

Razvoj softvera 2 – Beleške za vežbe

Nikola Ajzenhamer

Sadržaj

1. UVOD U MIKROSERVISNE APLIKACIJE	3
1. O KURSU	3
2. O MIKROSERVISNIM APLIKACIJAMA	3
3. RAZVOJ CATALOG.API MIKROSERVISA	3
PRIPREMA MONGODB KONTEJNERA	3
KREIRANJE PROJEKTA I INSTALIRANJE PAKETA	4
RAZVOJ MIKROSERVISA	5
POKRETANJE I DEBAGIRANJE APLIKACIJE	6
2. VIŠESTRUKI MIKROSERVISI. KONTEJNERIZACIJA APLIKACIJE.	7
1. RAZVOJ BASKET.API MIKROSERVISA	7
PRIPREMA REDIS KONTEJNERA	7
KREIRANJE PROJEKTA I INSTALIRANJE PAKETA	7
RAZVOJ MIKROSERVISA	8
POKRETANJE I DEBAGIRANJE APLIKACIJE	9
2. KONTEJNERIZACIJA PROJEKATA	9
POKRETANJE PROJEKTA IZ KOMANDNE LINIJE	10
POKRETANJE PROJEKTA IZ VISUAL STUDIO ALATA	11
NEKE NAPOMENE	12
DOCKER DESKTOP	12
3. SINHRONA KOMUNIKACIJA IZMEĐU MIKROSERVISA POMOĆU GRPC PROTOKOLA	14
1. PRIPREMA POSTGRESQL I PGADMIN KONTEJNERA	14
ALAT PGADMIN4	15
2. KREIRANJE ZAJEDNIČKOG PROJEKTA ZA API I GRPC PROJEKTE	16
KREIRANJE PROJEKTA I INSTALACIJA PAKETA	16
RAZVOJ BIBLIOTEKE KLASA ZA DISCOUNT PROJEKTE	16
3. KREIRANJE API PROJEKTA	17
RAZVOJ MIKROSERVISA	17
POKRETANJE I DEBAGIRANJE APLIKACIJE	18
4. KREIRANJE GRPC PROJEKTA	18
RAZVOJ MIKROSERVISA	19
POKRETANJE I DEBAGIRANJE APLIKACIJE	20
5. KORIŠĆENJE GRPC PROJEKTA U BASKET MIKROSERVISU	21
4. RAZVOJ VOĐEN DOMENOM. ČISTA ARHITEKTURA. CQRS.	23
1. RAZVOJ VOĐEN DOMENOM	23

2. ČISTA ARHITEKTURA	23
3. CQRS	24
4. IMPLEMENTACIJA ORDERING MIKROSERVISA	25
ORDERING.DOMAIN PROJEKAT	25
ORDERING.APPLICATION PROJEKAT	25
ORDERING.API PROJEKAT	28
5. ENTITY FRAMEWORK CORE	29
1. PRIPREMA SQL SERVER KONTEJNERA	29
2. ORDERING.INFRASTRUCTURE PROJEKAT	29
3. MIGRACIJE BAZA PODATAKA	31
4. KONTEJNERIZACIJA ORDERING.API PROJEKTA	32
6. ASINHRONA KOMUNIKACIJA IZMEĐU MIKROSERVISA POMOĆU REDA PORUKA	33
1. OSNOVNI ELEMENTI RABBITMQ POSREDNIKA	33
2. PRIPREMA RABBITMQ KONTEJNERA	34
3. EVENTBUS.MESSAGES PROJEKAT	35
4. OBJAVLJIVANJE PORUKE IZ BASKET API MIKROSERVISA	35
5. PRETPLAĆIVANJE NA PORUKE U ORDERING API MIKROSERVISU	36
6. KONTEJNERIZACIJA BASKET API I ORDERING API MIKROSERVISA SA RABBITMQ KONTEKSTOM	37
7. MREŽNI PROLAZI	38
1. KREIRANJE OCELOT MIKROSERVISA	39
2. RAZVOJ OCELOT MREŽNOG PROLAZA	40
3. KONTEJNERIZACIJA OCELOT MIKROSERVISA	41
8. BEZBEDNOST MIKROSERVISNIH APLIKACIJA	42
1. PODEŠAVANJE IDENTITYSERVER MIKROSERVISA I OSNOVNA IMPLEMENTACIJA	43
2. IMPLEMENTACIJA REGISTRACIJE KORISNIKA	45
3. IZDAVANJE JWT TOKENA	46
4. DODAVANJE AUTORIZACIJE	47
5. KONTEJNERIZACIJA IDENTITYSERVER MIKROSERVISA	48
6. ZAŠTITA DRUGIH MIKROSERVISA	49
DODATNA ZAŠTITA KORIŠĆENJEM TVRDNJI	50
7. REFRESH TOKENI	50
9. JEDNOSTRANIČNE KLIJENTSKE APLIKACIJE	54

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 seminarским 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

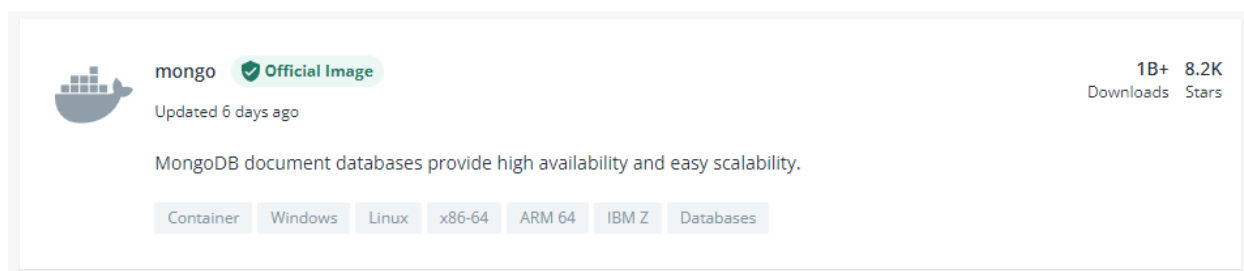
- Ukratko proći kroz tekst: <http://rs2.matf.bg.ac.rs/vezbe/ukratko-o-mikroservisima.pdf>

3. Razvoj Catalog.API mikroservisa

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

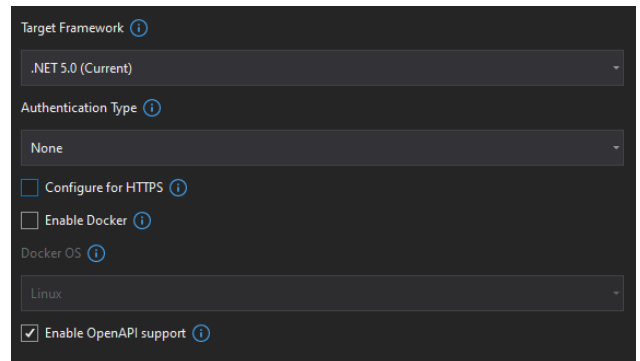
S obzirom da započinjemo razvoj „od nule“, potrebno je prvo da napravimo jedan *Solution* pre nego što kreiramo bilo koji projekat:

- 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:

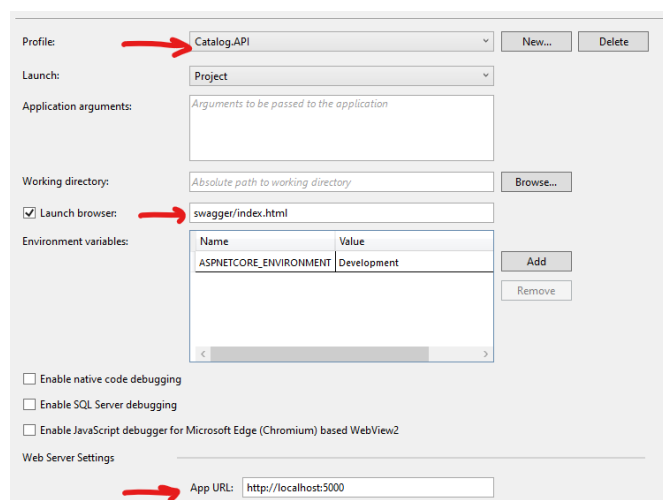
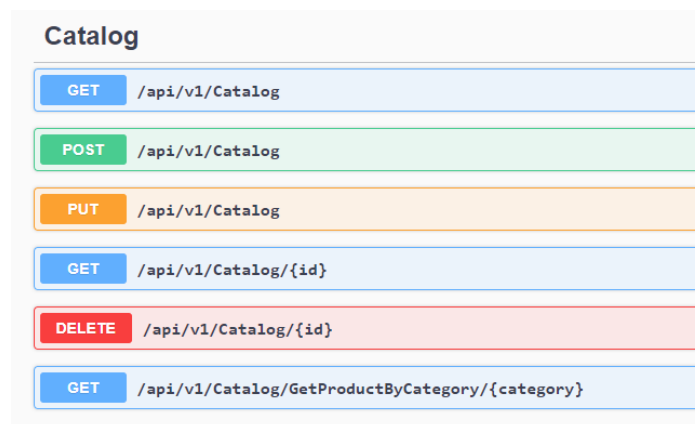
- MongoDB.Driver

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.



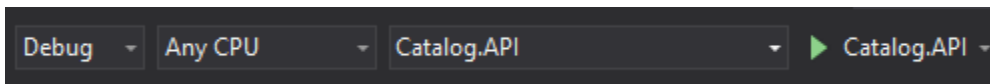
Zatim prelazimo na kodiranje, prema narednom redosledu:

- 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 debugiranje 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 debugiranje.

2. Višestruki mikroservisi. Kontejnerizacija aplikacije.

Tema ovih časova je rad sa Redis, distribuiranom 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**

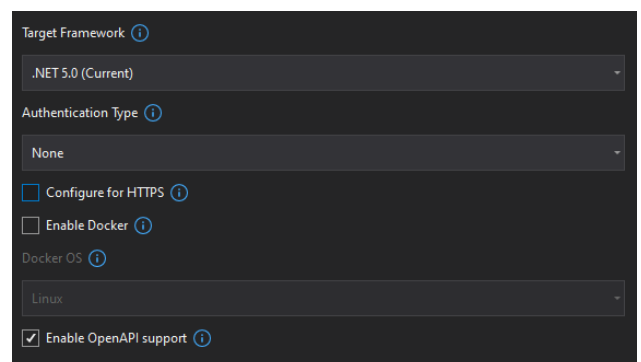
Kada se prikačimo za redis kontejner, dostupan nam je alat **redis-cli** kojim možemo izvršavati proizvoljne naredbe za upravljanje redis bazom. Neke osnovne komande ovog alata su:

- 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
- Odabrali 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:

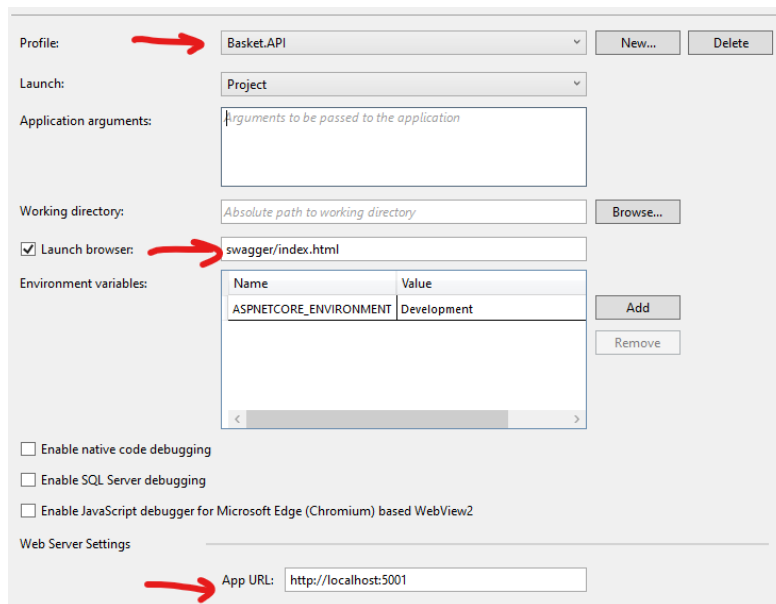
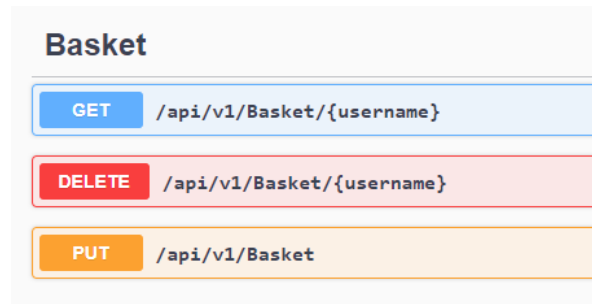
- Newtonsoft.Json
- Microsoft.Extensions.Caching.StackExchangeRedis

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.



