Izrada web aplikacija u ASP.NET Core tehnologiji

Braica, Šime

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Šime Braica

Izrada web aplikacija u ASP.NET Core tehnologiji

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Šime Braica

Matični broj: 0016155261

Studij: Informacijski i poslovni sustavi 1.2

Izrada web aplikacija u ASP.NET Core tehnologiji

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Marko Mijač

Varaždin, srpanj 2024.

Šime Braica

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Kroz ovaj završni rad opisana je ASP.NET Core tehnologija, tehnologija koja omogućuje pisanje backenda za web aplikacije u C# programskom jeziku. Primaran fokus bio je na izdvajanju glavnih značajki ove tehnologije te načina na koji se ista koristi za razvoj modernih web aplikacija. Tako su kroz poglavlja opisani pojmovi poput kontrolera, rutera, modela, objekata za rad s podacima i svi ostali pojmovi koji omogućuju lakše i bolje razumijevanje rada ove tehnologije. Kako backend razvoj aplikacija povkači rad sa bazom podataka, sesijama i tokenima opisani su principi rada sa Entity Framework-om, LINQ-om te ostalim bibliotekama potrebnim za efektivan i standardan pristup razvoju autorizacije i autentifikacije korisnika. Da bi aplikacija imala i vizualni dio za krajnje korisnike ukratko je opisan i Angular, razvojni okvir za frontend razvoj web aplikacija koji se lako integrira sa ASP.NET Core-om.

U praktičnom dijelu ovog završnog rada razvijena je full stack web aplikacija koristeći već spomenuti ASP.NET Core i Angular, a u sklopu aplikacije razvijeni su procesi autorizacije i autentifikacije korisnika te rada s podacima. Kako bi aplikacija pratila dobre prakse u razvoju je korištena višeslojna arhitektura, a svrha same aplikacije je prikazati praktičnu implementaciju teorijski obrađenog dijela.

Ključne riječi: ASP.NET Core, Angular, REST, API

Sadržaj

1. Uvod

Prije razvoja web aplikacija razvijale su se web stranice koje su po svojoj prirodi statičke, a čija je osnovna svrha bila prikaz statičkog, ne promjenjivog sadržaja. Kroz vrijeme potrebe korisnika su rasle te su tako web stranice postajale sve dinamičnije odnosno sadržavale su sadržaj koji se sve više mogao prilagodiit potrebama korisnika, a naposlijetku su iz toga proizašle web aplikacije koje kao primarnu funkciju imaju omogućiti korisnicima interaktivno i prilagodljivo iskustvo korištenja. No web aplikacije su po svojoj prirodi kompleksnije od web stranica pa se tako dijele na backend i frontend dio iz čega proizlaze i raznovrsne tehnologije koje se specijaliziraju za specifičan sloj koji želimo razviti.

Web aplikacije se često sastoje od više slojeva. No svaki sloj može se podijeliti na još podslojeva što je i današnja preporučena praksa u razvoju web aplikacija. Tako je kroz ovaj rad prikazana jedna od modernih tehnologija koja ne samo da po svojoj prirodi podržava, već i podupire korištenje višeslojnosti u razvijanju backend dijela aplikacije, ASP.NET Core. Ova tehnologija je po svojim mogućnostima osebujna zbog čega je i izbor mnogih programera jer nudi pregršt mogućnosti za pisanje koda na produkcijskoj razini. Tako je veliki dio ovog završnog rada posvećen isključivo ovoj tehnologiji te sistematizaciji i opisivanju svih njenih mogućnosti i značajki, ali ona ipak sama nije dovoljna za potpun razvoj web aplikacija. Tako je za frontend dio web aplikacije u ovom završnom radu korišten Angular. Angular je popularan programski okvir koji olakšava razvoj frontend dijela web aplikacija te se besprijekorno uklapa uz ASP.NET Core zbog TypeScript jezika koji koristi, a razvijen je od iste osobe koja je razvila i C# jezik koji se koristi u ASP.NET Core tehnologiji. Tako je dio rada posvećen i Angularu, a naposlijetku prikazana je i implementacija full stack web aplikacije koja primjenjuje značajke opisane u ovom radu te joj je glavna svrha omogućavanje upravljanja kolegijima profesorima koji ih drže.

2. Uvod u ASP.NET Core

2.1. Povijest ASP.NET Core

ASP:NET je evoluirao od tradicionalnog ASP (Active Server Pages) te se spojio sa Microsoft-ovim .NET tehnologijama [11]. Prvi put se pojavljuje na tržištu 2002 kao dio .NET razvojnog okvira. Početkom 2016. godine po prvi put se pojavljuje ASP.NET Core 1.0 koji je bio redizajnirana i optimizirana verzija tradicionalnog ASP.NET-a. Neke od bitnijih promjena koje su dodane su bile: Dependency Injection, podrška za druge operacijske sustave te kompatibilnost sa modernim klijentskim razvojnim okvirima (React, Angular, Vue.js). S uvodom .NET 8, Startup.cs više ne postoji. Umjesto toga, konfiguracija aplikacije se pojednostavljuje i integrira direktno u Program.cs datoteku, što dodatno smanjuje kompleksnost i poboljšava čitljivost koda.

2.2. ASP.NET MVC

ASP.NET MVC je razvojni okvir koji je razvijen od strane Microsoft-a za razvijanje dinamičkih web aplikacija . Ovaj razvojni okvir je napravljen po klasičnoj uzorku arhitekture „Model-View-Controller“. Model se sastoji od DTO klasa, servisa te klasa koji služe za pristup bazi. Služi da možemo raditi osnovne CRUD operacije nad bazom. „View“ se sastoji od klijentske strane te što korisnik vidi kada upali aplikaciju. „Controller“ služi kao posrednik između „Model“ i „View“ dijela aplikacije te manipulira i obrađuje podatke koje „Model“ dio šalje te ih priprema za „View“ dio. Svaki „View“ bi trebao imati svoj vlastiti kontroler. „Controller“ je klasa koja sadrži REST metode koje odgovaraju na HTTP zahtjeve.

