

1

# Ponovna upotreba i uslojavanje aplikacije

2020/21.10

# Ponovna upotreba softvera

2

- engl. *software reuse*
- Kako već razvijeni softver upotrijebiti za neku drugu svrhu ili neke druge korisnike?
  - potprogrami, komponente
  - oblikovni obrasci
  - aplikacijski okviri
  - standardne programske biblioteke
  - generatori programa
  - web-servisi
  - „gotovi” proizvodi spremni za prilagodbu (npr. SAP)
  - prilagodbe naslijeđenih (baštinjenih, engl. *legacy*) sustava
  - ...

# Podjela ponovne upotrebe po veličini dijelova

3

## ➤ Ponovna upotreba cijelog sustava

- postojećem sustavu se mijenja konfiguracija te se prilagođava novom korisniku uz eventualne manje promjene

## ➤ Ponovna upotreba dijela sustava

- dio postojećeg sustava srednje veličine (komponente razvijene za ponovnu upotrebu, web-servisi opće namjene) se ugrađuje u novi sustav

## ➤ Ponovna upotreba funkcije ili klase

- manji dio postojećeg sustava (određene funkcije i klase iz biblioteka općih namjena) se ugrađuje u novi sustav

# Podjela ponovne upotrebe po načinu ponovne upotrebe

4

- Ponovna upotreba programskog koda ili izvršivog programa
  - već napisani softver uključuje se u novi sustav
  - biblioteke, komponente, web-servisi, nove klase izvedene iz postojećih
- Ponovna upotreba modela ili idejnog rješenja
  - koriste se poznati oblikovni obrasci, iz postojećih UML dijagrama generira se programski kod

- Jedna ili više funkcionalnih komponenti (ili cijeli sustav) s kojim se komunicira putem javno objavljenih i precizno definiranih sučelja
  - mrežno dostupan podatkovni i/ili računalni resurs
  - osigurava mehanizam za pozivanje udaljenih postupaka
  - prima jedan ili više zahtjeva i vraća jedan ili više odgovora
- Princip crne kutije
- Heterogeni klijenti
- Koristi otvorene web standarde:
  - HTTP, XML, JSON, SOAP, OData, GraphQL, gRPC ...
  - ...

- SOA = Servisno orijentirana arhitektura
  - engl. Service Oriented Architecture
  - omogućava izradu distribuiranih aplikacija između različitih platformi koje komuniciraju razmjenom podataka među servisima
- Skup precizno definiranih, međusobno neovisnih servisa povezanih u logički jedinstvenu aplikaciju
  - objektno orijentirana aplikacija povezuje objekte
  - servisno orijentirana aplikacija povezuje servise
- Distribuirani sustav u kojem sudjeluje više autonomnih servisa međusobno šaljući poruke preko granica
  - granice mogu biti određene procesom, mrežom, ...

# Problemi i preporuke prilikom izradu servisa

7

- Sigurnost komunikacije
- Konzistentnost stanja
  - neuspjeh prilikom izvršavanja servisa ne smije ostaviti sustav u stanju pogreške
- Problem višenitnosti
- Pouzdanost i robusnost servisa
  - klijent treba znati je li servis primio poruku.
  - pogreške u servisu treba obraditi
- Interoperabilnost
  - Tko sve može pozvati servis?
- Skalabilnost
- Brzina obrade postupka

# Osnovna pravila servisno orijentirane arhitekture

8

- Jasno određene granice
  - jasno iskazana funkcionalnost i struktura podataka
  - implementacija = crna kutija
- Neovisnost servisa
  - servis ne ovisi o klijentu, nekom drugom servisu, lokaciji i vrsti instalacije
  - verzije se razvijaju neovisno o klijentu. Objavljene verzije se ne mijenjaju.
- Ugovor, a ne implementacija
  - korisnik servisa i implementator servisa dijele samo listu javnih postupaka i definiciju struktura podataka
    - dijeljeni podaci trebaju biti tipovno neutralni
    - tipovi specifični za pojedini jezik moraju se moći pretvoriti u neutralni oblik i obrnuto
  - implementacijski postupci ostaju tajna
- Semantika, a ne samo sintaksa
  - logička kategorizacija servisa, smisljeno imenovanje postupaka



# Standardi za web servise - SOAP

9

## ➤ SOAP – Simple Object Access Protocol

- donedavno *de facto* standard za izradu web servisa i razmjenu informacija u distribuiranim, heterogenim okruženjima  $\approx$  HTTP POST + XML
- orijentiran na akcije (engl. action driven)
- postupak web-servisa koji treba pozvati zapisan u sadržaju zahtjeva

## ➤ WSDL - Web Services Description Language

- XML shema za opis SOAP web-servisa
- definira format postupaka koje pruža Web servis