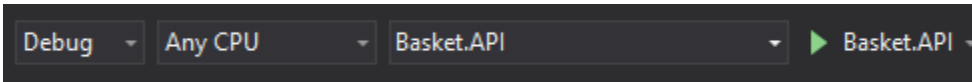
Zatim prelazimo na kodiranje, prema narednom redosledu:

- 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 debugiranje aplikacije

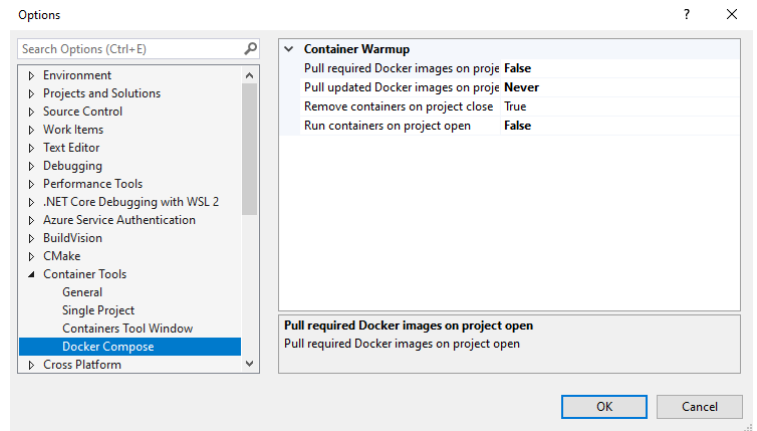
Iz gornjeg menija odabrati naredne opcije i pokrenuti aplikaciju:



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):

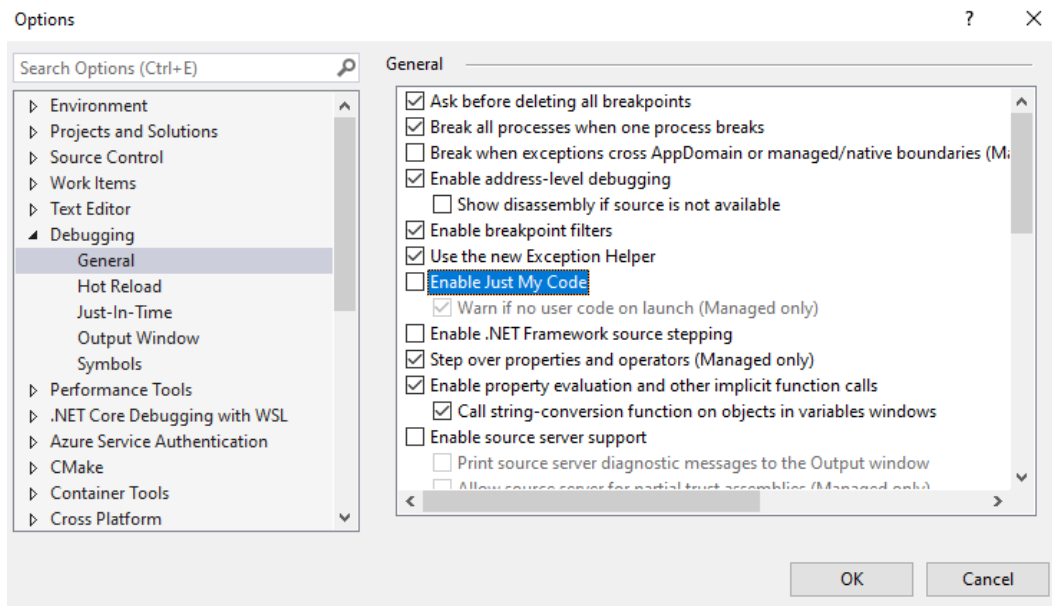
- Izgradnja svih kontejnera
 - **docker-compose build**
- Podizanje svih kontejnera
 - **docker-compose up -d**
- Izgradnja i podizanje svih kontejnera
 - **docker-compose up --build -d**
- Zaustavljanje svih kontejnera
 - **docker-compose stop**
- Zaustavljanje i uklanjanje svih kontejnera
 - **docker-compose 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**

Debugiranje projekata u kontejneru

Kako bismo omogućili debugiranje 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 debugira i kod koji je izgrađen u kontejnerima:

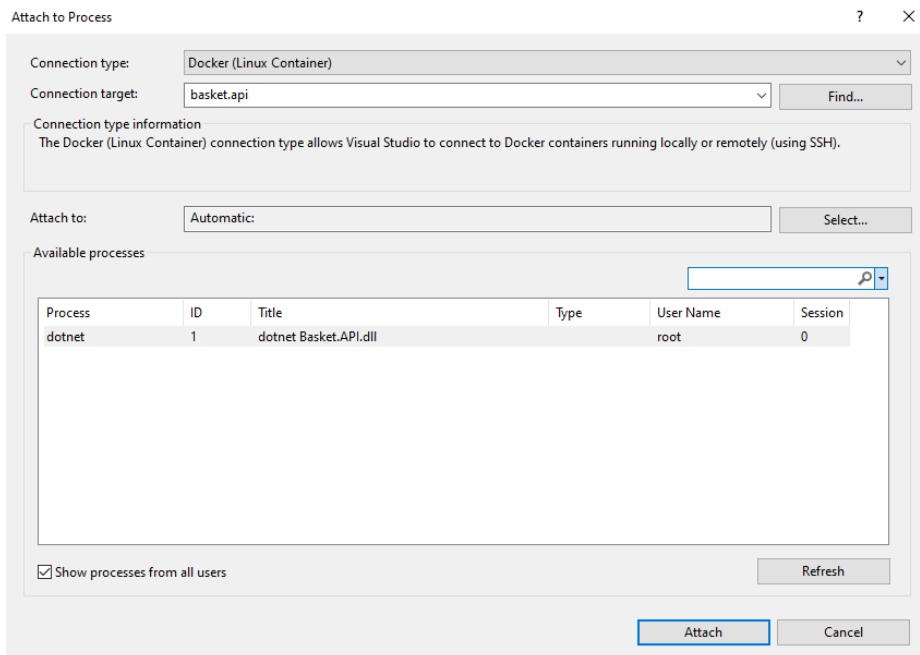
- Tools > Options
- Debugging > General

- Isključiti opciju „Enable Just My Code“, kao na slici ispod



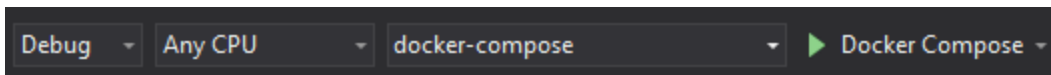
Sada možemo zakačiti debager:

- 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

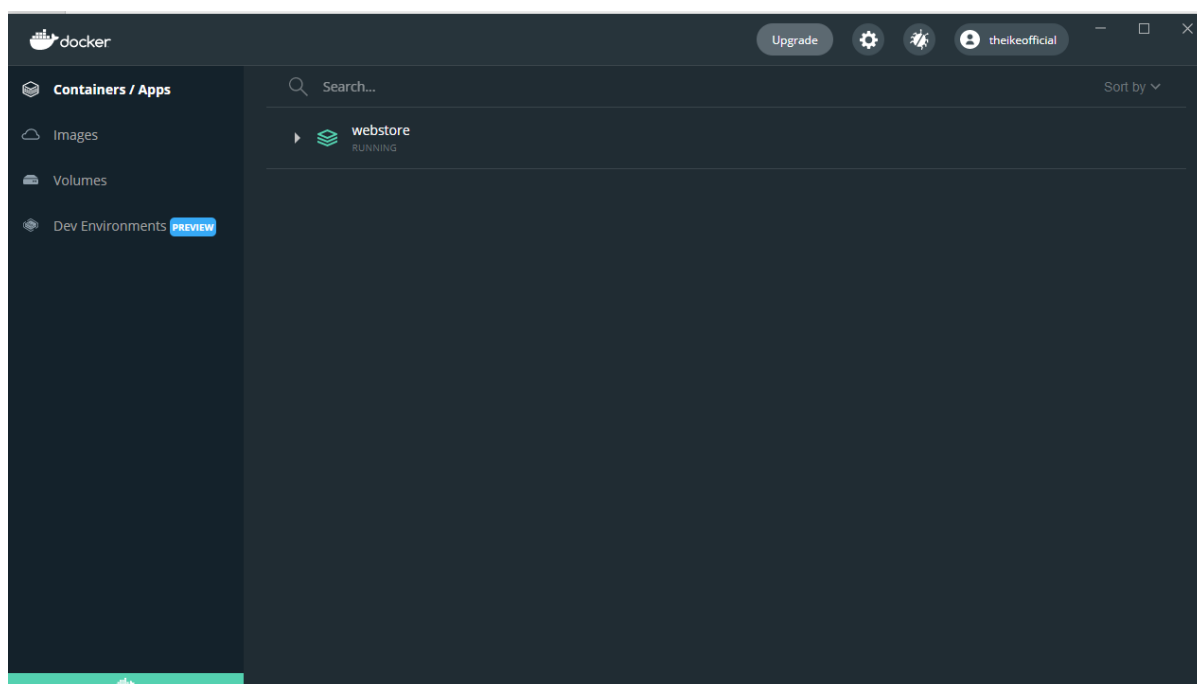
- Prednosti korišćenja su automatsko podizanje i automatsko debugiranje.
- Gašenjem projekta (bilo iz Visual Studio alata, gašenjem pregledača ili gašenjem terminala) se ne gase kontejneri, ali se ne mogu ugasiti iz **Docker Desktopa**, pa je neophodno uraditi **Build > Clean solution** iz glavnog menija – 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 prepisu sve promenljive okruženja iz Yaml datoteka u json pre pokretanja. Naravno, ovime se onemogućava pojedinačno pokretanje projekata iz Visual Studio alata.
- Debugiranje u kontejnerima je nešto sporije u odnosu na debugiranje 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.

The screenshot shows the Docker Desktop interface for a container named 'webstore'. The left sidebar lists 'Containers / Apps', 'Images', 'Volumes', and 'Dev Environments'. The main area displays a list of containers under the heading 'CONTAINERS':

- catalogdb mongo (RUNNING, PORT: 27017)
- basketdb redis:alpine (RUNNING, PORT: 6379)
- catalog.api catalogapi (RUNNING, PORT: 8000)
- basket.api basketapi (RUNNING, PORT: 8001)

The right pane shows the logs for the selected container, 'basketdb'. The logs include:

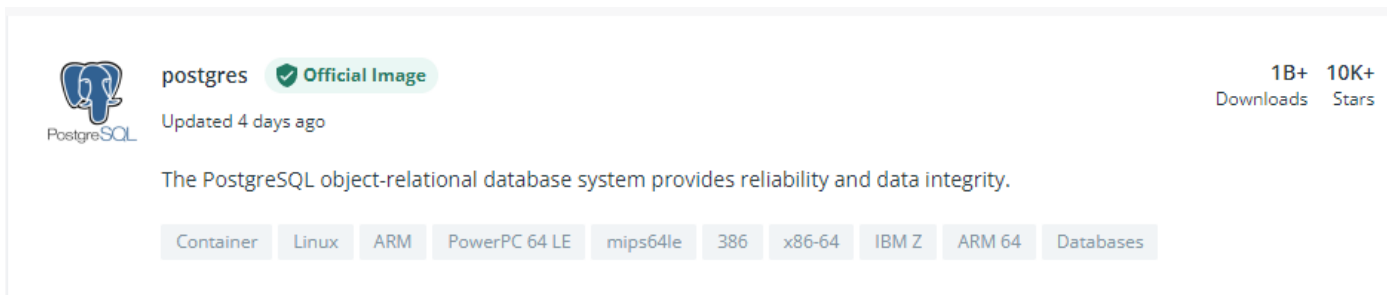
```
ta checkpoint timestamp: (0, 0) base write gen: 76"}}
basketdb | 1:C 22 Oct 2021 15:25:24.647 # Redis is starting o000o000
o000o
basketdb | 1:C 22 Oct 2021 15:25:24.647 # Redis version=6.2.6, bits=64, commit=000
00000, modified=0, pid=1, just started
basketdb | 1:C 22 Oct 2021 15:25:24.647 # Warning: no config file specified, using
the default config. In order to specify a config file use redis-
server /path/to/redis.conf
basketdb | 1:M 22 Oct 2021 15:25:24.647 * monotonic clock: POSIX clock_gettime
basketdb | 1:M 22 Oct 2021 15:25:24.648 * Running mode=standalone, port=6379.
basketdb | 1:M 22 Oct 2021 15:25:24.648 # Server initialized
basketdb | 1:M 22 Oct 2021 15:25:24.648 * Ready to accept connections
```

3. Sinhrona komunikacija između mikroservisa pomoću gRPC protokola

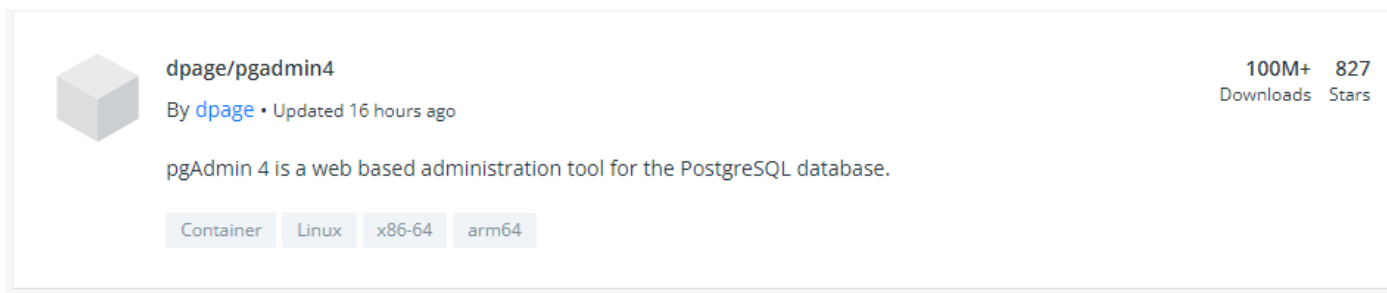
Tema ovih časova je ostvarivanje sinhronne komunikacije između mikroservisa implementiranjem gRPC protokola. Videćemo i rad sa PostgreSQL SUBP korišćenjem vrlo jednostavne biblioteke za ORP – Dapper.

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
- Odabрати 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
- Odabrali 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
- Odabrali Discount.Common
- Ok

Razvoj mikroservisa

API mikroservisa je opisan slikom pored.

