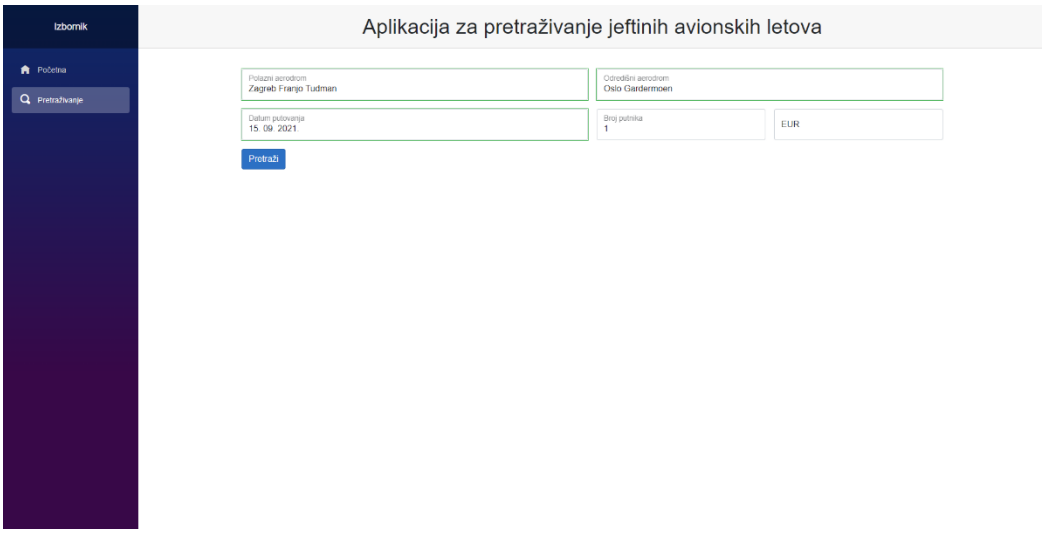
U ovom poglavlju će biti prikazan rad aplikacije. Prilikom pokretanja aplikacije, pojavljuje se početni zaslon prikazan na slici 42. Klikom na opciju „Pretraživanje“ se otvara stranica za pretraživanje jeftinih letova s formom za unos kriterija po kojima će se izvršiti pretraživanje. Svi kriteriji za unos su obavezni, što znači da korisnik neće moći izvršiti pretragu ukoliko nije unio neki od kriterija. Taj scenarij je prikazan na slici 43. Slika 44 prikazuje primjer unosa kriterija za pretragu, a slika 45 prikazuje kako izgleda ispis rezultata pretrage.



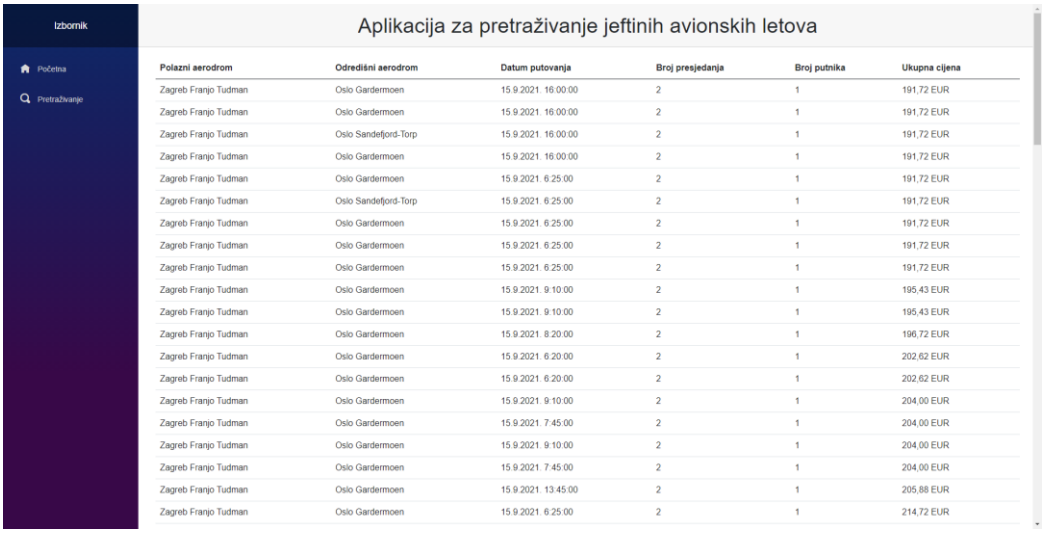

Slika 42. Početna stranica aplikacije (izrada autora)




