.NET Core

Predrag Glavaš- pedja.glavas@uns.ac.rs
Tanja Čubrić - radojcic.tanja@uns.ac.rs

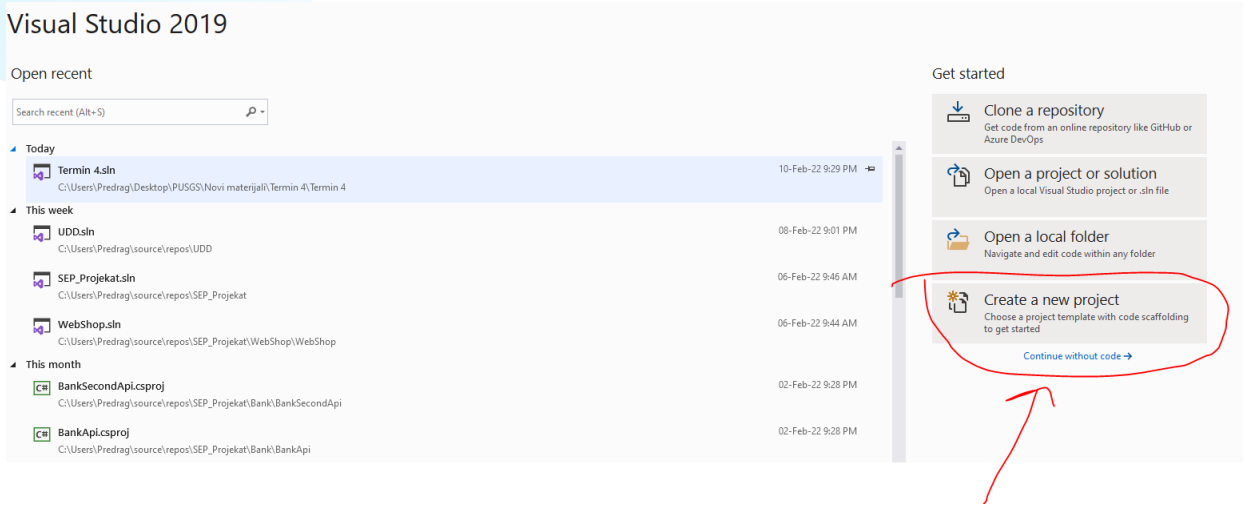
.NET Core web API

- .NET Core je radni okvir (frejmvork) za izradu web aplikacija , dok je web API tip web servisa koji funkcioniše kao RESTFUL API.
- Verzije : 2.2, 3.1, 5.0, 6.0
- Mi ćemo koristiti 5.0