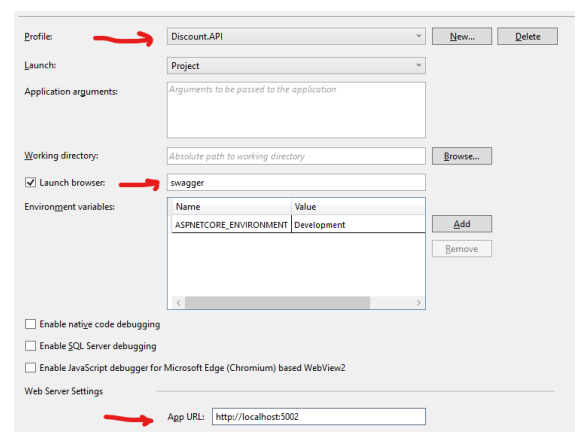
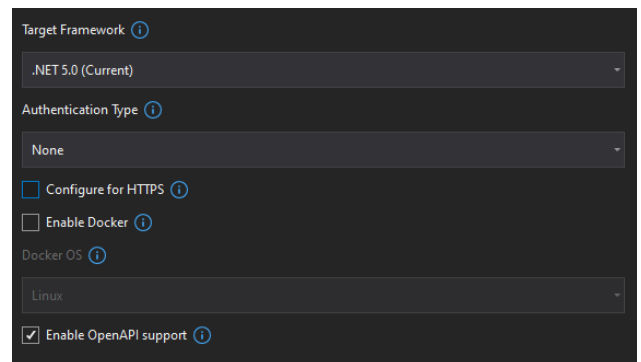
Prvo podešavamo opcije za pokretanje aplikacije:

- 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 debugiranje 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 PostgreSQL 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;"  
}
```

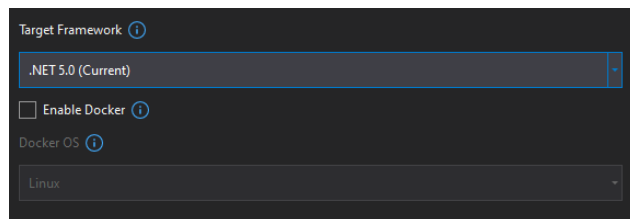
Sada dodati podršku za orkestrizaciju kontejnera za projekat Discount.API, pa pokrenuti sve kontejnere i testirati još jednom. Dopuniti **docker-compose.override.yml** datoteku narednim linijama:

```
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.

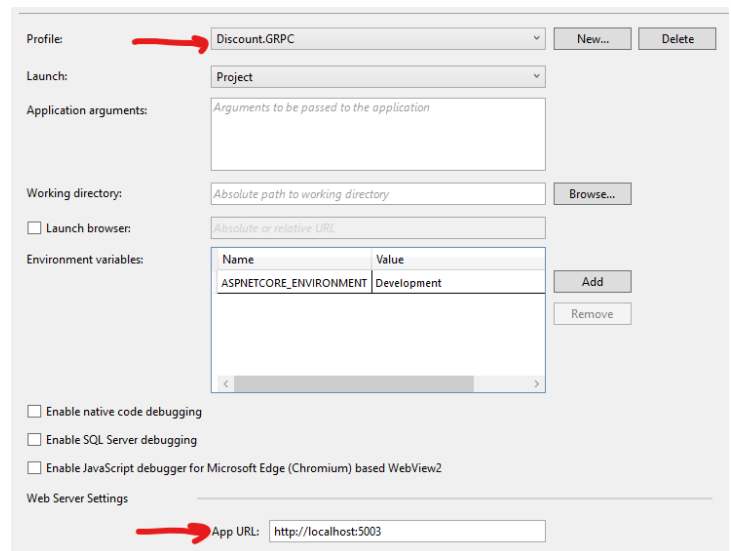
Razvoj mikroservisa

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();
});
```

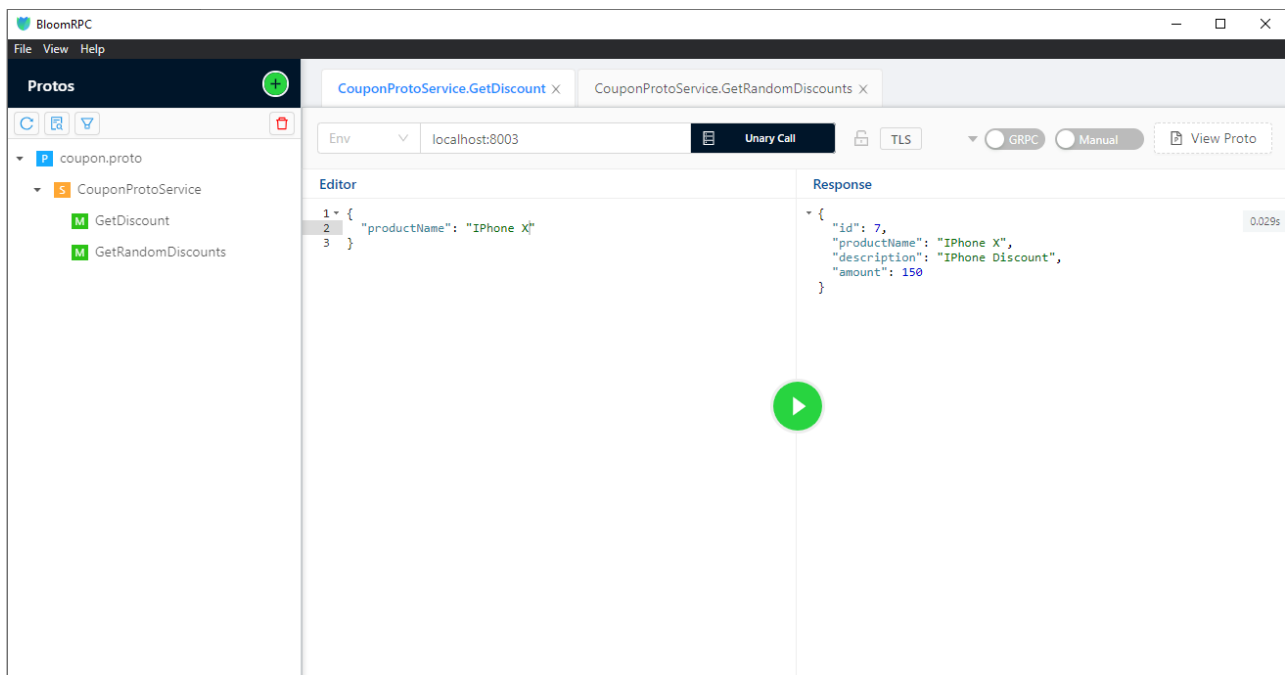
Pokretanje i debugiranje aplikacije

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 PostgreSQL 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"
}
```

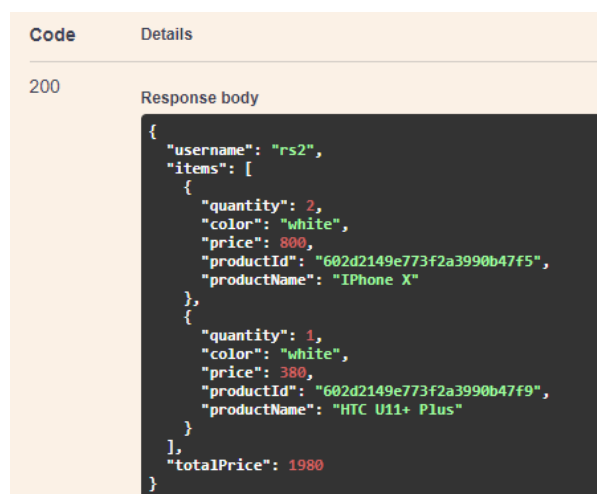
U `docker-compose.override.yml` datoteci:

```
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.



```
Code    Details
200     Response body
{
  "username": "rs2",
  "items": [
    {
      "quantity": 2,
      "color": "white",
      "price": 800,
      "productId": "602d2149e773f2a3990b47f5",
      "productName": "iPhone X"
    },
    {
      "quantity": 1,
      "color": "white",
      "price": 380,
      "productId": "602d2149e773f2a3990b47f9",
      "productName": "HTC U11+ Plus"
    }
  ],
  "totalPrice": 1980
}
```

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

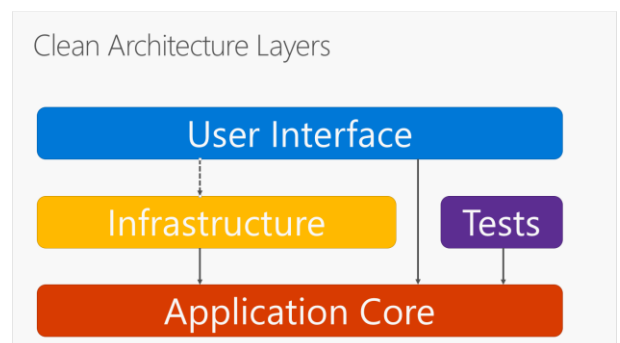
Slojevi u čistoj arhitekturi su:

- 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, ...)

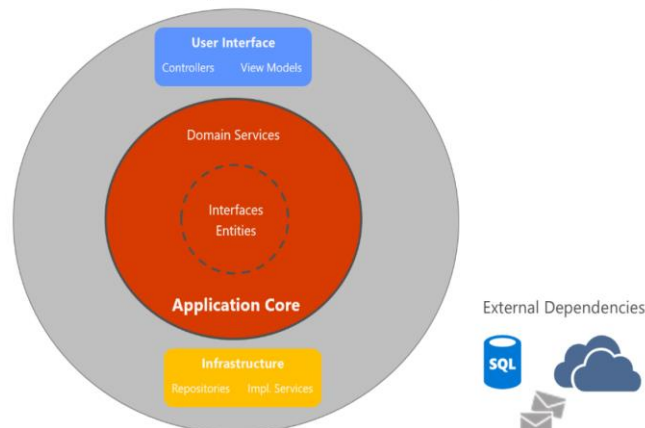
Na slici pored su prikazani ovi slojevi u horizontalnom rasporedu. Puna linija predstavlja zavisnost u fazi prevođenja koda, a isprekidana u fazi izvršavanja koda.

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

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

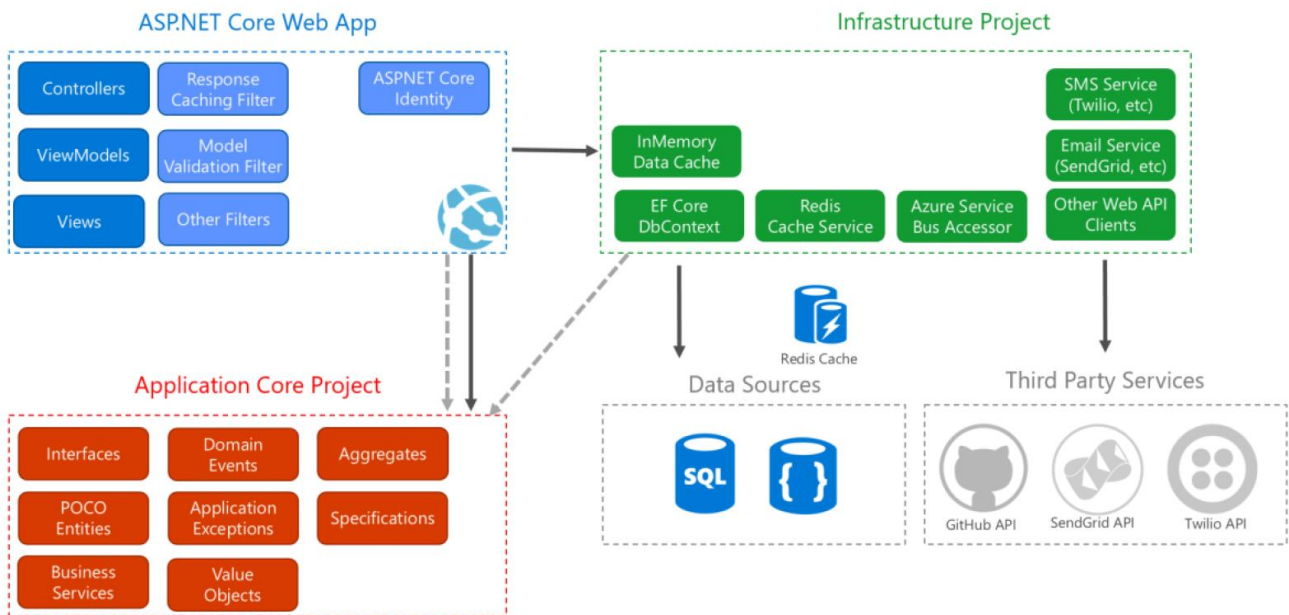


Clean Architecture Layers (Onion view)

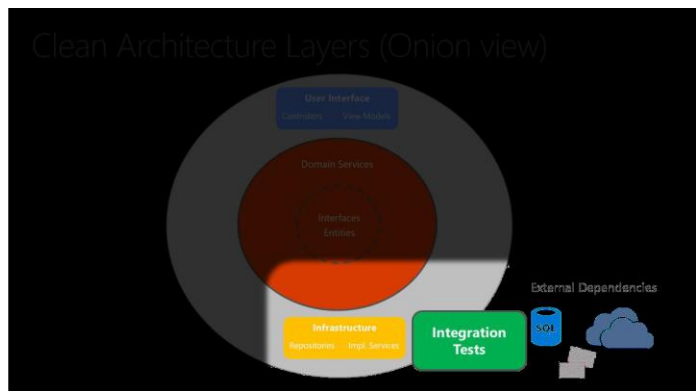
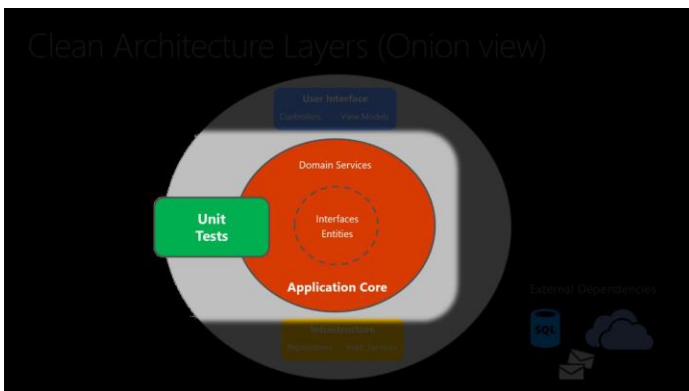


Naredna slika prikazuje još detaljniji pogled na istu arhitekturu:

ASP.NET Core Architecture



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 ubrzanje 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/eaDev/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:

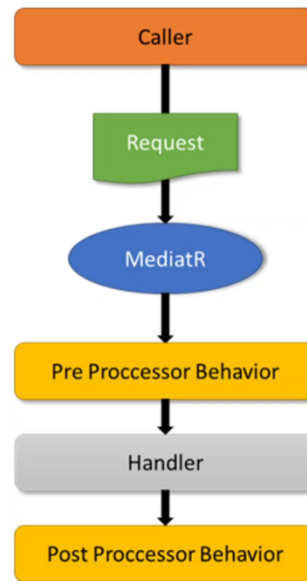
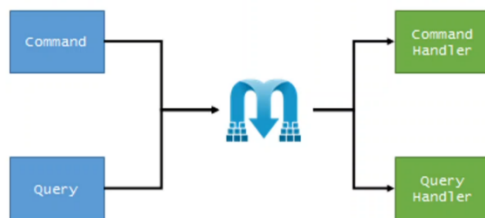
- Ordering.Domain

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.

MediatR Nuget Package



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/eaCatalog/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
 - Factories
 - IOrderFactory.cs
 - IorderViewModelFactory.cs

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

- Features
 - Orders

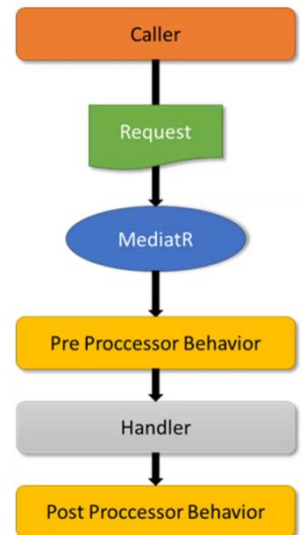
- Queries
 - GetListOfOrders
 - GetListOfOrdersQueryHandler.cs
- Commands
 - CreateOrder
 - CreateOrderCommandHandler.cs
 - UpdateOrder
 - UpdateOrderCommandHandler.cs
 - DeleteOrder
 - DeleteOrderCommandHandler.cs

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:

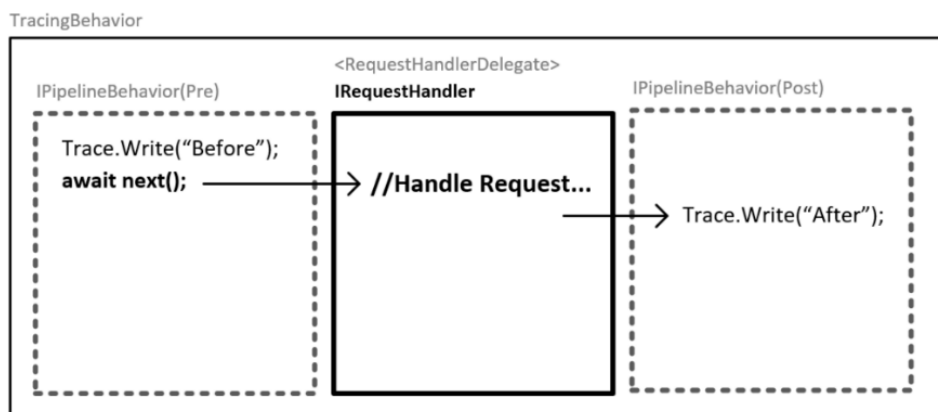
- Primer *pre processor behavior*: validacija podataka
- Primer *post processor behavior*: obrada grešaka

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



MediatR Pipeline Behavior



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 interfejs i klase koje drugi koriste. Redosled registracije ponašanja odgovara redosledu izvršavanja¹.

- 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.

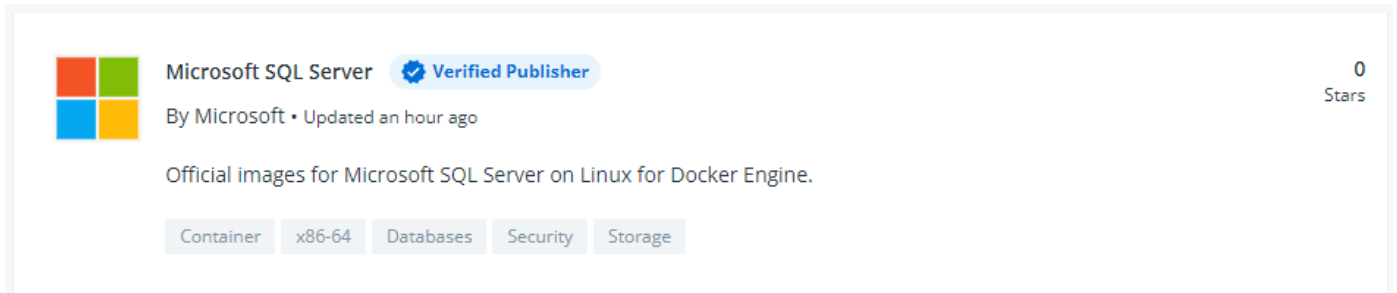
¹ Neko će se zapitati zašto registrujemo prvo **UnhandlerExceptionBehaviour**, pa tek onda **ValidationBehaviour** kada se prvo radi validacija, pa tek onda obrada grešaka. Važno je razumeti koje ponašanje se izvršava pre neke akcije, a koje ponašanje posle neke akcije. Ako pogledamo definicije ovih 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.

5. Entity Framework Core

Tema ovih časova je Entity Framework Core. U pitanju je radni okvir za ORP. Dodatno, korišćemo Sql Server RSUBP za skladištenje podataka.

1. Priprema SQL Server kontejnera

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



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

services:

orderdb:

image: mcr.microsoft.com/mssql/server:2017-latest

volumes:

mssql_data:

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

services:

orderdb:

container_name: orderdb

environment:

- SA_PASSWORD=MATF12345678rs2

- ACCEPT_EULA=Y

restart: always

ports:

- "1433:1433"

volumes:

- mssql_data:/var/opt/mssql/data

2. Ordering.Infrastructure projekat

Tip projekta: Class library

Paketi:

- Microsoft.EntityFrameworkCore.SqlServer
- MailKit
- MimeKit

Zavisnosti od drugih slojeva:

- Ordering.Application

U EF Core radnom okviru, prvi korak jeste definisanje konteksta za jednu bazu podataka. Ovaj kontekst služi za definisanje podešavanja i tabele koji su vezani za bazu podataka koja se koristi. Videćemo kasnije kako se koristi ovaj kontekst za definisanje migracija baza podataka.

Posebno obraćamo pažnju na implementaciju ORP između agregata/entiteta i vrednostih objekata. Vrednosti objekti nemaju identitet, pa po definiciji, ne smeju se čuvati kao nezavisni objekti u BP (tj. objekti koji imaju svoj, nezavisan identifikator). Više o ovoj strategiji u članku <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/implement-value-objects#persist-value-objects-as-owned-entity-types-in-ef-core-20-and-later>.

Implementacija:

- Persistence
 - EntityConfigurations
 - OrderItemEntityTypeConfiguration.cs
 - OrderEntityTypeConfiguration.cs
 - OrderContext.cs
 - OrderContextSeed.cs
- Repositories
 - RepositoryBase.cs²
 - OrderRepository.cs
- Factories
 - OrderFactory.cs
 - OrderViewModelFactory.cs
- Mail
 - EmailService.cs³⁴

Konačno, definišemo klasu proširenja **InfrastructureServiceRegistration** za ubrizgavanje zavisnosti implementiranih servisa i koristimo ovu klasu u API projektu. Ne zaboraviti na dodavanje niske za konekciju na Sql Server i podešavanja za elektronsku poštu⁵:

