Evolucija arhitekture modularne monolitne web aplikacije u mikroservisnu

Gambeta, Lukas

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:911230

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-17**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

Sveučilište u Rijeci, Fakultet informatike i digitalnih tehnologija

Sveučilišni prijediplomski studij Informatika

Lukas Gambeta

Evolucija arhitekture modularne monolitne web aplikacije u mikroservisnu

Završni rad

Mentor: doc. dr. sc. Vedran Miletić

Rijeka, 21.09.2023.

Sažetak

Ovaj završni rad istražuje evoluciju arhitekture web aplikacija, fokusirajući se na transformaciju modularnih monolitnih aplikacija u mikroservisnu arhitekturu. Počevši s pregledom monolitnih aplikacija, opisali smo njihove prednosti i ograničenja, istražujući kako su brz razvoj i jednostavno održavanje karakterizirali ovu tradicionalnu arhitekturu.

Nadalje, predstavili smo koncept modularnih monolita, gdje su monolitne aplikacije strukturirane na način da se sastoje od logički odvojenih modula ili komponenata. Ovaj pristup olakšava razvoj i održavanje, ali se suočava s izazovima u pogledu skalabilnosti i upravljanja kako aplikacija raste.

Kroz primjer implementacije koristeći ASP.NET Core, analizirali smo slojevitu arhitekturu modularnih monolita, uključujući Domain-Driven Design (DDD), Application, API, Persistence i Infrastructure slojeve. Razmotrili smo kako se DDD pristup primjenjuje na modele podataka, a Command and Query Responsibility Segregation (CQRS) obrazac na komande i upite za optimizaciju performansi.

Zatim smo istražili korake migracije iz modularnih monolita u mikroservise, gdje smo istaknuli prednosti postupnog prijelaza i upotrebu alata poput RabbitMQ za komunikaciju između modula. Analizirali smo i tehničke aspekte poput konfiguracije DbContext-a i integracije s mikroservisnom infrastrukturom.

Nakon što smo istražili transformaciju modularnih monolitnih aplikacija u mikroservisnu arhitekturu, detaljno smo opisali postupak deployanja naših mikroservisa u Azure okruženje.

Sadržaj

1. Uvod.....1
2. Monolitne Web Aplikacije.....2
3. Modularni Monoliti.....3
    3.1. Implementacija.....4
4. Mikroservisi.....14
    4.1. Migriranje modula u mikroservis.....15
    4.2. Deployment.....16
5. Zaključak.....20
6. Literautra.....21
7. Popis Slika.....22

1. Uvod

U svijetu razvoja softvera, arhitektura aplikacija igra ključnu ulogu u njihovom uspjehu i trajnosti. S rastom zahtjeva korisnika, tehnološkim inovacijama i promjenama u poslovnim modelima, organizacije se neprestano suočavaju s potrebom za evolucijom svojih aplikacija kako bi ostale konkurentne i funkcionalne. U tom kontekstu, ovaj završni rad istražuje razvoj arhitekture modularnih monolitnih web aplikacija prema mikroservisnoj arhitekturi.

Monolitne aplikacije su tradicionalna arhitektura u kojoj su sve komponente i funkcionalnosti integrirane unutar istog kodnog bloka. Iako su monoliti omogućavali brz razvoj i jednostavno održavanje za manje projekte, postaju izazovni kako aplikacija raste. Održavanje, skaliranje i fleksibilnost postaju složeniji, što dovodi do potrebe za promjenom arhitekture.

Modularni monoliti predstavljaju korak između tradicionalnih monolita i mikroservisne arhitekture. U ovom pristupu, monolitna aplikacija strukturirana je na način da se sastoji od logički odvojenih modula ili komponenata, što olakšava razvoj i održavanje. Svaki modul obavlja određeni skup funkcionalnosti ili usluga, ali svi moduli dijele isti kod.

U ovom završnom radu detaljno ćemo analizirati implementaciju modularnih monolita koristeći ASP.NET Core, razvojni okvir koji omogućava razvoj i organizaciju aplikacija. Proučit ćemo kako se različiti slojevi, poput sloja za pristup podacima, primjenjuju unutar ovog modela arhitekture. Također, istražit ćemo kako modularni monoliti predstavljaju korak prema mikroservisima, olakšavajući organizacijama postupni prijelaz na moderniju arhitekturu.

Kroz analizu implementacije i migracije, ovaj završni rad pružit će dublji uvid u evoluciju arhitekture aplikacija. Nastavljajući dalje, istražit ćemo korake potrebne za migraciju modularnih monolita u mikroservise, uz korištenje alata kao što je RabbitMQ za komunikaciju između modula. Kroz sve ove aspekte, istražit ćemo kako možemo modernizirati svoje aplikacije i ostati konkurentne u dinamičnom svijetu razvoja softvera.

E-commerce aplikacije su poznate po svojoj kompleksnosti jer moraju podržavati različite ključne funkcionalnosti kao što su upravljanje proizvodima, korisnicima, narudžbama, plaćanjem, recenzijama, ocjenama i mnogim drugim aspektima. Ta kompleksnost čini ih izazovnima za upravljanje i održavanje kao monolitne aplikacije, otvarajući prostor za primjenu mikroservisne arhitekture.

Zbog toga ćemo koristiti jednostavnu e-commerce aplikaciju kao primjer kako bi smo prikazali transformaciju monolitne arhitekture u mikroservisnu. Ova aplikacija služi kao platforma za trgovce da postave svoje proizvode ili usluge online, stvarajući digitalnu trgovinu koja je dostupna širokom krugu korisnika. Svrha aplikacije je stvaranje povezanog online tržišta koje olakšava trgovcima širenje svog poslovanja i korisnicima pristup proizvodima i uslugama.