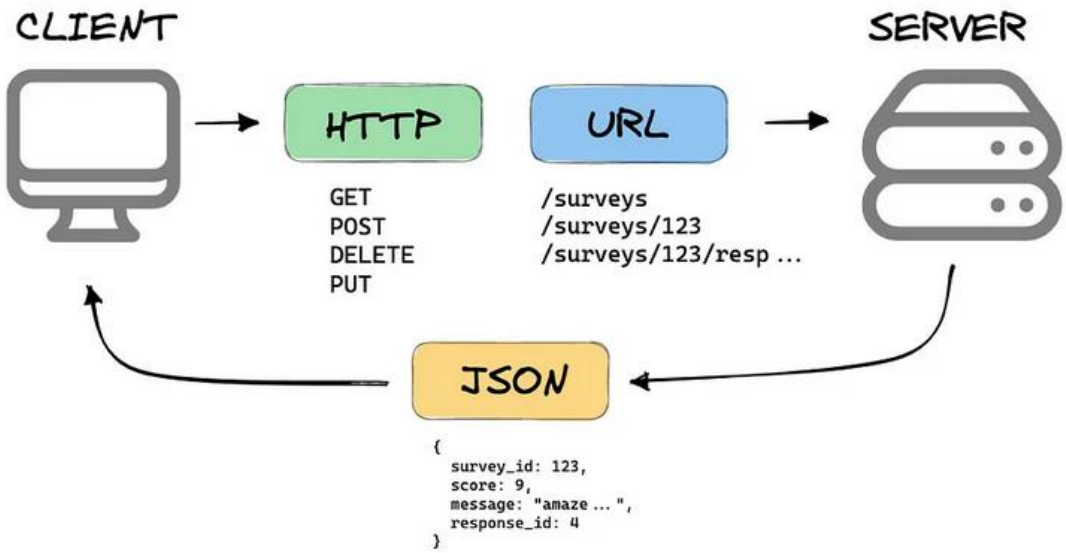
2.3. ASP.NET Core API

ASP.NET Core API je razvojni okvir koji služi za razvijanje brzih, cross-platform i modernih web API-ja. Neke od prednosti ASP.NET Core API-ja su to što ima “cross-platform” podršku što znači da se može pokretati na Windows, Linux ili macOS operacijskim sustavima, ima laku integraciju sa sustavima za kontejnerizaciju (“Docker”), verzioniranje (Github) te mogućnost integracije sa CI/CD sustavima.

2.3.1. RESTful-API

Web API (Application Programming Interface) služi za komunikaciju između 2 programa [10]. U suštini to je most između servera i klijenta. RESTful-API je tip API-ja koji prati REST

(Representational State Transfer) tip arhitekture. REST se ne sprema nigdje na serveru, to znači da se ne spremaju sesije na serveru nego već na klijentu. Naime REST API koristi HTTP metode: GET, POST, PUT, DELETE. GET putanja dohvaća resurs, POST putanja stvara novi resurs, PUT ažurira postojeći resurs te DELETE putanja briše postojeći resurs. Interakcije između klijenta i servera se odvijaju pomoću Request i Response sistema. Naime svaki request (zahtjev) se sastoji od 3 dijela: HTTP metode, zaglavlja te sadržaja, a svaki response (odgovor) se sastoji isto od 3 dijela: statusni kod, zaglavlje te sadržaja. REST API vraća odgovore većinom ili u JSON ili u XML formatu. Prednost korištenja klijent i server arhitekture je to što su kompletno neovisni te se mogu mijenjati neovisno jedan o drugome. Sve ove značajke povećavaju sigurnost te fleksibilnost u stvaranju i održavanju web aplikacija.



Slika 1: Prikaz interakcije poslužitelja i klijenta (izvor: [1])

Bitniji statusni kodovi

Tablica 1: Objašnjenja statusnih kodova (izvor: [7])

Statusni kod	Objašnjenje
200 (OK)	Označava da API uspješno obavio zadatak zadan od strane klijenta. Mora sadržavati tijelo odgovora.
201 (Created)	Označava da je API uspješno kreirao novi resurs.
400 (Bad request)	Koristi se kada bilo koji drugi 4XX tip statusnog koda nije primjenjiv. Često je se

	događa ako šaljemo zahtjev sa krivim parametrima.
401 (Unauthorized)	Označava da korisnik koji šalje zahtjev nije autentificiran.
403 (Forbidden)	Označava da autentificirani korisnik nije autoriziran da pristupi resursu.
404 (Not found)	Označava da klijent pokušava pristupiti resursu koji ne postoji.
500 (Internal server error)	Najčešći odgovor API-ja . Događa se isključivo na poslužiteljskoj strani te se može dogoditi kada niti jedan statusni kod odgovara.

2.3.2. Controllers

Kontroleri u ASP.NET Core-u su klase čija je svrha rad sa HTTP zahtjevima te obrađivanje istih [9]. Kontroleri mogu nasljeđivati od klase `ControllerBase` ili klase `Controller`. U slučaju da kontroler nasljeđuje od klase `ControllerBase` onda taj kontroler smije obrađivati HTTP zahtjeve te generiranje i slanje odgovora to jest rad sa request-response sistemom. Preporuka za korištenje klase `ControllerBase` je kada radimo isključivo sa API-ijma te servisima. U slučaju da želimo raditi ASP.NET Core MVC onda je bolje nasljeđivati od klase `Controller` jer ta klasa da je potpunu podršku za rad sa pogledima (eng. View). U kontekstu ovog završnog rada koristiti će se isključivo `ControllerBase`. Kada radimo sa kontrolerima bitno je za napomenuti važnost korištenja apstrakcija, korištenja DTO uzoraka dizajna te korištenje autorizacije, autentifikacije te validacije kao načina povećanja sigurnosti podataka i kontrolera.