```
"ConnectionStrings": {
  "OrderingConnectionString": "Server=localhost;Database=OrderDb;User Id=sa;Password=MATF12345678rs2;"
},
"EmailSettings": {
  "Mail": "destin.schroeder4@ethereal.email",
  "DisplayName": "Webstore App",
  "Password": "m7y9PEZzayQj7nkYCYZ",
  "Host": "smtp.ethereal.email",
  "Port": 587
}
```

² Važna napomena za efikasnu implementaciju operacija nad bazom podataka: **preferirati IQueryable interfejs nad IEnumerable interfejsom**. Više o tome na <https://www.tutorialspoint.com/what-is-the-difference-between-ienumerable-and-iqueryable-in-chash> i <https://medium.com/@mohamedabdeen/iqueryable-vs-ienumerable-in-net-92a15a803da3>.

³ Podešavanje za Gmail SMTP server: <https://support.google.com/a/answer/176600#zippy=%2Cuse-the-gmail-smtp-server>.

⁴ Članak koji opisuje jednostavno slanje elektronske pošte putem Gmail i Ethereal servisa: <https://codewithmukesh.com/blog/send-emails-with-aspnet-core/>.

⁵ Jednostavno kreiranje naloga za *mockup* SMTP server: <https://ethereal.email/>.

3. Migracije baza podataka

Migracija predstavlja proces inkrementalnog rekreiranja baze podataka na (najčešće) udaljenom SUBP. Pod inkrementalnim rekreiranjem podrazumevamo da se BP menja postepeno, tokom životnog ciklusa projekta. Migracije su važne zato što često ne možemo na početku projekta predvideti kako će sheme podataka tačno izgledati, već se vremenom te sheme menjaju, pa je neophodno promeniti i samu BP. Više o migracijama je dostupno u članku <https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>.

Migracija se može uraditi ručno, povezivanjem na SUBP i izvršavanjem naredbi (kao što smo radili za **Discount.API** projekat) ili automatski, korišćenjem raznih alata za automatizaciju ovog postupka. EF Core nudi takve alate koji su zasnovani na tzv. *code-first* pristupu, gde se na osnovu izvornog koda generišu klase koji će biti izvršeni nad SUBP-a kada se projekat pokrene. Ovi alati su dostupni u Visual Studio alatu kroz „Package Manager Console“ prozor, a iz terminala operativnog sistema kroz **dotnet ef** alat (više o ovom alatu na <https://docs.microsoft.com/en-us/ef/core/cli/dotnet>). Otvaranje prozora u Visual Studio alatu se može uraditi narednim koracima:

- Tools > Nuget Package Manager > Package Manager Console

Na početku, kada se kod implementira, potrebno je dodati inicijalnu migraciju.

- Otvoriti Package Manager Console
- Odabrati „Services\Ordering\Ordering.Infrastructure“ kao podrazumevani projekat na vrhu konzole
- Pokrenuti komandu „Add-Migration InitialCreate“

Rezultat ove akcije jeste generisana klasa koja je oblika **<VREMENSKA_ODREDNICA>_InitialCreate.cs** i nalazi se u direktorijumu **Migrations**. Ona sadrži metod **Up** za „podizanje“ migracije.

Postoji još nekoliko operacija koje su od značaja i koje su navedene u nastavku. Svaki put kada se neka od ovih operacija izvrši, kreira se nova klasa koja sadrži novi kod koji menja BP kako bi u svakom trenutku stanje u BP odgovaralo stanju u kodu.

- Dodavanje nove izmene:
 - PMC: <https://docs.microsoft.com/en-us/ef/core/cli/powershell#add-migration>
 - CLI: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#dotnet-ef-migrations-add>
- Poništavanje poslednje izmene:
 - PMC: <https://docs.microsoft.com/en-us/ef/core/cli/powershell#remove-migration>
 - CLI: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#dotnet-ef-migrations-remove>
- Izmena BP do poslednje, ili neke od ređene izmene:
 - PMC: <https://docs.microsoft.com/en-us/ef/core/cli/powershell#update-database>
 - CLI: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#dotnet-ef-database-update>
- Uklanjanje baze podataka:
 - PMC: <https://docs.microsoft.com/en-us/ef/core/cli/powershell#drop-database>
 - CLI: <https://docs.microsoft.com/en-us/ef/core/cli/dotnet#dotnet-ef-database-drop>

Sada je potrebno da napišemo klasu proširenja u **Ordering.API** projektu koju ćemo koristiti kako bismo rekli aplikaciji da izvrši migraciju:

- Extensions
 - HostExtensions.cs
- Program.cs

Sada je moguće testirati API projekat iz Visual Studio okruženja. Pre dodavanja migracije bi pokušaj pristupanja rezultovao greškom u fazi povezivanja na bazu podataka.

4. Kontejnerizacija Ordering.API projekta

Prvo dodajemo podršku za orkestrizaciju za Docker Compose alat. Zatim, dodaje mo naredne opcije u **docker-compose.override.yml** datoteci:

services:

ordering.api:

container_name: ordering.api

environment:

- ASPNETCORE_ENVIRONMENT=Development

- "ConnectionStrings:OrderingConnectionString=Server=orderdb;Database=OrderDb;User

Id=sa;Password=MATF12345678rs2"

depends_on:

- orderdb

ports:

- "8004:80"

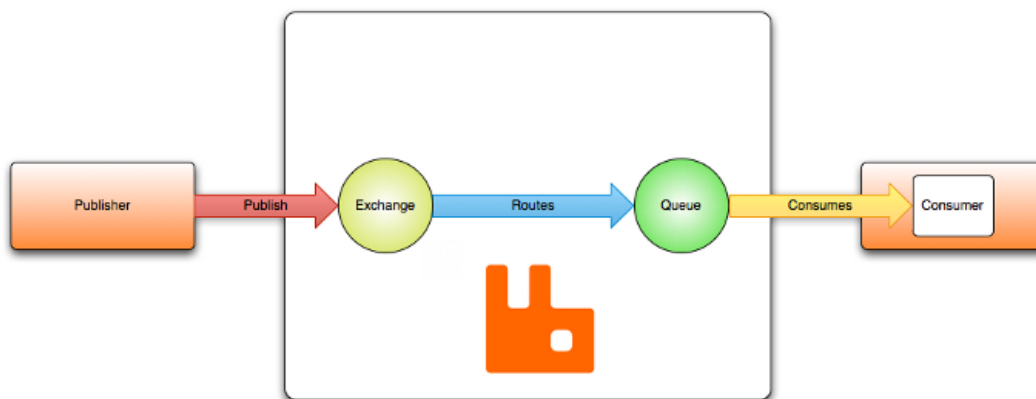
6. Asinhrona komunikacija između mikroservisa pomoću reda poruka

Tema ovih časova je ostvarivanje asinhronne komunikacije između mikroservisa implementiranjem RabbitMQ reda poruka.

1. Osnovni elementi RabbitMQ posrednika

Posrednici (eng. *broker*) predstavljaju sisteme koji učestvuju u komunikaciji između više aplikacija radi ostvarivanja nekog slučaja upotrebe. U savremenim aplikacijama, posrednici se koriste za implementaciju asinhronih akcija na nivou sistema, odnosno, za ostvarivanje asinhronne komunikacije između komponenti tog sistema. RabbitMQ je primer jednog posrednika. Da bismo razumeli kako posrednici omogućavaju asinhronu saradnju, potrebno je da razumemo osnovne elemente AMQP 0-9-1 protokola koji se koristi u RabbitMQ sistemu.

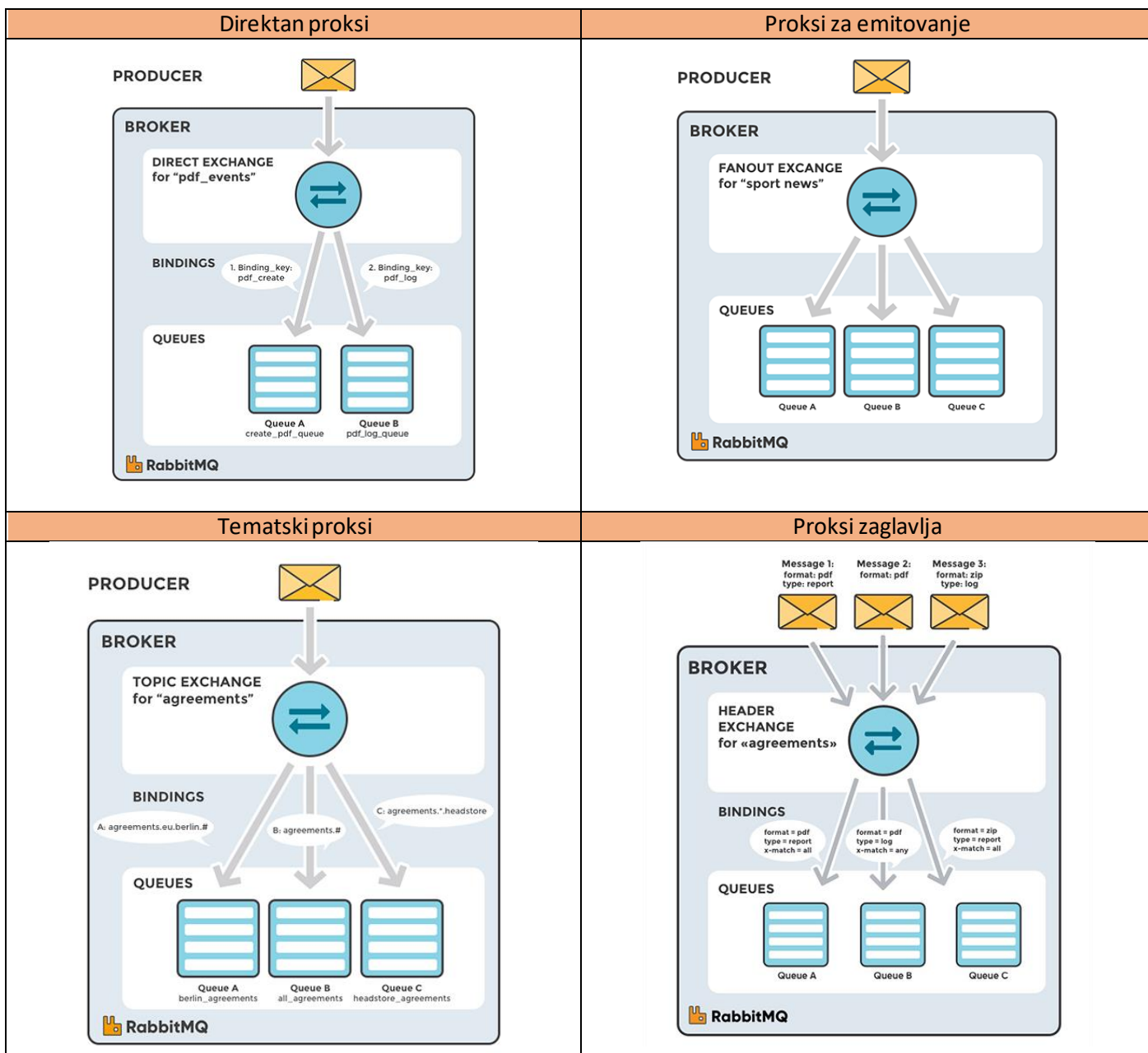
"Hello, world" example routing



Osnovni elementi AMQP (Advanced Message Queuing Protocol) protokola su:

- *Poruka* (eng. *message*) predstavlja objekat koji nosi neku informaciju.
- *Proizvođač* (eng. *producer*) jeste aplikacija koja kreira i šalje poruke.
- *Red* (eng. *queue*) predstavlja bafer koji čuva poruke. Svaki red može imati neke važne karakteristike kojima se može definisati specifično ponašanje tog reda:
 - Ime (za prepoznavanje reda),
 - Trajnost (red „preživljava“ restartovanje posrednika),
 - Ekskluzivnost (koristi se samo od strane jedne konekcije i biće uništen kada se konekcija završi),
 - Automatsko brisanje (red koji ima makar jednog pretplatnika biće obrisano kada se poslednji pretplatnik otplati sa njega),
 - Argumenti (opciona svojstva kao što su, na primer, najveća dužina reda, vreme tokom kojeg će poruka ili red ostati živi, itd.).
- *Pretplatnik* (eng. *consumer*) predstavlja aplikaciju koja dobija poruke.
- *Proksi* (eng. *exchange*) jeste podsistem koji dobija poruke od proizvođača i odlučuje na koji red (ili na više njih) dodaje poruku koju je dobio. Kako bi odlučio koji redovi će dobiti novu poruku, proksiji moraju definisati svoj tip. U RabbitMQ imamo četiri tipa proksija:
 - *Direktan proksi* (eng. *direct*), koji šalje poruku u redove na osnovu *ključa za rutiranje* (routing key). Red veže direktan proksi za sebe pomoću nekog ključa za rutiranje K . Kada nova poruka sa ključem za rutiranje R dođe do direktnog proksija, taj proksi šalje poruku na red ako važi $R=K$.

- *Proksiza emitovanje* (eng. *fanout*), koji šalje poruku svim redovima koji su vezani za njega, bez obzira na ključeve za rutiranje.
- *Tematski proksi* (eng. *topic*), koji šalje poruku na jedan red ili više njih na osnovu poklapanja između ključa za rutiranje i šablona koji se koristio za vezivanje reda na proksi.
- *Proksizaglavlja* (eng. *headers*), koji koristi složeniji skup atributa za određivanje pravila za rutiranje u odnosu na jednostavan ključ za rutiranje. Ovaj skup atributa nazivamo *zaglavljem* poruke.



2. Priprema RabbitMQ kontejnera

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

rabbitmq Official Image

Updated 10 hours ago

RabbitMQ is an open source multi-protocol messaging broker.

1B+ **4.1K**

Downloads Stars

Container
Linux
ARM 64
IBM Z
386
x86-64
PowerPC 64 LE
riscv64
ARM
Messaging Services

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

