Razvoj softvera 2 – Beleške za vežbe

Nikola Ajzenhamer

Sadržaj

1. Uvod u mikroservisne aplikacije

Tema ovih časova je upoznavanje studenata sa planom kursa i obavezama na kursu, kao i razvijanje jednostavnog mikroservisa sa MongoDB SUBP.

1. O kursu

- Sajt kursa: http://rs2.matf.bg.ac.rs/
- Prezentacija: http://rs2.matf.bg.ac.rs/vezbe/o-kursu.pdf
- O seminarskim radovima: http://rs2.matf.bg.ac.rs/seminarski-radovi/
- Neophodni alati:
    - Visual Studio (Windows), Visual Studio for Mac (OSX), Visual Studio Code (Windows, OSX, Linux), Jetbrains Rider (Windows, OSX, Linux)
    - .NET 5 (Windows, OSX, Linux)
    - Docker Desktop (Windows, OSX), Docker Server (Linux)

2. O mikroservisnim aplikacijama

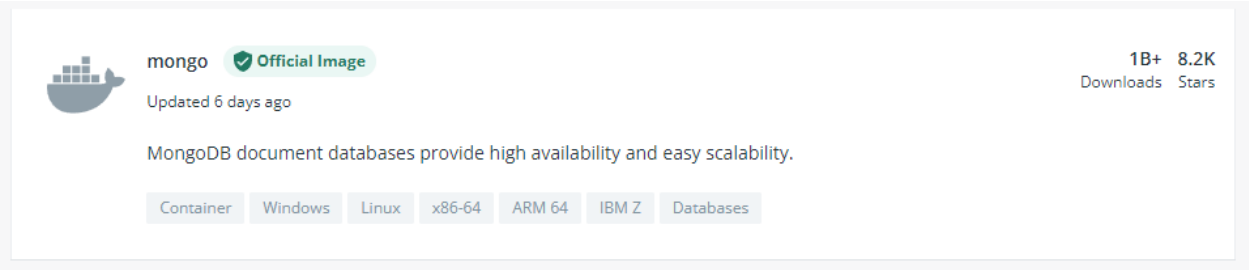
- Ukratko proći kroz tekst: http://rs2.matf.bg.ac.rs/vezbe/ukratko-o-mikroservisima.pdf

3. Razvoj Catalog.API mikroservisa

- Priprema MongoDB kontejnera
- Kreiranje projekta i instalacija paketa
- Razvoj mikroservisa
- Pokretanje i debagiranje aplikacije

Priprema MongoDB kontejnera

Na stranici https://hub.docker.com/ uneti „mongo“ u polje za pretragu. Otvoriti sledeću stranicu:



Osnovni MongoDB pojmovi: baza dokumenata, dokumenti, jedinstveni ključevi (_id), kolekcije, indeksi (nad _id).

Pokretanje mongo kontejnera:

- Otvoriti Powershell
- **docker run --name mongo_catalog -p 27017:27017 -d mongo**

Ove naredbe ispisuju identifikator kontejnera u kojem je pokrenut MongoDB SUBP. Korisne docker naredbe:

- Ispisuje sve pokrenute kontejnere
    - **docker ps**
- Ispisuje sve kontejnere
    - **docker ps -a**
- Ispisuje samo identifikatore pokrenutih kontejnera

    - **docker ps -q**
- Pokreće ugašeni kontejner
    - **docker start IDENTIFIKATOR_KONTEJNERA**
- Zaustavlja pokrenuti kontejner
    - **docker stop IDENTIFIKATOR_KONTEJNERA**
- Uklanja kontejner
    - **docker rm IDENTIFIKATOR_KONTEJNERA**
- Uklanja sve kontejnere sa sistema
    - **docker rm $(docker ps -aq)**
- Kreira novi kontejner na osnovu slike čiji je naziv **IME_SLIKE** na hub.docker.com i pokreće ga
    - **docker run IME_SLIKE**
- Kreira novi kontejner na osnovu slike i pokreće ga, pri čemu mu daje ime **IME_KONTEJNERA**
    - **docker run --name IME_KONTEJNERA IME_SLIKE**
- Kreira novi kontejner na osnovu slike i pokreće ga, pri čemu se vrši preslikavanje portova, tako što se **UNUTRAŠNJI_PORT** preslikava na **SPOLJNI_PORT** u localhost-u
    - **docker run -p SPOLJNI_PORT:UNUTRAŠNJI_PORT IME_SLIKE**
- Kreira novi kontejner na osnovu slike i pokreće ga, ali u pozadini (terminal se ne blokira)
    - **docker run -d IME_SLIKE**
- Čita (i prati, ako se navede opcija **-f**) sve logove za pokrenuti kontejner
    - **docker logs -f IME_KONTEJNERA**
- Pokreće interaktivni terminal u kontejneru. Ovo je korisno za izvršavanje proizvoljnih naredba
    - **docker exec -it IME_KONTEJNERA /bin/bash**

Kada se prikačimo za mongo kontejner, dostupan nam je CLI alat **mongo** kojim možemo izvršavati proizvoljne naredbe za upravljanje mongo bazom. Neke osnovne komande ovog alata su:

- Prikazuje sve baze podataka
    - **show dbs**
- Bira bazu podataka **BAZA_PODATAKA** za koju će se odnositi sve dalje naredbe (odabrana BP biće dostupna kroz objekat **db**)
    - **use BAZA_PODATAKA**
- Čitanje svih dokumenata iz kolekcije **IME_KOLEKCIJE**
    - **db.IME_KOLEKCIJE.find({})**
- Unošenje novog dokumenta u kolekciju **IME_KOLEKCIJE**
    - **db.IME_KOLEKCIJE.insertOne({ name: 'Pera', prezime: 'Perić' })**
- Ažuriranje
    - **db.IME_KOLEKCIJE.updateOne({ name: 'Pera' }, { $set: { izmenjen: true } })**
- Brisanje
    - **db.IME_KOLEKCIJE.deleteOne({ name: 'Pera' })**

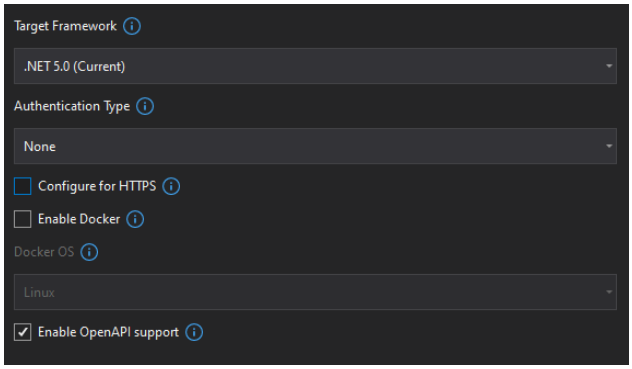
Kreiranje projekta i instaliranje paketa

S obzirom da započinjemo razvoj „od nule“, potrebno je prvo da napravimo jedan *Solution* pre nego što kreiramo bilo koji projekat:

- File > New > Project
- Blank Solution
- Popuniti neophodnim podacima:
    - Solution name: **Webstore**
    - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA**
    - Solution: **Create new solution**
- Create

Sada možemo da kreiramo nove projekte:

- Desni klik na ime *Solution-a*
- Add > New Project
- ASP.NET Core Web API
- Popuniti neophodnim podacima:
    - Project name: **Catalog.API**
    - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Services\Catalog**
- Next
- Odabrati opcije kao na slici pored:
- Create



Pre nego što krenemo sa razvojem, potrebno je da instaliramo neophodne pakete. Otvoriti *NuGet Package Manager* sledećim koracima:

- Desni klik na naziv projekta
- Manage NuGet Packages

Prvo ćemo ažurirati sve pakete koji su do sada instalirani:

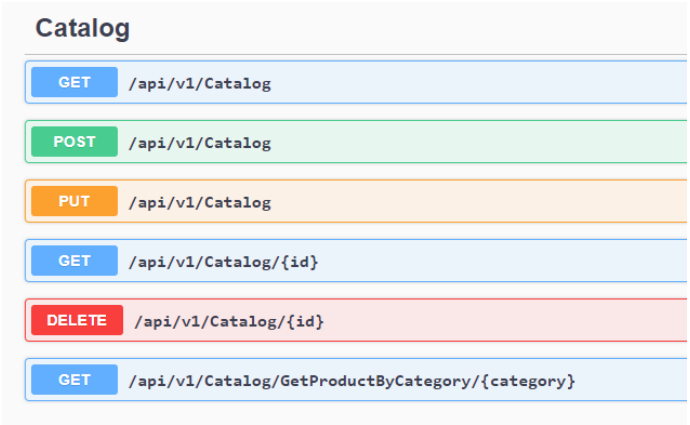
- Updates
- Select all packages
- Update

Zatim je potrebno otvoriti tab „Browse“ i tu pronaći i instalirati sledeće pakete:

- MongoDB.Driver

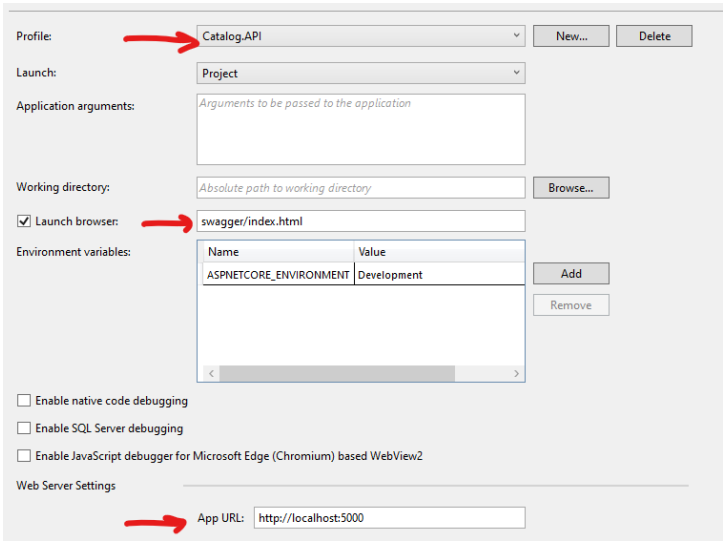
Razvoj mikroservisa

API mikroservisa je opisan slikom pored.



Prvo podešavamo opcije za pokretanje aplikacije:

- Desni klik na naziv projekta
- Properties
- Debug
- Podesiti opcije sa slike ispod.