2.3.3. Actions

Akcije unutar kontrolera su metode koje su odgovorne za rad sa HTTP zahtjevima i odgovorima. Akcija može biti tip GET, POST, PUT ili DELETE. Akcije vraćaju podatke u JSON ili XML formatu. Često su asinkrone što znači da njihov rad ne blokira glavnu dretvu. Definiranje asinkronih akcija se uvelo prvi put sa ASP.NET 4.0 koji je kasnije evoluirao u ASP.NET Core. Asinkrone operacije se definiraju pomoću ključnih riječi `async` i `await`. Svaka akcija koja nema definirane HTTP zahjeve je non-action. Te akcije su često tipa protected ili internal te se koriste kao način stvaranja apstrakcije unutar kontrolera. Definiramo akcije pomoću ključnih riječi `ActionResult`, `ViewResult` (samo za ASP.NET Core MVC), `JsonResult` (vraća Json tip

podatka), `RedirectResult` (redirektira na neki drugi URL), `FileResult` (vraća neki dokument) te `ContentResult` (vraća HTML). Prednost `ActionResult` je mogućnost vraćanja statusnih kodova poput `OkResult`, `NotFoundResult`, `BadRequestResult`.

2.3.4. Routers

Rutiranje u kontekstu ASP.NET Core služi da postavi HTTP zahtjeve na određene URL-ove (URL endpoints). Rutere definiramo sa [Route] atributom poviše kontrolera te moramo definirati HTTP metodu: [HttpGet], [HttpPost], [HttpPut] ili [HttpDelete]. Ruteri mogu imati prefikse. Kada definiramo ruter na razini kontrolera onda svaka ruta tog kontrolera započinje sa tom rutom. Bitno je za napomenuti da više HTTP zahtjeva može imati isti endpoint jedino ako imaju drugačije HTTP metode. Kod stvaranja rutera bitno se držati nekih konvencija stvaranja naziva. Recimo da radimo putanju za dohvaćanje svih studenata. Ta putanja bi se trebala zvati `api/students`, odnosno ne bi se trebala zvati `api/getStudents`.
Ovdje vidimo da prilikom stvaranja resursa da uvijek stavljamo imenicu, a ne ime HTTP metode.

2.3.5. Models

Modeli su obične C# klase koje služe kao reprezentacija podataka iz baze podataka te služe za serijalizaciju i deserijalizaciju zahtjeva i odgovora. U primjeru ovog završnog rada kreiranje modela će biti obavljeno od stranje Entity Framework Core-a. Modeli mogu stvarati validaciju unosa podataka sa atributima poput *[Required]* ili *[StringLength].*

2.3.6. Dependency Injection

“Dependency Injection” je uzorak dizajna koji se često koristi u objektno-orijentiranim jezicima kao što je C#. “Dependency Injection” stvara kod čitljivijim, fleksibilnijim i robusnijim. Glavni cilj ovoga uzorka dizajna je poboljšati modularnost aplikacije te smanjenje ovisninost između komponenti. Neke od vrsta “Dependency Injection” su: constructor injection, property injection, te method injection.

- **Constructor injection** – ovo je najčešći i najviše korišten oblik “Dependency Injection”-a. To znači da klasa prima druge ovisnosti kroz svoj konstruktor.
- **Property injection** – ovisnosti injektiramo pomoću javnih svojsta klasa (“get” i “set”). Ne koristi se često jer otežava upravljanje ovisnosti koje ne korstimo.
- **Method injection** - ovisnosti injektiramo kroz neku metodu neke klase. Ovaj pristup nam pomaže da određene metode koriste određene ovisnosti samo kada ih trebaju.

2.4. Razvojno okruženje

Visual Studio je jedan od najpopularnijih razvojnih okruženja danas. Visual Studio je jedan od najpopularnijih IDE (Integrated Development Environment) trenutno te danas uključuje debugiranje koda, integraciju sa sustavima za upravljanje kontrolom (github), automatizirano testiranje te pomoć u pisanju koda (github copilot). Povijest Visual Studia počinje 1989. godine kada Microsoft izdaje Visual Basic. Prva izdanje Visual Studia je 1997 te uključuje jezike poput Visual Basic 5.0, Visual C++, Visual J++. Danas Visual Studio 2022 uključuje podršku za većinu popularnih programskih jezika i framework-a poput: C++, C#, F#, Python, Javascript, Angular, React.

Visual studio nudi programerima brojne značajke koje poboljšavaju produktivnost kao: debbugiranje koda, integracija sa sustavima za verzioniranje, automatizirano testiranje te pomoć u pisanju koda (Intellisense i GitHub Copilot).