## ➤ UDDI - Universal Description, Discovery, and Integration

- propisuje način dokumentiranja servisa Discovery (URI i WSDL opisi) koji bi se objavljivali u registracijskim bazama (npr. <http://uddi.xml.org>)
- ideja UDDI napuštena

## ➤ WS-\*

- skup standarda za sigurnost, podatke, opise i koordiniranje SOAP web-servisa

# Standardi za web servise - REST

10

- REST = Representational State Transfer
  - alternativa SOAP protokolu
    - nije protokol, već način izrade servisa (konceptualno ne mora biti baziran nad HTTP protokolom)
  - orijentiran na resurse (engl. *resource driven*)
- REST web-servisi izlažu „resurse”.
  - adresa servisa jednoznačno određuje resurs
  - osmislio Roy Fielding u doktorskoj disertaciji 2000.
- REST web-servisi izlažu „resurse”.
  - Adresa servisa jednoznačno određuje resurs
  - SOA → ROA (*Resource Oriented Architecture*)
- Akcije određene vrstom HTTP zahtjeva
  - GET – dohvat podataka
  - POST - stvaranje novog podatka
  - PUT i PATCH – izmjene cijelog ili dijela podatka
  - DELETE – brisanje podatka

# REST vs SOAP

11

- Slanjem SOAP poruke na pristupnu točku provjerava se sadržaj i na osnovu sadržaja se određuje postupak koji se treba izvršiti
- Korištenjem REST-a postupak se određuje na osnovu url-a i http metode (GET, POST, DELETE, PUT)
- Prednosti REST-a u odnosu na SOAP
  - veći broj potencijalnih klijenata
    - dovoljna je podrška za Http i XML ili JSON
    - nije potrebno implementirati složene WS-\* standarde
  - lakše cacheiranje
  - manje poruke
    - POX (Plain old XML) – XML poruke bez SOAP zaglavlja
    - JSON – (JavaScript Object Notation)
- Nedostatak : nema wsdl-a, veća mogućnost nepravilne poruke
  - praktično se rješava alatima za dokumentiranje servisa (Swagger/OpenAPI) i generatorima klijentskih razreda

## ➤ GET: Dohvat podataka

- Dohvat svih podataka i/ili jednog podatka po primarnom ključu

- (Opcionalno) dohvat na osnovu kriterija pretrage

- Vraća rezultat ili HTTP status 404

## ➤ POST: Kreiranje novog podatka

- Za uspješno stvoreni podatak vraća HTTP status 201 (te često lokaciju stvorenog resursa i sam resurs)

## ➤ PUT/PATCH: Ažuriranje cijelog podatka (PUT) ili dijela podatka (PATCH)

- Nakon uspješnog ažuriranja vraća HTTP status 200 ili 204

## ➤ DELETE: Brisanje podatka

- Ako je podatak uspješno obrisani ili je već ranije bio obrisani vraća HTTP status 200 ili 204

## ➤ U kompletu čini serverski Web API

# Richardsonov model zrelosti

13

- 4 nivoa zrelosti servisa prema ograničenjima REST-a
  - Leonard Richardson 2008.
  - <https://martinfowler.com/articles/richardsonMaturityModel.html>
- Nivo 0: RPC/RPI (Remote Procedure Call/Invocation) preko HTTP-a
- Nivo 1: Izlaganje resursa umjesto metoda
  - interakcija vezana za konkretni resurs (URL identificira resurs)
- Nivo 2: Pravilna upotreba vrsta HTTP zahtjeva, statusa i sadržaja odgovara (npr. lokacija dodanog resursa)
  - Napomena: WebAPI iz primjera koji slijedi pripada ovom nivou
- Nivo 3: Odgovor sadrži hipermedijske elemente (npr. poveznice na ostale resurse i akcije koje se mogu napraviti s tim resursom)
  - Hypermedia as the Engine of Application State (HATEOAS)
  - omogućavaju samostalno otkrivanje ostalih resursa/mogućnost servisa
- Kreator REST-a Roy Fielding smatra da je nivo 3 preduvjet da bi se nešto nazivalo REST servisom
  - <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

# Standardi za web servise - OData

14

- OData – Open Data Protocol
  - Standard za izgradnju RESTful API-a
  - Standardizira oblik adrese pojedinog REST servisa
  - Definira sintaksu za oblikovanje/filtriranje rezultata
  - Nudi informacije o entitetima s kojima pojedini servis radi
- Za detalje o standardu pogledati na <http://www.odata.org>
- U nastavku slijedi primjer s WebAPI-em koji bi pripadao nivou 2 Richardsonog modela zrelosti