Kreiranje projekta

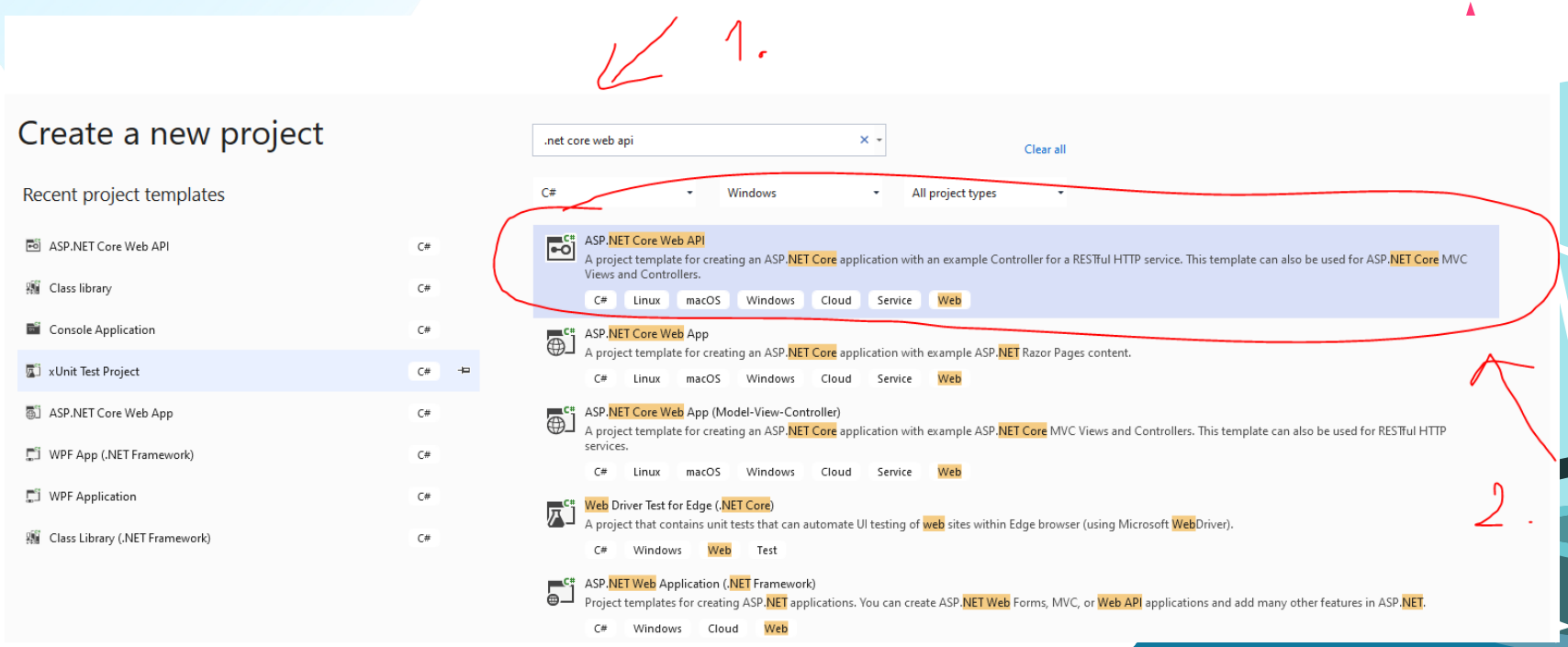
Kreiranje projekta

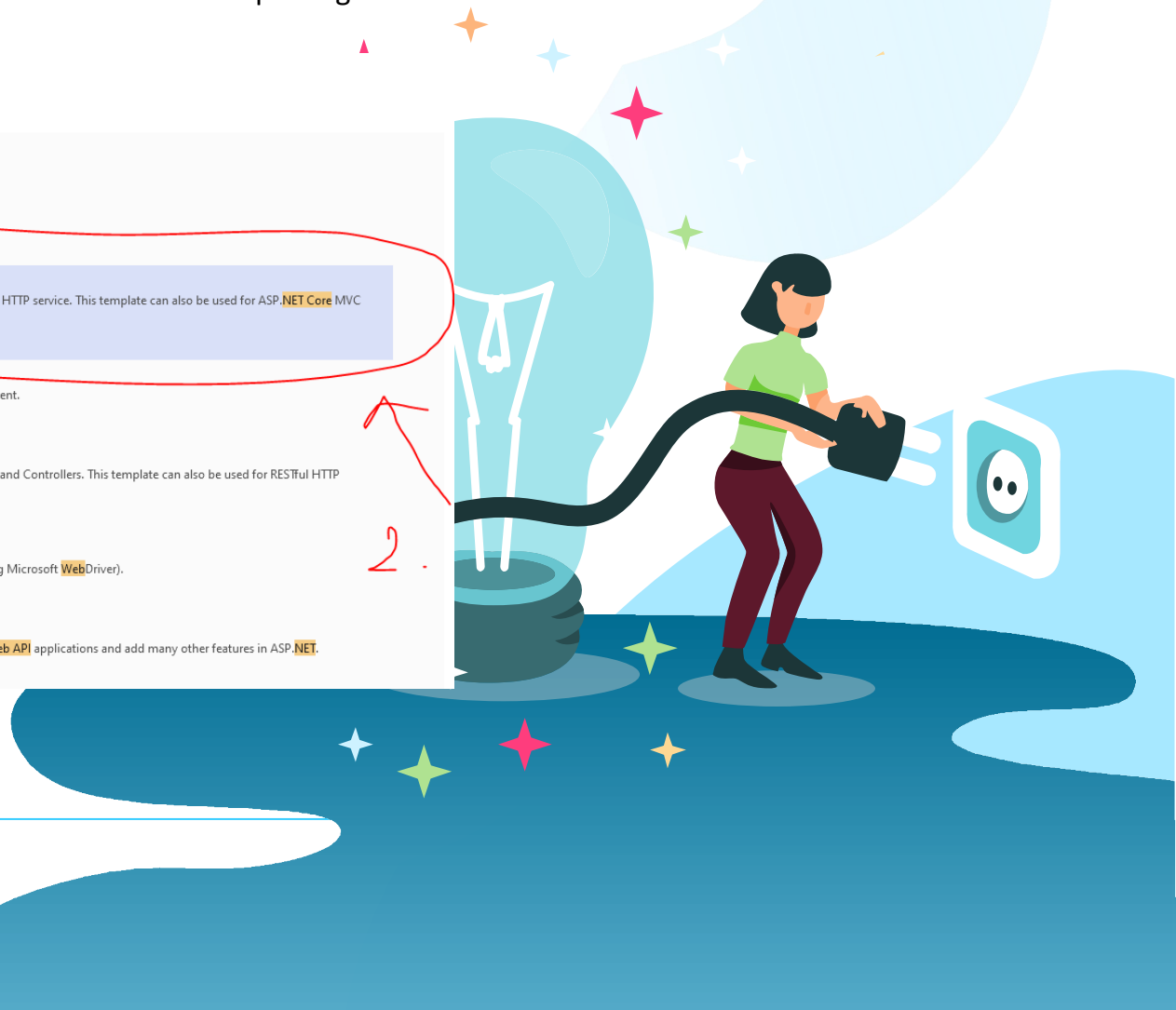
- Otvorimo Visual Studio, a zatim odaberemo: Create a new project