```
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
```

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

```
services:
  rabbitmq:
    container_name: rabbitmq
    restart: always
    ports:
      - "5672:5672"
      - "15672:15672"
```

Koristimo „management“ verziju kontejnera kako bismo dobili i UI alat koji se zove *Management plugin* i koji nam omogućava da pratimo stanje u redu poruka. Sam red poruka će biti pokrenut na portu **5672**, dok će alat biti pokrenut na portu **15672**. Više o Management pluginu na: <https://www.rabbitmq.com/management.html>.

Podrazumevano korisničko ime i lozinka za pristup alatu je **guest/guest**.

3. EventBus.Messages projekat

Ovaj projekat nam služi za definisanje zajedničkih elemenata koji će imati svi mikroservisi, te je ovo dobro mesto za definisanje klasa za poruke koje će mikroservisi razmenjivati, kao što je poruka „kupac je završio kupovinu“ u našoj prodavnici.

Tip projekta: Class library

Implementacija:

- Events
 - IntegrationBaseEvent.cs
 - BasketCheckoutEvent.cs
- Common
 - EventBusConstants.cs

4. Objavljivanje poruke iz Basket API mikroservisa

U **Basket.API** projektu moramo dodati referencu na **EventBus.Messages** projekat.

Dodatno, potrebno je da instaliramo NuGet pakete za korišćenje redova poruka:

- MassTransit
- MassTransit.RabbitMQ
- MassTransit.AspNetCore
- AutoMapper.Extensions.Microsoft.DependencyInjection

U **Startup.cs** datoteci dodati podešavanje za red poruka i ubrizgavanje zavisnosti:

```
services.AddMassTransit(config => {
    config.UsingRabbitMq((ctx, cfg) => {
        cfg.Host(Configuration["EventBusSettings:HostAddress"]);
    });
});
```

```
services.AddMassTransitHostedService();
```

U `appsettings.development.json` datoteci dodati URI za konekciju na red poruka:

```
"EventBusSettings": {
  "HostAddress": "amqp://guest:guest@localhost:5672"
}
```

Implementacija:

- Entities
 - BasketCheckout.cs
 - BasketItem.cs
- Controllers
 - BasketController.cs
- Mapper
 - BasketProfile.cs

Dodati DI za AutoMapper u `Startup.cs` datoteci.

5. Pretplaćivanje na poruke u Ordering API mikroservisu

Dodati iste reference, NuGet pakete i podešavanja kao za **Basket.API** projekat u prethodnoj sekciji. Jedina razlika jeste što se Ordering API koristi kao pretplatnik u redu poruka, pa je neophodno dopuniti podešavanje za MassTransit⁶.