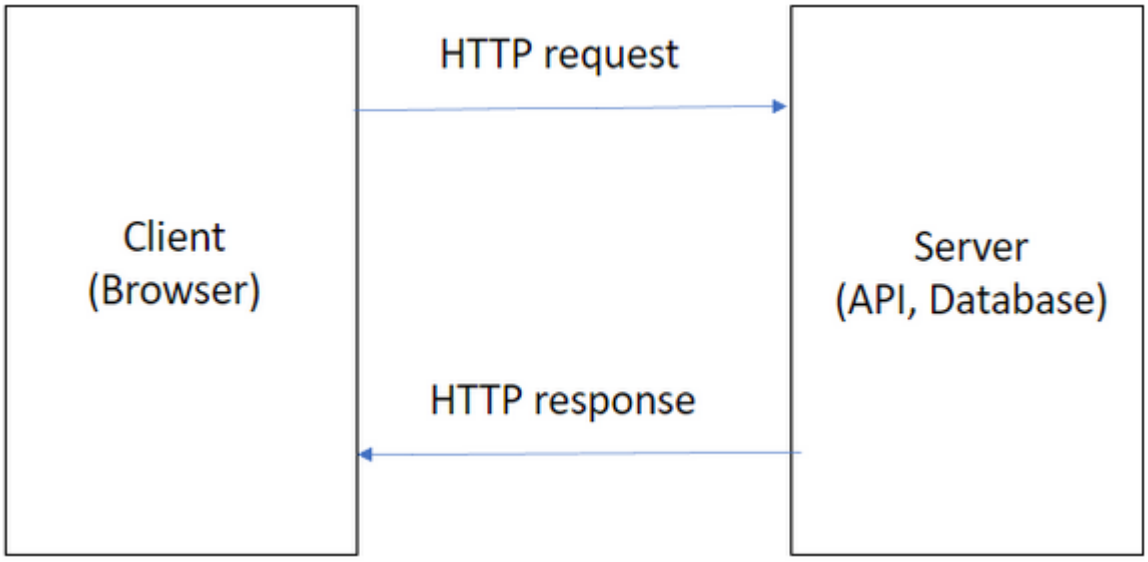
3. ASP.NET Core API

Kao što sam već napisao ASP.NET Core API je programski okvir u C# jeziku koji služi za izgradnju API-ja. ASP.NET Core pruža višeplatformsku podršku te ima podršku za RESTful API te sve metode (GET, POST, PUT, DELETE). ASP.NET Core API bi najbolje usporedio sa metaforom konobara u restoranu. Recimo da je kuhinja naš poslužitelj, a gost klijent. Mi kao vlasnici restorana sada moramo napraviti nekakav “most” između njih tj. kako da gost komunicira sa kuhinjom odnosno klijent sa poslužiteljom. Tu dolazi API, odnosno u našem slučaju konobar te prima narudžbe od gosta te šalje ih kuhinji. Kada je narudžba gotova onda kuhinja preko konobara (API-ja) šalje narudžbu gostu. Iz ovoga primjera se može vidjeti uloga API-ja u odnosu klijenta i poslužitelja.

3.1. Arhitektura projekta

Svaki projekt mora biti izgrađen na nekom arhitekturalnom uzorku. U ovom poglavlju ću objasniti neke od bitnijih arhitektura ASP.NET Core-a. Višeslojna arhitektura označava više slojeva (N) od kojih svaki ima svoje odgovornosti. Korištenje višteslojne arhitektura znatno povećava bolju čitljivost koda, organizaciju odgovornosti, sigurnost te fleksibilnost [17]. Jedna od glavnih prednosti je fleksibilnost korištenja ove arhitekture: mijenjanje koda u jednom sloju ne utječe na sljedeći sloj i obrnuto. Zove se višeslojna arhitektura (eng. *nlayer*) jer ima više od 3 sloja. Česti slojevi koji se mogu vidjeti u ovoj arhitekturi su prezentacijski sloj, sloj za poslovna

pravila te sloj za upravljanje podacima [17]. U ovom završnom radu implementiram poslužiteljsko-klijentsku arhitekturu unutar višeslojne arhitekture. Ovo je pogodno zato što višeslojna arhitektura razdvaja projekt na slojeva te je lako raspoznati koji sloja odgovora klijentu, a koji sloj odgovora poslužitelju. U ovom konkretnom primjeru klijentska strana je sloj UI dok su slojevi API, BAL, DAL i DTO sve dio poslužiteljske strane. Klijentsko-poslužiteljska arhitektura znači da imamo dvije strane aplikacije, a to su klijent i poslužitelj. U klijentsko poslužiteljskoj arhitekturi klijent služi samo za prikazivanje podataka dok poslužitelj obavlja svu potrebnu manipulaciju podataka. Glavna ideja je da prilikom korištenja aplikacije klijent šalje zahtjeve na poslužitelj te poslužitelj šalje odgovor. Ovaj pristup razbija odgovornosti te je jasan protok podataka. Klijentsko poslužiteljska arhitektura je osnova za bilo kakvu web aplikaciju. Pomoću sljedeće slike ću detaljno objasniti proces klijentsko poslužiteljske arhitekture:

