


Razvoj primijenjene programske potpore

14. Command-Query Separation

Uslojavanje programskog koda (1)

- Primjer:  MVC \ Controllers \ *Controller.cs
 - Upravljači iz primjera vrše prihvata ulaznih argumenata, dohvat podataka i pripremu modela
 - Problem je u načinu dohvat podataka
 - umjesto na „što“, upravljač u primjeru fokusiran na „kako“ (slaganje EF upita)
 - „prepametn“ upravljač → debeli klijent
 - Što ako umjesto EF-a treba koristiti neku drugu tehniku pristupa podacima?
 - Hoće li entiteti i dalje biti isti?
 - ...i hoće li ih biti ako se ne koristi ORM?
 - Što ako su podaci agregirani iz više izvora?
 - Što ako želimo dodati neko zajedničko ponašanje na svim mjestima (validacija prije snimanja, praćenje traga i slično)
 - ...

```
public IActionResult Index(int page = 1, int sort = 1,
{
    int pagesize = appData.PageSize;
    var query = ctx.Artikl.AsNoTracking();
    int count = query.Count();

    var pagingInfo = new PagingInfo
    {
        CurrentPage = page,
        Sort = sort,
        Ascending = ascending,
        ItemsPerPage = pagesize,
        TotalItems = count
    };
    if (page < 1)
    {
        page = 1;
    }
    else if (page > pagingInfo.TotalPages)
    {
        return RedirectToAction(nameof(Index), new { page = 1 });
    }

    System.Linq.Expressions.Expression<Func<Artikl, obje
    switch (sort)
    {
        case 1:
            orderSelector = a => a.SlikaArtikla; //ima smisl
            break;
        case 2:
            orderSelector = a => a.SifArtikla;
            break;
        case 3:
            orderSelector = a => a.NazArtikla;
            break;
        case 4:
            orderSelector = a => a.JedMjere;
            break;
        case 5:
            orderSelector = a => a.CijArtikla;
            break;
        case 6:
            orderSelector = a => a.ZastUsluga;
            break;
    }
    if (orderSelector != null)
    {
        query = ascending ?
            query.OrderBy(orderSelector) :
            query.OrderByDescending(orderSelector);
    }

    var artikli = query
        .Select(a => new ArtiklViewModel
        {
            SifraArtikla = a.SifArtikla,
            NazivArtikla = a.NazArtikla,
            JedinicaMjere = a.JedMjere,
```

Uslojavanje programskog koda (2)

- Dio problema riješen
 - odvajanjem koda za sortiranje u posebne metode, ali i dalje ostaje problem vezanosti za tehnologiju
 - pogledima koji koriste prezentacijske modele
- ... ali ostaje problem vezanosti za tehnologiju (EF i relacijske baze podataka)
- Ideja: Apstrahirati pristup podacima tako da aplikacija ovisi o sučeljima kojima se definira interakcija sa slojem pristupa podacima
 - Izdvojiti poslovna pravila i složenu validaciju

Repozitoriji umjesto konkretnog ORM-a? (1)

- Zamjenom ORM alata može se pretpostaviti da će i dalje postojati isti koncepti i entiteti u neznatno izmijenjenom obliku
 - entiteti su posljedica modela baze podataka
- Konkretni objekti tipa `DbSet<T>` iz EF-a mogu se zamijeniti sučeljima s postupcima za CRUD operacije - *repozitoriji*
 - kontekst iz EF-a se mijenja sučeljem koje koordinira više repozitorija i predstavlja jedinstveni kontekst pristupa podacima - *Unit of Work*
 - ako se i ne koristi ORM alat, mogu postojati repozitoriji i *Unit of Work*
 - u tom slučaju entiteti su zamijenjeni razredima kojima se opisuju izlazni podaci iz postupaka
- Upravljači tada ovise o sučeljima repozitorija, a konkretna implementacija se umetne tehnikom *Dependency Injection*
 - u ovom slučaju rješava se samo dio problema
 - skriva način perzistencije objektnog modela u relacijsku bazu i djelomično olakšava testiranje programa
 - moguće napisati testnu implementaciju repozitorija

Repozitoriji umjesto konkretnog ORM-a? (2)


- Nije promijenjen način rukovanja repozitorijima iz upravljača
 - prethodno prikazani upravljač bi se neznatno promijenio – i dalje bi slagao upit, ali ovaj put nad nekim nepoznatim repozitorijem
 - Koji je tip povratne vrijednosti kod operacija čitanja podataka?
 - `IQueryable<T>` bi omogućio daljnje upite (filtriranje, sortiranje, ...)
 - Iz čega je nastao taj `IQueryable`? Što sve podržava?
 - `IEnumerable<T>` - nije li možda upit već evaluiran, pa su svi podaci morali biti dovučeni u memoriju?
- Što ako podaci nisu u bazi podataka ili su agregirani iz više izvora?
 - kako dodatno oblikovati upit po želji?
- Bilo bi dobro kad bi upravljač pozivao jedan postupak koji bi mu vratio upravo one podatke koje treba
 - krivi smjer rješenja: proširiti repozitorije s dodatnim metodama
 - traži po nekoj vrijednosti, po kombinaciji vrijednosti, vrati samo dio složene po nekom kriteriju, ...
 - stvara prevelike repozitorije koje nije lako implementirati i otežava održavanje i testiranje