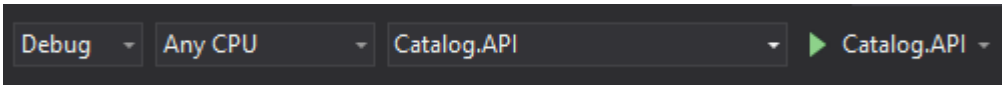
Zatim prelazimo na kodiranje, prema narednom redosledu:

- Entities
  - Product.cs
- Data
  - ICatalogContext.cs
  - CatalogContext.cs
  - CatalogContextSeed.cs
- Repositories
  - IProductRepository.cs
  - ProductRepository.cs
- Controllers
  - CatalogController.cs

Sada je preostalo da dodamo ubrizgavanje zavisnosti, što se nalazi u **Startup.cs** datoteci.

Pokretanje i debagiranje aplikacije

Iz gornjeg menija odabrati naredne opcije i pokrenuti aplikaciju:



Postaviti u nekom zahtevu tačku prekida i prolaziti kroz kod. Prikazivati Visual Studio okruženje za debagiranje.

2. Višestruki mikroservisi. Kontejnerizacija aplikacije.

Tema ovih časova je rad sa Redis, distrubuiranom keš memorijom i kontejnerizacija projekata pomoću Docker i Docker Compose alata.

1. Razvoj Basket.API mikroservisa

Priprema Redis kontejnera

Na stranici https://hub.docker.com/ uneti „redis“ u polje za pretragu. Otvoriti sledeću stranicu:

Osnovni Redis pojmovi: baza ključ-vrednost, keš memorija, prednosti i ograničenja (https://redis.io/topics/faq).

Pokretanje redis kontejnera:

- Otvoriti Powershell
- **docker run -d -p 6379:6379 --name redis_basket redis**

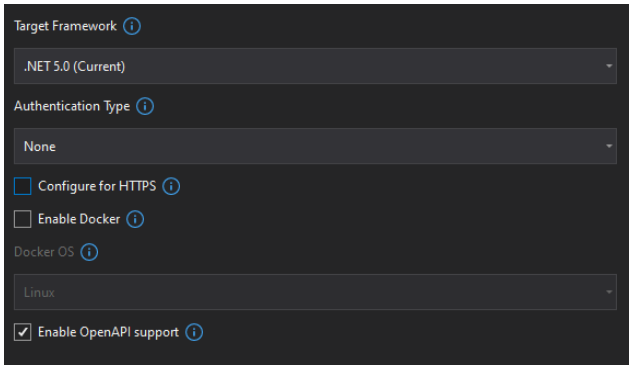
Kada se prikačimo za redis kontejner, dostupan nam je alat **redis-cli** kojim možemo izvršavati proizvoljne naredbe za upravljanje redis bazom. Neke osnovne komande ovog alata su:

- Provera da li je baza spremna (očekuje se odgovor PONG)
  - ping
- Postavljanje vrednosti
  - set **KLJUČ VREDNOST**
- Čitanje vrednosti
  - get **KLJUČ**

Kreiranje projekta i instaliranje paketa

Dodajemo novi projekat u okviru već napravljenog *Solution-a*:

- Desni klik na ime *Solution-a*
- Add > New Project
- ASP.NET Core Web API
- Popuniti neophodnim podacima:
  - Project name: **Basket.API**
  - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Services\Basket**
- Next
- Odabrati opcije kao na slici pored.
- Create



Pre nego što krenemo sa razvojem, potrebno je da instaliramo neophodne pakete. Otvoriti *NuGet Package Manager* sledećim koracima:

- Desni klik na naziv projekta
- Manage NuGet Packages

Prvo ćemo ažurirati sve pakete koji su do sada instalirani:

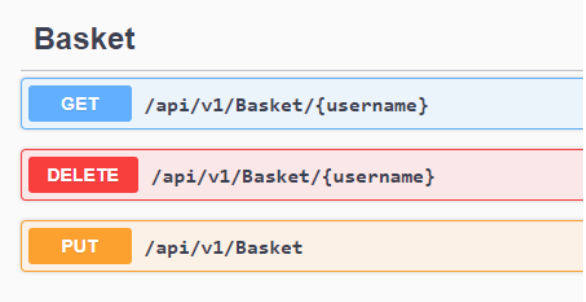
- Updates
- Select all packages
- Update

Zatim je potrebno otvoriti tab „Browse“ i tu pronaći i instalirati sledeće pakete:

- Newtonsoft.Json
- Microsoft.Extensions.Caching.StackExchangeRedis

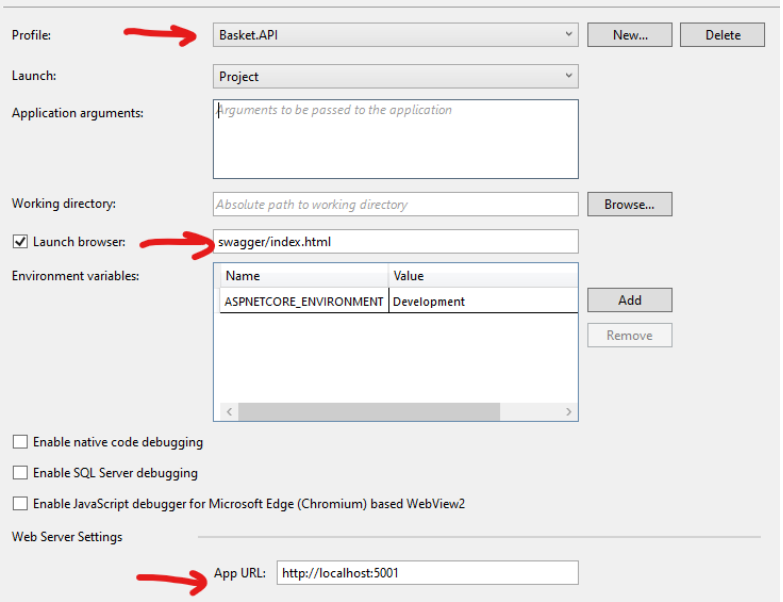
Razvoj mikroservisa

API mikroservisa je opisan slikom pored.



Prvo podešavamo opcije za pokretanje aplikacije:

- Desni klik na naziv projekta
- Properties
- Debug
- Podesiti opcije sa slike ispod.



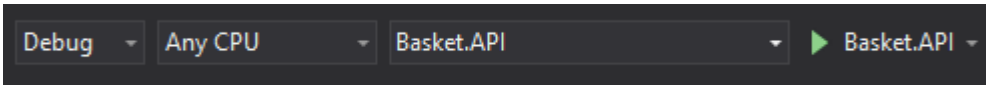
Zatim prelazimo na kodiranje, prema narednom redosledu:

- Entities
  - ShoppingCartItem.cs
  - ShoppingCart.cs
- Repositories
  - IBasketRepository.cs
  - BasketRepository.cs
- Controllers
  - BasketController.cs

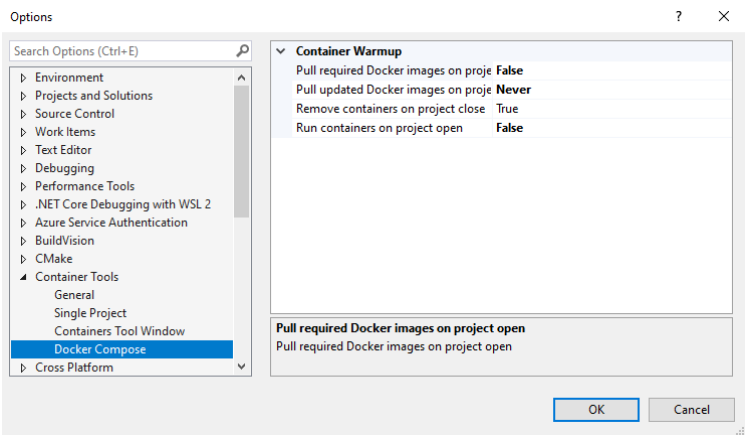
Sada je preostalo da dodamo ubrizgavanje zavisnosti, što se nalazi u **Startup.cs** datoteci.

Pokretanje i debagiranje aplikacije

Iz gornjeg menija odabrati naredne opcije i pokrenuti aplikaciju:



2. Kontejnerizacija projekata



Pre nego što bilo šta uradimo, preporuka je da postavimo naredne opcije u Visual Studio alatu, kako bismo izbegli neka suvišna pokretanja Docker Compose alata:

- Tools > Options
- Otvoriti Container Tools grupu opcija
- Odabrati opciju Docker Compose
- Odabrati opcije sa naredne slike:

Dodavanje podrške za Docker Compose projektu:

- Desni klik na naziv projekta
- Add > Container Orchestrator Support
- Docker Compose
- Ok
- Linux
- Ok

Proći kroz generisani **Dockerfile** i objasniti neke najvažnije elemente, a zatim objasniti **docker-compose.yml** i **docker-compose.override.yml** datoteke.