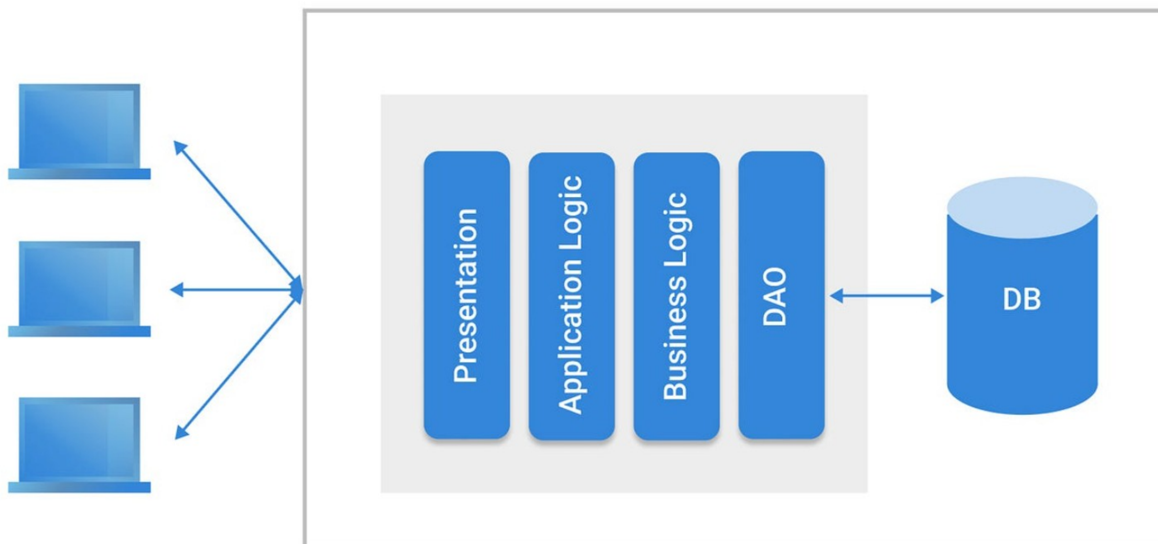
2. Monolitne Web Aplikacije

Monolitna web aplikacija predstavlja tip aplikacije koja se temelji na arhitekturi u kojoj su svi njezini dijelovi i funkcionalnosti integrirani unutar istog koda. Ova arhitektura često se koristi za manje ili početne projekte zbog svoje jednostavnosti i brzine razvoja. U takvoj aplikaciji, različiti dijelovi, kao što su korisničko sučelje, poslovna logika i baza podataka, često su usko povezani i dijele isti kontekst izvršavanja. [1]

Jedan od ključnih karakteristika monolitnih aplikacija je njihova duboka povezanost (*engl. tight coupling*) između različitih komponenata i servisa unutar aplikacije. To znači da ako napravite promjenu u jednom dijelu aplikacije, ta promjena može imati neslućene posljedice na druge dijelove aplikacije. Ovo može znatno otežati proces održavanja i nadogradnje aplikacije, jer svaka promjena zahtijeva pažljivo testiranje kako bi se osiguralo da nema negativnih utjecaja na druge dijelove. [1]

Osim toga, skaliranje monolitne aplikacije obično zahtijeva povećanje broja instanci cijele aplikacije. Ovo može biti neučinkovito i skupo, jer se cijela aplikacija mora skalirati čak i ako je samo mali dio aplikacije pod povećanim opterećenjem. [2]

Unatoč svojim ograničenjima, monolitne web aplikacije imaju svoje mjesto u svijetu razvoja softvera, posebno za manje projekte s ograničenim resursima.Njihova jednostavnost i brzina razvoja često čine monolitnu arhitekturu atraktivnim izborom za projekte koji ne zahtijevaju visoku skalabilnost i složenost. Međutim, kako aplikacija raste i postaje složenija, monolitna arhitektura može postati prepreka za postizanje visokih razina skalabilnosti, fleksibilnosti i upravljanja aplikacijom.



Slika 1: Monolitna arhitektura

3. Modularni Monoliti

Modularni monoliti su monolitne web aplikacije koje su strukturirane na način da se sastoje od logički odvojenih modula ili komponenata. Svaki modul obavlja određeni skup funkcionalnosti ili usluga, iako su svi moduli unutar istog koda. Ova arhitektura omogućava organizaciju aplikacije na način koji olakšava razvoj i održavanje. [3]

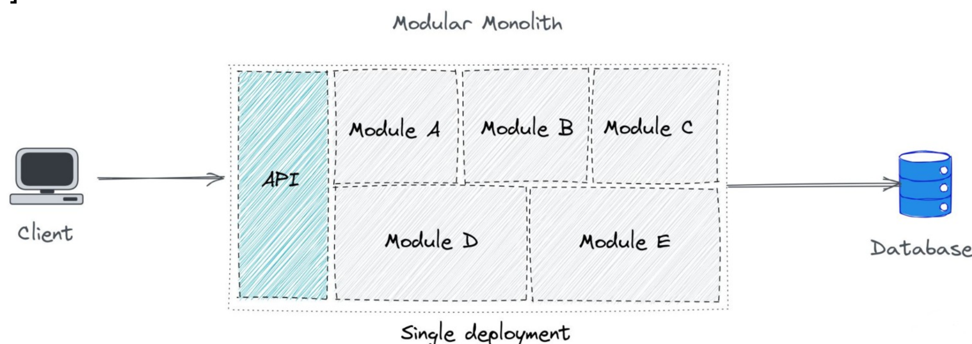
Međutim, skaliranje modularnog monolita zahtijeva skaliranje cijelog monolita, jer su moduli često međusobno povezani i dijele isti kontekst izvršavanja. To znači da će povećanje opterećenja na jednom dijelu aplikacije zahtijevati skaliranje cijele aplikacije,

Ključna značajka modularnih monolita je da se komponente ili moduli unutar aplikacije ne moraju nužno preklapati ili biti tijesno povezani, kao što je često slučaj s tradicionalnim monolitima. Umjesto toga, svaki modul može biti dizajniran da bude relativno autonomna cjelina s vlastitim skupom funkcionalnosti. Ovo omogućava razvojnim timovima da rade na različitim modulima neovisno, što poboljšava produktivnost i omogućava brže inovacije.

Svaki od modula u modularnom monolitu može imati svoj vlastiti skup tehnologija, bazu podataka i biblioteka, pružajući timovima slobodu odabira najprikladnijih alata za njihov specifičan zadatak. To olakšava razvojni proces i omogućava timovima da budu produktivni u svojim specifičnim oblastima bez potrebe za dubokim razumijevanjem cijelog aplikacijskog koda.

Još jedna važna prednost modularnih monolita leži u tome što omogućuju postepen prelazak s klasičnih monolitnih aplikacija na modernije arhitekture poput mikroservisa. Organizacije koje već imaju modularne monolite ne moraju se suočiti s radikalnim prelaskom na mikroservisnu arhitekturu, što može biti vrlo složeno i skupo. Umjesto toga, mogu postepeno migrirati svoje module u mikroservise kako rastu i razvijaju se.