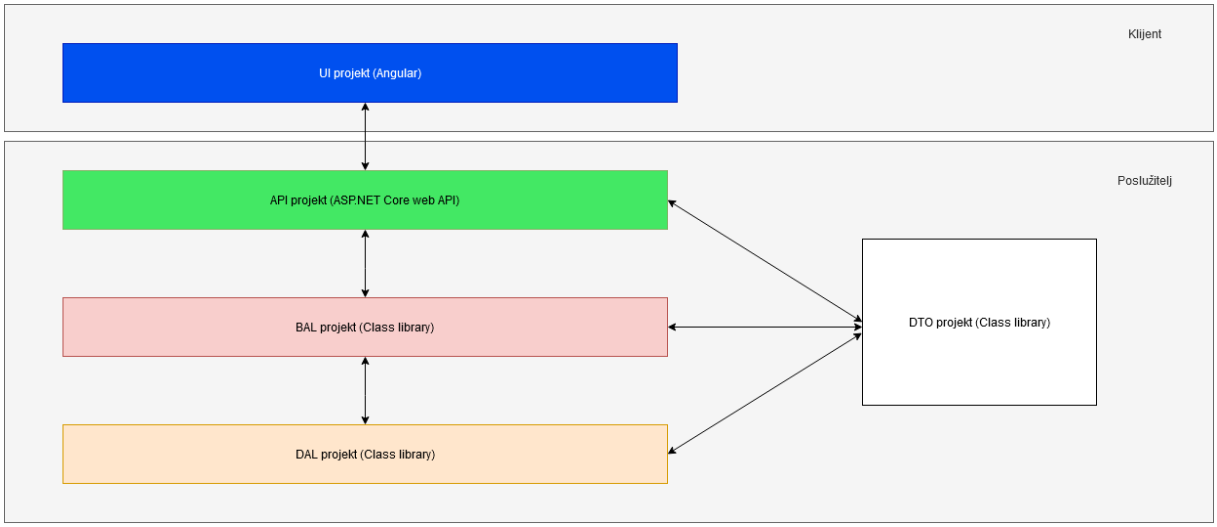


Slika 2: Klijentsko poslužiteljska arhitektura (izvor: slika autora)

Na ovoj slici vidimo da klijent (Angular) šalje HTTP zahtjev na poslužitelj. Svaki zahtjev mora imati svoju pripradajuću metodu bila ona: GET, POST, PUT, DELETE. Nakon slanja HTTP zahtjeva na poslužitelj onda poslužitelj obrađuje zahtjev te vraća odgovor u obliku JSON resursa ili statusni kod.

Ovaj pristup olakšava skaliranje projekta te lakši rad u timu. Ovaj projekt ima 5 slojeva, a to su: UI (Angular), API (ASP.NET Core Web API), BAL (Class Library), DAL (Class Library) te DTO (Class Library). UI sloj služi za prikaz podataka na Web-u te UI dohvaća podatke pomoću API-ja. API služi za posluživanje REST putanja te nema nikakvu poslovnu logiku nego poziva BAL (Business Logic Layer). U BAL-u se nalazi sva poslovna logika te on poziva DAL (Data Access Layer) koji direktno komunicira sa bazom. Svi slojevi osim UI sloja koriste DTO sloj za

manipulaciju sa podacima. Na projektu su obrađene sljedeće tablice iz baze podataka: StudyPrograms, Courses, CourseInAcademicYears, AcademicYears te Teachers.



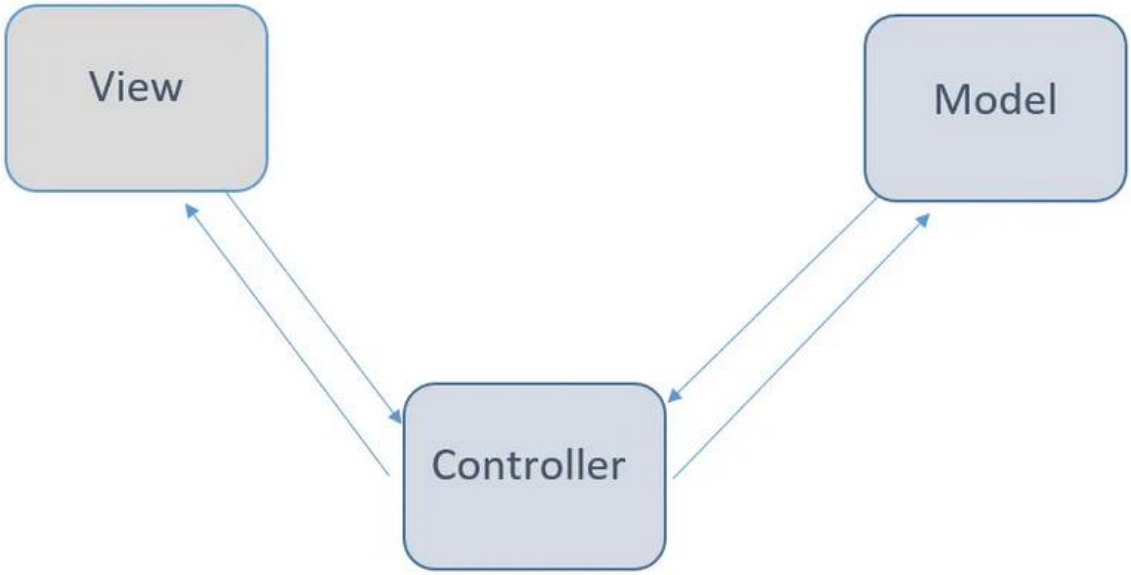
Slika 3: Višeslojna arhitektura u kontekstu klijentsko poslužiteljske arhitekture (izvor: slika autora)

U sljedećoj tablici ću opisati prednosti i nedostatke višeslojne arhitekture:

Tablica 2: Objašnjenja prednosti i nedostataka višeslojne arhitekture (izvor: autor)

Prednosti	Nedostatci
Svaki sloj ima predefiniranu ulogu	Moguće ponavljanje funkcionalnosti i koda
Možemo mijenjati sloj bez da utječemo na druge slojeve	Debuggiranje je teže
Testiranje je jednostavnije jer se svaki sloj može testirati odvojeno	Pisanje više testova
Lako za održavanje te skaliranje	Svaki sloj može imati svoju verziju
Svaki sloj implementira svoje sigurnosne značajke	Konstantna međusobna komunikacija smanjuje performanse
Jednostavno integriranje više tehnologija u više slojeva	Visoka početna krivulja učenja (za korištenje višeslojne arhitektura potrebno je znati mnogo više uzoraka dizajna)
Jednostavnija zamijena tehnologija	Rizik od pretjerane složenosti dizajna softvera

Još jedan bitan arhitekturalni uzorak dizajniranja web aplikacija pomoću ASP.NET Core-a je MVC uzorak. MVC uzorak kao što i ime sugerira se sastoji od 3 dijela: Model, Pogled (eng. *View*) te Kontroler (eng .*Controller*). Ovaj pristup razvoju olakšava održavanje te povećava fleksibilnost. Pomoću sljedeće slike ću opisati glavnu ideju MVC arhitekture.