Kreiranje projekta

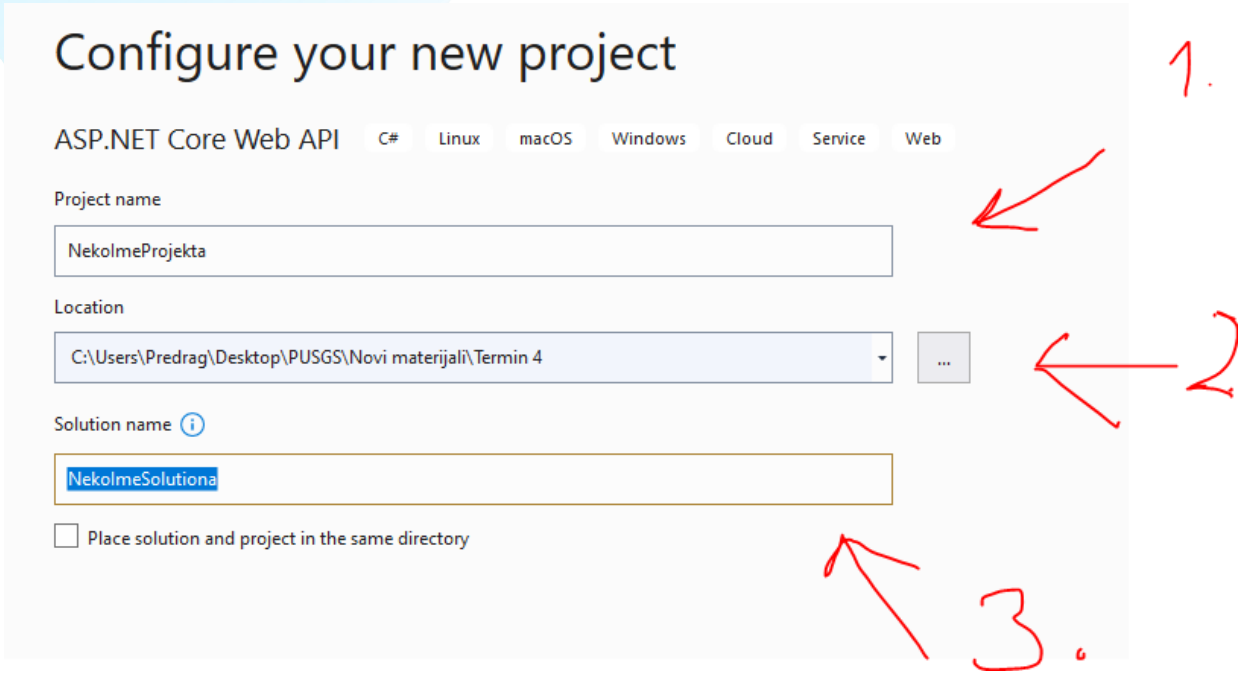
- Pojaviće se prozor za izbor tipa projekta, mi biramo ASP .NET Core web API, možemo to ukucati i u pretragu