Odvajanje upita od naredbi

- Više upita u istom sučelju narušava SOLID principe
 - *Single Responsibility, Open/Closed i Interface Segregation Principle*
- *Command-query separation*
 - odvojena sučelja za čitanje podataka (upiti, engl. *queries*) od onih koji mijenjaju podatke (naredbe, engl. *commands*)
 - ovisno o rješenju mogu koristiti različita spremišta
 - u implementaciji i naredbe i upiti mogu koristiti repozitorije

→ Svaki upit predstavljen jednim sučeljem

Opisi upita

- „Upit” specificira tip rezultata tog upita za neke ulazne podatke
 - ne mora nužno imati argumente (svojstva)
 - može se opisati generičkim sučeljem
 - Primjer:  CommandQueryCore \ IQuery.cs

```
public interface IQuery<TResult> {}
```

- Primjeri opisa upita

 CommandQuerySample \ Contract\ Queries \ ...

- opis upita koji treba vratiti broj mjesta ovisno o traženom tekstu

```
public class MjestoCountQuery : IQuery<int> {  
    public string SearchText { get; set; }  
}
```

- opis upita koji vraća podatke o mjestu s određenim identifikatorom


```
public class MjestoQuery : IQuery<DT0s.Mjesto> {  
    public int Id { get; set; }  
}
```

Objekti koji „putuju” kroz slojeve

- DTO – Data Transfer Objects
- Čisti, podatkovni razredi koji služe za prijenos podataka između slojeva
- U jednostavnim primjerima dolazi do dupliciranja istih razreda, ali potrebno zbog potencijalnih promjena u budućnosti
 - promjene naziva u bazi podataka ili vrste spremišta ne smije utjecati na opise podataka koji su dogovoreni s konzumentima web-servisa
 - može se koristiti *AutoMapper* ili neki drugi alat za automatsko kopiranje vrijednosti iz objekta jednog tipa u objekt drugog tipa
 - posebno praktično ako su nazivi svojstava isti
 - Preslikavanja postavljena u zasebnim klasama (pri inicijalizaciji AutoMappera dovoljno navesti neku klasu iz projekta)

```
services.AddAutoMapper(typeof(Startup),  
                        typeof(Util.ApiModelsMappingProfile));
```



Primjer složenijeg upita

- Primjeri opisa upita  Contact \ Queries \ ...
 - opis upita koji treba vratiti podskup mjesta poredanih po određenim atributima i u ovisnosti o postavljenjem tekstu pretrage

```
public class MjestaQuery : IQuery<IEnumerable<DTOs.Mjesto>> {  
    public string SearchText { get; set; }  
    public int? From { get; set; }  
    public int? Count { get; set; }  
    public SortInfo Sort { get; set; }  
}
```

```
public class SortInfo {  
    public enum Order {  
        ASCENDING, DESCENDING  
    }  
    public List<KeyValuePair<string, Order>> ColumnOrder  
        { get; set; } = new List<KeyValuePair<string, Order>>();  
    ...  
}
```


Rukovatelj upitom

- Upit (engl. *query*) opisan razredom *IQuery<TResult>* bit će izvršen u nekom rukovatelju upita (engl. *query handler*)
- Sučeljem se standardizira kako rukovatelji upita izgledaju
 - Primjer:  CommandQueryCore \ IQueryHandler.cs

```
public interface IQueryHandler<TQuery, TResult>
    where TQuery : IQuery<TResult> {
    Task<TResult> Handle (TQuery query);
}
```

- Upit predstavlja opis upita (ulazne podatke i povratnu vrijednost), a rukovatelj izvršava tako opisani upit
 - preciznije, sučelje propisuje implementaciju rukovatelja određenim upitom

Primjeri rukovatelja upitom

- Primjeri sučelja u  Contract \ QueryHandlers \ ...
- Opis rukovatelja upitom za dohvat broja mjesta
 - obrađuje upit postavljen razredom MjestoCountQuery i mora isporučiti cijeli broj kao rezultat

```
public interface IMjestoCountQueryHandler :  
    IQueryHandler<MjestoCountQuery, int> { }
```