```
services.AddMassTransit(config => {
  config.AddConsumer<BasketCheckoutConsumer>();
  config.UsingRabbitMq((ctx, cfg) => {
    cfg.Host(Configuration["EventBusSettings:HostAddress"]);
    cfg.ReceiveEndpoint(EventBusConstants.BasketCheckoutQueue, c =>
    {
      c.ConfigureConsumer<BasketCheckoutConsumer>(ctx);
    });
  });
});
services.AddMassTransitHostedService();
```

Implementacija:

- EventBusConsumers
 - BasketCheckoutConsumer.cs
- Mapper
 - OrderingProfile.cs

Dodati DI za AutoMapper i klasu `BasketCheckoutConsumer` u `Startup.cs` datoteci.

⁶ Ovo ne znači da Ordering API ne može biti i proizvođač i pretplatnik, već samo ako je i pretplatnik, onda treba registrovati klase koje će obrađivati pretplate na odgovarajuće poruke.

6. Kontejnerizacija Basket API i Ordering API mikroservisa sa RabbitMQ kontekstom

S obzirom da smo dodali novu projektnu referencu (**EventBus.Messages**) u ovim mikroservisima, potrebno je da ažuriramo **Dockerfile** datoteke kako bi se projekti ispravno izgradili. To je moguće uraditi ručno, ali bolje je da ostavimo Visual Studio alatu da to automatski uradi za nas. U oba projekta uraditi naredne korake:

- Obrisati Dockerfile datoteku.
- Desni klik na ime projekta > Add > Docker Support...
- Target OS: Linux.

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

services:

basket.api:

container_name: basket.api

environment:

- ASPNETCORE_ENVIRONMENT=Development
- "CacheSettings:ConnectionString=basketdb:6379"
- "GrpcSettings:DiscountUrl=http://discount.grpc"
- "EventBusSettings:HostAddress=amqp://guest:guest@rabbitmq:5672"

depends_on:

- basketdb
- discount.grpc
- rabbitmq

ports:

- "8001:80"

ordering.api:

container_name: ordering.api

environment:

- ASPNETCORE_ENVIRONMENT=Development
- "ConnectionStrings:OrderingConnectionString=Server=orderdb;Database=OrderDb;UserId=sa;Password=MATF12345678rs2"
- "EventBusSettings:HostAddress=amqp://guest:guest@rabbitmq:5672"

depends_on:

- orderdb
- rabbitmq

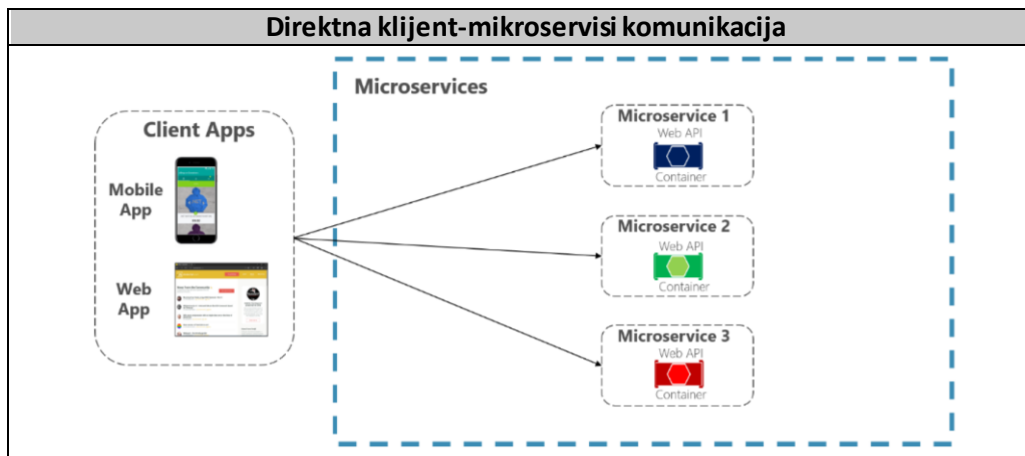
ports:

- "8004:80"

7. Mrežni prolazi

Tema ovih časova je izgradnja mrežnog prolaza pomoću Ocelot sistema.

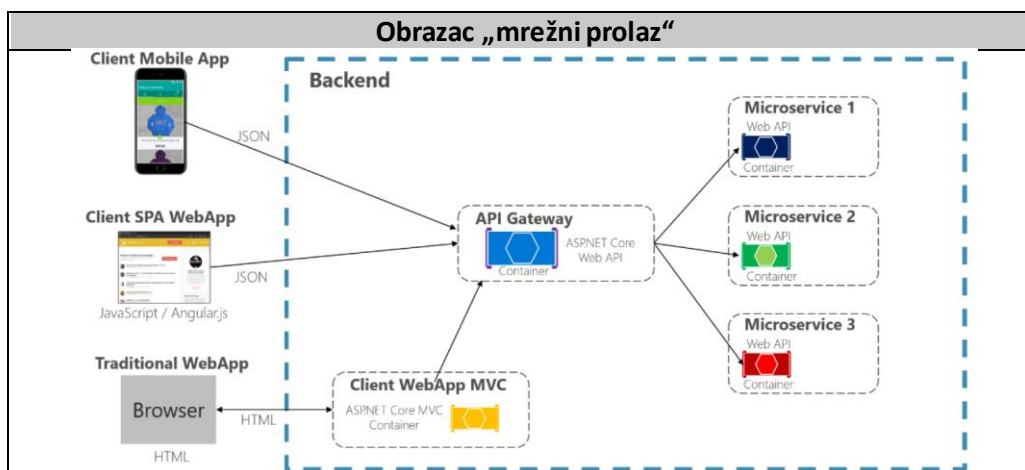
Svaki mikroservis koji radi u produkciji ima svoj jedinstveni HTTP URI putem kojeg se može komunicirati sa njime od strane nekih klijentskih aplikacija. Ukoliko klijenti imaju informaciju o tim URI-jima, onda oni mogu direktno slati zahteve tim mikroservisima.



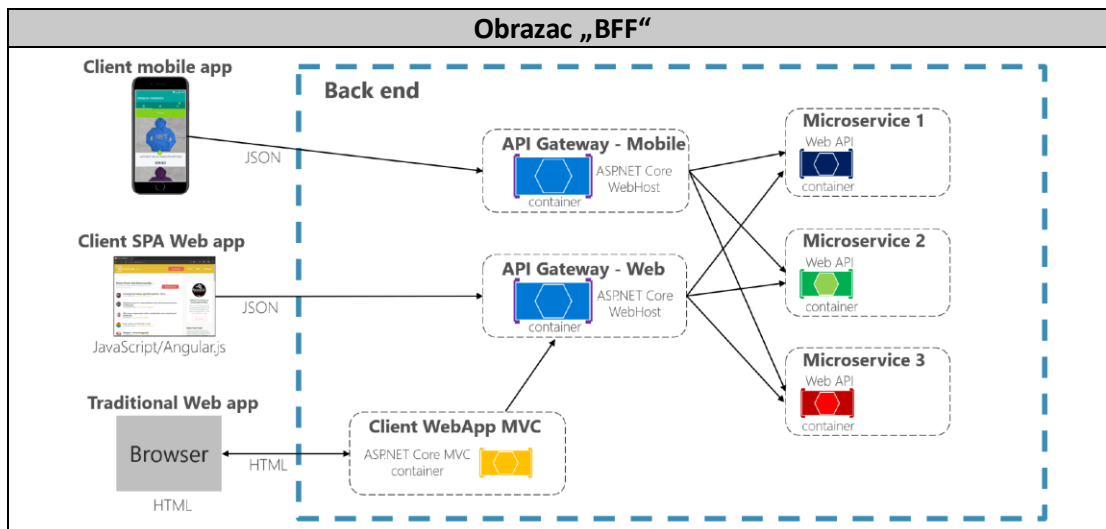
Međutim, ovde imamo nekoliko potencijalnih problema:

- Ako za neki slučaj upotrebe klijentska aplikacija treba da kontaktira više mikroservisa, onda time dolazi u situaciju da pravi veliki broj HTTP poziva. Takođe, klijentska aplikacija mora da „agregira“ rezultate od tih mikroservisa, čime se uvećava njena složenost. U idealnom scenariju, ovakve „agregacije“ rezultata bi trebalo da se rade na serverskoj strani jer su joj dostupni efikasniji protokoli za komunikaciju (na primer, gRPC).
- U ovakvom modelu, neke stvari se moraju implementirati na svakom mikroservisu, kao što je bezbednost mikroservisa. Na primer, ako neki klijent pošalje zahtev mikroservisima Basket, Ordering i Catalog za izvršavanje nekog slučaja upotrebe, **svi ti mikroservisi** moraju prvo da identifikuju koji korisnik je u pitanju (autentifikacija), a potom da provere da li taj korisnik ima neophodne privilegije da izvrši odgovarajuću akciju (autorizacija).
- Klijentske aplikacije ne mogu da koriste pogodne protokole za komunikaciju sa servisima, kao što su AMQP, veće se ovakvi zahtevi moraju interno transformisati.
- Ne postoji način za definisanje „fasada“ za specifične tipove aplikacija, kao što su mobilne aplikacije.

Svi ovi problemi se mogu rešiti uvođenjem *mrežnog prolaza* (eng. *gateway*).



Često se koristi i više mrežnih prolaza za različite vrste aplikacija. Tada govorimo o *BFF* (eng. *Backend For Frontend*) obrascu. Ideja je da za svaku vrstu aplikacija definišemo različite API-je kojima oni pristupaju. Na primer, ako je klijentska aplikacija uprošćena verzija veb aplikacije, onda nema razloga da ona pristupa svim API-jima koji su dostupni veb aplikaciji.



1. Kreiranje Ocelot mikroservisa

Kreiramo novi projekat:

- Desni klik na naziv *Solution*-a
- Add > New Solution Folder
- Imenujemo direktorijum „ApiGateways“
- Desni klik na „ApiGateways“
- Location: Add > New Project
- ASP.NET Core Empty
- Popuniti neophodnim podacima:
 - Project name: **SPAGateway**
 - Location: **LOKACIJA_LOKALNOG_REPOZITORIJUMA\ApiGateways**
- Next
- Odabrati opciju **.NET 5.0 (Current)**
- Create

Instalirati naredne NuGet pakete:

- Ocelot

Sada je potrebno izmeniti podešavanja:

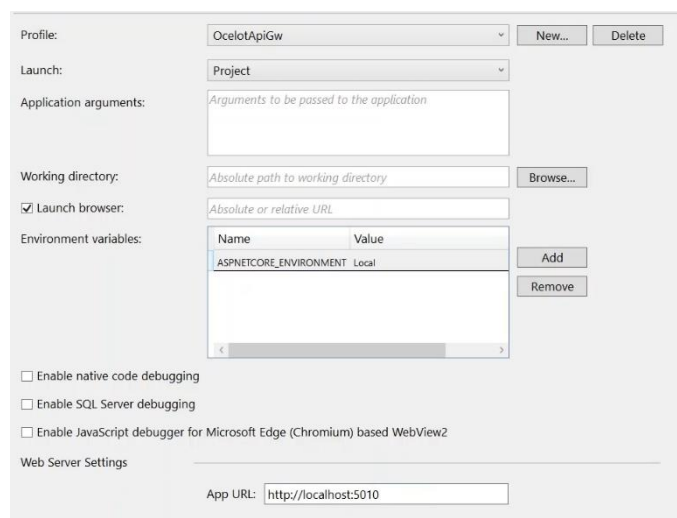
- Desni klik na naziv projekta > Properties
- Podesiti kao na slici pored.

U datoteci **Startup.cs** dodati DI za Ocelot:

services.AddOcelot();

U istoj datoteci u **Configure** metodi dodati *pipeline* za Ocelot middleware:

await app.UseOcelot();



2. Razvoj Ocelot mrežnog prolaza

Podešavanje Ocelot-a se vrši pomoću JSON datoteka. Da bismo napravili preslikavanje ruta javnog API-ja na rute ka mikroservisima, potrebno je da napravimo JSON datoteku koja će opisivati ova preslikavanja.

Kreiramo 3 ovakve JSON datoteke:

- **ocelot.json** koja definiše opšta pravila.
- **ocelot.Local.json** koja definiše pravila kada pokrećemo aplikacije lokalno.
- **ocelot.Development.json** koja definiše pravila kada pokrećemo aplikacije iz Docker-a.

Na osnovu tipa okruženja (**ASPNETCORE_ENVIRONMENT**), sistem treba da učitava odgovarajuću datoteku. U datoteci **Program.cs** dodajemo konfiguraciju za učitavanje JSON datoteke:

Host.CreateDefaultBuilder(args)

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddJsonFile($"ocelot.{hostingContext.HostingEnvironment.EnvironmentName}.json", true, true);
})
...

```

Ocelot JSON datoteka se obično sastoji od dva svojstva: **Routes** i **GlobalConfiguration**. Svojstvo **Routes** predstavlja niz koji sadrži definicije preslikavanja ruta, a svojstvo **GlobalConfiguration** predstavlja objekat koji sadrži podešavanja koja važe za sve rute.

Svako preslikavanje ruta se definiše jednim JSON objektom koji se nalazi u nizu **Routes**. Ovi objekti imaju nekoliko vrsta svojstava, od kojih su najvažniji oni koji počinju rečima „Downstream“ i „Upstream“. Prva svojstva se odnose na unutrašnje putanje ka mikroservisima, a druga svojstva se odnose na putanje koje će biti vidljive klijentskim aplikacijama. Svako preslikavanje ruta mora imati i „Downstream“ i „Upstream“ svojstva.

U narednoj tabeli je dat primer jednog preslikavanja skupa ruta iz Catalog API-ja:

U lokalnom okruženju:	U Docker okruženju:
<pre>{ "DownstreamPathTemplate": "/api/v1/Catalog", "DownstreamScheme": "http", "DownstreamHostAndPorts": [{ "Host": "localhost", "Port": "5000" }], "UpstreamPathTemplate": "/Catalog", "UpstreamHttpMethod": ["GET", "POST", "PUT"] }</pre>	<pre>{ "DownstreamPathTemplate": "/api/v1/Catalog", "DownstreamScheme": "http", "DownstreamHostAndPorts": [{ "Host": "catalog.api", "Port": "80" }], "UpstreamPathTemplate": "/Catalog", "UpstreamHttpMethod": ["GET", "POST", "PUT"] }</pre>

Objekat **GlobalConfiguration** obavezno ima svojstvo **BaseUrl** koji predstavlja URI preko kojeg će klijenti kontaktirati Ocelot radi. U našem slučaju, to će biti <http://localhost:5010>, kao što smo definisali u podešavanjima projekta iznad.

Ocelot podržava neke korisne opcije, kao što su:

- Limitiranje zahteva: <https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html>
- Keširanje: <https://ocelot.readthedocs.io/en/latest/features/caching.html>
- Agregiranje zahteva: <https://ocelot.readthedocs.io/en/latest/features/requestaggregation.html>
- Autentikacija: <https://ocelot.readthedocs.io/en/latest/features/authentication.html>

- Autorizacija: <https://ocelot.readthedocs.io/en/latest/features/authorization.html>

3. Kontejnerizacija Ocelot mikroservisa

Dodavanje podrške za Docker Compose projektu:

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

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

services:

ocelotapigateway:

container_name: ocelotapigateway

environment:

- ASPNETCORE_ENVIRONMENT=Development

depends_on:

- catalog.api

- basket.api

- discount.api

- ordering.api

ports:

- "8010:80"

8. Bezbednost mikroservisnih aplikacija

Tema ovih časova jeste uvođenje sigurnosnih mehanizama u mikroservisne aplikacije.

Do sada smo naučili kako da razvijamo mikroservisne aplikacije. Trenutno stanje naše aplikacije je takvo da svako može da pošalje bilo koji dostupni HTTP zahtev našim API-jima. Takve API-je nazivamo *javnim*. Međutim, često ne želimo da proizvoljni klijenti, aplikacije ili neki drugi sistemi (u daljem tekstu, *akteri*) imaju javni pristup našim API-jima. Zbog toga je neophodno da uvedemo neke sigurnosne mehanizme:

- **Autentifikacija** (eng. *authentication*) predstavlja identifikovanje aktera i verifikaciju tog identiteta.

Identifikovanje se može vršiti na nekoliko načina:

- Identifikovanje na osnovu *saznanja* (knowledge-based authentication)
 - Lozinka, PIN, ...
- Identifikovanje na osnovu *vlasništva* (ownership-based authentication)
 - Sertifikat, kartica, USB, ...
- Identifikovanje na osnovu *nasleđa* (inherence-based authentication)
 - Otisak prsta, DNA sekvenca, ...

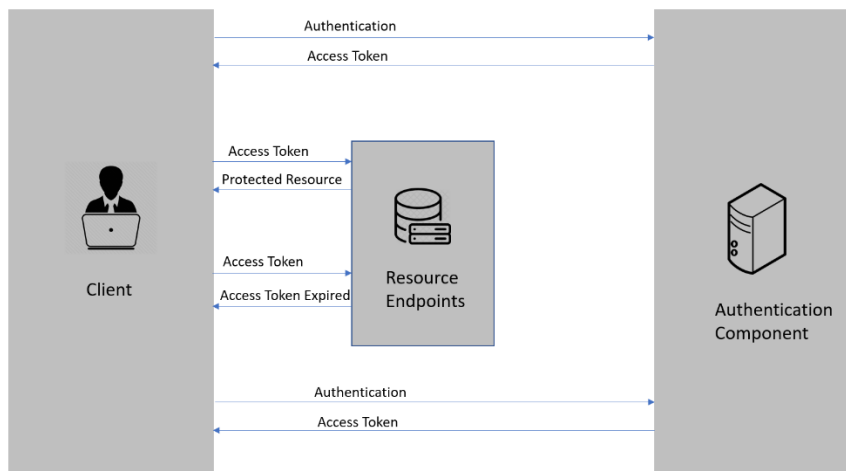
Nekada sistemi koriste više od jedne vrste autentifikacije i to su primeri dvofaktornih sistema autentifikacije (tzv. *2FA sistemi*). Na primer, nakon uspešnog prijavljivanja pomoću korisničkog imena i lozinke (saznanje), aplikacija zahteva od korisnika da potvrdi prijavljivanje na mobilnom uređaju otiskom prsta (nasleđe).

- **Autorizacija** (eng. *authorization*) predstavlja proces odlučivanja da li je identifikovanom akteru dopušteno da izvrši odgovarajuću akciju u sistemu. Postoji nekoliko načina za izvršavanje autorizacije:
 - Jedan od najpopularnijih jeste *obezbeđivanje bezbednosti na osnovu uloga* (eng. *role-based security*). U ovom modelu, prava za pristup se ne daju konkretnim korisnicima, već apstrakcijama koje se nazivaju *uloge* (eng. *role*). Pojedinačnim korisnicima se onda pridružuju uloge, preko kojih oni dobijaju pristup.
 - Drugi način jeste *obezbeđivanje bezbednosti na osnovu tvrdnji* (eng. *claim-based security*). Osnovni aspekt ovog pristupa jeste skup *tvrdnji* (eng. *claim*). Jedna tvrdnja predstavlja jednu pretpostavku koju entitet ima o sebi. Na primer, neki korisnik može imati pridružene naredne tvrdnje:
 - Ime ovog korisnika je Pera.
 - Perina adresa elektronske pošte je pera.peric@rs2.matf.bg.ac.rs.
 - Pera ima 25 godina.
 - Pera može da briše korisnike.