Dodati naredne resurse u **docker-compose.yml** datoteku:

```
services:
  catalogdb:
    image: mongo
  basketdb:
    image: redis:alpine

volumes:
  mongo_data:
```

Dodati naredne resurse u **docker-compose.override.yml** datoteku:

```
services:
  catalogdb:
    container_name: catalogdb
    restart: always
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

  basketdb:
    container_name: basketdb
    restart: always
```

```
  ports:
    - "6379:6379"

  catalog.api:
    container_name: catalog.api
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - "DatabaseSettings:ConnectionString=mongodb://catalogdb:27017"
    depends_on:
      - catalogdb
    ports:
      - "8000:80"

  basket.api:
    container_name: basket.api
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - "CacheSettings:ConnectionString=basketdb:6379"
    depends_on:
      - basketdb
    ports:
      - "8001:80"
```

Pokretanje projekta iz komandne linije

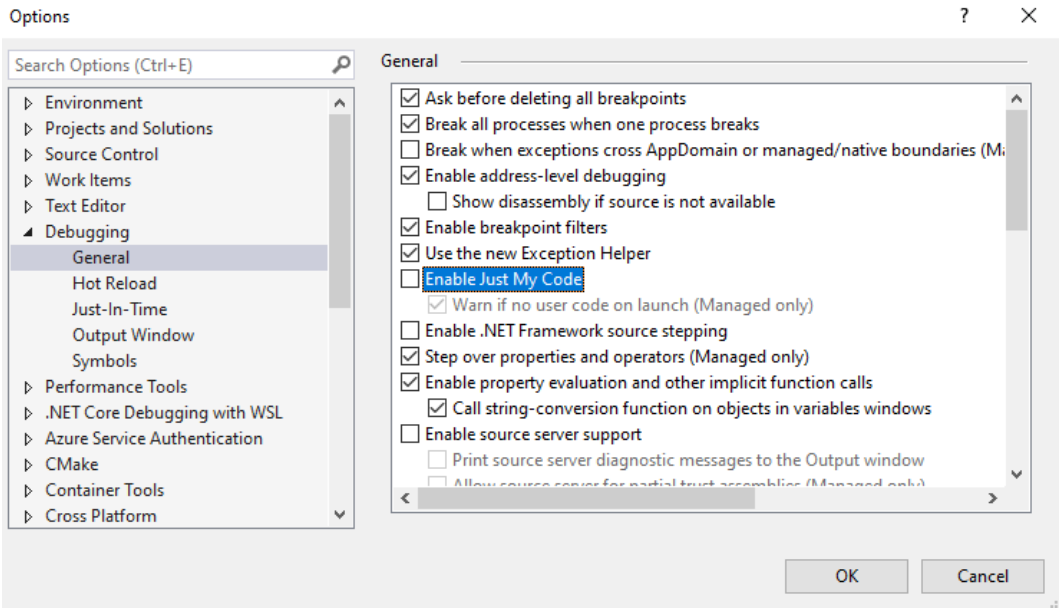
Pokretanje kontejnera iz komandne linije se vrši alatom **docker-compose** koja ima nekoliko važnih komandi (opcija **-d** označava da će se proces nastaviti u pozadini, kako se ne bi blokirao terminal):

- Izgradnja svih kontejnera
  - **docker-compose build**
- Podizanje svih kontejnera
  - **docker-compose up -d**
- Izgradnja i podizanje svih kontejnera
  - **docker-compose up --build -d**
- Zaustavljanje svih kontejnera
  - **docker-compose stop**
- Zaustavljanje i uklanjanje svih kontejnera
  - **docker-compose down**
- Specifikovanje datoteka koje se koriste za podizanje/spuštanje svih kontejnera
  - **docker-compose -f docker-compose.yml -f docker-compose.override.yml up --build -d**
  - **docker-compose -f docker-compose.yml -f docker-compose.override.yml**

Debagiranje projekata u kontejneru

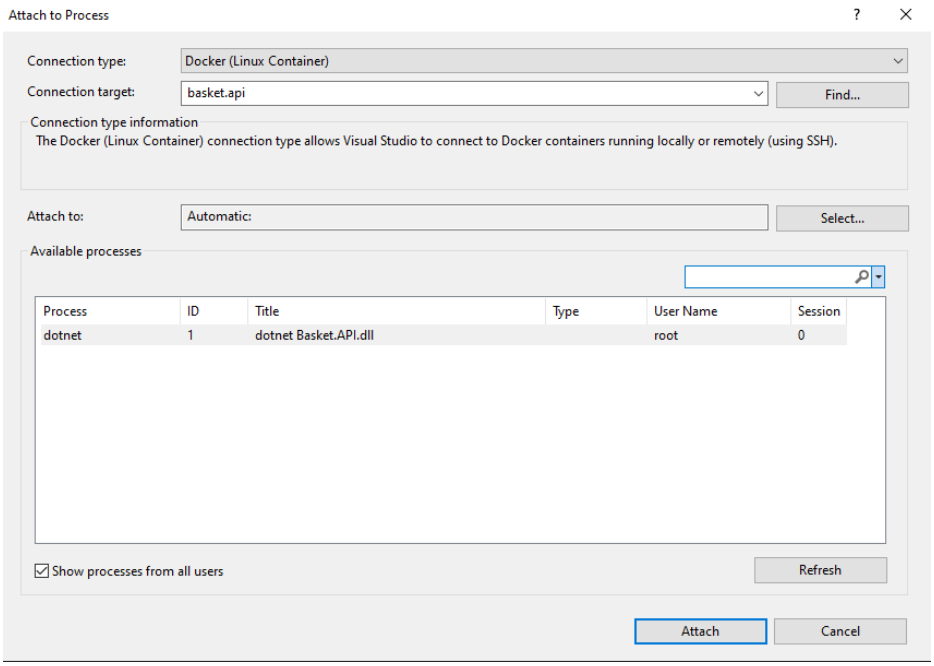
Kako bismo omogućili debagiranje projekata koji se pokreću u kontejneru, potrebno je da ručno *zakačimo* debager za proces koji se izvršava u kontejneru. Pre toga, potrebno je isključiti opciju „Enable Just My Code“ u podešavanjima, kako bismo instruisali Visual Studio da debagira i kod koji je izgrađen u kontejnerima:

- Tools > Options
- Debugging > General
- Isključiti opciju „Enable Just My Code“, kao na slici ispod



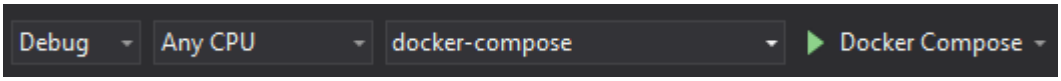
Sada možemo zakačiti debager:

- Debug > Attach to Process
- Za „Connection type“ odabrati Docker (Linux Container)
- Odabrati dugme Find
- Nakon nekoliko sekundi bi trebalo da se pojavi spisak svih podignutih kontejnera. Odabrati, na primer, basket.api projekat, pa dugme Ok
- U tabeli „Available processes“ bi trebalo da se pojavi „dotnet“ proces, kao na slici ispod
- Odabrati taj proces, pa dugme Attach
- Odabrati opciju „Managed (.NET Core for Unix)“, pa dugme Ok
- Posle nekoliko sekundi, debager će biti *zakačen* za proces



Pokretanje projekta iz Visual Studio alata

Nakon što smo napravili **docker-compose** projekat, potrebno je da odaberemo naredne opcije u glavnom meniju:



Klikom na pokretanje se vrše naredne akcije:

- Izgrađuje se kod iz *Solution-a*
- Izgrađuju se kontejneri
- Pokreću se kontejneri
- Pokreće se debager i automatski se zakači za izvršni kod

Neke napomene

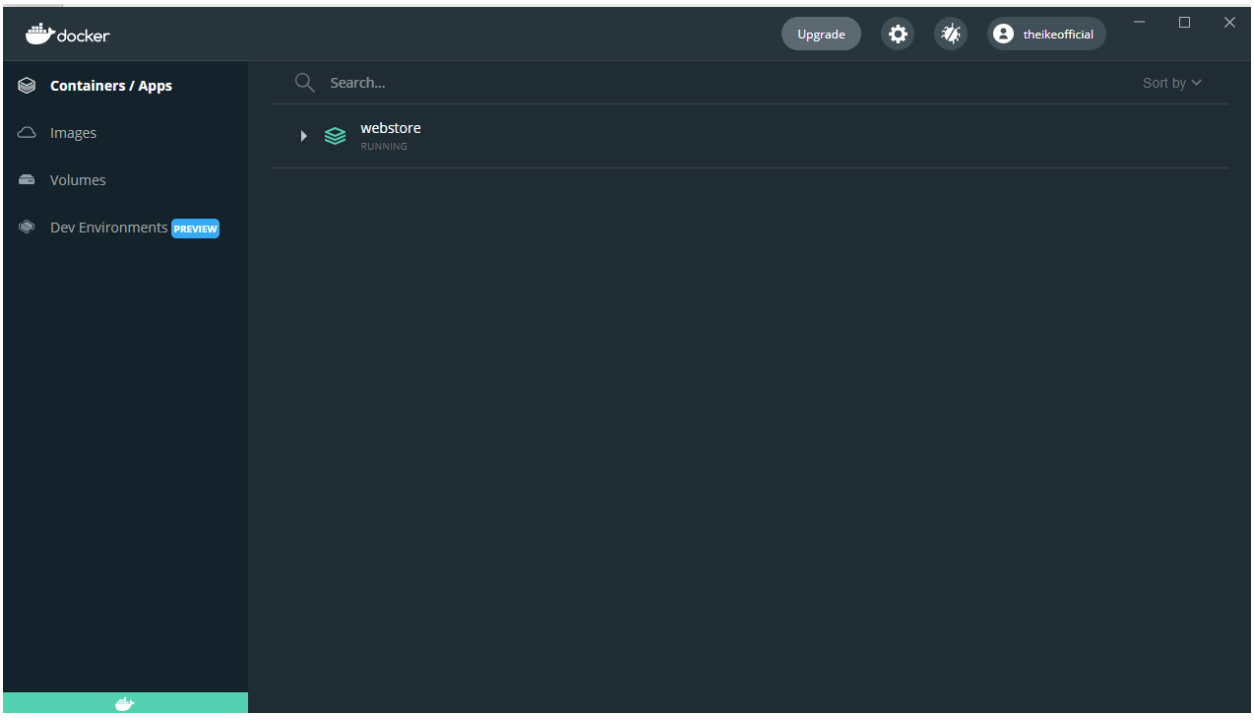
- Prednosti korišćenja su automatsko podizanje i automatsko debagiranje.
- Gašenjem projekta (bilo iz Visual Studio alata, gašenjem pregledača ili gašenjem terminala) se ne gase kontejneri, ali se ne mogu ugasiti iz **Docker Desktopa**, pa je neophodno uraditi **Build > Clean solution** iz glavnog menija – obavezno pre zatvaranja Visual Studio alata.
- Treba imati u vidu da će se konfiguracija prvo pročitati iz **appsettings.json** i **appsettings.Development.json** datoteka nego iz **docker-compose.yml** datoteka. Preporuka je da se prepišu sve promenljive okruženja iz Yaml datoteka u json pre pokretanja. Naravno, ovime se onemogućava pojedinačno pokretanje projekata iz Visual Studio alata.
- Debagiranje u kontejnerima je nešto sporije u odnosu na debagiranje aplikacija koje su pokrenute na host računaru, ali funkcioniše identično.