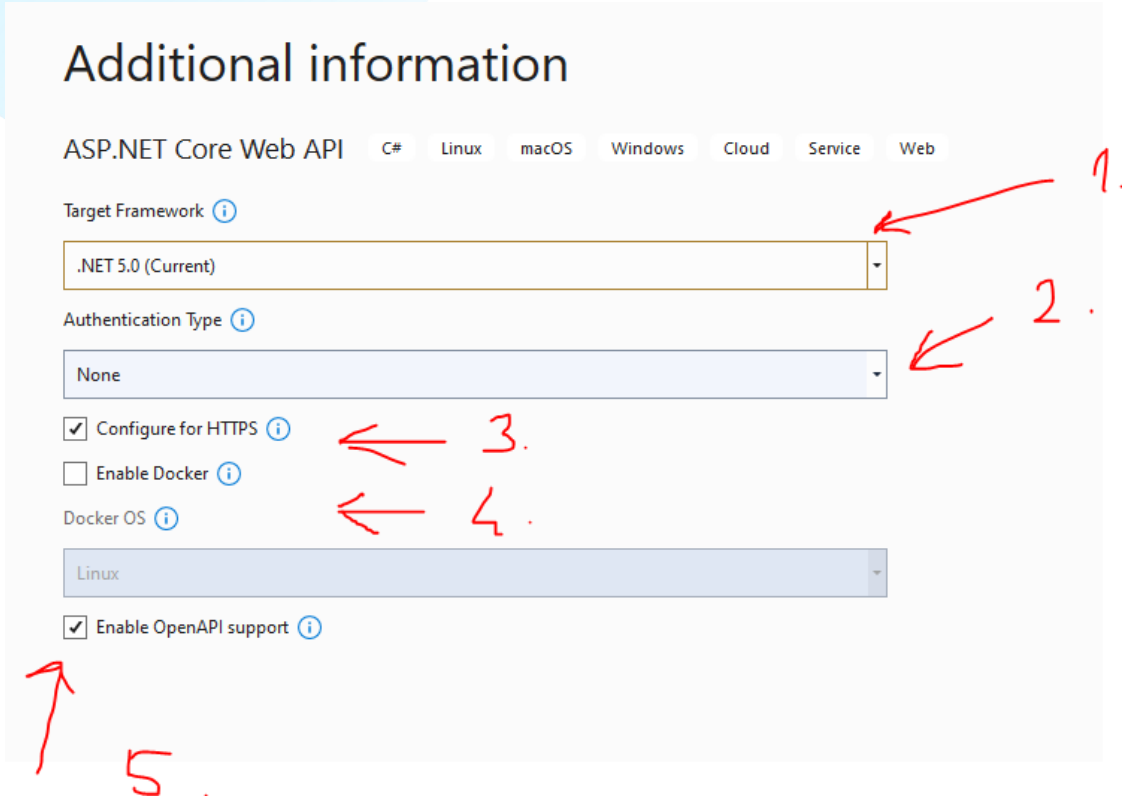
Kreiranje projekta

- Na sledećem prozoru biramo putanju na kojoj će se projekat kreirati, ime projekta i ime solutiona