Postupak migracije modula u mikroservise je tehnika koja se naziva Strangler obrazac (*engl. pattern*). Strangler obrazac je tehnika migracije softverskog sustava koja uključuje postepeno zamjenjivanje starijeg ili monolitnog dijela aplikacije novim ili mikroservisnim komponentama. U slučaju modularnih monolita, organizacije koriste sličan pristup postepenog razdvajanja funkcionalnosti i migracije prema mikroservisnoj arhitekturi kako bi izbjegli radikalnu promjenu. [4]



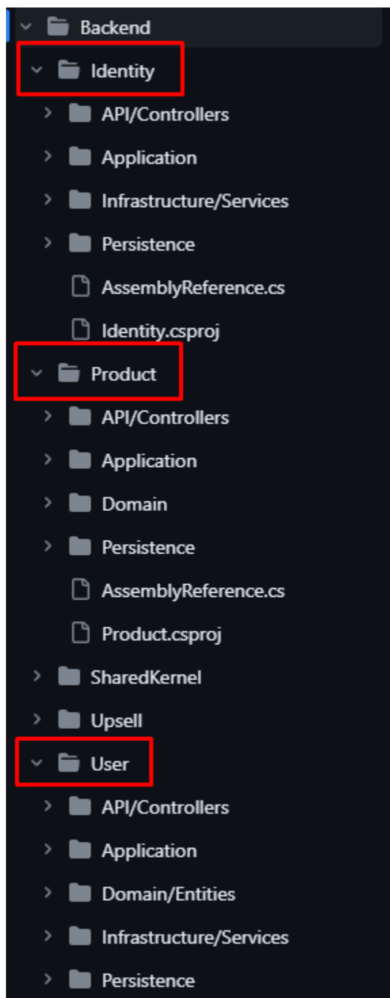
Slika 2: Modularna monolitna arhitektura

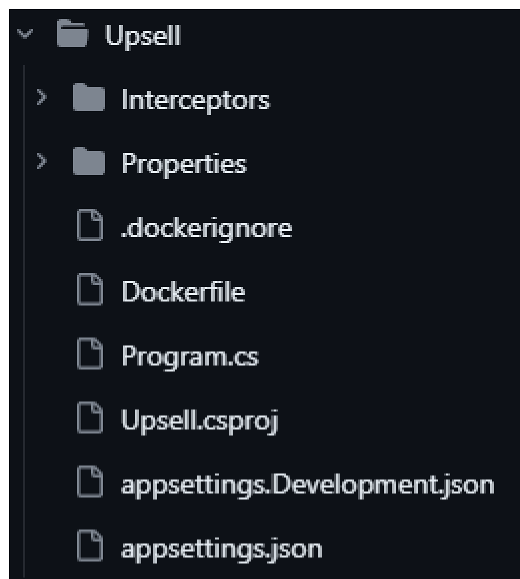
3.1. Implementacija

Za implementaciju modularnog monolita odabrali smo ASP.NET Core razvojni okvir, prateći koncept koji omogućava strukturiranje aplikacije na način da se sastoji od logički odvojenih modula ili komponenata. Svaki modul je samostalna cjelina koja obavlja specifičan skup funkcionalnosti ili usluga, a svi su oni integrirani unutar istog monolita.

Glavni projekt naše aplikacije zove se Upsell te predstavlja centralnu točku aplikacije koja sadrži osnovne komponente kao što su konfiguracija (appsetting) i Program.cs datoteka koja sadrži servise koji se injektiraju (*engl, dependency injection*) unutar aplikacije.

Modularni monolit ove web aplikacije sastoji se od tri biblioteke klasa (*engl. class library*) User, Product i Identity te svaka biblioteke reprezentira točno jedan modul u monolitu. Svaki od ovih modula fokusira se na jedan aspekt funkcionalnosti i sadrži sve potrebne klase, kontrolere, servise i resurse specificirane za taj modul.





Slika 4: Monolitna arhitektura – folder struktura

Slika 3: Modularna monolitna arhitektura – folder struktura

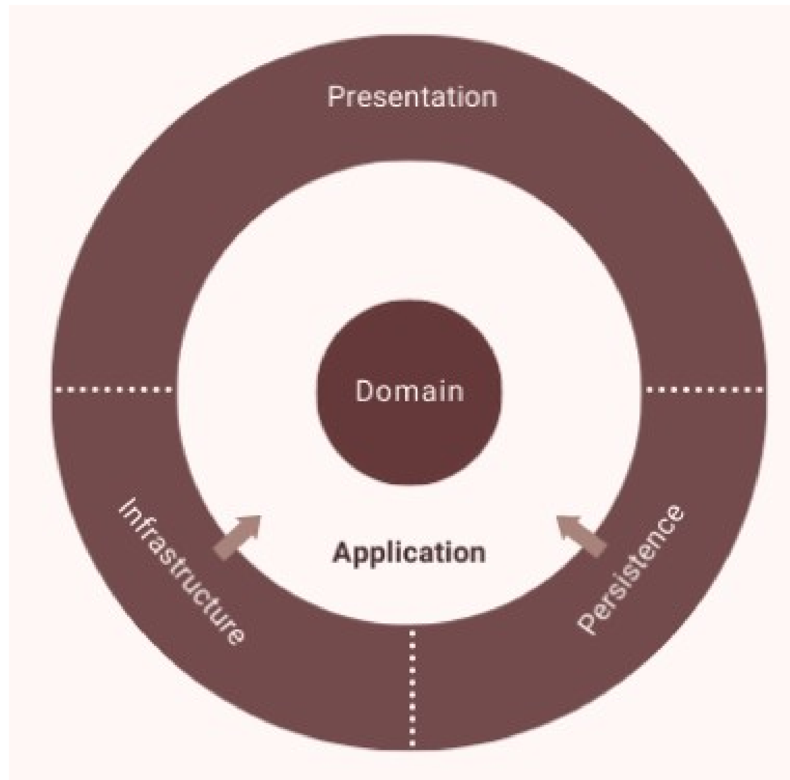
Svaki pojedinačni projekt uključuje jednu datoteku Program.cs. To znači da, kao u slučaju tradicionalnih monolitnih aplikacija, naš modularni monolit također ima samo jednu Program.cs datoteku. U toj datoteci konfiguriramo sve servise i komponente koje će biti dostupne unutar tog projekta ili monolita. To uključuje i dodavanje DbContext-a koji nam omogućuje izvođenje osnovnih operacija (dodavanje, čitanje, ažuriranje, brisanje) nad bazom podataka.