Docker Desktop

Alat Docker Desktop nam služi za upravljanje kontejnerima iz grafičke korisničke aplikacije. Na narednoj slici možemo videti prikaz nakon pokretanja. Sa strane možemo birati neke od tabova koji nam daju uvid u naredne elemente za rad sa kontejnerima:

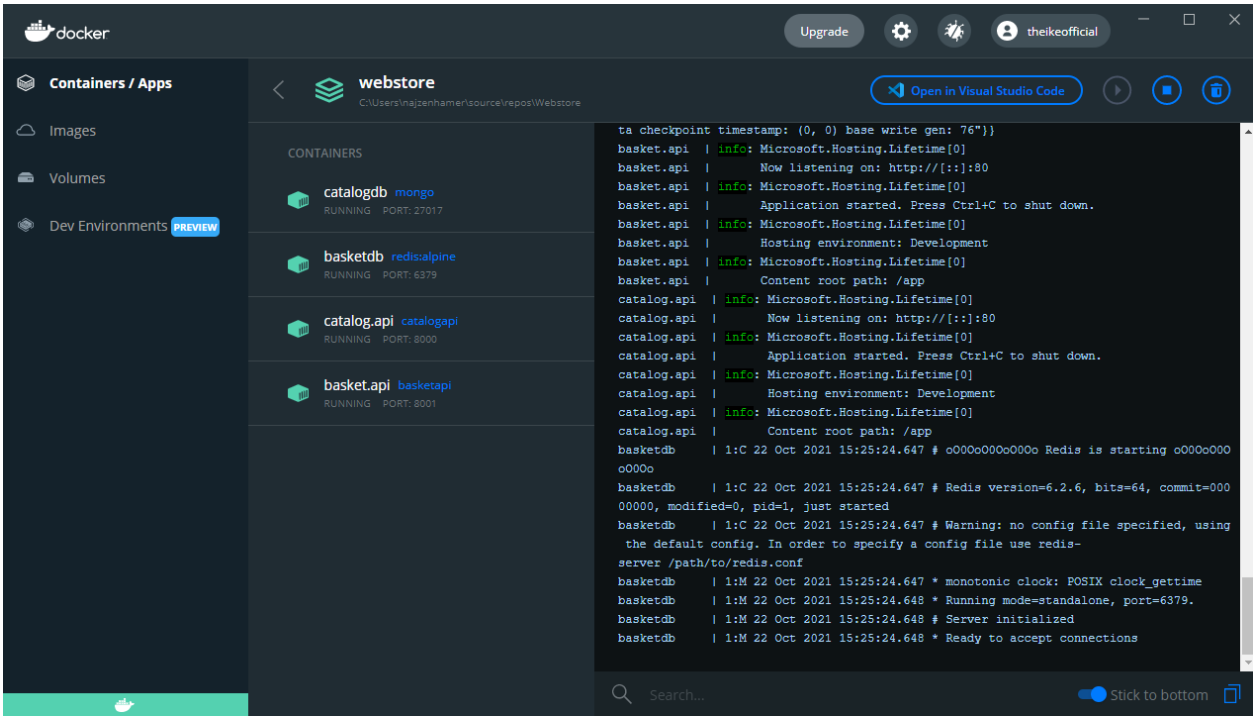
- **Containers/Apps** prikazuje pregled svih kontejnera koji postoje na sistemu
- **Images** prikazuje pregled svih slika koji su dovučeni na sistemu
- **Volumes** prikazuje pregled svih „diskova“ koje kontejneri koriste za trajno skladištenje datoteka
- **Dev Environments** prikazuje pregled okruženja za razvoj za jednostavnu kolaboraciju razvijalaca softvera u timu

Nama će najznačajniji biti prvi pregled.



Kontejneri mogu biti pokrenuti pojedinačno, ili kao deo neke mreže, tj. orkestra kontejnera. Na slici ispod je prikazana orkestrizacija celokupne aplikacija *Webstore* koja se, u ovom trenutku, sastoji od četiri kontejnera. Sa

desne strane možemo videti zajedničke dnevnike, a klikom na konkretan kontejner, biće nam prikazan dnevnik samo za taj kontejner.

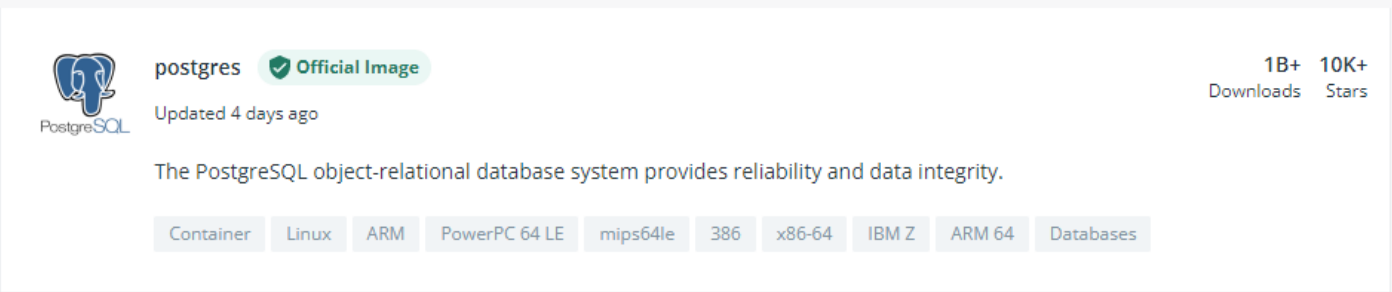


3. Sinhrona komunikacija između mikroservisa pomoću gRPC protokola

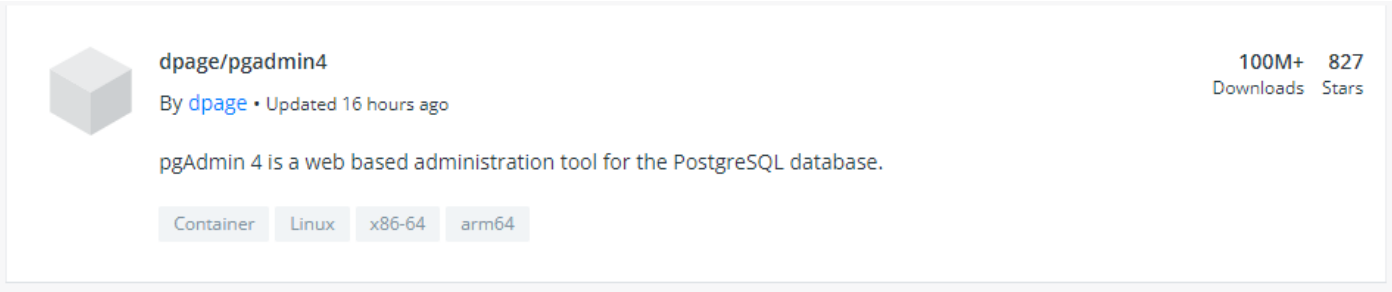
Tema ovih časova je ostvarivanje sinhrone komunikacije između mikroservisa implementiranjem gRPC protokola. Videćemo i rad sa PostgreSQL SUBP korišćenjem vrlo jednostavne biblioteke za ORP – Dapper.

1. Priprema PostgreSQL i pgadmin kontejnera

Na stranici https://hub.docker.com/ uneti „postgres“ u polje za pretragu. Otvoriti sledeću stranicu:



Na stranici https://hub.docker.com/ uneti „pgadmin“ u polje za pretragu. Otvoriti sledeću stranicu:



Dodati naredne resurse u **docker-compose.yml** datoteku:

```
services:
  discountdb:
    image: postgres

  pgadmin:
    image: dpage/pgadmin4

volumes:
  mongo_data:
  postgres_data:
  pgadmin_data:
```

Dodati naredne resurse u **docker-compose.override.yml** datoteku:

```
services:
  discountdb:
    container_name: discountdb
    environment:
      - POSTGRES_USER=admin
```

```
    - POSTGRES_PASSWORD=admin1234
    - POSTGRES_DB=DiscountDb
  restart: always
  ports:
    - "5432:5432"
  volumes:
    - postgres_data:/var/lib/postgresql/data/

 pgadmin:
  container_name: pgadmin
  environment:
    - PGADMIN_DEFAULT_EMAIL=razvoj.softvera.matf@gmail.com
    - PGADMIN_DEFAULT_PASSWORD=admin1234
  restart: always
  ports:
    - "5050:80"
  volumes:
    - pgadmin_data:/root/.pgadmin
```

Alat pgAdmin4

Nakon pokretanja kontejnera, alat je dostupan na adresi http://localhost:5050. Prijaviti se na sistem korišćenjem adrese elektronske pošte i lozinke koji su navedeni u datoteci iznad.

Dodavanje novog servera:

- Add New Server
- General
  - Name: **DiscountServer**
- Connection
  - Host name/address: **discountdb**
  - Port: **5432**
  - Maintenance database: **postgres**
  - Username: **admin**
  - Password: **admin1234**
  - Napomena za podatke iznad: username i password treba da budu isto kao u **docker-compose.override.yml** datoteci.
- Save

Podesiti inicijalnu shemu baze podataka DiscountDB:

- Tools > Query Tool
- Izvršiti naredni upit:

```
CREATE TABLE Coupon (
  ID SERIAL PRIMARY KEY NOT NULL,
  ProductName VARCHAR(24) NOT NULL,
  Description TEXT,
  Amount INT
);
```

Sada kreirajmo nekoliko inicijalnih vrednosti:

```
INSERT INTO Coupon (ProductName, Description, Amount) VALUES ('IPhone X', 'IPhone Discount', 150);
INSERT INTO Coupon (ProductName, Description, Amount) VALUES ('Huawei Plus', 'Huawei Discount', 110);
```

INSERT INTO Coupon (ProductName, Description, Amount) VALUES ('Xiaomi Mi 9', 'Xiaomi Discount', 75);
INSERT INTO Coupon (ProductName, Description, Amount) VALUES ('Samsung 10', 'Samsung Discount', 100);

Nakon svake izvršene naredbe koja menja shemu je potrebno osvežiti pogled u *Browser-u* kako bi se videle izmene.

2. Kreiranje zajedničkog projekta za API i gRPC projekte

S obzirom da koristimo iste resurse u dva projekta, bilo bi dobro da izdvojimo sve zajedničke stvari iz tih projekata u jednu biblioteku klasa. Ovime izbegavamo dupliciranje koda, ali polako uvodimo slojevitost u našim projektima. Ovu ideju ćemo detaljnije produbiti kada budemo govorili o čistoj arhitekturi.