Slika 4: MVC Arhitektura (izvor: [19])

Povijest MVC arhitekture počinje sa Trygve Reenskauh-om 1978. godine. Iako je stvorena prvo za grafička sučelja danas se isključivo koristi kao jedan od načina izgradnje web aplikacija [19]. Model dio MVC-a služi za upravljanje sa podacima. Na njega možemo gledati kao DAL i BAL sloj u višeslojnoj arhitekturi. To znači da je model zadužen za direktnu komunikaciju sa bazom te sva poslovna pravila. Podaci iz modela se šalju direktno na kontroler. Pogled u MVC arhitekturi se može usporediti sa prezentacijskim slojem u višeslojnoj arhitekturi. Pogled ima sva pravila za grafičko sučelje te preko njega korisnici aplikacije interaktiraju sa našim sustavom. Kao i višeslojna arhitektura, MVC ima kontroler. Kontroler dohvaća podatke pomoću modela te ih šalje pogledu da se prikažu korisniku. Kontroler ne upravlja poslovnom logikom i ne komunicira direktno sa bazom podataka. Pomoću sljedeće tablice ću navesti prednosti i nedostatke MVC arhitekture.

Tablica 3: Objašnjenja prednosti i nedostataka MVC arhitekture (izvor: [19])

Prednosti MVC arhitekture	Nedostatci MVC arhitekture
MVC arhitektura dobro razdvaja slojeve grafičkog sučelja, poslovne logike te upravljanja sa podacima.	Početna krivulja učenja je visoka.
Komponente koje stvorimo možemo ponovno koristiti.	Moguće stvaranje ne potrebnih komponenata.
Olakšava testiranje.	Nije korisno za manje sustave i aplikacije.

3.2. DTO – Data Transfer Object

DTO (eng. *Data Transfer Object*) je jednostavan objekt koji služi za prijenos podataka između 2 sloja u aplikaciji [9]. Najčešće prenosi podatke između poslužitelja i klijenta. Povećavaju brzinu aplikacije te povećavaju sigurnost. U ovom projektu stvaramo model baze podataka pomoću Entity framwork Core-a te on nam stvara model baze podataka. Kada želimo upravljati podacima iz baze podataka, korištenje modela može usporavati aplikaciju te dohvaćati podatke koji nam ne trebaju (navigacijska svojstva). DTO se često stvara sa ključnom riječi `record`. Ključna riječ `record` samo znači da se vrijednosti atributa ne mogu mijenjati poslije inicijalizacije. DTO može enkapsulirati validacijsku logiku. Validacijska logika u ovom slučaju označava atribute sa deklaracijama kao [StringLength] ili [Required]. U teoriji niti jedan DTO nebi smio imati nikakvu logiku [12]. DTO je iznimno koristan kada želimo dohvaćati podatke koje želimo te ovaj pristup ubrzava aplikaciju. DTO naravno i može u sebi imati podatke iz više izvora. Sve ove značajke čine DTO korisnim uzorkom dizajna.

3.3. CORS

Cross-Origin Resource Sharing (CORS) je sigurnosna funkcionalnost pomoću koje poslužiteljska strana omogućava koji klijenti mogu koristiti taj sadržaj [9]. U kontekstu ASP.NET Core API-ja CORS se definira u Program.cs klasi. Ovo u suštini omogućava samo predefiniranim stranicima da koriste resurse poslužitelja. O primjeru ovog završnog rada ja moram slati podatke sa ASP.NET Core API projekta na Angular klijentsku stranu.
Ovako sam definirao CORS u Program.cs-u.

```
builder.Services.AddCors(options => {
    options.AddPolicy(MyAllowSpecificOrigins,
        policy => {
            policy.WithOrigins("http://localhost:4200")
                  .AllowAnyHeader()
                  .AllowAnyMethod()
                  .AllowCredentials();
        });
});
```

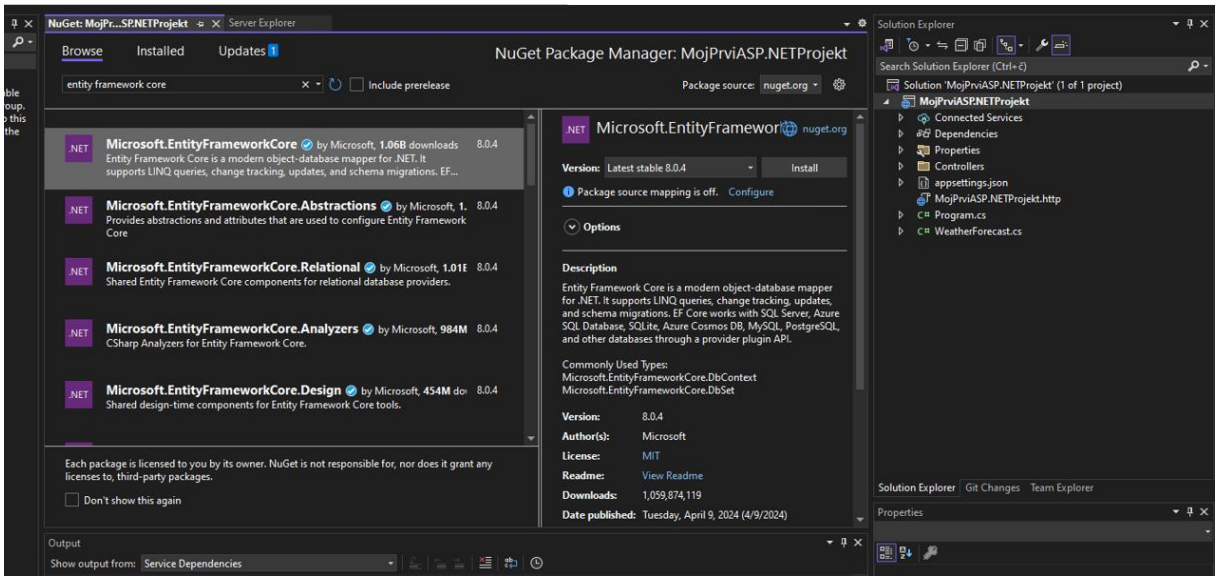
Ovdje sam definirao da stranica *http:///localhost:4200* (angular klijent) ima pristup svim resursima API-ja.