Na slici vidimo registraciju više instanci DbContext-a. Ovo znači da svaki modul unutar naše aplikacije može imati svoju zasebnu bazu podataka. Također, ovaj pristup omogućuje svakom modulu da koristi različitu bazu podataka prema svojim potrebama i zahtjevima, što znači da nije obavezno da svi moduli koriste istu bazu podataka, već mogu biti prilagođeni za različite svrhe ili dijelove aplikacije.

```
builder.Services.AddDbContext<IdentityDbContext>((serviceProvider, options) =>
{
    var interceptor = serviceProvider.GetService<AuditableEntitiesInterceptor>();
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("Identity"))
        .AddInterceptors(interceptor);
});

builder.Services.AddDbContext<UserDbContext>((serviceProvider, options) =>
{
    var interceptor = serviceProvider.GetService<AuditableEntitiesInterceptor>();
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("User"))
        .AddInterceptors(interceptor);
});

builder.Services.AddScoped<IProductDbContext>(provider => provider.GetRequiredService<ProductDbContext>());
builder.Services.AddScoped<IIdentityDbContext>(provider => provider.GetRequiredService<IdentityDbContext>());
builder.Services.AddScoped<IUserDbContext>(provider => provider.GetRequiredService<UserDbContext>());

builder.Services.AddDbContext<ProductDbContext>((serviceProvider, options) =>
{
    var interceptor = serviceProvider.GetService<AuditableEntitiesInterceptor>();
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection"), b => b.MigrationsAssembly("Product"))
        .AddInterceptors(interceptor);
});
```

Slika 5: Program.cs

U arhitekturi modularnog monolita, svaki modul slijedi princip "čiste arhitekture" *(engl. clean architecture)*, što omogućuje organizaciju koda na način koji poboljšava održavanje, testiranje i skalabilnost. Ovaj pristup možemo podijeliti na nekoliko ključnih slojeva koji zajedno čine svaki modul.



Slika 6: Clean architecture

Jedan od najvažnijih koncepta u "čistoj arhitekturi" jest DDD (Domain-Driven Design) koji predstavlja pristup razvoju softvera koji se fokusira na duboko razumijevanje poslovne domene. U početnim fazama razvoja, DDD sloj može sadržavati anemične entitete[1] (*engl. anemic entities*) budući da još uvijek nema značajne količine poslovne logike.

Slijedeći sloj u arhitekturi predstavlja aplikacijski sloj, obuhvaćajući sučelja, komande i upite. U kontekstu ovog sloja, posebno se ističu komande i upiti koji su ključni dijelovi CQRS (Command and Query Responsibility Segregation) obrasca. Ovaj obrazac naglašava važnost optimizacije performansi i unapređenja arhitekture aplikacije.Optimizacija performanse često se postiže razdvajanjem baze podataka na dvije komponente: bazu za čitanje i bazu za pisanje. Kada su baze razdvojene onda ih je lakše optimizirati u svrhu akceleracije radnji koje izvršavamo nad istima.

```csharp
using AutoMapper;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Product.Application.Interfaces;
using SharedKernel.Types;

namespace Product.Application.Queries;

public record GetCategoriesQuery() : IRequest<Response<IEnumerable<GetCategoriesQueryModel>>>;
public class GetCategoriesQueryHandler : IRequestHandler<GetCategoriesQuery, Response<IEnumerable<GetCategoriesQueryModel>>>
{
    private readonly IProductDbContext _context;
    public GetCategoriesQueryHandler(IProductDbContext context, IMapper mapper)
    {
        _context = context;
    }
    public async Task<Response<IEnumerable<GetCategoriesQueryModel>>> Handle(GetCategoriesQuery request, CancellationToken cancellationToken)
    {
        var categories = await _context.Category.ToListAsync(cancellationToken);

        var categoriesQueryModel = categories.Select(c => new GetCategoriesQueryModel
        {
            Id = c.Id,
            ParentCategoryId = c.ParentCategoryId,
            Name = c.Name,
            Image = c.Image
        });

        return new Response<IEnumerable<GetCategoriesQueryModel>> { Success = true, Data = categoriesQueryModel };
    }
}

public class GetCategoriesQueryModel
{
    public int Id { get; set; }
    public int? ParentCategoryId { get; set; }
    public string Name { get; set; }
    public string Image { get; set; }
}
```

Slika 7: CQRS Query

```
using MediatR;
using Product.Application.Interfaces;
using SharedKernel.Types;

namespace Product.Application.Commands
{
    public record DeleteProductCommand(int ProductId) : IRequest<Response<string>>;

    public class DeleteProductCommandHandler : IRequestHandler<DeleteProductCommand, Response<string>>
    {
        private readonly IProductDbContext _context;

        public DeleteProductCommandHandler(IProductDbContext context)
        {
            _context = context;
        }

        public async Task<Response<string>> Handle(DeleteProductCommand request, CancellationToken cancellationToken)
        {
            var existingProduct = await _context.Product.FindAsync(request.ProductId);

            if (existingProduct == null)
            {
                return new Response<string>
                {
                    Success = false,
                    ErrorMessage = "Product not found."
                };
            }

            _context.Product.Remove(existingProduct);

            await _context.SaveChangesAsync(cancellationToken);

            return new Response<string>
            {
                Success = true,
            };
        }
    }
}
```

Slika 8: CQRS Command

Primjećujemo da rukovatelji komandi i upita u našem primjeru implementiraju sučelje IRequestHandler. Ova implementacija je bitna jer se usko veže uz još jedan važan obrazac koji se koristi u kontekstu CQRS-a, a to je Mediator obrazac.

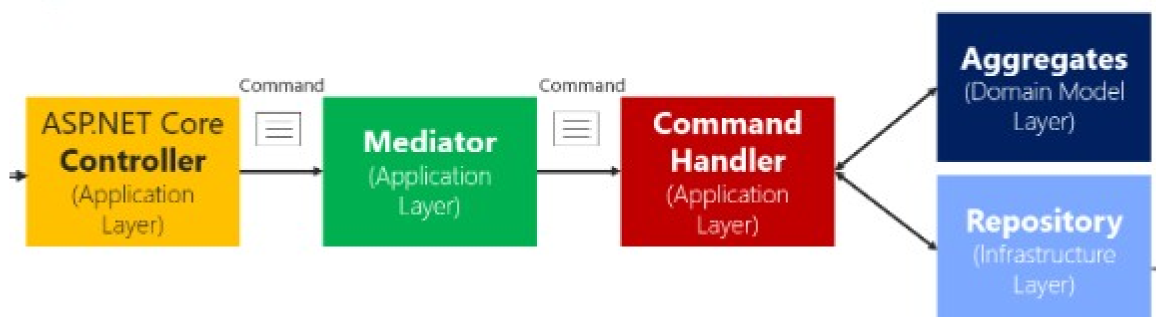
Mediator obrazac omogućava decentralizirano upravljanje komunikacijom između različitih komponenata sustava. Umjesto da komponente komuniciraju izravno jedna s drugom, Mediator obrazac uvodi posrednika (mediatora) koji upravlja razmjenom poruka između komponenata. Ovaj pristup čini sustav fleksibilnijim i olakšava održavanje, jer se komunikacijska logika izdvaja iz samih komponenata.