Kreiranje projekta

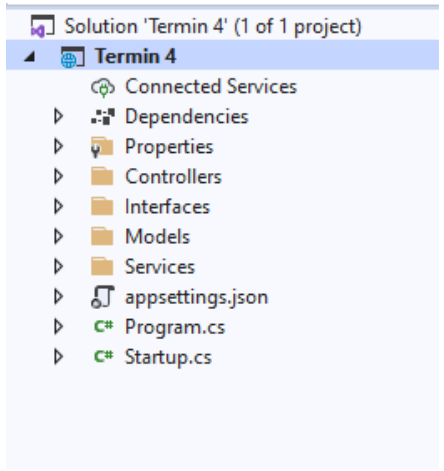
- Na kraju biramo verziju frejmvorka, tip autentifikacije, bezbednosnu šemu (HTTPS), da li želimo doker podršku i da li želimo Swagger.



Struktura web API projekta

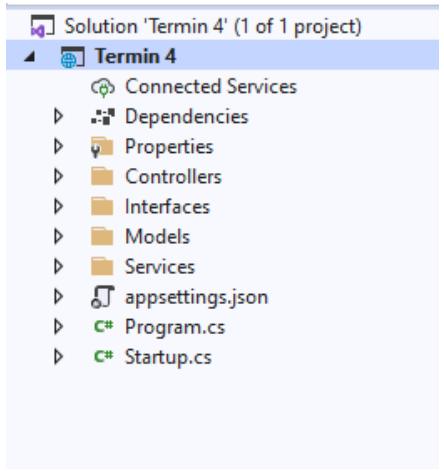
Struktura projekta

- .NET Core web API projekat ima strukturu kao sa slike ispod.
- U klasi Program.cs započinje izvršavanje koda nakon što se projekat pokrene, odatle se poziva klasa Startup.cs koja konfiguriše sve što je neophodno za rad projekta nakon pokretanja.
- Datoteka appsettings.json čuva neka podešavanja u JSON formatu (npr adrese nekih eksternih servisa koje gađamo i slično). Podacima odavde se može pristupiti preko IConfiguration interfejsa



Struktura projekta

- Pri dodavanju novih klasa poželjno ih je grupisati logički po folderima, modele u folder Models, servise u Services, itd..
- Nekada se softverski slojevi dele ne po folderima, već kreiranjem zasebnih Class library projekata za svaki sloj aplikacije. Mi nećemo koristiti taj pristup.



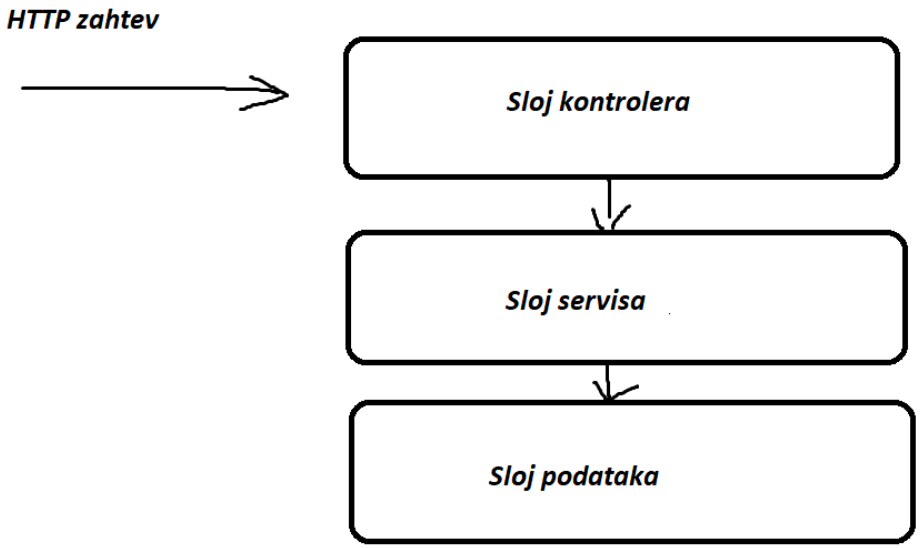
Startup.cs klasa

- Startup.cs klasa sa svoje 2 metode Configure i ConfigureServices predstavlja mesto gde se podešava većina bitna stvari za aplikaciju.
- U metodi ConfigureServices se dodaju zavisnosti u kontejner za injekciju, nije bitno kojim redom se koja zavisnost dodaje.
- U metodi metodi Configure se podešava redosled izvršavanja funkcija Midlveru prilikom prijema HTTP zahteva.
- Jako je bitno da se u Configure metode pozivaju tačno definisanim redosledom (pogledati Microsoft dokumentaciju) inače će aplikacija odbiti da se startuje ili će odbijati validne HTTP zahteve.