Kreiranje projekta i instalacija paketa

Dodajemo novi projekat u okviru već napravljenog *Solution-a*:

- Desni klik na ime *Solution-a*
- Add > New Project
- Class library
- Popuniti neophodnim podacima:
  - Project name: **Discount.Common**
  - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Services\Discount**
- Next
- Odabrati opciju **.NET 5.0 (Current)**
- Create

Pre nego što krenemo sa razvojem, potrebno je da instaliramo neophodne pakete. Otvoriti *NuGet Package Manager* sledećim koracima:

- Desni klik na naziv projekta
- Manage NuGet Packages

Prvo ćemo ažurirati sve pakete koji su do sada instalirani:

- Updates
- Select all packages
- Update

Zatim je potrebno otvoriti tab „Browse“ i tu pronaći i instalirati sledeće pakete:

- Npgsql
- Dapper
- AutoMapper.Extensions.Microsoft.DependencyInjection
- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Binder
- Microsoft.Extensions.DependencyInjection.Abstractions

Razvoj biblioteke klasa za Discount projekte

Kodiranje vršimo prema narednom redosledu:

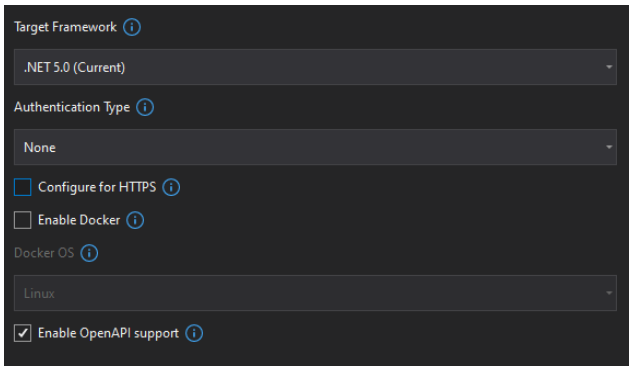
- Entities
  - Coupon.cs
- Data
  - ICouponContext.cs
  - CouponContext.cs
- DTOs
  - BaseCouponDTO.cs

    - BaseIdentityCouponDTO.cs
    - CouponDTO.cs
    - CreateCouponDTO.cs
    - UpdateCouponDTO.cs
- Repositories
    - ICouponRepository.cs
    - CouponRepository.cs
- Extensions
    - DiscountCommonExtensions.cs

3. Kreiranje API projekta

Dodajemo novi projekat u okviru već napravljenog *Solution-a*:

- Desni klik na ime *Solution-a*
- Add > New Project
- ASP.NET Core Web API
- Popuniti neophodnim podacima:
    - Project name: **Discount.API**
    - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA \Services\Discount**
- Next
- Odabrati opcije kao na slici pored:
- Create

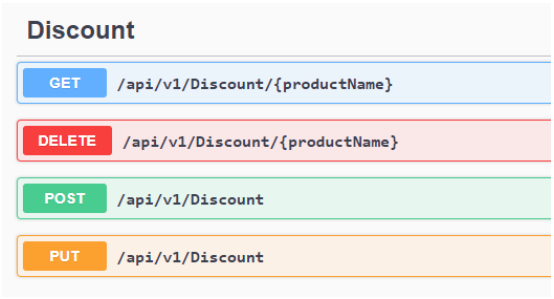


Pre nego što krenemo sa razvojem, potrebno je da dodamo referencu na prethodno napravljeni projekat **Discount.Common**:

- Desni klik na naziv projekta
- Add > Project Reference
- Odabrati Discount.Common
- Ok

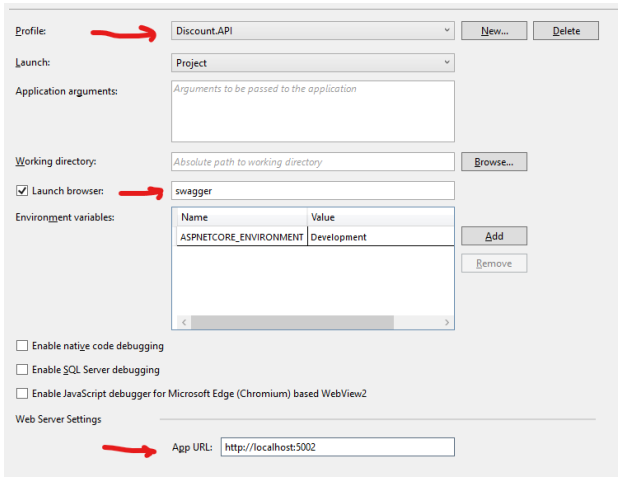
Razvoj mikroservisa

API mikroservisa je opisan slikom pored.



Prvo podešavamo opcije za pokretanje aplikacije:

- Desni klik na naziv projekta
- Properties
- Debug
- Podesiti opcije sa slike pored.



Zatim prelazimo na kodiranje, prema narednom redosledu:

- Controllers
    - DiscountController.cs

Sada je preostalo da dodamo ubrizgavanje zavisnosti, što se nalazi u Startup.cs datoteci.

Pokretanje i debagiranje aplikacije

Pokrenuti prvo **docker-compose** projekat iz terminala, a ujedno pokrenuti samo Discount.API projekat iz Visual Studio alata. Discount.API projekat bi trebalo da se poveže sa PostreSQL bazom podataka ukoliko se u **appsettings.Development.json** datoteci doda naredna konfiguracija:

```
"DatabaseSettings": {
  "ConnectionString": "Server=localhost;Port=5432;Database=DiscountDb;User Id=admin;Password=admin1234;"
}
```

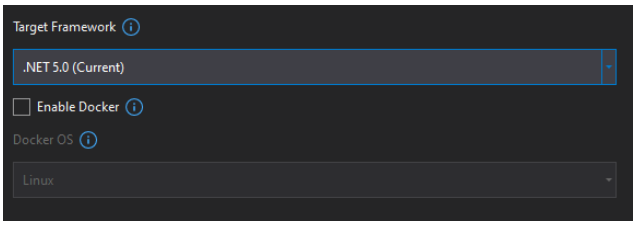
Sada dodati podršku za orkestrizaciju kontejnera za projekat Discount.API, pa pokrenuti sve kontejnere i testirati još jednom. Dopuniti **docker-compose.override.yml** datoteku narednim linijama:

```
discount.api:
  container_name: discount.api
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - "DatabaseSettings:ConnectionString=Server=discountdb;Port=5432;Database=DiscountDb;User Id=admin;Password=admin1234;"
  depends_on:
    - discountdb
  ports:
    - "8002:80"
```

4. Kreiranje gRPC projekta

Dodajemo novi projekat u okviru već napravljenog *Solution-a*:

- Desni klik na ime *Solution-a*
- Add > New Project
- ASP.NET Core gRPC Service
- Popuniti neophodnim podacima:
  - Project name: **Discount.GRPC**
  - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Services\Discount**
  - Next
- Odabrati opcije kao na slici pored.
- Create



Pre nego što krenemo sa razvojem, potrebno je da dodamo referencu na prethodno napravljeni projekat **Discount.Common**:

- Desni klik na naziv projekta
- Add > Project Reference
- Odabrati Discount.Common
- Ok

Sada ažurirajmo neophodne pakete:

- Desni klik na naziv projekta
- Manage NuGet Packages
- Updates
- Select all packages
- Update

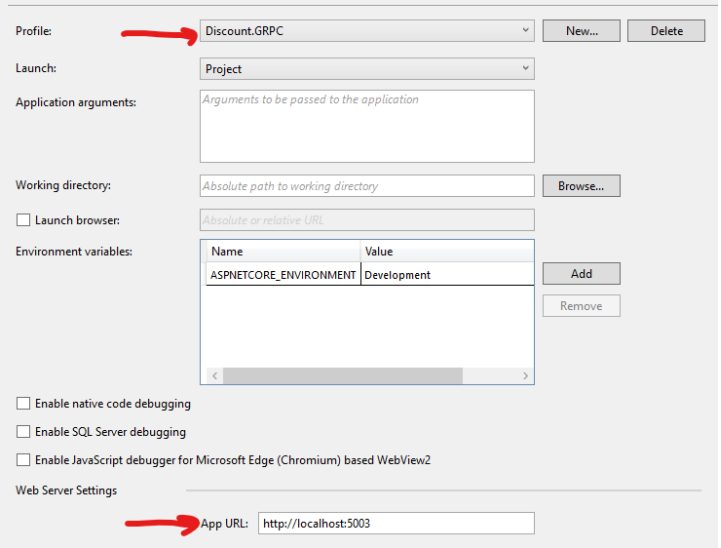
Zatim je potrebno otvoriti tab „Browse“ i tu pronaći i instalirati sledeće pakete:

- Google.Protobuf
- Grpc.Core
- Grpc.Tools

Ako se i dalje pojavljuje greška u Build prozoru, samo očistiti i ponovo izgraditi projekat.

Razvoj mikroservisa

Prvo podešavamo opcije za pokretanje aplikacije:



- Desni klik na naziv projekta
- Properties
- Debug
- Podesiti opcije sa slike pored:

Da bismo implementirali gRPC protokol, potrebno je da postoje dve strane u komunikaciji: server i klijent.

Server je onaj mikroservis koji opslužuje druge nekim servisima, a klijent je onaj mikroservis koji koristi usluge nekih servera. U našem primeru, server je Discount.GRPC zato što on omogućava da se pretraže kuponi za dati proizvod, a Basket.API je klijent zato što on koristi tu uslugu pretrage kupona kako bi odredio konačnu cenu potrošačke korpe.

Za gRPC implementaciju su potrebne dve stvari:

- *proto buffer* datoteka koja opisuje gRPC servise
- C# klasa koja implementira te opisane servise

Generisanje *proto buffer* datoteke se radi narednim koracima:

- Desni klik na direktorijum *Protos*
- Add > New Item
- Protocol Buffer File
- Name: coupon.proto

Sada je potrebno definisati servise koje želimo da omogućimo. U tu svrhu se koristi *proto3* sintaksa (više o ovom jeziku na https://developers.google.com/protocol-buffers/docs/proto3). Obavezno postaviti da se datoteka koristi za potrebe definisanja gRPC servera:

- Desni klik na *proto buffer* datoteku > Properties
- Build action: Protobuf compiler
- gRPC Stub Classes: Server Only

Kada se završi pisanje *proto buffer* datoteke, možemo generisati implementaciju narednim koracima:

- Desni klik na direktorijum *Services*
- Add > Class
- Name: CouponService
- Naslediti klasu **CouponProtoService.CouponProtoServiceBase**
- Desni klik na našu klasu > Generate overrides

Sada smo dobili potpise metoda koje implementiramo kako god želimo. Ova klasa predstavlja na neki način kontroler, samo što koristi gRPC protokol umesto HTTP.