Sloj "API" sastoji se od kontrolera koji služe kao izloženi sloj za komunikaciju s vanjskim sustavima ili korisnicima. Kontroleri obrađuju zahtjeve i prosljeđuju ih aplikacijskom sloju za obradu. Ovaj sloj često implementira RESTful API-je ili druge protokole za komunikaciju.

```
[Route("api/[controller]")]
[ApiController]
public class CategoryController : ControllerBase
{
    private readonly IMediator _mediator;

    public CategoryController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet("categories")]
    public async Task<ActionResult<string>> GetCategories()
    {
        return Ok(await _mediator.Send(new GetCategoriesQuery()));
    }
}
```

Slika 9: Controller

Na slici vidimo kontroler koji šalje zahtjev aplikacijskom sloju putem posrednika. U pozadini posrednik ima riječnik (*engl. dictionary)* koji se populira prilikom implementacije IRequestHandler sučelja te isti koristi kako bi znao gdje preusmjeriti zahtjev.



Slika 10: CQRS & Mediator

Korištenjem Mediator obrasca za prijenos zahtjeva iz kontrolera u aplikacijski sloj, postižemo čistu i organiziranu arhitekturu. Mediator djeluje kao posrednik koji omogućava komunikaciju između različitih komponenata aplikacije, a kontroler ne mora biti svjestan svih detalja implementacije u aplikacijskom sloju. Ovaj pristup čini kod kontrolera čitkim i olakšava održavanje aplikacije.

U arhitekturi modularnih monolita, sloj „Persistence” igra ključnu ulogu u interakciji s bazom podataka. Ovaj sloj odgovoran je za komunikaciju s bazom podataka, izvršavanje različitih operacija nad njom i omogućavanje aplikaciji pristup podacima.

Centralna komponenta ovog sloja je DbContext, koja predstavlja vezu između aplikacije i baze podataka. DbContext je od suštinskog značaja jer omogućava definiranje i mapiranje modela podataka na tablice u bazi te pruža mehanizme za izvršavanje upita i ažuriranje podataka.

Što se tiče konfiguracije entiteta, svaki DbSet unutar DbContext-a ima svoju konfiguraciju. Ova konfiguracija uključuje definiranje kako se entitet mapira na tablicu u bazi, postavljanje primarnih ključeva, veza između entiteta i druga pravila koja utječu na ponašanje baze podataka.

Entity Framework (EF) često se koristi kao alat u sloju "Persistence" modularnih monolita jer olakšava konfiguraciju i upravljanje entitetima te omogućava izvođenje učinkovitih upita prema bazi podataka. Kroz EF, razvojni timovi mogu usredotočiti se na poslovnu logiku umjesto na složenosti komunikacije s bazom podataka.

```
public class ProductDbContext : DbContext, IProductDbContext
{
    public ProductDbContext(DbContextOptions<ProductDbContext> options) : base(options) { }

    public DbSet<Category> Category { get; set; }
    public DbSet<Domain.Entities.Product> Product { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        new CategoryEntityConfiguration().Configure(modelBuilder.Entity<Category>());
        new ProductEntityConfiguration().Configure(modelBuilder.Entity<Domain.Entities.Product>());
    }
}
```

Slika 11: DbContext

Naposljetku, sloj "Infrastructure" uključuje sve tehničke aspekte potrebne za podršku aplikaciji, kao što je integracija s RabbitMQ za komunikaciju između modula. Ovaj sloj brine se o infrastrukturnim detaljima kako bi omogućio nesmetano izvršavanje modula te pravilno funkcioniranje cijelog modularnog monolita.

Važno je napomenuti da RabbitMQ, iako nije obvezan za komunikaciju između modula unutar modularnog monolita, postaje ključan element u procesu prijelaza s modularnog monolita na mikroservisnu arhitekturu. U ovom prijelaznom razdoblju, RabbitMQ igra ključnu ulogu kao posrednik u komunikaciji između modula koji se postupno razdvajaju i migriraju prema mikroservisima. Stoga se može reći da RabbitMQ postaje ključan korak u tranziciji prema modernijoj arhitekturi.

```
public class IdentityRabbitMqProducer : IIdentityRabbitMqProducer
{
    private readonly IConfiguration _configuration;
    public IdentityRabbitMqProducer(IConfiguration configuration) { _configuration = configuration; }
    public void SendMessage<T>(T message)
    {
        var factory = new ConnectionFactory()
        {
            HostName = _configuration["RabbitMQ:RabbitMQHost"],
            Port = _configuration.GetValue<int>("RabbitMQ:RabbitMQPort")
        };

        var connection = factory.CreateConnection();
        using
        var channel = connection.CreateModel();

        channel.QueueDeclare("identity", exclusive: false);

        var json = JsonConvert.SerializeObject(message);
        var body = Encoding.UTF8.GetBytes(json);

        channel.BasicPublish(exchange: "", routingKey: "identity", body: body);
    }
}
```

Slika 12: RabbitMq producer

```
_rabbitMqProducer.SendMessage(new IntegrationEvent<RegisterUserCommand>()
{
    Name = nameof(IdentityEventType.UserRegistered),
    Data = request
});
```

Slika 13: RabbitMq Message

```
public class UserRabbitMqConsumer : BackgroundService
{
    private readonly IUserEventProcessor _userEventProcessor;
    private readonly IConfiguration _configuration;
    private IConnection _connection;
    private IModel _channel;
    public UserRabbitMqConsumer(IUserEventProcessor userEventProcessor, IConfiguration configuration)
    {
        _userEventProcessor = userEventProcessor;
        _configuration = configuration;

        InitializeRabbitMQ();
    }

    public void InitializeRabbitMQ()
    {
        var factory = new ConnectionFactory()
        {
            HostName = _configuration["RabbitMQ:RabbitMQHost"],
            Port = _configuration.GetValue<int>("RabbitMQ:RabbitMQPort")
        };

        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
        _channel.QueueDeclare("identity", exclusive: false);
    }

    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        var consumer = new EventingBasicConsumer(_channel);

        consumer.Received += (model, eventArgs) =>
        {
            var body = eventArgs.Body.ToArray();
            var message = Encoding.UTF8.GetString(body);
            _userEventProcessor.Process(message);
        };

        _channel.BasicConsume("identity", autoAck: true, consumer);

        return Task.CompletedTask;
    }
}
```

Slika 14: RabbitMq consumer

RabbitMQ je popularna open-source aplikacija za upravljanje redovima poruka (*engl. message queues*) koja omogućava asinkronu komunikaciju između različitih komponenata ili servisa u aplikaciji.