Troslojna arhitektura web aplikacija

Arhitektura

- Kako kod koji pišemo treba da ima neku logičku strukturu, delimo ga u slojeve gde će svaki sloj biti zadužen za određeni deo posla.
- Troslojna arhitektura podrazumeva postojanje 3 sloja u web aplikacijama: Kontrolerski, Servisni i sloj podataka
- Slojevi mogu međusobno komunicirati ali samo preko apstrakcija (interfejsa) i to tako da se međusobno ne preskaču i da niži sloj ne vidi viši (videti sliku).



Sloj kontrolera

- Na vrhu hijerarhije slojeva nalazi se sloj kontrolera, on je ujedno i ulazna tačka za spoljne zahteve ka našoj aplikaciji.
- Svaka metoda kontrolera se mapira na jedinstvenu URL putanju, uz to da mora biti definisan i HTTP glagol nad metodom (GET, POST, PUT..)
- Ulazne tačke koje su logički povezane treba staviti u isti kontroler. Primer: sve operacije nad korisnicima staviti u UserController
- Kontroleri „poznaju“ HTTP zahteve i kako na njih odgovoriti, ali ne bi smeli da direktno kontaktiraju bazu podataka niti da obrađuju podatke. Te zadatke delegiraju sloju Servisa s kojim direktno komuniciraju.
- Ako se radi o kontrolerima RESTFUL web servisa, oni bi morali vraćati odgovarajuće HTTP odgovore, OK-200, NotFound-404, itd..
- Pri definisanju URL-ova voditi se pravilima nazivanja RESTFUL tačaka pristupa
- https://restfulapi.net/resource-naming/

Sloj servisa

- Sloj servisa u web aplikacijama je zadužen za biznis logiku aplikacije. Biznis logika može biti: sortiranje nizova, slanje mejlova ili bilo šta vezano za namenu aplikacije.
- Sloj servisa NE SME poznavati HTTP zahteve niti odgovore i NE SME direktno komunicirati sa bazom podataka. Za komunikaciju sa bazom se obraća sloju podataka.
- Sloj servisa ne bi smeo da direktno poziva metode kontrolera!
- Ukoliko je neophodno može vršiti validacije podataka. U slučaju nevalidnosti podataka ili greške pri izvršavanju trebao bi da baci izuzetak i tako javi sloju iznad (kontrolerskom sloju) šta tačno nije u redu.
- Dozvoljeno je da u slučaju greške vrati podatke o grešci kao povratnu vrednost kontroleru koji ga je pozvao.

Sloj podataka

- Sloj podataka je najniži u troslojnoj hijerarhiji i zadužen je za direktan pristup podacima smeštenim obično u bazama podataka.
- Jedini sme direktno da komunicira sa bazom, ali NE SME pozivati metodu nijednog sloja iznad njega.
- U .NET Core-u ovaj sloj je implementiran kao DbContext klasa (više o tome kasnije), ali se inače pravi kao Repository ili Repository+Unit of work šablon.
- Mišljenja o tome da li je potrebno „umotati“ DbContext u Repository su vrlo podeljena, ali neka smernica je da ako postoji mogućnost da će se menjati tehnologija baze podataka (SQL, NoSQL) onda treba implementirati i Repository.
- Sloj podataka bi trebao da enkapsulira upite ka bazi podataka i da vraća modele kao odgovor.