Ne zaboraviti da je potrebno da se u **Startup.cs** doda rutiranje gRPC servisa u HTTP *pipeline-u*:

```
endpoints.MapGrpcService<CouponService>();
```

kao i ubrizgavanje zavisnosti i preslikavanje neophodnih modela:

```
services.AddDiscountCommonServices();
services.AddAutoMapper(configuration =>
{
   configuration.CreateMap<Coupon, CouponModel>().ReverseMap();
});
```

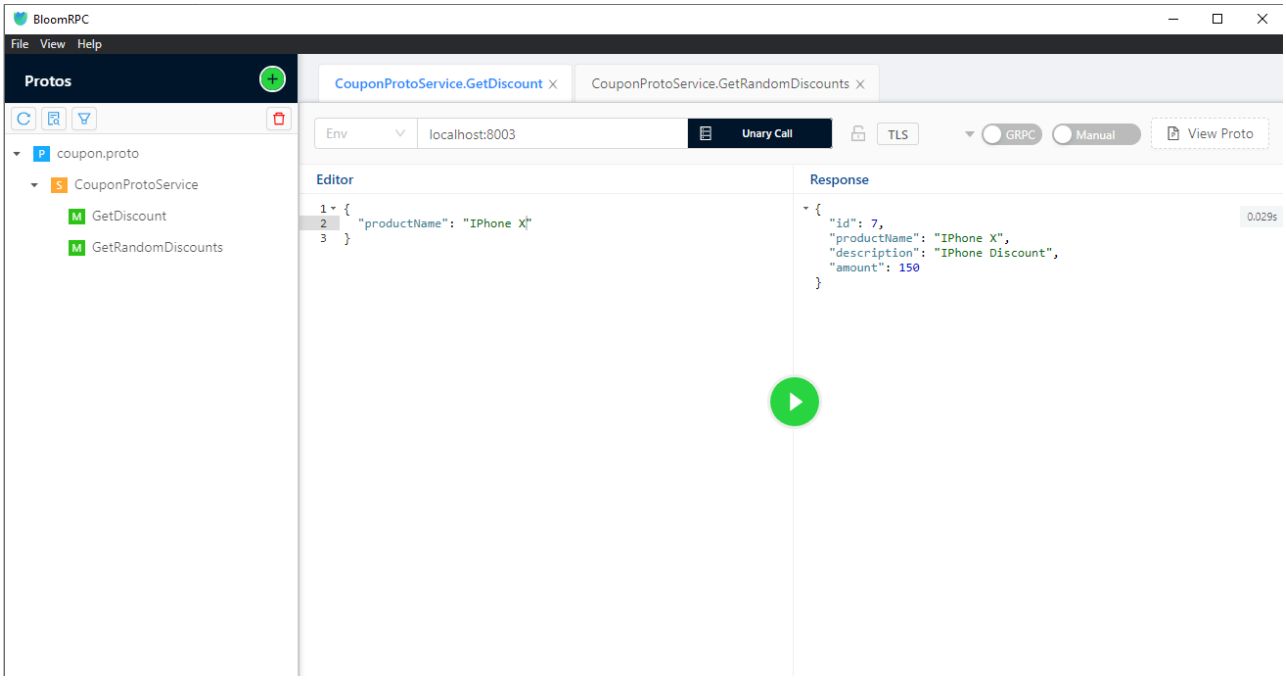
Pokretanje i debagiranje aplikacije

Pokrenuti prvo **docker-compose** projekat iz terminala, a ujedno pokrenuti samo Discount.GRPC projekat iz Visual Studio alata. Discount.GRPC projekat bi trebalo da se poveže sa PostreSQL bazom podataka ukoliko se u **appsettings.Development.json** datoteci doda naredna konfiguracija:

```
 "DatabaseSettings": {
   "ConnectionString": "Server=localhost;Port=5432;Database=DiscountDb;User Id=admin;Password=admin1234;"
 }
```

Otvoriti BloomRPC alat, učitati *proto buffer* datoteku i testirati gRPC servis slanjem zahteva kao na narednoj slici. Pre prikazivanja slike napomenimo sledeće:

- Dok se testira projekat kad je pokrenut lokalno, u alatu upisati adresu **localhost:5003**
- Dok se testira projekat u kontejneru, u alatu upisati adresu **localhost:8003**



Sada dodati podršku za orkestrizaciju kontejnera za projekat Discount.GRPC, pa pokrenuti sve kontejnere i testirati još jednom. Dopuniti **docker-compose.override.yml** datoteku narednim linijama:

```
 discount.grpc:
   container_name: discount.grpc
   environment:
     - ASPNETCORE_ENVIRONMENT=Development
     - "DatabaseSettings:ConnectionString=Server=discountdb;Port=5432;Database=DiscountDb;User
Id=admin;Password=admin1234;"
   depends_on:
```

```
    - discountdb
  ports:
    - "8003:80"
```

5. Korišćenje gRPC projekta u Basket mikroservisu

Vratimo se sada na Basket.API mikroservis kako bismo dodali neophodan kod za ostvarivanje gRPC komunikacije ka Discount.GRPC mikroservisu. Koraci koje je neophodno izvršiti su slični kao za server, ali neke stvari je moguće ubrzati:

- Registracija gRPC servisa
- Implementacija klijentske logike
- Dodavanje informacija o gRPC komunikaciji i ubrizgavanje zavisnosti

Da bismo registrovali novi gRPC servis, potrebno je da uradimo naredne korake:

- Desni klik na naziv Basket.API projekta
- Add > Connected Service
- U okviru „Service References (OpenAPI, gRPC, WCF Web Service)“ odabrati dugme „+“
- gRPC
- File > Browse
- Pronaći *proto buffer* datoteku koju smo implementirali na serveru. Trebalo bi da bude na putanji oblika **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Services\Discount\Discount.GRPC\Protos\coupon.proto**
- Select the type of class to be generated: **Client**
- Finish
- Close

Sada možemo da napravimo klasu koju ćemo koristiti u ovom projektu, a koja se oslanja na automatski generisanog klijenta u gRPC komunikaciji:

- GrpcServices
  - CouponGrpcService.cs
- Controllers
  - BasketController.cs

Konačno, neophodno je da u **Startup.cs** datoteci navedemo podešavanja za gRPC komunikaciju. Najpre, potrebno je da navedemo koji gRPC servis „gađa“ generisani klijent, kao i da ubrizgamo klasu koju smo mi implementirali iznad:

```
services.AddGrpcClient<CouponProtoService.CouponProtoServiceClient>
   (o => o.Address = new Uri(Configuration["GrpcSettings:DiscountUrl"]));
services.AddScoped<CouponGrpcService>();
```

Očigledno, neophodno je dodati adresu u konfiguraciju, jedanput u datoteci **appsettings.Development.json** i jedanput u **docker-compose.override.yml**. Dodatno, u drugoj datoteci je potrebno navesti da Basket.API mikroservis očekuje da Discount.GRPC servis bude pokrenut pre njega.

U **appsettings.Development.json** datoteci:

```
 "GrpcSettings": {
   "DiscountUrl": "http://localhost:5003"
 }
```

U **docker-compose.override.yml** datoteci:

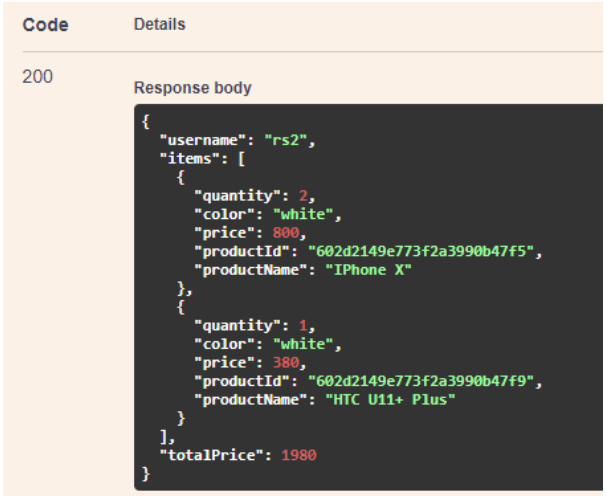
```
 basket.api:
   container_name: basket.api
```

```
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - "CacheSettings:ConnectionString=basketdb:6379"
    - "GrpcSettings:DiscountUrl=http://discount.grpc"
  depends_on:
    - basketdb
    - discount.grpc
  ports:
    - "8001:80"
```

Za testiranje PUT zahteva za Basket.API mikroservis se može koristiti naredni JSON kod:

```
{
 "username": "rs2",
 "items": [
  {
   "quantity": 2,
   "color": "white",
   "price": 950,
   "productId": "602d2149e773f2a3990b47f5",
   "productName": "IPhone X"
  },
  {
   "quantity": 1,
   "color": "white",
   "price": 380,
   "productId": "602d2149e773f2a3990b47f9",
   "productName": "HTC U11+ Plus"
  }
 ]
}
```

Ono što možemo primetiti sa slike ispod jeste da se u odgovoru zahteva primenio kupon na uređaj „IPhone X“, ali da je uređaj „HTC U11+ Plus“ zadržao svoju originalnu cenu.



4. Razvoj vođen domenom. Čista arhitektura. CQRS.

Tema ovih časova je razvoj složenijih mikroservisa. Razvoj vođen domenom (*Domain-Driven* Design, DDD) omogućuje nam da, vodeći se OOP principima, implementiramo poslovne procese u skladu sa zahtevima koji su prepoznati od strane eksperata u domenu. Čista arhitektura nam omogućava da grupišemo kolekcije klasa na način koji omogućava nesmetan razvoj. CQRS pristup predstavlja razdvajanje odgovornosti operacija koji se odigravaju u sistemu.