3.4. Entity Framework Core

Entity framework Core je ORM (eng. *Object-relational mapping*) stvoren od strane Microsoft-a 2006. godine. Služi za povezivanje sa bazom i stvaranje visoke razine apstrakcije pri radom sa podacima iz baze. Može se koristiti isključivo na .NET projektima.

3.4.1. Instalacija Entity Framework Core-a

Da bi Entity framework Core pravilno funkcionirao moram instalirati sljedeće: *Microsoft.EntityFrameworkCore, Micorsoft.EntityFrameworkCore.Design te Micorsoft.EntityFrameworkCore.SqlServer.*



Slika 5: Prikaz preuzimanja Entity Framework Core-a (izvor: slika autora)

Nakon uspješne instalacije moramo izvršiti sljedeću naredbu u Developer powershell-u.

```
dotnet ef dbcontext scaffold "Data Source=31.147.206.65;Initial
Catalog=CourseManagerTest;User ID=sbraica;Password=******;Connect
Timeout=30;Encrypt=True;Trust Server Certificate=True;Application
Intent=ReadWrite;Multi Subnet Failover=False"
Microsoft.EntityFrameworkCore.SqlServer -o Models
```

dotnet – komanda za interakciju sa .NET CLI
ef – Entity Framework
dbcontext scaffold – komanda pomoću koju „preslikavamo“ bazu podataka u obliku klasa. Ovo nam omogućava da komuniciramo sa bazom preko LINQ-a.

Nakon što izvršimo ovu naredbu trebali bi dobiti entitete iz baze podataka te klasu dbcontext.

3.4.1.1. Arhitektura Entity Framework Core-a

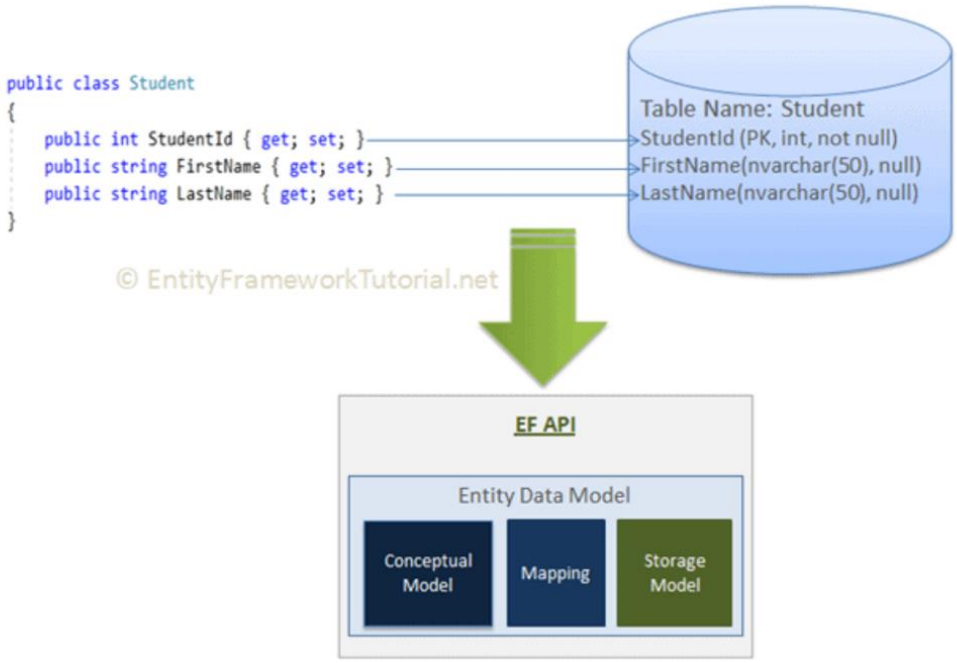
API za Entity Framework Core funkcionira pomoću:

Conceptual model – je model klasa i dbcontext-a stvoren iz baze

Storage model – je model baze podataka iz koje želimo napraviti EF

Mapping - način na koji se conceptual model mapira na storage model

DB context – klasa koja se generira prilikom pokretanja EF Core-a koja je u suštini predstavlja most između baze podataka i aplikacije. Pruža metode za stvaranje upita te spremanje promjena (*Remove, Add, Update, SaveChanges*).