# Kostur Web API upravljača

15

```
[Route("api/[controller]")]
public class ValuesController : BaseController {
    // GET: api/values
    [HttpGet]
    public IEnumerable<string> Get() {
        return new string[] { "value1", "value2" };
    }
    // GET api/values/5
    [HttpGet("{id}")]
    public string Get(int id) {
        return "value";
    }
    // POST api/values
    [HttpPost]
    public void Post([FromBody]string value) { }
    // PUT api/values/5
    [HttpPut("{id}")]
    public void Put(int id, [FromBody]string value) { }
    // DELETE api/values/5
    [HttpDelete("{id}")]
    public void Delete(int id) { }
```

# Izvori podataka za argumente Web API servisa

16

- Ispred tipa i naziva argumenta postupka Web API servisa može se navesti neki od atributa
  - [FromBody], [FromForm], [FromHeader], [FromQuery], [FromRoute], [FromServices]
- Za upravljače označene s [ApiController] ne vrijede pravila, tj. redoslijed kao kod upravljača u MVC-u. Ako nije naveden niti jedan, onda se koriste sljedeća pravila
  - jednostavni tipovi (int, string, ...) trebaju biti dio *query stringa*
  - složeni tipovi se nalaze u tijelu zahtjeva
  - Datoteke su dio podataka forme
- **Postupak može imati samo jedan složeni podatak čiji su podaci iz tijela zahtjeva.**
- Detaljnije na <https://docs.microsoft.com/en-us/aspnet/core/web-api/#binding-source-parameter-inference>



# MVC – Web API : Razlike u *Startup.cs*

17

## ➡ Web API


```
public void ConfigureServices(IServiceCollection services) {  
    services.AddControllers() ...  
  
    public void Configure(...) {  
        app.UseEndpoints(endpoints => {  
            endpoints.MapControllers();  
        });  
    }
```

## ➡ MVC

```
public void ConfigureServices(IServiceCollection services) {  
    services.AddControllersWithViews()  
  
    public void Configure(...) {  
        app.UseEndpoints(endpoints => {  
            endpoints.MapDefaultControllerRoute();  
        });  
    }
```

# Primjer Web API servisa – Web API za mjesta

18

- Definira se slično kao i upravljači u MVC-u, ali ima definirano vlastito usmjeravanje, ne vraća pogled već podatke i/ili statusni kod
  - nasljeđuje *ControllerBase*
- Označen atributom *ApiController*
  - **Automatska provjera valjanost modela. Ako model nije valjan rezultat je status 400 - BadRequest**
- Može se implementirati kao debeli klijent ili uslojavanjem aplikacije (opisano uskoro)
  - Primjer:  CommandQuerySample \ WebServices \ Controllers \ MjestoController.cs

```
[ApiController]
[Route (" [controller] ")]
public class MjestoController : ControllerBase ...

...
[HttpGet (" {id} ", Name = "DohvatiMjesto")]
public async Task<ActionResult<Mjesto>> Get (int id)
```