U kontekstu modularnog monolita, RabbitMQ Processor igra ključnu ulogu u omogućavanju komunikacije između različitih modula ili servisa. Umjesto da izravno pozivaju ili komuniciraju s drugim modulima, moduli šalju poruke RabbitMQ Processoru, a zatim RabbitMQ Processor šalje te poruke drugim modulima ili servisima putem RabbitMQ sustava.

Ovo je posebno korisno kada je potrebno ostvariti asinkronu komunikaciju ili kada je potrebno poslati poruku ili zahtjev jednom modulu, a da se ne čeka odmah na odgovor. RabbitMQ Processor omogućava rasterećenje modula od izravne komunikacije i olakšava asinkronu obradu poruka.

S druge strane, RabbitMQ Consumer je komponenta koja služi za primanje i obradu poruka koje su poslane putem RabbitMQ sustava. RabbitMQ Consumer pretplaćuje se na određeni red poruka i čeka da stigne poruka. Kada poruka stigne, RabbitMQ Consumer je odgovoran za obradu te poruke prema definiranoj logici ili akcijama koje su potrebne.

RabbitMQ Consumeri su ključni za obradu poruka koje su poslane iz drugih modula ili servisa. Oni čekaju i reagiraju na dolazne poruke te obavljaju potrebne zadatke ili akcije na temelju njih. Ovo omogućava modularnost i fleksibilnost u aplikaciji, jer različiti Consumeri mogu biti odgovorni za različite vrste poruka i obradu. [7]

U našoj aplikaciji, tijekom procesa registracije, Identity modul provjerava postoji li već registrirani korisnik. Ako korisnik nije registriran, Identity modul izvršava registraciju i šalje poruku User modulu. Ova komunikacija putem poruke omogućava razdvajanje odgovornosti Identity modula i User modula te predstavlja važan aspekt modularne arhitekture.

Identity modul odgovara za autentifikaciju i registraciju korisnika, dok User modul odgovara za pohranu i upravljanje informacijama o korisnicima. Razdvajanje ovih odgovornosti omogućava svakom modulu da se fokusira na svoje specifične zadatke i poslovnu logiku.

Ovaj pristup omogućava fleksibilnost i skalabilnost u razvoju aplikacije, jer svaki modul ima jasno definiranu ulogu i neovisnost o ostalim modulima. Također, olakšava se održavanje i nadogradnja aplikacije jer se svaki modul može razvijati i testirati zasebno, a promjene u jednom modulu ne moraju nužno utjecati na druge module. To je esencijalna karakteristika modularnih monolita.

4. Mikroservisi

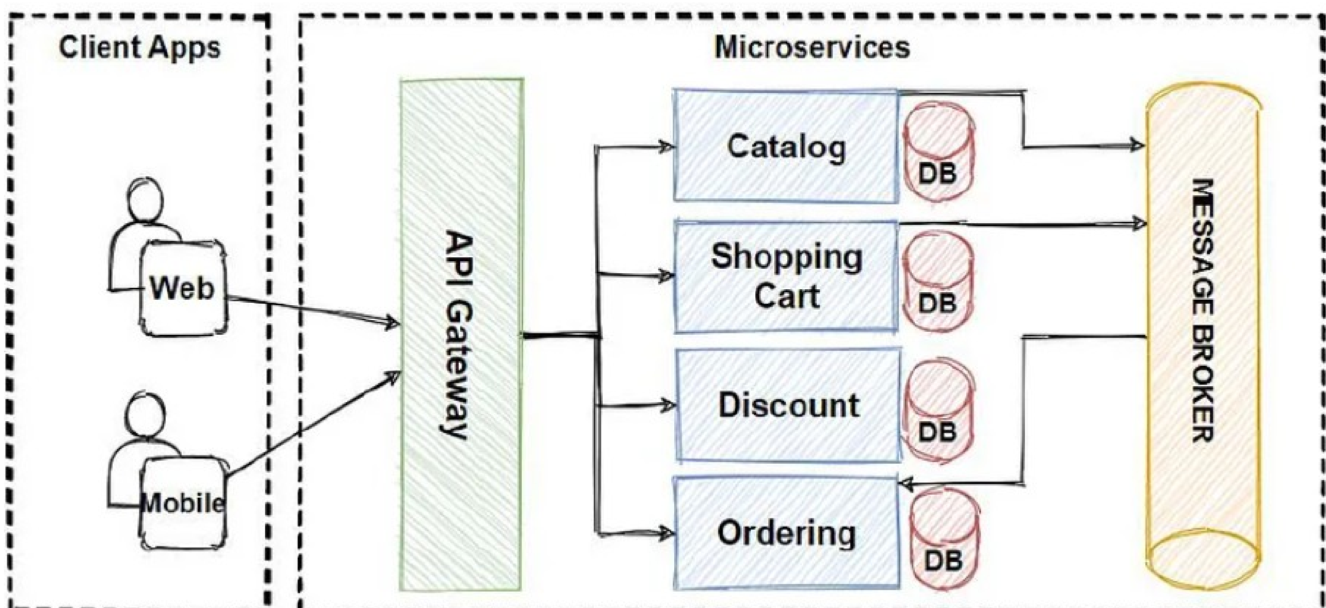
Mikroservisna arhitektura predstavlja revolucionarnu promjenu u načinu kako razmišljamo i razvijamo aplikacije. Umjesto da cijelu aplikaciju promatramo kao jedan monolitan entitet, mikroservisi razbijaju aplikaciju na manje, autonomne servise koji obavljaju specifične zadatke. Svaki servis može se razvijati, skalirati i održavati neovisno o drugim servisima u aplikaciji. [5]

Jedna od glavnih prednosti mikroservisne arhitekture leži u njezinoj skalabilnosti. Umjesto skaliranja cijele aplikacije, mikroservisi omogućavaju skaliranje samo onih dijelova aplikacije koji zahtijevaju više resursa ili su izloženi većem opterećenju. Ovo znači da se resursi mogu učinkovito iskoristiti, a aplikacija može bolje reagirati na promjenjive zahtjeve. [6]

Osim toga, mikroservisna arhitektura promiče brži razvoj i inovacije. Svaki servis može biti razvijan od strane posebnog tima, što omogućava paralelan rad na različitim dijelovima aplikacije. To rezultira bržim isporukama novih funkcionalnosti i boljom prilagodljivošću aplikacije na promjene na tržištu. [5]

Važno je napomenuti da mikroservisna arhitektura donosi i izazove. Upravljanje većim brojem servisa može biti kompleksno, a komunikacija između servisa zahtijeva posebnu pažnju. Osim toga, potrebna je odgovarajuća infrastruktura i alati za nadzor i upravljanje servisima.