1. Razvoj vođen domenom

- Proći kroz prezentaciju „Razvoj vođen domenom“: http://rs2.matf.bg.ac.rs/vezbe/ddd.pdf

2. Čista arhitektura

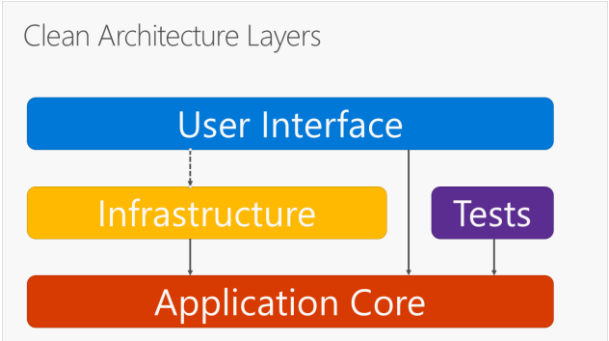
Čista arhitektura (ovde se više misli na dizajn, s obzirom da je ovo arhitektura u okviru jednog mikroservisa, a ne celog sistema) podrazumeva da se kod unutar mikroservisa podeli na slojeve. Slojevi mogu da zavise jedni od drugih, pri čemu razlikujemo:

- Zavisnost u fazi prevođenja koda
- Zavisnost u fazi izvršavanja aplikacije

Slojevi u čistoj arhitekturi su:

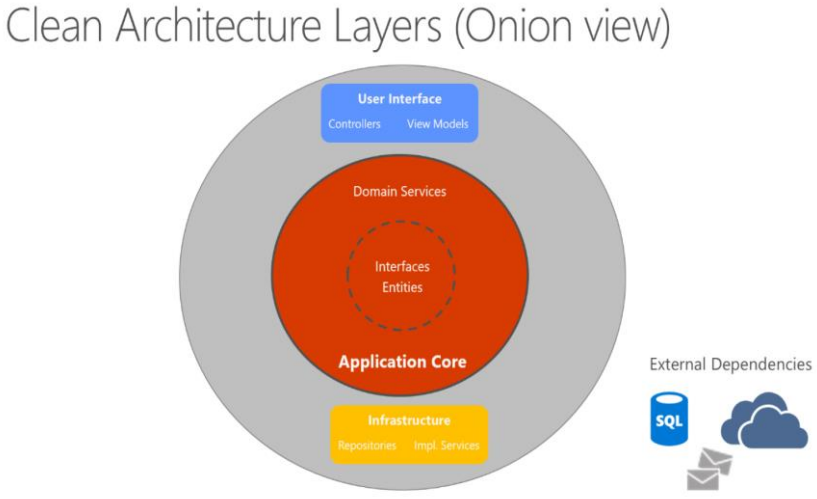
- Sloj domena (jezgro aplikacije koje čine POCO klase koje opisuju domen sistema)
- Sloj aplikacije (interfejsi koji se oslanjaju na logiku iz domena, a predstavljaju uslove koje ostali resursi moraju da zadovoljavaju)
- Sloj infrastrukture (implementacija interfejsa iz sloja aplikacije, često korišćenjem spoljašnjih resursa, kao što je komunikacija za raznim SUBP, skladištenje dokumenata u oblacima, slanje elektronske pošte, ...)
- Sloj API-ja ili korisničkog interfejsa (implementacija korisničkih zahteva, generisanje HTML stranica, ...)

Na slici pored su prikazani ovi slojevi u horizontalnom rasporedu. Puna linija predstavlja zavisnost u fazi prevođenja koda, a isprekidana u fazi izvršavanja koda.

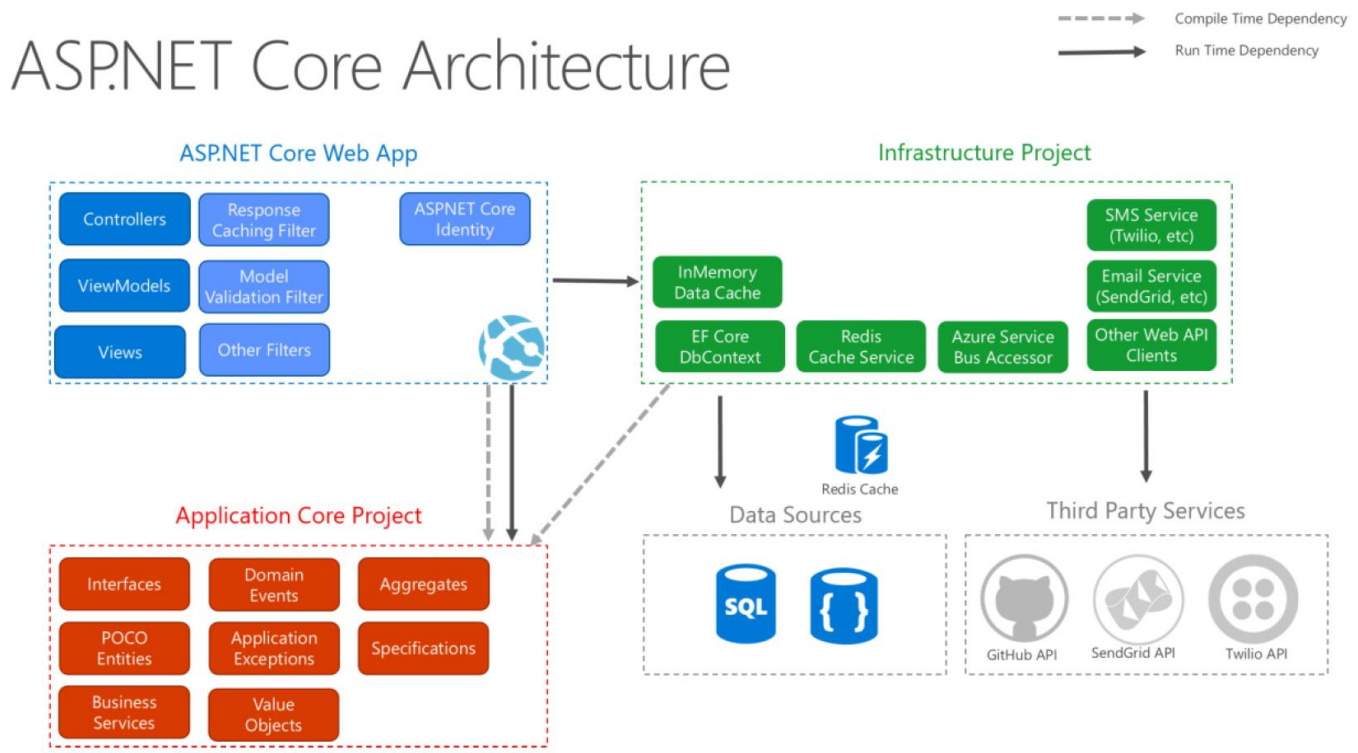


Slika ispod prikazuje drugi pogled na istu arhitekturu, pri čemu slojevi koji su spolja zavise od slojeva koji su unutra:

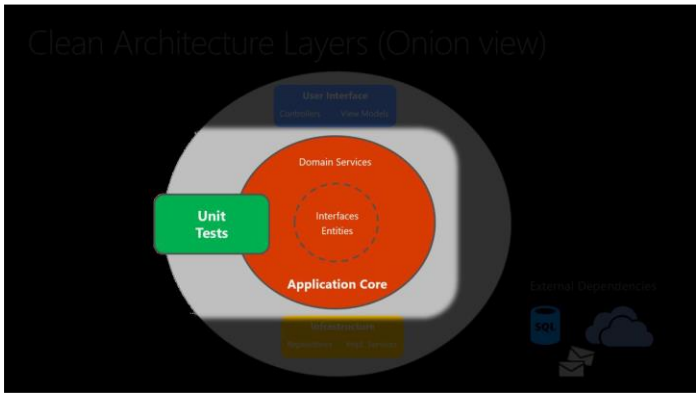
- API/UI i infrastruktni slojevi zavise od aplikacionog sloja
- Aplikacioni sloj zavisi od domena

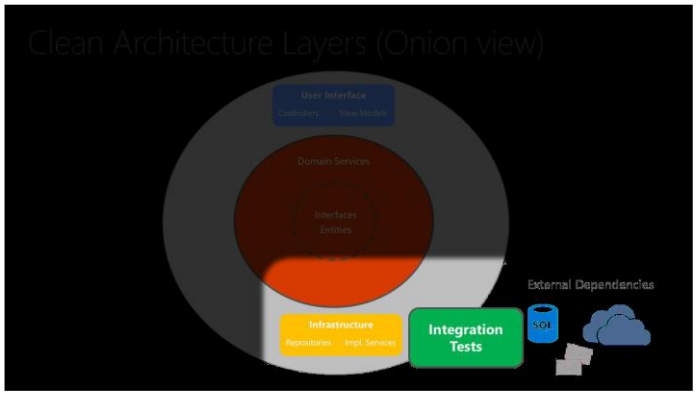


Naredna slika prikazuje još detaljniji pogled na istu arhitekturu:



Važno je uzeti u obzir i mesto testova u ovoj arhitekturi. Ono što je ispostavlja kao velika prednost čiste arhitekture jeste što razdvajanje koda po slojevima omogućava ograničenost konteksta koja je neophodna testovima, kao i izolovanost testova po slojevima. Naredne dve slike prikazuju na kojim mestima bi se testovi jedinica koda i integracioni testovi smestili u ovoj arhitekturi.





Drugim rečima, s obzirom da domenski i aplikacioni slojevi ne zavise od arhitekture, vrlo je jednostavno pisati testove koji su izolovani samo za te slojeve. Sa druge strane, s obzirom da se API/UI ne oslanja na tipove iz infrastrukturnog sloja, vrlo je jednostavno zameniti implementacije koje se koriste u obradi zahteva, bilo za potrebe izvršavanja testova ili radi izmena u zahtevima aplikacije (u ovome nam značajno pomaže i ubrizgavanje zavisnosti).

3. CQRS

Ukratko proći kroz članak https://martinfowler.com/bliki/CQRS.html i skrenuti pažnju na postojanje koncepta Event Sourcing (više o tome sa primerima implementacije na https://martinfowler.com/eaaDev/EventSourcing.html). Pogledati i Microsoftov priručnik za obrasce na temu Event Sourcing-a (https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing).