# ActionResult i ProblemDetails

19

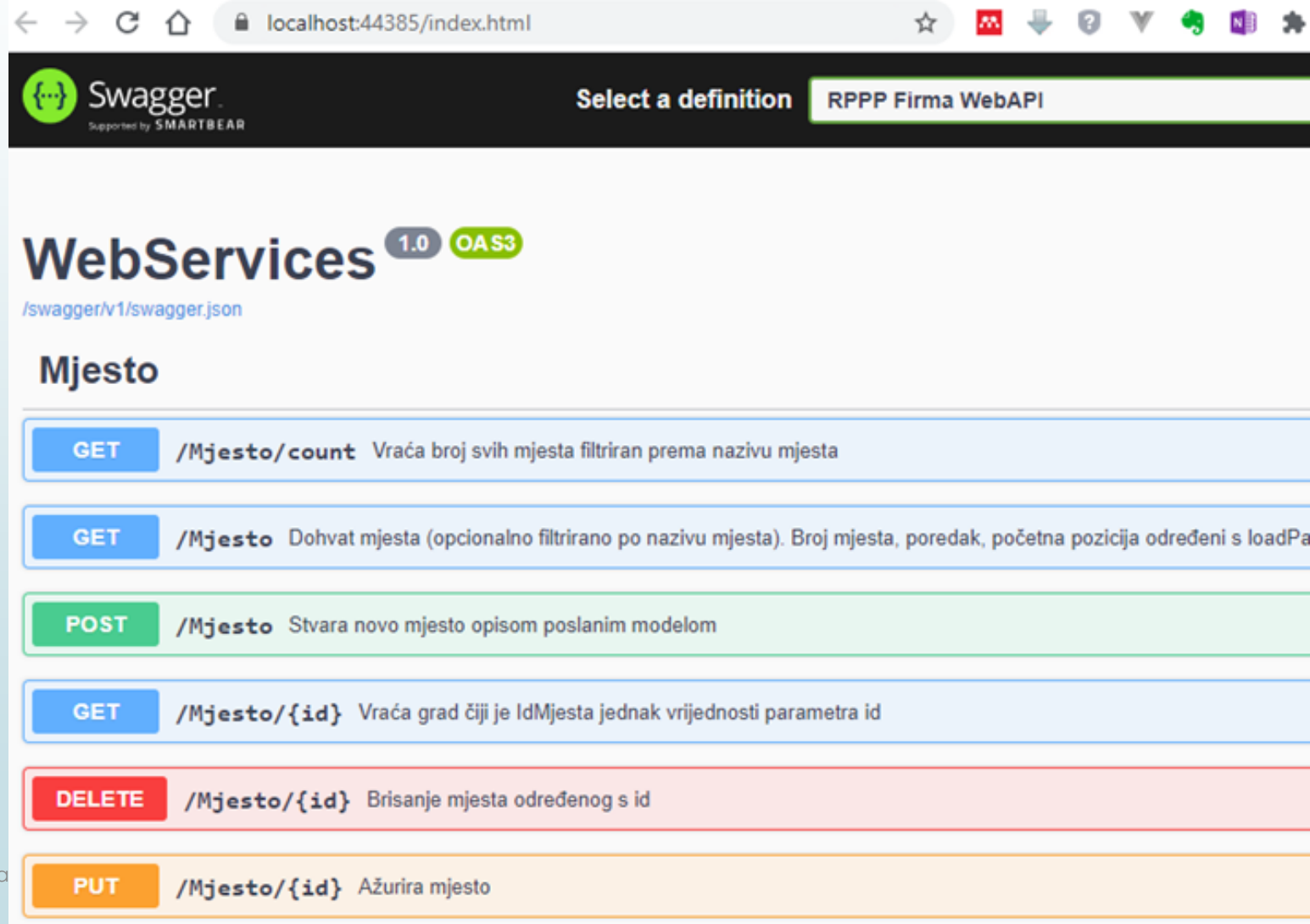
- U prethodnom primjeru postupak je mogao vratiti
  - konkretno mjesto (objekt tipa *Mjesto*) – posljedično status 200
  - status + poruku
    - u takvim slučajevima koristi se povratni tip *ActionResult<T>*
      - definira implicitnu konverziju iz T u *ActionResult<T>*
- Poruke o pogrešci bi trebale biti u skladu s RFC 7807
  - <https://tools.ietf.org/html/rfc7807>
- U primjeru se može koristiti *NotFound()* (obratiti pažnju da treba koristiti varijantu bez argumenata) ili *Problem* koji vraća *ProblemDetails* u skladu s RFC 7807

```
public async Task<ActionResult<Mjesto>> Get(int id) {  
    ...  
    if (mjesto == null)  
        return Problem(statusCode: StatusCodes.Status404NotFound,  
                        detail: $"No data for id = {id}");  
    else return mjesto;  
}
```

# Dokumentacija za web servise


20

- Za dokumentiranje Web API servisa koristi se alat *Swagger*
- Osim informacija o tipovima podataka i rezultata, omogućava i pozivanje servisa



# Aktivacija Swaggera (1)

21

- Dodati NuGet paket Swashbuckle.AspNetCore
- U Startup.cs aktivirati Swagger
  - Swagger automatski pronalazi sve upravljače i attribute Http[Get|Post|...]
    - S ApiExplorerSettings(IgnoreApi = true) moguće izbaciti željene upravljače
  - U postavkama projekta uključiti kreiranje XML dokumentacije
    - Navesti putanje do svih xml datoteka koje treba uključiti
  - Primjer  CommandQuerySample \ WebServices \ WebServices \ Startup.cs

```
public void ConfigureServices(...  
    ...  
    services.AddSwaggerGen(c => {  
        var xmlFile = $"{Assembly.GetExecutingAssembly().  
                                .GetName().Name}.xml";  
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);  
        c.IncludeXmlComments(xmlPath);  
    });  
    ...
```

# Aktivacija Swaggera (2)

22


➡ Primjer  CommandQuerySample \ WebServices \ WebServices \ Startup.cs

```
public void Configure(...  
    ...  
    app.UseSwagger();  
    app.UseSwaggerUI(c => {  
        c.SwaggerEndpoint("/swagger/v1/swagger.json",  
                           "Firma WebAPI");  
        c.RoutePrefix = string.Empty;  
    });  
    ...
```

- ➡ Nakon navedenih postavki automatski generirana dokumentacija za WebApi servise je dostupna na <http://.../route-prefix/>
  - ➡ Npr. <https://localhost:44385/>

# Informacije o statusnim porukama

23

- Swagger pretpostavlja da svi postupci vraćaju status 200
  - U slučaju da nije tako, potrebno eksplicitno navesti moguće vrijednosti iznad postupka
  - Koristi se atribut `ProducesResponseType`
  - Primjer  ... \ WebServices \ Controller \ MjestoController.cs

```
[HttpPut("{id}", Name = "AzurirajMjesto")]  
[ProducesResponseType (StatusCodes.Status204NoContent)]  
[ProducesResponseType (StatusCodes.Status404NotFound)]  
[ProducesResponseType (StatusCodes.Status400BadRequest)]  
public async Task<IActionResult> Update(int id, Mjesto model) {  
    ...  
}
```