Injekcija zavisnosti (dependency injection)

Injekcija zavisnosti

- Injekcija zavisnosti prestavlja mehanizam za dodelu zavisnosti klasama koje ih koriste. Recimo ako UserController komunicira sa UserService servisom ne može to činiti direktno već samo preko interfejsa.
- „Sakrivanjem“ implementacija iza interfejsa postižemo slabiju povezanost između implementacije i apstrakcije i lakše testiranje metoda.
- .NET Core ima veoma moćan i jednostavan mehanizam za injekciju zavisnosti, u okviru Startup.cs klase u metodi ConfigureServices moguće je dodati interfejse i zavisnosti u kontejner, odakle ih .NET Core injektuje automatski gde god su potrebne u aplikaciji.
- Zavisnosti se mogu dodati sa tri različita životna ciklusa: Scoped, Transient i Singleton
- Kako bi injektor znao gde sve treba da injektuje zavisnost kroz konstruktor, potrebno je apstrakciju zavisnosti (interfejs) dodati u konstruktor klase kojoj je zavisnost potrebna.

Scoped životni ciklus

- Scoped životni ciklus je najčešće korišćen.
- Zavisnosti definisane sa ovim životnim ciklusom će biti injektovane sa istom referencom u okviru jednog HTTP zahteva.
- Za primer IScopedRaceService to znači da će injektor zavisnosti kreirati jedan objekat RaceService servisa i dodeliti ga svakoj klasi kojoj je potreban u okviru jednog HTTP zahteva
- Već pri narednom zahtevu, napraviće se nova instanca servisa i svako stanje iz prethodnog zahteva će biti izgubljeno.
- Testirajte ovo ponašanje pozivanjem POST /api/races/scoped metode servisa iz primera, a zatim GET /api/races/scoped. Trka dodata prvim zahtevom više ne postoji slanjem narednog zahteva.

```
services.AddScoped<IScopedGetterService, GetterService>(); /
services.AddSingleton<ISingletonRaceService, RaceService>();
services.AddTransient<ITransientRaceService, RaceService>();
```

Transient životni ciklus

- Zavisnosti definisane sa ovim životnim ciklusom će biti injektovane sa različitom referencom uvek, čak i u okviru istog HTTP zahteva.
- Obično se koristi za „lagane“ servise koji nikad ne čuvaju stanje
- Za primer ITransientRaceService to znači da će injektor zavisnosti različit objekat RaceService servisa dodeliti svakoj klasi kojoj je potreban u okviru jednog HTTP zahteva
- U primeru koda RaceService je potreban Race kontroleru i Getter servisu. Kreiraće se 2 različite instance i dodeliti kroz konstruktor.
- Testirajte ovo ponašanje pozivanjem /api/races/transient metode servisa iz primera. Trka dodata ovim zahtevom neće biti vraćena u odgovoru jer Getter servis koji vraća odgovor koristi drugu (različitu) instancu Race servisa.

```
services.AddTransient<ITransientGetterService, GetterService>();
```

Singleton životni ciklus

- Zavisnosti definisane sa ovim životnim ciklusom će biti injektovane sa istom referencom uvek, čak i u okviru više HTTP zahteva.
- Obično se koristi za servise koji moraju da čuvaju stanje ili koji ni nemaju stanje pa se želi uštedeti na memoriji korišćenjem jedne instance
- Za primer ISingletonRaceService na slici ispod to znači da će injektor zavisnosti isti objekat RaceService servisa dodeliti svakoj klasi kojoj je potreban ikada.
- Testirajte ovo ponašanje pozivanjem POST /api/races/singleton metode servisa iz primera a zatim GET api/races/singleton. Trka dodata prvim zahtevom vidljiva je i u odgovoru na prvi zahtev i među trkama dobavljenim drugim zahtevom.

```
services.AddSingleton<ISingletonRaceService, RaceService>();
```

Termin 4 - zadatak

- Kreirati novi .NET Core web API projekat
- Napraviti model studenta sa poljima : ime, prezime i broj indeksa
- Napraviti model profesora sa poljima ime, prezime, plata
- Napraviti studentski i profesorski servis
- Napraviti studentski i profesorski kontroler
- Studentski servis dodati kao scoped, a profesorski kao singleton
- U oba kontrolera napraviti POST i GET metode za kreiranje i dobavljanje resursa
- Ispoštovati RESTFUL naming konvencije

THANK YOU