U sklopu mikroservisne arhitekture, svaki servis ima svoju bazu podataka, što omogućava veću neovisnost i fleksibilnost. Ovo također znači da je svaki servis odgovoran za svoje podatke i poslovnu logiku, čime se izbjegavaju tijesne veze između servisa.



Slika 15: Mikroservisna arhitektura

4.1. Migriranje modula u mikroservis

Prijelaz iz modularnog monolita na mikroservisnu arhitekturu olakšan je zahvaljujući već postojećoj modularnosti u strukturi aplikacije. Često se prilikom implementacije modularne monolitne arhitekture stvaraju tijesno povezani moduli iz kojih se razvija mnogo komplikacija prilikom migracije ne mikroservisnu arhitekturu. Mi smo to izbjegli tako što smo od samog početka implementirali komunikaciju između modula putem RabbitMq-a. Također, i ako su do sada sve tablice bile u jednoj bazi, zato što svaki modul ima svoj DbContext s lakoćom možemo izdvojiti tablice iz postojeće baze u novu bazu podataka.

Prvi korak je kreiranje novog projekta koji će predstavljati mikroservis. Ovaj projekt sadrži svoj vlastiti Program.cs, koji služi kao ulazna točka za mikroservis. Iz Program.cs modularnog monolita, izdvajamo sve servise koji su specifični za taj određeni modul i prenosimo ih u novostvoreni projekt za mikroservis.

Nakon što smo prenijeli servise, kopiramo sve relevantne direktorije i datoteke iz modula u novi mikroservisni projekt. Ovo uključuje sve komponente potrebne za funkcionalnost tog modula.

Također, moramo stvoriti novu bazu podataka za mikroservis u SQL Server Management Studiju (SSMS) i prilagoditi Connection String u konfiguracijskoj datoteci (appsettings) mikroservisa kako bismo usmjerili mikroservis prema novoj bazi.

Nakon kreiranja nove baze podatala, provodimo migraciju kako bismo osigurali da nova baza odgovara potrebama mikroservisa. Također je potrebno populirati novu bazu podataka s postojećim podacima te izbrisati tablice iz modularnog monolita:

```
SET IDENTITY_INSERT [Identity].dbo.IdentityUser ON;

INSERT INTO [Identity].dbo.IdentityUser ([Id], [Email], [PasswordHash], [CreatedAt], [ModifiedAt])
SELECT [Id], [Email], [PasswordHash], [CreatedAt], [ModifiedAt] FROM dbo.IdentityUser

SET IDENTITY_INSERT [Identity].dbo.IdentityUser OFF;
```

Slika 16: SQL naredbe za prijenos podataka

Kako bi finalizirali migraciju modularne monolitne arhitekture u mikroservisnu moramo ovaj postupak ponoviiti za svaki modul. Također valja napomenuti da modularan monolit i mikroservisi mogu koegzistirati. Ako imamo puno modula, normalna je praksa postepeno migrirati module u mikroservise. U tom slučaju imamo hibrid modularnog monolita i mikroservisa, odnosno modularan monolit sa mikroservisima.

4.2. Deployment

Kako bi deployali naše mikroservise, potrebno je prvo izgraditi Docker sliku iz Dockerfile koristeći naredbu: *docker build -t upsell:dev .*

```
1  FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
2  WORKDIR /app
3  EXPOSE 80
4  EXPOSE 443
5
6  FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
7  WORKDIR /src
8  COPY ["MSIdentity.csproj", "MSIdentity/"]
9  RUN dotnet restore "MSIdentity/MSIdentity.csproj"
10
11 WORKDIR "/src/MSIdentity"
12 COPY . .
13
14 RUN dotnet build "MSIdentity.csproj" -c Release -o /app/build
15
16 FROM build AS publish
17 RUN dotnet publish "MSIdentity.csproj" -c Release -o /app/publish /p:UseAppHost=false
18
19 FROM base AS final
20 WORKDIR /app
21 COPY --from=publish /app/publish .
22 ENTRYPOINT ["dotnet", "MSIdentity.dll"]
```

Slika 17: Dockerfile

Nadalje se prijavljujemo u Azure naredbom:

az login

Potrebno je kreirati grupu resursa (*engl. Resource Group*):

az group create -l westeurope -n upsell

Kako bi deployali naše Docker slike potrebno je kreirati Azure Container Registry:

az acr create -g upsell -n upsell cr --sku Basic --admin-enabled

te se u isti ulogirati:

az acr login -n upsell

Zatim je potrebno učitati docker slike u Azure Container Registry:

docker tag msuser:dev upsell.azurecr.io/msuser:v1.0
docker push upsell.azurecr.io/msuser:v1.0

Potom stvaramo service principal odnosno identitet koji ćemo koristiti prilikom uporabe usluga Azur-a:

```
az ad sp create-for-rbac --skip-assignment
```

Nakon toga stvaramo novu ulogu:

```
az acr show --name upsell --resource-group sonukuberg --query "id" --output tsv
az role assignment create --assignee "cfed7ab3-8db9-4bba-86c9-2311d505d2ad" --
role Reader --scope <Put value from above query>
```

Sada stvaramo AKS (Azure Kubernetes Service) kluster:

```
az aks create
--name sonukubeakscluster
--resource-group sonukuberg
--node-count 1
--generate-ssh-keys
--service-principal "cfed7ab3-8db9-4bba-86c9-2311d505d2ad"
--client-secret "7193c36f-3123-4453-8ee7-2aa7bf071007"
```

Kako bi upravljali klasterom potrebno je instalirati *kubectl:*

```
az aks install-cli
```

Sada je potrebno dobiti kluster vjerodajnice (*engl. cluster credentials):*

```
az aks get-credentials --name sonukubeakscluster --resource-group sonukuberg
kubectl get nodes
```