Kao što vidimo, u ovom modelu korisnik sistemu daje na raspolaganju skup tvrdnji, a ne uloge. Ovaj skup tvrdnji se prosleđuje kao deo *sigurnosnog tokena* (eng. *security token*).

Naravno, sistem mora da zna da odredi da li može da veruje nekom skupu tvrdnji. Ovo poverenje se obično implementira posredstvom posebne aplikacije u koju naš sistem ima poverenje. Takva aplikacija često i sama izdaje skupove tvrdnji i vrši proveru validnosti nekog skupa tvrdnji. Ako ona utvrdi verodostojnost skupa tvrdnji, onda i naš sistem ima poverenje u taj skup tvrdnji. Ovakva aplikacija se naziva *izdavač ovlašćenja* (eng. *issuing authority*, skr. *IA*). Deo IA koji prihvata zahteve za tokenima se naziva *servis bezbednostih tokena* (eng. *security token service*, skr. *STS*). Ako se izdavanje tokena prepusti nekoj drugoj aplikaciji umesto IA, onda se ta aplikacija naziva *provajder identiteta* (eng. *identity provider*, skr. *IdP*). U tom slučaju, IdP validira kredencijale aktera i iskomunicira validnost kredencijala nazad to STS. Ako su kredencijali validni, STS izdaje token sa tvrdnjama. Akter onda može da prosledi dobijeni token našem sistemu koji validira token, izdvoji tvrdnje iz tokena, prepozna identitet na osnovu tvrdnji i nivo pristupa sistemu na osnovu tih tvrdnji.

Tok koji ćemo koristiti je prikazan na slici ispod.



1. Podešavanje IdentityServer mikroservisa i osnovna implementacija

Kreirajmo novi ASP.NET Core Web Api projekat (.NET 5.0) naziva **IdentityServer** na putanji **LOKACIJA_LOKALNOG_REPOZITORIJUMA\Webstore\Security** i postavimo ga da se pokreće na adresi **http://localhost:4000**. Instalirajmo naredne NuGet pakete:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.SqlServer
- AutoMapper.Extensions.Microsoft.DependencyInjection
- Microsoft.AspNetCore.Authentication.JwtBearer

S obzirom da ćemo i u ovom mikroservisu koristiti SQL Server, ažuriramo **docker-compose.yml** i **docker-compose.override.yml** datoteke tako da se mikroservis **orderdb** preimenuje u **mssql**. Naredne slike prikazuju izmene.

Datoteka **docker-compose.yml**.

```

- orderdb:
+ mssql:
  image: mcr.microsoft.com/mssql/server:2017-latest
  
```

Datoteka **docker-compose.override.yml**.

```

@@ -38,8 +38,8 @@ services:
  volumes:
    - pgadmin_data:/root/.pgadmin

- orderdb:
- container_name: orderdb
+ mssql:
+ container_name: mssql
  environment:
    - SA_PASSWORD=MATF12345678rs2
    - ACCEPT_EULA=Y

@@ -105,10 +105,10 @@ services:
  container_name: ordering.api
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
-   - "ConnectionStrings:OrderingConnectionString=Server=orderdb;Database=OrderDb;User Id=sa;Password=MATF12345678rs2"
+   - "ConnectionStrings:OrderingConnectionString=Server=mssql;Database=OrderDb;User Id=sa;Password=MATF12345678rs2"
+   - "EventBusSettings:HostAddress=amqp://guest:guest@rabbitmq:5672"
  depends_on:
-   - orderdb
+   - mssql
  - rabbitmq
  ports:
    - "8004:80"
  
```

Pređimo na implementaciju mikroservisa:

- Entities
 - User.cs
- Data
 - ApplicationContext.cs
- Extensions
 - IdentityExtensions.cs

Zatim ažuriramo **Startup.cs** kako bismo dodali pozive kreiranih ekstenzija u **ConfigureServices** metodu:

```
// Services relevant to Identity
services.AddAuthentication();
```

```
services.ConfigurePersistence(Configuration);
services.ConfigureIdentity();
```

Dodatno, u metodu **Configure** dodajemo naredne pozive⁷:

```
app.useAuthentication();
app.useAuthorization();
```

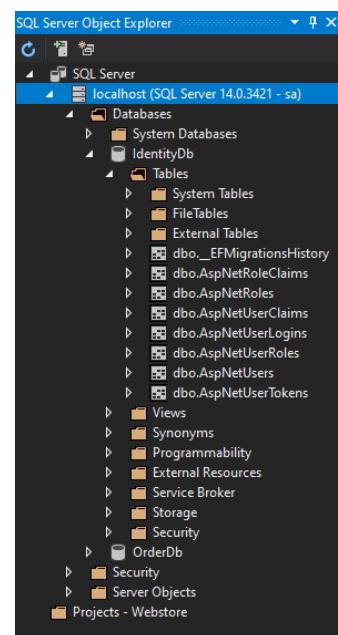
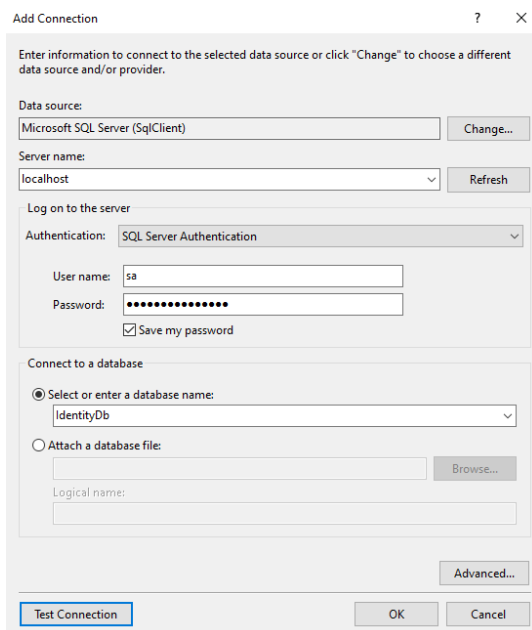
Sada izvršimo naredne naredbe kako bismo dodali migraciju za dodavanje Identity tabela:

```
Add-Migration CreatingIdentityTables
Update-Database
```

Ukoliko želimo da vidimo kreirane tabele, možemo se povezati na SQL Server iz Visual Studio alata:

- Tools > Connect to Database...
- Uneti podatke kao na slici ispod (levo).
- Klikom na dugme Test Connection možemo isprobati da li je konekcija uspešna.
- OK.

U SQL Server Object Explorer-u bi trebalo da se prikažu tabele kao na slici ispod (desno).



⁷ Veoma je važno da ovi pozivi budu u ovom redosledu i da se nalaze između poziva metoda **UseRouting** i **UseEndpoints**.

Konačno, podesimo i dve uloge koje naši korisnici mogu da imaju – administrator i kupac:

- Data
 - Configuration
 - RoleConfiguration.cs

Dodajemo preopterećenje metoda **OnModelCreating** u klasi **ApplicationContext** koje će uključiti implementiranu konfiguraciju za uloge. Kreirajmo novu migraciju:

Add-Migration AddedRolesToDb

Update-Database

Možemo primetiti da tabela `dbo.AspNetRoles` sada ima dve nove uloge:

	Id	Name	NormalizedNa...	ConcurrencySt...
▶	36-de9bba3e33b7	Administrator	ADMINISTRATOR	33abd2c1-16ab...
	e824f9b7-b233-...	Buyer	BUYER	86a9bdd2-c343...
⊕	NULL	NULL	NULL	NULL

2. Implementacija registracije korisnika

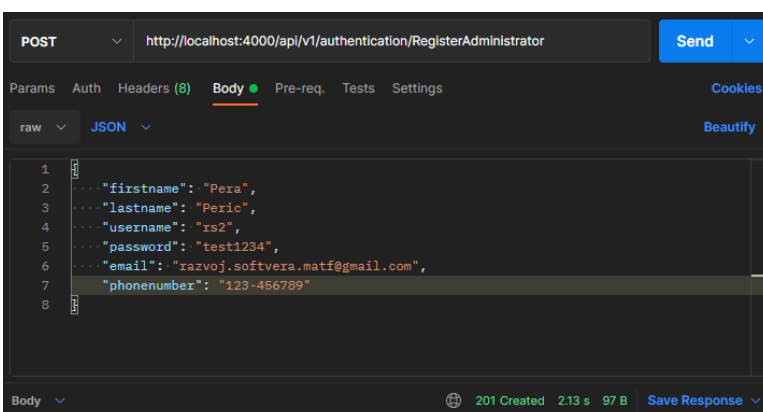
Implementiramo novi kontroler koji ćemo koristiti za potrebe autentifikacije. Dodajemo dva metoda - jedan za registraciju kupaca, a drugi za registraciju administratora.

- DTOs
 - NewUserDto.cs
- Mapper
 - IdentityProfile.cs
- Controllers
 - AuthenticationController.cs
 - AuthenticationControllerBase.cs

Dodajemo metod **ConfigureMiscellaneousServices** u **IdentityExtensions** klasi i pozivamo ga u **Startup** klasi.

Testirajmo implementirane metode.

```
{
  "firstname": "Pera",
  "lastname": "Peric",
  "username": "rs2",
  "password": "test1234",
  "email": "razvoj.softvera.matf@gmail.com",
  "phonenumber": "123-456789"
}
```



```
info: IdentityServer.Controllers.AuthenticationController[0]
      Successfully registered a user: rs2.
info: IdentityServer.Controllers.AuthenticationController[0]
      Added a role Administrator to user rs2.
```

Dodajmo i jednog korisnika koji nije administrator, već kupac:

```
{
  "firstname": "Ana",
  "lastname": "Peric",
  "username": "anaperic",
  "password": "test1234",
  "email": "ana.peric@matf.rs",
  "phonenumber": "123-456789"
}
```

3. Izdavanje JWT tokena

Dodajmo naredna podešavanja u `appsetting.development.json` datoteci:

```
"JwtSettings": {
  "validIssuer": "Webstore Identity",
  "validAudience": "Webstore",
  "secretKey": "MyVerySecretMessage"
}
```

Zatim dodajemo novi metod `ConfigureJWT` u `IdentityExtensions` klasi i pozivamo ga u `ConfigureServices` metodu.

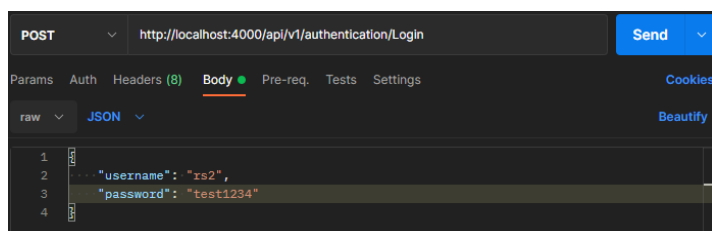
Redosled implementacije:

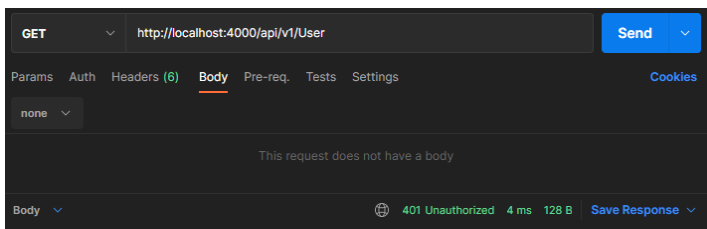
- DTOs
 - `UserCredentialsDto.cs`
- Services
 - `IAuthenticationService.cs`
 - `AuthenticationService.cs`

Dodajemo DI za napisani servis:

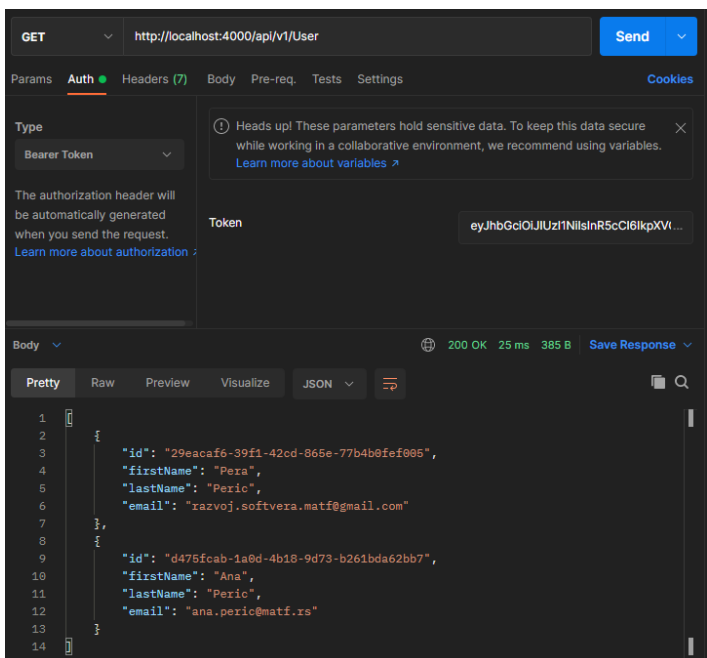
```
public static IServiceCollection ConfigureMiscellaneousServices(this IServiceCollection services)
{
    ...
    // Other
    services.AddScoped<IAuthenticationService, AuthenticationService>();
    ...
}
```

Dodajemo metod `Login` za prijavljivanje u `AuthenticationController` klasi. Sada možemo da testiramo implementaciju:

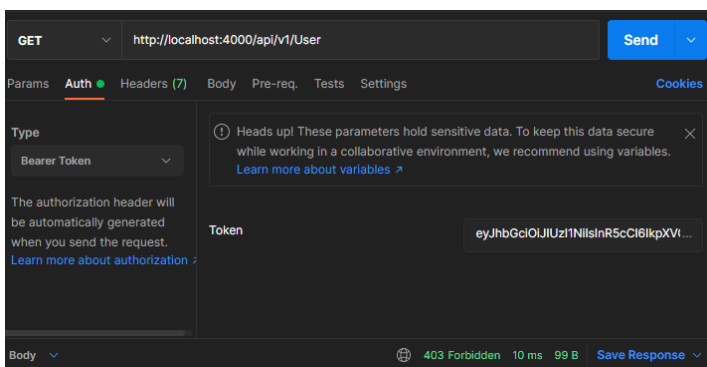




Dobijamo odgovor 401 zato što se nismo autentifikovali. Da bismo to uradili u alatu Postman, potrebno je da dodamo JWT token kao *Bearer* shemu u *Auth* pogledu:



S obzirom da svi metodi nad ovim kontrolerom zahtevaju da autentifikovani korisnik ima ulogu administratora, ako probamo da izvršimo isti zahtev, ali koristeći JWT nekog kupca, onda ćemo dobiti statusni kod 403 kao odgovor:



Isto ovo dodajemo za **RegisterAdministrator** akciju, kako bi samo administratori mogli da kreiraju druge administratore.