- Opis rukovatelja upitom za dohvat mjesta s određenom oznakom
 - obrađuje upit postavljen razredom MjestoQuery i mora isporučiti podatkovni objekt s podacima o traženom mjestu

```
public interface IMjestoQueryHandler :  
    IQueryHandler<MjestoQuery, DTOs.Mjesto> { }
```

Primjeri implementacije rukovatelja upitom

- Primjer sučelja  DAL \ QueryHandlers \ ...
 - dohvaća konkretno mjesto i vraća odgovarajući DTO

```
public class MjestaQueryHandler : IMjestaQueryHandler {  
    private readonly FirmaContext ctx;  
    public MjestaQueryHandler(FirmaContext ctx) {  
        this.ctx = ctx;  
    }  
    public async Task<DTOs.Mjesto> Handle(MjestoQuery query)  
    {  
        var mjesto = await ctx.Mjesto  
            .Where(m => m.IdMjesta == query.Id)  
            .Select(m => new DToS.Mjesto {  
                IdMjesta = m.IdMjesta ...  
            })  
            .FirstOrDefaultAsync();  
  
        return mjesto;  
  
        ...  
    }  
}
```


Korištenje rukovatelja upitom iz upravljača

- Upravljač ovisi o sučelju, a konkretnu implementaciju rukovatelja upitom prima u konstruktoru preko DI-a

- Primjer  ... \ WebServices \ Controllers \ MjestoController.cs

```
public class MjestoController ...
    public MjestoController(IMjestoQueryHandler mjestoQueryHandler,
                           IMjestaQueryHandler mjestaQueryHandler,
                           IMjestaCountQueryHandler mjestaCountQueryHandler,
    ...
        this.mjestoQueryHandler = mjestoQueryHandler;
        ...
    public async Task<ActionResult<Mjesto>> Get(int id) {
        var query = new MjestoQuery { Id = id };
        var mjesto = await mjestoQueryHandler.Handle(query);
        if (mjesto == null)
            return Problem(statusCode: StatusCodes.Status404NotFound,
                           detail: $"No data for id = {id}");
        else
            return mjesto;
```

Postavljanje ovisnosti

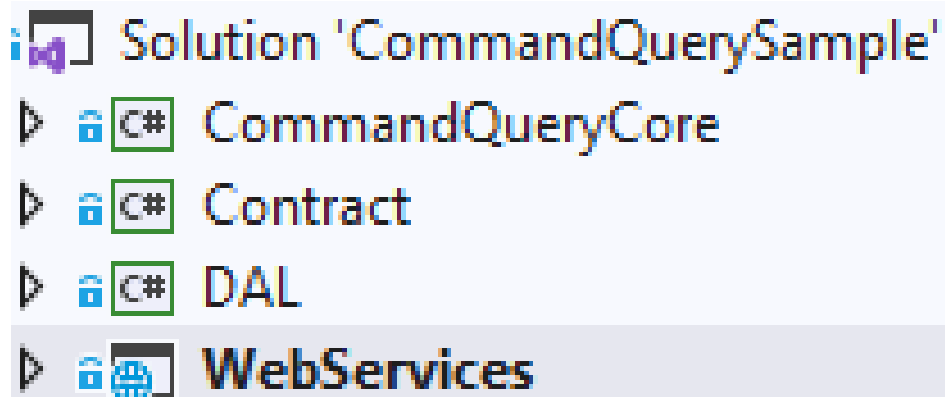
- Povezivanje za DI postavljano u razredu Startup web-aplikacije
 - Primjer  CommandQuerySample \ WebServices \ Startup.cs

```
public class Startup {  
    public void ConfigureServices(IServiceCollection services) {  
        ...  
        services.AddTransient<IMjestaQueryHandler,  
                                MjestaQueryHandler>();  
        services.AddTransient<IMjestoQueryHandler,  
                                MjestoQueryHandler>();  
        services.AddTransient<IMjestoCountQueryHandler,  
                                MjestoCountQueryHandler>();  
        ...  
    }  
}
```


- Umjesto repetitivnog pisanja može se riješiti refleksijom tražeći i registrirajući sve klase koje implementiraju neki *IQueryHandler<, >* iz *typeof(ArtiklQueryHandler).Assembly*

Međusobne ovisnosti projekata

- Smisao pojedinog projekta
 - *CommandQueryCore* – generička sučelja za upit, naredbu i njihove rukovatelje
 - *Contract* – opis svih upita koji se koriste u rješenju i pomoćni razreda koji služe za razmjenu podataka s pozivateljem
 - Data transfer objekti (Value object) – sadrže samo vrijednosti
 - *DAL* – implementacija postupaka iz Firma.DataContract
- Nijedan element web-aplikacije (upravljači, pogledi, ...) osim razreda Startup ne zna za projekt DAL i ovise samo o projektu Contract
 - Startup.cs postavlja Dependency Injection pa stoga mora postojati referenca iz projekta WebServices na DAL