4. Implementacija Ordering mikroservisa

U nastavku prikazujemo samo redosled implementacije slojeva i elemenata u okviru tih slojeva. Očekuje se da su čitaoci u stanju da rekonstruišu kod na osnovu ovih beleški i koda sa repozitorijuma.

Ordering.Domain projekat

Tip projekta: Class library

Paketi: /

Zavisnosti od drugih slojeva: /

Implementacija:

- Common
  - EntityBase.cs
  - ValueObjectBase.cs
  - AggregateRoot.cs
- Entities
  - OrderItem.cs
- Exceptions
  - OrderingDomainException.cs
- ValueObjects
  - Address.cs
- Aggregates
  - Order.cs

Ordering.Application projekat

Tip projekta: Class library

Paketi:

- MediatR
- MediatR.Extensions.Microsoft.DependencyInjection
- FluentValidation
- FluentValidation.DependencyInjectionExtensions
- Microsoft.Extensions.Logging.Abstractions

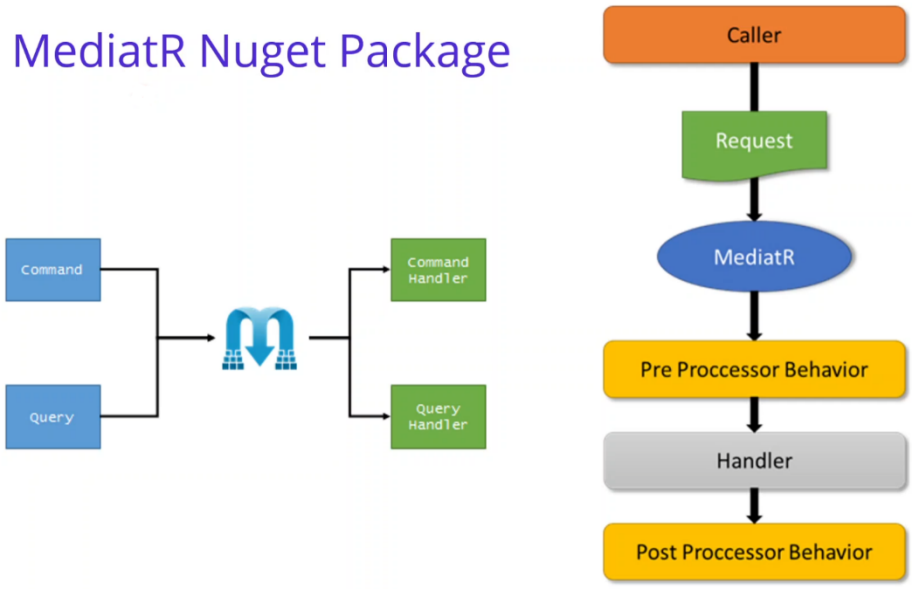
Zavisnosti od drugih slojeva:

- Ordering.Domain

Implementacija:

- Contracts
  - Persistence
    - IAsyncRepository.cs
    - IOrderRepository.cs
  - Infrastructure
    - IEmailService.cs
- Models
  - Email.cs
  - EmailSettings.cs

Pre nego što nastavimo sa implementacijom, vredno je pomenuti kako nam MediatR paket omogućava da implementiramo CQRS.



Dakle, prvo je neophodno da definišemo *upite* i *naredbe*, a zatim da definišemo tzv. *handler* klase, koji će obrađivati zahteve. Preporučuje se da *upiti* i *naredbe* imaju nezavisne modele, kako bi bili u skladu sa CQRS obrascem. Mi ćemo sve klase držati u aplikativnom sloju, ali ćemo ih koristiti potpuno nezavisno.

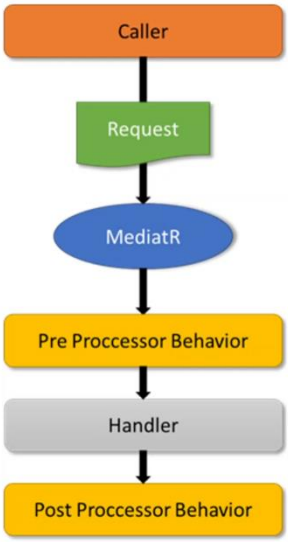
Dakle, prvo definišemo *upite* i *naredbe* za CQRS obrazac korišćenjem MediatR paketa. Takođe, želimo da definišemo i *Data Transfer Object*-e (https://martinfowler.com/eaaCatalog/dataTransferObject.html), koji se često u CQRS obrascu nazivaju *ViewModel*-i ako se koriste kao rezultati *upita*.

- Features
  - Orders
    - Queries
      - GetListOfOrders
        - GetListOfOrdersQuery.cs
      - ViewModels
        - OrderViewModel.cs
        - OrderItemViewModel.cs
    - Commands
      - DTOs
        - OrderItemDTO.cs
      - CreateOrder
        - CreateOrderCommand.cs
      - UpdateOrder
        - UpdateOrderCommand.cs
      - DeleteOrder
        - DeleteOrderCommand.cs
- Contracts
  - Factories
    - IOrderFactory.cs
    - IorderViewModelFactory.cs

Zatim, dodajemo *handler* klase za napisane *upite* i *naredbe*.

- Features
  - Orders
    - Queries
      - GetListOfOrders

            - GetListOfOrdersQueryHandler.cs
        - Commands
            - CreateOrder
                - CreateOrderCommandHandler.cs
            - UpdateOrder
                - UpdateOrderCommandHandler.cs
            - DeleteOrder
                - DeleteOrderCommandHandler.cs

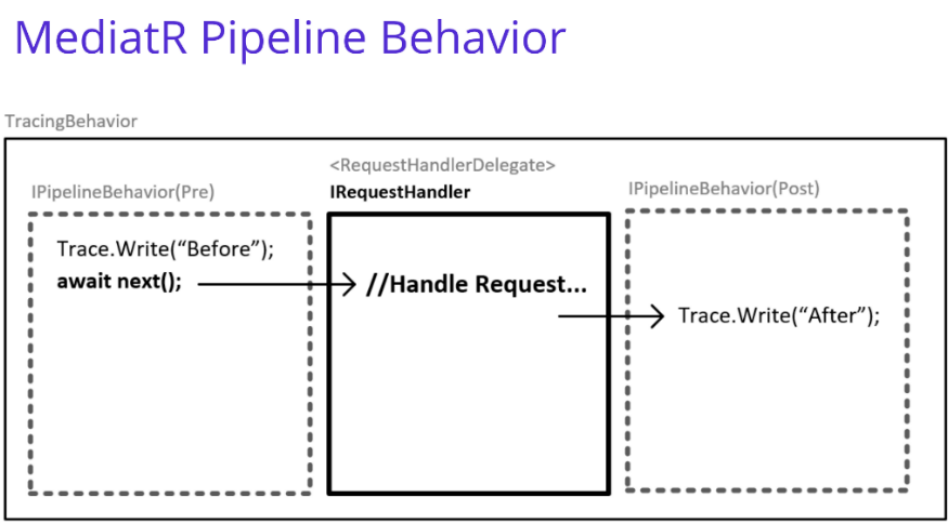


Dobra stvar kod MediatR paketa jeste što nam omogućava da izvršavamo akcije pre i posle pozivanja *handler* operacija, kao na slici pored. Ono što dobijamo na ovaj način jeste razdvajanje implementacije poslovne logike u našem sistemu (što se implementira u samim *handler* klasama) od raznih drugih operacija koje su neophodne za izvršavanje poslovne logike, ali nisu deo nje:

- Primer *pre processor behavior*: validacija podataka
- Primer *post processor behavior:* obrada grešaka

Sada ćemo videti kako je moguće dodavati ove akcije jednostavno pomoću MediatR paketa. Za početak, definišimo validatore za obradu *naredbi* i izuzetke koji se mogu javljati:

- Features
    - Orders
        - Commands
            - CreateOrder
                - CreateOrderCommandValidator.cs
            - UpdateOrder
                - UpdateOrderCommandValidator.cs
            - DeleteOrder
                - DeleteOrderCommandValidator.cs
- Exceptions
    - EntityNotFoundException.cs
    - ValidationFailedException.cs



Sada možemo dodati implementacije *ponašanja*. Pre implementacije, važno je razumeti kako jedna funkcija poziva drugu u obradi zahteva, što je ilustrovano na slici iznad. Više o *ponašanjima* je dostupno na https://github.com/jbogard/MediatR/wiki/Behaviors.

- Behaviours
    - ValidationBehavior.cs
    - UnhandledExceptionBehavior.cs

Konačno, neophodno je da definišemo *klasu proširenja* za ubrizgavanje zavisnosti svih servisa koje smo definisali. Važno je napomenuti da će se ova klasa koristiti samo u API sloju, pošto je to sloj koji predstavlja Web API projekat, dok aplikacioni sloj samo definiše interfejse i klase koje drugi koriste. Redosled registracije ponašanja odgovara redosledu izvršavanja[1].

- ApplicationServiceRegistration.cs

Naravno, ovu klasu koristimo u API projektu.

Ordering.API projekat

Tip projekta: ASP.NET Core Web API

Paketi:

- Microsoft.EntityFrameworkCore.Tools
- Polly

Zavisnosti od drugih slojeva:

- Ordering.Application
- Odering.Infrastructure

Podešavanja:

- App URL: http://localhost:5004

Implementacija:

- Controllers
    - OrderController.cs

Napomenimo da ovde nije završena implementacija ovog projekta i da ćemo se vratiti na njega kada završimo implementaciju sloja infrastrukture.

[1] Neko će se zapitati zašto registrujemo prvo **UnhandlerExceptionBehaviour**, pa tek onda **ValidationBehaviour** kada se prvo radi validacija, pa tek onda obrada grešaka. Važno je razumeti koje ponašanje se izvršava pre neke akcije, a koje ponašanje posle neke akcije. Ako pogledamo definicije ovih klasa, vidimo da se u klasi **ValidationBehaviour** poziv „**next()**“ izvršava tek na kraju metoda **Handle**, što znači da se logika ovog ponašanja izvršava *pre* obrade zahteva. Sa druge strane, u **Handle** metodu klase **UnhandlerExceptionBehaviour**, poziv „**next()**“ se izvršava na samom početku (**try** blok), a sam čin obrade grešaka (**catch** blok) biće izvršen tek ako prilikom obrade zahteva bude ispaljen neki izuzetak, što znači da se logika ovog ponašanja izvršava *posle* obrade zahteva.