5. Kontejnerizacija IdentityServer mikroservisa

Dodajmo podršku za orkestrizaciju implementiranom projektu. Izmeniti **docker-compose.override.yml** datoteku:

identityserver:

container_name: identityserver

environment:

- ASPNETCORE_ENVIRONMENT=Development


```
- "ConnectionStrings:IdentityConnectionString=Server=mssql;Database=IdentityDb;User
Id=sa;Password=MATF12345678rs2"
depends_on:
- mssql
ports:
- "4000:80"
```

6. Zaštita drugih mikroservisa

U svakom mikroservisu u kojem želimo da primenimo JWT *Bearer* strategiju kako bismo zaštitili akcije, potrebno je da uradimo naredne korake:

- Instalirati NuGet paket **Microsoft.AspNetCore.Authentication.JwtBearer**
- Specifikovati koje rute zahtevaju autorizaciju (i, eventualno, koju ulogu):

```
namespace Basket.API.Controllers
{
    [Authorize(Roles = "Buyer")]
    [ApiController]
    [Route("api/v1/[controller]")]
    public class BasketController : ControllerBase
    ...
}
```

- Ažurirati datoteku **appsettings.development.json** tako da sadrži neophodne JWT informacije:

```
"JwtSettings": {
  "validIssuer": "Webstore Identity",
  "validAudience": "Webstore",
  "secretKey": "MyVerySecretMessage"
}
```

- Ažurirati **Startup** klasu tako što se u **ConfigureServices** metodu doda naredni kod:

```
// JWTSecurity
var jwtSettings = Configuration.GetSection("JwtSettings");
var secretKey = jwtSettings.GetSection("secretKey").Value;

services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
```

```

        ValidIssuer = jwtSettings.GetSection("validIssuer").Value,
        ValidAudience = jwtSettings.GetSection("validAudience").Value,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
    };
});

```

a u Configure metodu doda naredni kod:

```

app.UseAuthentication();
app.UseAuthorization();

```

Dodatna zaštita korišćenjem tvrdnji

Obezbeđivanje zaštite korišćenjem uloga nam pomaže da sprečimo čitavu klasu korisnika da pristupi akcijama koje im nisu dostupne u sistemu. Međutim, ovo nam često nije dovoljno. Na primer, u **BasketController** klasi smo rekli da samo korisnici koji imaju ulogu „kupac“ mogu da izvršavaju akcije definisane u tom kontroleru, kao što je dohvaćanje sadržaja korpe. Međutim, ne postoji provera da li je kupac koji dohvata korpu isti onaj kupac čija se korpa dohvata. Drugim rečima, jedan kupac može dohvatiti korpu za nekog drugog kupca, sve dok zna njegovo korisničko ime.

Rešenje se nalazi u implementiranju zaštite korišćenjem tvrdnji. Sve klase koje naslede **ControllerBase** imaju na raspolaganju objekat **User** koji sadrži informacije o autentifikovanim korisnicima. Pomoću ovog objekta možemo dohvatiti informacije o tvrdnjama, na primer:

```

public async Task<ActionResult<ShoppingCart>> GetBasket(string username)
{
    if (User.FindFirstValue(ClaimTypes.Name) != username)
    {
        return Forbid();
    }

    var basket = await _repository.GetBasket(username);
    return Ok(basket ?? new ShoppingCart(username));
}

```

Ovako se možemo obezbediti da kupci mogu da dohvataju korpu samo za sebe. Zapravo, uvođenjem autentifikacije i koncepta tvrdnji, moguće je promeniti API tako da više ne zahteva korišćenje korisničkog imena:

```

public async Task<ActionResult<ShoppingCart>> GetBasket()
{
    var username = User.FindFirstValue(ClaimTypes.Name);
    if (username == null) // Vrednost za username će biti null ako ne postoji tvrdnja ClaimTypes.Name.
    {
        return Forbid();
    }

    var basket = await _repository.GetBasket(username);
    return Ok(basket ?? new ShoppingCart(username));
}

```

7. Refresh tokeni

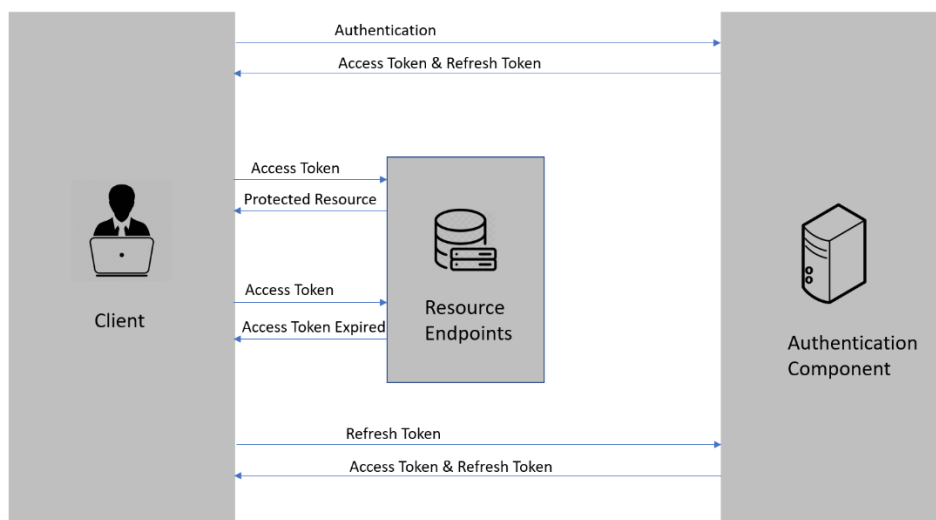
Tokeni koje smo do sada izdvojali se nazivaju još i *access tokeni* zato što se koriste za pristup akcijama u sistemu. Svako ko ima validan access token može izvršavati dozvoljene akcije. Kada token postane nevalidan, korisnici moraju da se ponovo autentifikuju kako bi dobili novi token. U ovom toku imamo neke probleme:

- Ako je vreme za koje token postaje nevalidan premalo (na primer, 15 minuta), onda korisnici moraju često da se autentifikuju. Korisnicima je prečesta autentifikacija veoma naporna.
- Ako je vreme za koje token postaje nevalidan preveliko (na primer, mesec dana), onda korisnici ne moraju toliko često da se autentifikuju. Međutim, sada rizikujemo da neko treće lice dobije pristup našem sistemu ukoliko (na neki način) sazna za taj token.

Drugi problem se može rešiti u postojećem sistemu tako što se za svakog korisnika trajno čuvaju svi izdati access tokeni (na primer, u bazi podataka). Zatim, prilikom korišćenja nekog access tokena, sistem treba da proveri da li je access token i dalje povezan sa korisnikom (tj. da li se nalazi u bazi podataka). U slučaju da se otkrije da je došlo do zloupotrebe access tokena, jednostavno se mogu obrisati svi access tokeni za datog korisnika, čime se poništavaju svi access tokeni. Korisnik onda mora ponovo da se autentifikuje kako bi pristupio sistemu, ali dalja zloupotreba poništenih access tokena nije moguća. Očigledan, a znajačan, problem sa ovim pristupom jeste taj da svaki poziv ka API-ju našeg sistema zahteva pretragu access tokena u bazi podataka. Ovime se značajno usporava rad sistema.

Rešenje se sastoji u uvođenju nove vrste tokena i promeni autorizacionog toka. *Refresh tokeni* predstavljaju kredencijale koji se koriste za izdavanje novih access tokena. Kada access token istekne, umesto da zahtevamo od korisnika da se autentifikuje pomoću svojih kredencijala, sistem može iskoristiti (validan) refresh token kako bi dobio novi access token od komponente za autentifikaciju. Naravno, ako je refresh token nevalidan, onda sistem zahteva ponovnu autentifikaciju od korisnika. Međutim, na ovaj način možemo iskoristiti kratkotrajne access tokene, a po pravilu su refresh tokeni dužeg trajanja⁸.

Tok kojise koristi u slučaju korišćenja access i refresh tokena je prikazan na slici ispod.



Za početak, ažurirajmo **appsettings.development.json** datoteku da obuhvati podešavanje za isticanje refresh tokena:

"RefreshTokenExpires": 30

Dopunimo **User** model tako da sadrži informacije o svim izdatim refresh tokenima i njihovim vremenskim odrednicama za isticanje:

- Entities
 - RefreshToken.cs
 - User.cs
- Data
 - Configurations

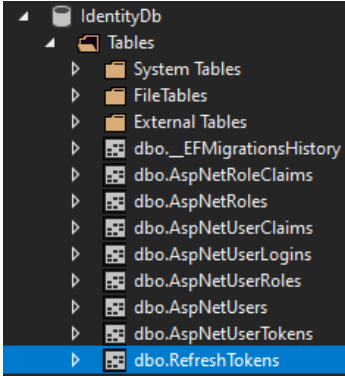
⁸ Ne postoje tvrda pravila koliko dugo neki token treba da živi, ali access tokeni ne bi trebalo da važe duže od 15 minuta. U slučaju da treće lice sazna za ovakav token, on će imati pristup sistemu najduže 15 minuta, nakon čega token postaje nevalidan. Ovo je uglavnom prihvatljivo. Dužina trajanja refresh tokena se uglavnom određuje time koliko dugo želimo da dozvolimo korisnicima da koriste sistem bez da moraju da se autentifikuju ponovo.

- UserConfiguration.cs
 - ApplicationContext.cs

Sada nam je potrebna nova migracija:

Add-Migration AddedRefreshTokensToUser Update-Database

Možemo primetiti novu tabelu u *SQL Server Object Explorer* pogledu:



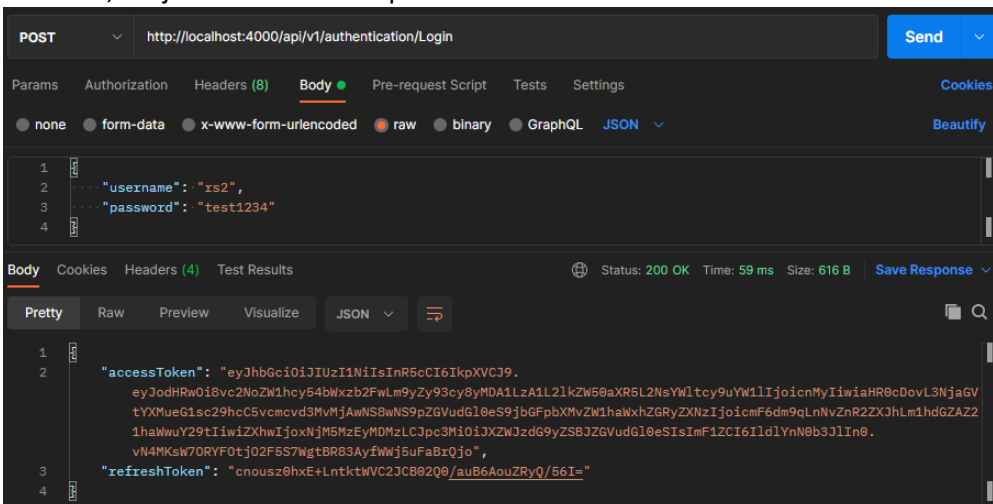
Sada prelazimo na implementaciju metoda za izdavanje nove vrste tokena. Usput ćemo refaktorirati servis za autentifikaciju:

- DTOs
 - AuthenticationModel.cs
 - RefreshTokenModel.cs
- Services
 - IAuthenticationService.cs
 - AuthenticationService.cs
- Controllers
 - AuthenticationController.cs

Ako sada dva puta pozovemo Login API za istog korisnika, dobićemo naredno stanje u **dbo.RefreshTokens** tabeli:

	Id	Token	ExpiryTime	UserId
	1c371082-1638-...	+YCy4Et3CkSG...	11-Jan-22 12:12:04	149ea247-cc31-...
	df813d6b-0d71...	cnouz0hxE+Ln...	11-Jan-22 12:12:13	149ea247-cc31-...
	NULL	NULL	NULL	NULL

Naravno, ovaj API treba da vrati podatke o oba tokena:



9. Jednostranične klijentske aplikacije

Tema ovih časova je razvoj jednostraničnih aplikacija pomoću Angular 12.

Literatura:

- Angular dokumentacija: <https://angular.io/docs>
- Odlomci iz skripte „Programiranje za veb“: <https://www.nikolaajzenhamer.rs/assets/pdf/pzv.pdf#page=241>

Angular video lekcije:

- Reaktivna paradigma u JS i biblioteka RxJS: <https://youtu.be/3QC45tpjwRk>
- Komponente: <https://youtu.be/C-G7wzMEbE4>
- Ugrađene direktive, razmena podataka između komponenti i životni ciklus komponenti: <https://youtu.be/p1H4Bptc4UM>
- Servisi, rutiranje na klijentu, formulari i filteri: <https://youtu.be/N2e-7FgbCVY>
- HTTP komunikacija: <https://youtu.be/IQs2HKtVWa0>