- Alternativno, upravljač se može označiti s `[APIConventions]` (uz pretpostavku da se u realizaciji držimo opisane konvencije)

# Informacije o povratnim vrijednostima

24

- Swagger provjerava tip povratne vrijednosti iz servisa
  - određeno s `ActionResult<povratni_tip>`
- Ako je povratna informacija tipa `IAsyncResult` Swagger ne može znati što je povratni tipa, pa se mogu koristiti atributi `ProducesResponseType` i (izvedeni iz njega) `SwaggerResponseAttribute`
  - izvedeni razred sadrži mogućnosti dodavanja opisa

```
[HttpGet("{oznDrzave}", Name = "DohvatiDrzavu")]
[ProducesResponseType(typeof(Drzava), (int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
public async Task<IActionResult> Get(string oznDrzave)

...

[HttpGet]
[SwaggerResponseAttribute((int)HttpStatusCode.OK,
    typeof(IEnumerable<Drzava>),
    Description = "Vraća enumeraciju država")]
public async Task<IEnumerable<Drzava>> Get()
```

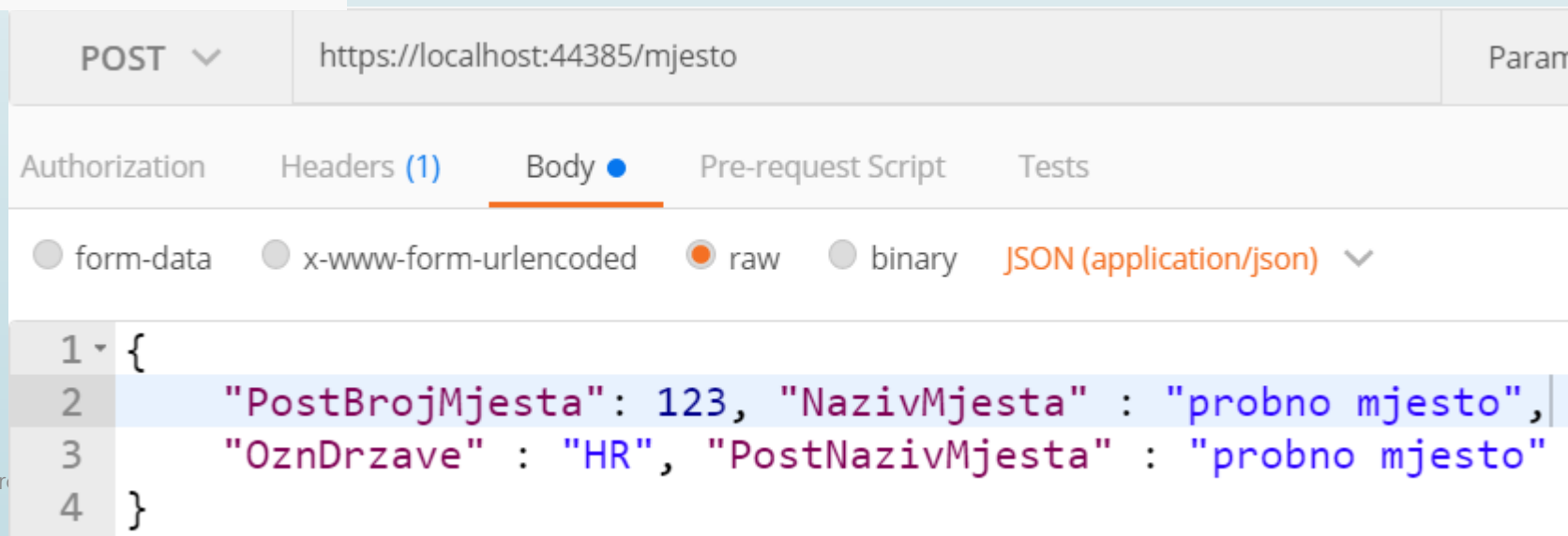
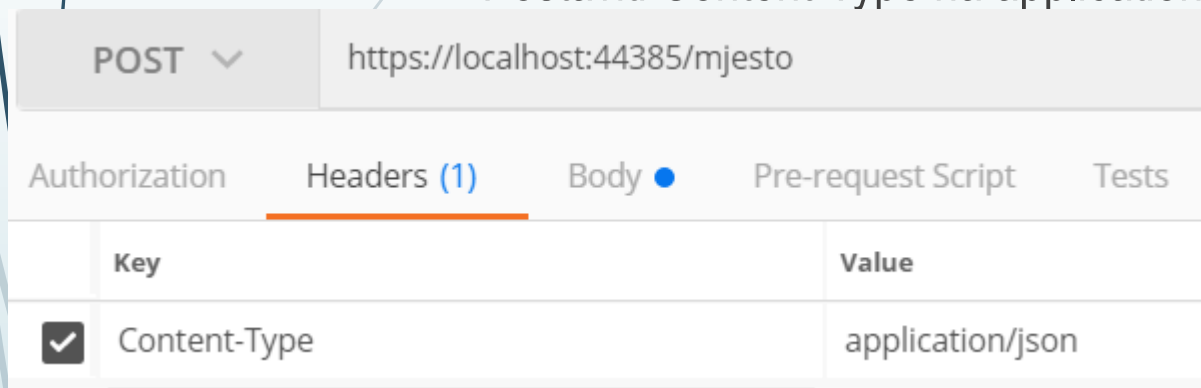