Nakon toga je potrebno kreirati deployment konfiguracijski file (deployment.yml) unutar kojeg specificiramo kontenjere koje ćemo deployati:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: upsell-deployment
  labels:
    app: upsell
spec:
  replicas: 1 #5
  template:
    metadata:
      name: upsell
      labels:
        app: upsell
    spec:
      containers:
        - name: upsell
          image: upsell.azurecr.io/upsell:v1.0
          imagePullPolicy: IfNotPresent
        - name: msuser
          image: upsell.azurecr.io/msuser:v1.0
          imagePullPolicy: IfNotPresent
        - name: msidentity
          image: upsell.azurecr.io/msidentity:v1.0
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
  selector:
    matchLabels:
      app: upsell
---
apiVersion: v1
kind: Service
metadata:
  name: upsell-pocservice-emp
spec:
  selector:
    app: upsell
  ports:
    - port: 80
  type: LoadBalancer

```

Slika 18: Deployment.yml

Nakon čega je potrebno pozvati slijedeću komandu nebi li deployali kontenjere na AKS:

kubectl create -f deployment.yml

Provjera deploymenta:

kubectl get services

```
NAME                   TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
kubernetes             ClusterIP      10.0.0.1       <none>           443/TCP        7h3m
msidentity-service     LoadBalancer   10.0.4.201     <pending>        80:32238/TCP   6h38m
msuser-service         LoadBalancer   10.0.163.64    51.138.110.122   80:31518/TCP   6h38m
upsell-pocservice-emp  LoadBalancer   10.0.76.37     <pending>        80:31932/TCP   6h37m
upsell-service         LoadBalancer   10.0.168.90    51.138.109.184   80:30185/TCP   6h38m
```

Slika 19: Deployani mikroservisi

Iz zadnje naredbe možemo zaključiti da su upsell i msuser mikroservisi uspješno deployani. Nažalost nemožemo reći isto i za identity mikroservis, razlog tome je što besplatna verzija AKS-a ima ograničen broj podova koji možemo koristit. [8]

5. Zaključak

U ovom završnom radu razmatrali smo evoluciju arhitekture aplikacija, s posebnim naglaskom na prijelaz iz monolitne arhitekture u modularne monolite i mikroservisnu arhitekturu. Monolitne aplikacije nude brz razvoj i jednostavnu implementaciju, ali se suočavaju s izazovima održavanja i skaliranja kako rastu i postaju složenije. Modularni monoliti predstavljaju prijelazni korak između tradicionalnih monolita i mikroservisa, omogućujući organizacijama da postupno moderniziraju svoje postojeće aplikacije.

U kontekstu modularnih monolita, analizirali smo implementaciju koristeći ASP.NET Core. Svaki modul je organiziran kao zasebna klasna biblioteka, što olakšava razvoj i održavanje. Uz korištenje CQRS obrasca i Mediator obrasca, postigli smo bolju organizaciju i performanse u aplikaciji. EF Core je omogućio pristup podacima s preciznim konfiguracijama za svaki modul.

Prijelaz s modularnih monolita na mikroservise zahtijeva minimalne napore, jer su moduli već organizirani kao zasebni projekti. Migracija podataka i servisa u novi mikroservis relativno je jednostavna, a RabbitMQ se koristi za komunikaciju između modula/mikroservisa.

Napomena je da RabbitMQ nije nužan za komunikaciju između modula u modularnom monolitu, ali je ključan kao međukorak u tranziciji prema mikroservisima. Ovaj pristup omogućuje organizacijama postupno usvajanje mikroservisne arhitekture bez potrebe za radikalnim promjenama.

Uz praktičan prikaz koraka za implementaciju, migraciju i deploy, ovaj završni rad pruža uvid u modernizaciju arhitekture aplikacija i olakšava prilagodbu rastućim zahtjevima i tehnološkim promjenama. S modularnim monolitima i mikroservisima, organizacije imaju fleksibilnost i skalabilnost potrebnu za uspjeh u današnjem digitalnom svijetu.

6. Literautra

[1] Rahul Awati | Ivy Wigmore, techtarget, Monolithic architecture, https://www.techtarget.com/whatis/definition/monolithic-architecture, svibanj 2022

[2] Benji Vesterby, Understanding Monolithic Architectures: Benefits, Scaling, and Pain Points, https://benjiv.com/understanding-monolithic-architecture/#:~:text=to%20this%20process.-,How%20do%20you%20scale%20a%20monolith%3F,them%20behind%20a%20load%20balancer, 10. ožujka 2021

[3] Medium, Mehmet Ozkaya, Microservices Killer: Modular Monolithic Architecture, https://medium.com/design-microservices-architecture-with-patterns/microservices-killer-modular-monolithic-architecture-ac83814f6862, 16. veljače 2023

[4] Redhat, Bob Reselman, The pros and cons of the Strangler architecture pattern, https://www.redhat.com/architect/pros-and-cons-strangler-architecture-pattern#:~:text=The%20Strangler%20pattern%20is%20one,system%20pass%20through%20the%20facade, 27. svibanj 2021

[5] microservices.io, Chris Richardson, https://microservices.io/

[6] tutorialspoint,, Microservice Architecture – Scaling, https://www.tutorialspoint.com/microservice_architecture/microservice_architecture_scaling.htm#:~:text=Scaling%20is%20a%20process%20of,advance%20features%20of%20the%20application

[7] RabbitMq, Introduction, https://www.rabbitmq.com/tutorials/tutorial-one-python.html

[8] Sonu Kumar, Deploy your MicroService to Azure Container Services (AKS), https://sksonudas.medium.com/deploy-your-first-service-to-azure-container-services-aks-de5ed3e9ff31, 24. travnja 2020.

7. Popis Slika

Slika 1: Monolitna arhitektura 2
Slika 2: Modularna monolitna arhitektura 3
Slika 3: Modularna monolitna arhitektura – folder struktura 4
Slika 4: Monolitna arhitektura – folder struktura 4
Slika 5: Program.cs 5
Slika 6: Clean architecture 6
Slika 7: CQRS Query 7
Slika 8: CQRS Command 8
Slika 9: Controller 9
Slika 10: CQRS & Mediator 9
Slika 11: DbContext 10
Slika 12: RabbitMq producer 11
Slika 13: RabbitMq Message 11
Slika 14: RabbitMq consumer 12
Slika 15: Mikroservisna arhitektura 14
Slika 16: SQL naredbe za prijenos podataka 15
Slika 17: Dockerfile 16
Slika 18: Deployment.yml 18
Slika 19: Deployani mikroservisi 19