Slika 43. Obavezan unos kriterija u aplikaciji (izrada autora)




Slika 44. Unos valjanih kriterija u aplikaciju (izrada autora)




| Polazni aerodrom | Odredišni aerodrom | Datum putovanja | Broj presjedanja | Broj putnika | Ukupna cijena |
|---|---|---|---|---|---|
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 16:00:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 16:00:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Sandefjord-Torp | 15.9.2021. 16:00:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 16:00:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:25:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Sandefjord-Torp | 15.9.2021. 6:25:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:25:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:25:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:25:00 | 2 | 1 | 191,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 9:10:00 | 2 | 1 | 195,43 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 9:10:00 | 2 | 1 | 195,43 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 8:20:00 | 2 | 1 | 196,72 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:20:00 | 2 | 1 | 202,62 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:20:00 | 2 | 1 | 202,62 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 9:10:00 | 2 | 1 | 204,00 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 7:45:00 | 2 | 1 | 204,00 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 9:10:00 | 2 | 1 | 204,00 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 7:45:00 | 2 | 1 | 204,00 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 13:45:00 | 2 | 1 | 205,88 EUR |
| Zagreb Franjo Tudman | Oslo Gardermoen | 15.9.2021. 6:25:00 | 2 | 1 | 214,72 EUR |

Slika 45. Rezultati pretrage (izrada autora)

# 7. Zaključak

Razvoj kvalitetne, modularne i fleksibilne aplikacije baš i nije lagan zadatak. Kroz ovaj rad se moglo vidjeti na koje sve elemente treba posvetiti pozornost prilikom pristupa razvoju softvera prateći dobre prakse programiranja. Prvo se istaknula uloga arhitekture i dizajna softverskog rješenja. Arhitektura igra neizmjerno veliku ulogu u procesu razvoja, a ako se ne pristupi dobro dizajniranju i osmišljavanju sustava s arhitekturalnog stajališta, može doći do katastrofalnih posljedica. Napravljena je usporedba arhitektura usmjerenih na bazu podataka i arhitektura usmjerenih na domenu poslovanja te su istaknuti glavni razlozi zašto pristup usmjeren prema podacima nije dobar u kontekstu razvoja složenih poslovnih aplikacija.

Zatim je istaknuta važnost jasne i koncizne arhitekture, kao i činjenica da struktura direktorija projekta treba oslikavati primijenjeni arhitekturalni pristup razvoja softverskog sustava (vrišteća arhitektura). Nakon toga je objašnjen pojam dizajna upravljanog domenom poslovanja kao i uloga ograničenog konteksta, sveprisutnog jezika, entiteta i vrijednosnih objekata te agregata i agregatnog korijena. Predstavljene su heksagonalna, onion i čista arhitektura kao ključne arhitekture usmjerene na domenu poslovanja. Naglašena je važnost poštovanja pravila ovisnosti u kontekstu čiste arhitekture i objašnjen je tijek kontrole po slojevima.

U konačnici su svi objašnjeni koncepti stavljeni u praksu kroz razvoj aplikacije za pretraživanje jeftinih avionskih letova. Struktura aplikacije je podijeljena na četiri modula koji oslikavaju četiri sloja čiste arhitekture. Zatim je naglašena važnost testova, implementirani su jedinični testovi te je objašnjen proces kontinuirane integracije i isporuke aplikacije. Primijenjeni pristup u razvoju aplikacije se naravno može i dodatno poboljšati, a jedan od primjera poboljšanja bi bio primjena funkcionalnog programiranja za modeliranje problemske domene, o čemu govori Scott Wlaschin [18]. I za sam kraj će biti istaknuta sljedeća misao: čist kod je ujedno i strukturiran kod, a strukturiran kod je odraz čiste arhitekture.