# Kako isprobati POST i ostale zahtjeve?

25

➡ Postman – REST Client (<https://www.getpostman.com/>)

- ➡ Alternativa Fiddler ili slični alati, vlastita konzolna aplikacija, generirani klijent preko Swaggera (o tome malo kasnije)
- ➡ Postaviti Content-Type na application/json i poslati odgovarajući JSON



# Kako isprobati POST i ostale zahtjeve? (2)

26

Body

Cookies

Headers (6)

Test Results

Status: 201 Created

Pretty

Raw

Preview

JSON ▼

1 {

2 "IdMjesta": 36460,

3 "PostBrojMjesta": 123,

4 "NazivMjesta": "probno mjesto",

5 "PostNazivMjesta": "probno mjesto",

6 "OznDrzave": "HR",

7 "NazivDrzave": "Croatia"

8 }

**Content-Type** → application/json; charset=utf-8

**Date** → Mon, 11 Jan 2021 10:26:36 GMT

**Location** → https://localhost:44385/Mjesto/36460

# Kako isprobati POST i ostale zahtjeve? (3)

27

- U slučaju validacijske pogreške, odgovor sadrži poruku o pogrešci i statusni kod 400 (*Bad Request*)
  - Posljedica validacijskih atributa (ili *FluentValidationa*) te atributa [*ApiController*] na upravljaču




The screenshot shows a REST client interface with tabs for Body, Cookies, Headers (5), and Test Results. The status bar indicates 'Status: 400 Bad Request'. The response body is displayed in 'Pretty' format as a JSON object. The JSON object contains a 'type' pointing to an RFC 7231 section, a 'title' describing the error, a 'status' of 400, a 'traceId', and an 'errors' object. The 'errors' object has a 'PostBrojMjesta' property with a message in Croatian: 'Dozvoljeni raspon: 10-60000'.

```
1 {
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "One or more validation errors occurred.",
4   "status": 400,
5   "traceId": "|91f38987-4e468597ec15583c.",
6   "errors": {
7     "PostBrojMjesta": [
8       "Dozvoljeni raspon: 10-60000"
9     ]
10  }
```

# Uslojavanje programskog koda (1)

28

- Primjer:  MVC \ Controllers \ ArtiklController.cs
- U navedenom primjeru upravljač vrši prihvata ulaznih argumenata, dohvat podataka i pripremu modela
  - problem je u načinu dohvat podataka
    - umjesto na „što“, upravljač u prethodnom 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?
    - ...

```
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
    }

    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)

29

- 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)

30

- 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 tako mijenja sa 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, ali konkretna implementacija se umetne tehnikom *Dependency Injection*
  - u ovom slučaju rješava 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)

31

- 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 nekoliko metoda koje primaju različite parametre
    - 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

32

- 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



- „Upit” specificira tip rezultata tog upita za neke ulazne podatke

- ne mora nužno imati argumente (svojstva)

- može se opisati generičkim sučeljem

- Primjer:  CommandQuerySample \ 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<DTOs.Mjesto> {  
    public int Id { get; set; }  
}
```

# Objekti koji „putuju” kroz slojeve

34

- DTO – *Data Transfer Objects*
- Čisti, podatkovni razredi koji služe za prijenos podataka između slojeva, odnosno za prijenos između servisa i njegovih klijenata
  - Klijent i server se moraju dogovoriti oko podataka
- 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 AutoMapper-a dovoljno navesti neku klasu iz projekta)

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

# Primjer složenijeg upita

35

➡ Primjeri opisa upita  CommandQuerySample \ Contract \ Queries \ ...