Naredbe (1)

- „Naredba” predstavlja podatke koje neki rukovatelj akcijom treba zaprimiti i odraditi
 - rukovatelja se može se opisati generičkim sučeljem
 - Primjer:  CommandQueryCore \ ICommandHandler.cs

```
public interface ICommandHandler<TCommand> {  
    Task Handle(TCommand command);  
}
```

- Sama naredba je podatkovni objekt, a ne neka akcija
 - Primjer:  Contract \ Commands \ UpdateMjesto.cs

```
public class UpdateMjesto {  
    public int IdMjesta { get; set; }  
    public string NazivMjesta { get; set; }  
    public int PostBrojMjesta { get; set; }  
    public string OznDrzave { get; set; }  
    public string PostNazivMjesta { get; set; }  
}
```


Naredbe (2)

- Smije li naredba vraćati vrijednost? – predmet diskusija

- Primjer:  CommandQueryCore \ ICommandHandler.cs


```
public interface ICommandHandler<TCommand, TKey> {  
    Task<TKey> Handle(TCommand command);  
}
```

- Primjer:  Contract \ Commands \ AddMjesto.cs

```
public class AddMjesto {  
    public string NazivMjesta { get; set; }  
    public int PostBrojMjesta { get; set; }  
    public string OznDrzave { get; set; }  
    public string PostNazivMjesta { get; set; }  
}
```


- Kasnije definirana referenca tipa
ICommandHandler<AddMjesto, int>

Implementacija rukovatelja naredbom

- Implementacija naredbe u podatkovnom sloju
 - Primjer  DAL \ CommandHandlers \ MjestoCommandHandler.cs
 - Može se razdvojiti na 3 rukovatelja, ali nije praktično

```
public class MjestoCommandHandler : ICommandHandler<DeleteMjesto>,
ICommandHandler<AddMjesto, int>, ICommandHandler<UpdateMjesto> {
    private readonly FirmaContext ctx;
    public MjestoCommandHandler(FirmaContext ctx) {
        this.ctx = ctx;
    }
    public async Task<int> Handle(AddMjesto command) {
        var mjesto = new Mjesto {
            NazMjesta = command.NazivMjesta,
            ...
        };
        ctx.Add(mjesto);
        await ctx.SaveChangesAsync();
        return mjesto.IdMjesta; ...
    }
}
```

Korištenje rukovatelja upitom iz upravljača

- Upravljač ovisi o sučelju, a konkretnu implementaciju rukovatelja upitom prima u konstruktoru preko DI-a
 - Primjer  ... \ WebServices \ Controllers \ MjestoController.cs

```
public class MjestoController ...
    public MjestoController(IMjestoQueryHandler mjestoQueryHandler,
        ICommandHandler<AddMjesto, int> addMjestoCommandHandler,
        ...

    public async Task<IActionResult> Create(Mjesto model) {
        AddMjesto command = mapper.Map<AddMjesto>(model);
        int id = await addMjestoCommandHandler.Handle(command);

        var addedItem = await mjestoQueryHandler.Handle(
            new MjestoQuery { Id = id });



        return CreatedAtAction(nameof(Get), new { id }, addedItem);
    }
```

Dodatne zanimljivosti u projektu

- Primijetiti da je izvedena validacija naredbi za dodavanje i ažuriranje mjesta
 - Dodatno u odnosu na validaciju DTO-a
 - Ovom validacijom se provjera jedinstvenost para (pbr, država) neovisno o načinu izvedbe podatkovnog sloja
- Razred ValidateCommandBeforeHandle služi kao dekorator postojećeg rukovatelja nekom naredbom na način da prvo izvrši validaciju naredbe
 - Upravljači ovoga nisu svjesni – ovise samo u sučelju koje opisuje traženi rukovatelj naredbom

```
services.AddTransient<ICommandHandler<AddMjesto, int>,  
    ValidateCommandBeforeHandle<AddMjesto, int,  
        MjestoCommandHandler>>());
```

Nedostatci pristupa

- Zamorno registriranje ovisnosti velikog broj rukovatelja
- Previše složeno za male projekte
- Paradoksalno, ali ponekad teško za testiranje
- Constructor over-injection
 - Razred radi previše toga?
 - Koristiti umetanje u pojedinoj akciji s [FromServices]?
 - Rješenje: obrazac *Medijator*
 - <https://github.com/jbogard/MediatR>
 - Konstruktori ovise o medijatoru, a trivijalno se registriraju svi rukovatelji iz nekog projekta
 - Primjer  CqsWithMediator \ *
 - Moguće definirati akcije koje treba izvesti prije ili poslije rukovanja upitom/naredbom
 - Primjer  ... \ Contract \ Validation \ ValidationPipeline.cs