# Popis literature


[1] J. Atwood, „Who's Your Coding Buddy?“, *Coding Horror*, velj. 25, 2009. https://blog.codinghorror.com/whos-your-coding-buddy/ (pristupljeno tra. 09, 2021).

[2] A. Molochko, „Clean Architecture : Part 1 – Database vs Domain“, *Software Architecture*, kol. 06, 2017. https://crosp.net/blog/software-architecture/clean-architecture-part-1-databse-vs-domain/ (pristupljeno tra. 09, 2021).

[3] *Floor Plan - St. Rose of Lima Catholic Church - Reno, NV*. Pristupljeno: svi. 09, 2021. [Na internetu]. Dostupno na: https://strosereno.com/pictures/2016/2/STROSE_LAYOUT_2016.jpg

[4] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. London, England: Prentice Hall, 2018.

[5] V. Khorikov, „Domain-centric vs data-centric approaches to software development“, *Enterprise Craftsmanship*, stu. 19, 2015. https://enterprisecraftsmanship.com/posts/domain-centric-vs-data-centric-approaches/ (pristupljeno lip. 09, 2021).

[6] I. Jacobson, *Object-oriented software engineering: a use case driven approach*. [New York] : Wokingham, Eng. ; Reading, Mass: ACM Press ; Addison-Wesley Pub, 1992.

[7] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley, 2004.

[8] D. Pereira, „Clean architecture series - Part 1“, *David Pereira*, svi. 02, 2020. https://pereiren.medium.com/clean-architecture-series-part-1-f34ef6b04b62 (pristupljeno lip. 09, 2021).

[9] D. Pereira, „Clean architecture series - Part 2“, *David Pereira*, velj. 24, 2020. https://pereiren.medium.com/clean-architecture-series-part-2-56197c4b9d58 (pristupljeno lip. 09, 2021).

[10] D. Pereira, „Clean architecture series - Part 3“, *David Pereira*, kol. 03, 2020. https://pereiren.medium.com/clean-architecture-series-part-3-a0c150551e5f (pristupljeno lip. 09, 2021).

[11] A. Molochko, „Clean Architecture : Part 2 – The Clean Architecture“, *Software Architecture*, lip. 10, 2017. https://crosp.net/blog/software-architecture/clean-architecture-part-2-the-clean-architecture/ (pristupljeno svi. 09, 2021).

[12] A. Jain, „Primitive Obsession - code smell that hurt people the most“, sij. 30, 2019. https://www.linkedin.com/pulse/primitive-obsession-code-smell-hurt-people-most-arpit-jain (pristupljeno ruj. 09, 2021).

[13] D. Roth, R. Anderson, i S. Luttin, „Introduction to ASP.NET Core“, tra. 17, 2020. https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0 (pristupljeno ruj. 09, 2021).

[14] M. J. Price, *C# 9 and .NET 5 - modern cross-platform development: build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code*. Birmingham: Packt Publishing, 2020.

[15] G. McLean Hall, *Adaptive code via c#: Agile coding with design patterns and solid principles*. Redmond, Washington: Microsoft Press, 2014.

[16] CloudBees, „What is Jenkins?“, 2021. https://www.cloudbees.com/jenkins/what-is-jenkins (pristupljeno lis. 09, 2021).

[17] Emtec Digital Think Tank, „Octopus for Automated Deployment in DevOps models“, svi. 06, 2018. https://www.emtec.digital/think-hub/blogs/octopus-automated-deployment-in-devops/ (pristupljeno lis. 09, 2021).

[18] S. Wlaschin i B. MacDonald, *Domain modeling made functional: tackle software complexity with domain-driven design and F♯*, Version: P1.0. Raleigh, North Carolina: The Pragmatic Bookshelf, 2018.

# Popis slika