➡ opis upita koji treba vratiti podskup mjesta poredanih po određenim atributima i u ovisnosti o postavljenom 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

36

- 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: `CommandQuerySample \ 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

37

➤ Primjeri sučelja u  CommandQuerySample \ 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

38

➡ Primjer sučelja  CommandQuerySample \ 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

39

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

➡ Primjer  CommandQuerySample \ 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

40

- 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*

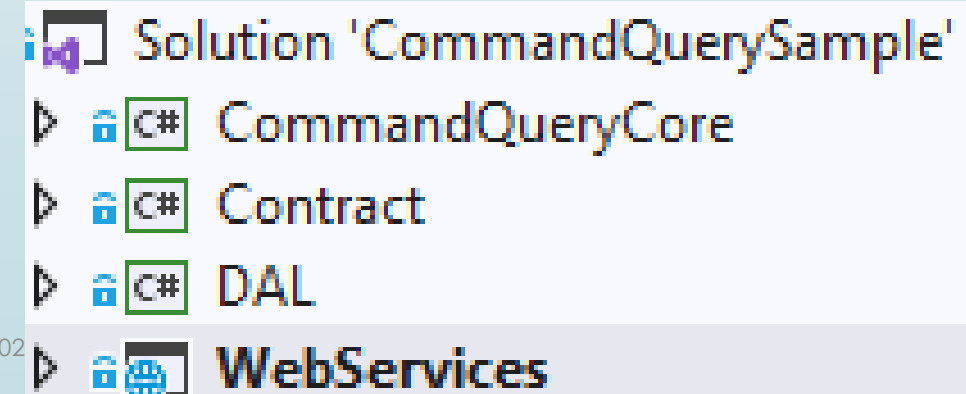


# Međusobne ovisnosti projekata

41

## ➤ 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)

42

➡ „Naredba” predstavlja podatke koje neki rukovatelj akcijom treba zaprimiti i odraditi

➡ rukovatelja se može se opisati generičkim sučeljem

➡ Primjer:  CommandQuerySample \ CommandQueryCore \ ICommandHandler.cs

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

➡ Sama naredba je podatkovni objekt, a ne neka akcija

➡ Primjer:  CommandQuerySample \ 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)

43

➡ 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

44

## ➤ Implementacija naredbe u podatkovnom sloju


➤ Primjer:  ... \ Firma.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; ...
    }
}
```

# Web API – dodavanje mjesta

45


- Postupak POST proizvoljnog imena u kojem se model rekonstruira iz tijela zahtjeva
  - Ako je model ispravan posprema podatak u bazu podataka i vraća 201
    - Uz status vraća pohranjeni podatak i adresu podatka
      - Adresa je dio zaglavlja odgovora, podatak je u tijelu
  - Neispravni model uzrokuje statusnu poruku 400 (BadRequest)
  - Primjer:  CommandQuerySample \ WebServices \ Controllers \ MjestoController.cs

```
[HttpPost(Name = "DodajMjesto")]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
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);
}
```

# Web API – ažuriranje mjesta

46


## ➤ Postupak PUT proizvoljnog imena

- U primjeru se šalje cijeli model, a ne samo parcijalne promjene (PATCH)
- Identifikator mjesta iz zahtjeva i onaj iz modela moraju biti jednaki
- Uspješna promjena podatka vraća statusnu poruku 204
- Neispravni podaci: 400 ili 404 (ako mjesto ne postoji)
- Primjer:  ... WebServices \ Controllers \ MjestoController.cs

```
public async Task<IActionResult> Update(int id, Mjesto model) {  
    if (model.IdMjesta != id)  
        return Problem(statusCode: StatusCodes.Status400BadRequest,  
                        detail: $"Different ids {id} vs {model.IdMjesta}");  
    else {  
        var mjesto = await mjestoQueryHandler.Handle(new MjestoQuery {  
            Id = id  
        });  
        if (mjesto == null)  
            return Problem(statusCode: StatusCodes.Status404NotFound, detail: ...  
UpdateMjesto command = mapper.Map<UpdateMjesto>(model);  
await updateMjestoCommandHandler.Handle(command);  
return NoContent() ...  
    }
```

# Web API – brisanje mjesta

47

- Postupak DELETE proizvoljnog imena sa šifrom države u adresi
- Uspješno brisanje vraća statusnu poruku 204, a za nepostojeće mjesto vraća se 404
- Primjer:  ... WebServices \ Controllers \ MjestoController.cs

```
[HttpDelete("{id}", Name = "ObrisiMjesto")]
public async Task<IActionResult> Delete(int id) {
    var mjesto = await mjestoQueryHandler.Handle(new MjestoQuery {
        Id = id
    });

    if (mjesto == null) {
        return Problem(statusCode: StatusCodes.Status404NotFound,
            detail: $"Invalid id = {id}");
    }
    else {
        await deleteMjestoCommandHandler.Handle(new DeleteMjesto(id));
        return NoContent();
    }
}
```

# Što u slučaju pogreške unutar upravljača?

48




- Pogreška prilikom spremanja podatka?
- Neuhvaćena iznimka?
- ...
- Različiti pristupi
  - „Zabijanje glave u pijesak”
    - pravimo se da se iznimka neće dogoditi
    - Ako se dogodi izaziva statusnu poruku broj 500 (Interval Server Error) koja korisniku ne znači previše, a može otkriti neželjene interne podatke
  - „Sve OK”
    - Čitavi programski kôd servisa omotan u try-catch block, a rezultat je razred koji sadrži omotane podatke koje je trebao vratiti i informaciju o uspješnosti postupka i eventualnoj pogrešci
    - Korisnik uvijek dobiva statusnu poruku 200
  - Nešto treće?



# Primjer korištenja servisa koristeći *jTable*

49

## ➤ Proučiti sljedeće sadržaje

-  WebServices \ WebServices \ wwwroot \ mjesta.html  
<https://localhost:44385/mjesta.html>
-  WebServices \ WebServices \ Controllers \ jTable \ jTableController.cs
-  WebServices \ WebServices \ Controllers \ jTable \ MjestoJTableController.cs
-  WebServices \ WebServices \ Controllers \ jTable \ LookupController.cs
-  WebServices \ WebServices \ ViewModels \ jTable \ \*

## ➤ jTable ne radi s WebAPI servisima direktno, već koristi POST, pa je potreban *wrapper*.

- jTable koristi koncept da je poziv uvijek uspješan (status 200) uz poruku je li akcija uspjela ili ne

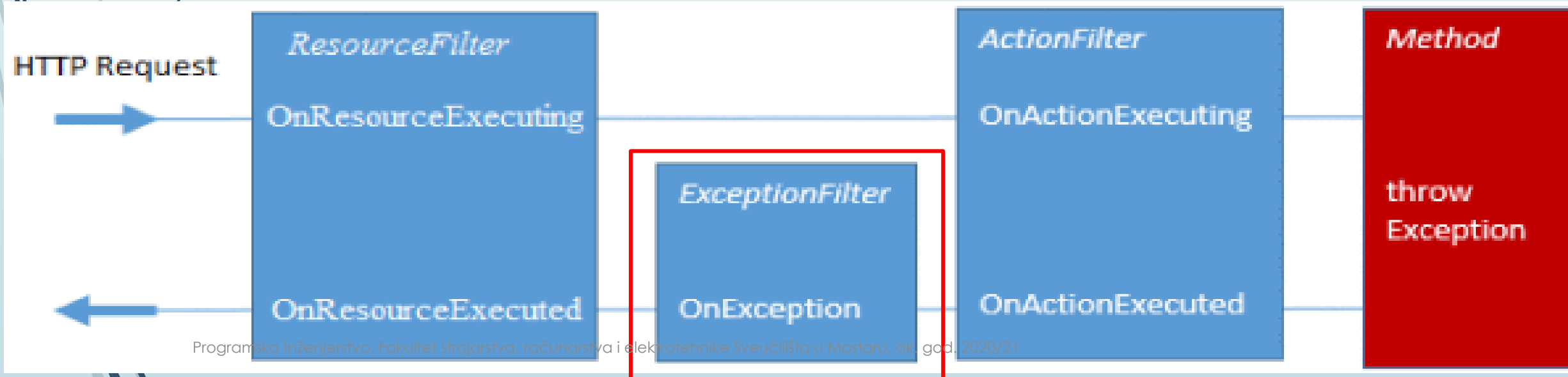
- U slučaju validacijske pogreške *wrapper* ne vraća 400, već 200 uz (primjerice) sadržaj

```
{ "Result": "ERROR",    "Message":  
    "PostBrojMjesta: Dozvoljeni raspon: 10-60000; " }
```

# Obrada iznimke korištenjem atributa

50

- Umjesto napornog pisanja try-catch blokova u svakom postupku (a i za slučaj nepredviđene iznimke), obrada iznimke se može centralizirati pomoću ExceptionFiltera
  - Napisati vlastiti atribut implementiranjem sučelja `IFilter` ili izvođenjem iz razreda `ExceptionFilterAttribute`
  - Koristi se kao tip unutar atributa `TypeFilter`, npr. `[TypeFilter(typeof(ErrorStatusTo200WithErrorMessage))]`
  - Primjeri:
    - 📁 WebServices \ Util \ ExceptionFilters \ ProblemDetailsForSqlException.cs
    - 📁 ... ExceptionFilters \ ErrorStatusTo200WithErrorMessage.cs
    - 📁 WebServices \ Controllers \ Jtable \ JTableController.cs



# Dodatne zanimljivosti u projektu

51

- 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>>() ;
```

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

## ➡ Command-query separation

- ➡ <https://cuttingedge.it/blogs/steven/pivot/entry.php?id=92>
- ➡ <https://www.future-processing.pl/blog/cqrs-simple-architecture/>
- ➡ <https://martinfowler.com/bliki/CQRS.html>