Slika 6: Prikaz arhitekture Entity Framework Core-a (izvor: [3])

Lazy Loading – tehnika pomoću koja učitava entitete tek kada im eksplicitno pristupimo. Iznimno je korisna tehnika koja optimizira rad sa bazom podataka tako da učitava samo podatke kada ih trebamo. Lazy Loading povećava efikasnost tako što radi upite nad bazom samo kada treba, te pojednostavljuje proces razvoja tako što ne moramo eksplicitno specificirati koje entitete treba učitati. Iako Lazy Loading povećava efikasnost dohvaćanja podataka iz baze također mora više upita raditi kada radi upit nad entitetima koji imaju više relacija. Neke od metoda Lazy Loading-a su *Collection() i Reference().*

Eager Loading – tehnika koja za razliku od Lazy Loading-a učitava sve entitete te njihove relacije kada napravimo upit nad bazom. Eager Loading poboljšava performanse zato što radi samo jedan upit na bazom da dohvati sve entitete i njihove relacije. Naravno ako koristimo Eager Loading moramo paziti da ne dohvaćamo previše podataka od jednom te opterećujemo bazu, isto se mora i eksplicitno navesti koje entitete želimo dohvatiti upitom što može smanjiti čitljivost koda i povećati cijene održavanja. Neke od metoda Eager Load-a su *Include() te ThenInclude().*

3.4.2. Usporedba Entity Framework Core-a sa Dapper-om

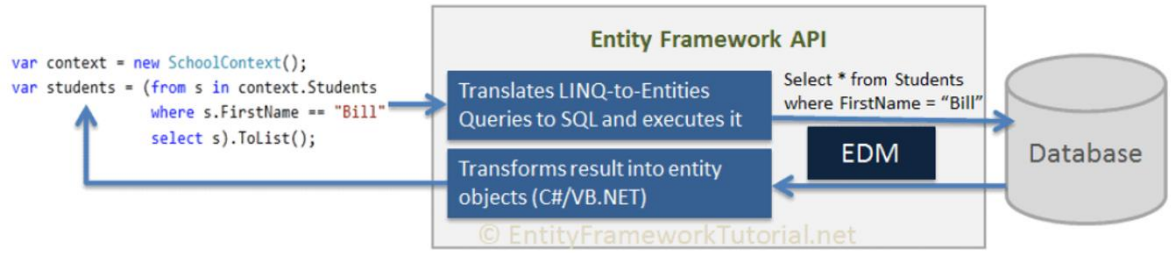
Kao i EF Core Dapper je ORM, ali služe drugačijim svrhama. Dapper je brz, lagan te efikasan te radi na niskoj razini apstrakcije mikro ORM, dok EF Core pruža visoku razinu apstrakcije, sporiji je od Dapper-a te kompliciraniji od Dapper-a. U sljedećoj tablici ću objasniti prednosti EF Core-a te prednosti Dapper-a

Tablica 4: Razlika EF Core-a i Dapper-a (izvor: [7])

Prednosti EF Core-a	Prednosti Dapper-a
Veća produktivnost	Brže performanse
Brži razvoj softvera	Bolji za rad sa puno podataka
Visoka razina apstrakcije	Minimalne apstrakcije
LINQ	Bolji za manje projekte
Jednostavnije stvaranje kompleksnijih modela	Bolja i lakša komunikacija sa bazom („stored“ procedure)
„Code-First Development“	SQL upiti

3.5. LINQ

LINQ (Language Intergated Query) je način pristupanja bazi koji omogućuje programerima da pišu upite nad bazom u programskom jeziku u kojem programiraju. LINQ daje programeru mnoštvo metoda koji omogućavaju filtriranje, sortiranje, agregaciju



Slika 7: Prikaz primjene LINQ-a u kontekstu EF Core-a (izvor: [4])

Ovdje vidimo da Entity Framework prevodi LINQ upite nad bazom u SQL upite koristeći EDM te vraća rezultat. EDM (Entity Data Model) je model koji opisuje strukturu podataka baze podataka. To uključuje sve relacije i sve entitete. LINQ je stvoren od strane Microsofta kao dio .NET framework-a sa ciljem da olakša upite nad bazom podataka. Stvoren je 2007. godine u sklopu .NET Framework 3.5. Iako se LINQ ne može koristiti u drugim programskim jezicima osim C# postoje jako slične alternative poput: Java ima Stream API, Python i Javascript imaju metode poput .map(), .filter() i .reduce().

```
var context = new NoviKontekst();

var studenti = (from s in context.Students
                where s.FirstName == "Sime"
                orderby s.FirstName ascending
                select s).ToList();
```

U ovom primjeru stvorili smo novu instancu konteksta baze podataka. Nakon toga deklariramo varijablu studenti te pomoću LINQ-a radimo upit nad bazom. U ovom specifičnom primjeru iz baze želimo dohvatiti sve studente čije prvo ime (s.FirstName) je Sime te sortirati te iste uzlazno te pretvoriti u listu sa metodom .ToList(). Moramo koristiti .ToList() da pretvorimo entitete dohvaćene iz baze u listu objekata.

3.6. Sigurnost ASP.NET Core Api-ja

3.6.1. BCrypt

Bcrypt je „hashing“ algoritam baziran na Blowfish Cypher-u. Blowfish Cypher je stvoren 1993 sa ciljem da bude algoritam opće namjene [13]. Bcrypt koristi 128-bitovni salt te enkriptira 192 bitovnu vrijednost. Hashiranje je način spremanja podataka koji osigurava jednostrani pristup. Bcrypt funkcionira na način da „uzme“ input password-a te vrati hashiranu vrijednost tog istog password-a. Prilikom stvaranja hashirane lozinke Bcrypt algoritam prvo mora generirati sol.

Sol u ovom slučaju je nasumični niz znakova. Generiranje sol-i otežava hakeru da koristi tablice riječi ili rječnike za napad. Nakon što je sol generirana Bcrypt algoritam vrši više iteracija samog sebe. Ovo usporava proces hashiranja, ali osigurava veću sigurnost. Povećanje ili smanjenje broja iteracija se određuje sa workFactor parametrom. U ovom primjeru workFactor je 12 što znači da će se algoritam izvesti 2^12 puta. Bcrypt algoritam automatski dodaje sol vrijednostima što znači da će dvije vrijednosti iako iste imati potpuno drugačiju hashiranu vrijednost [13].

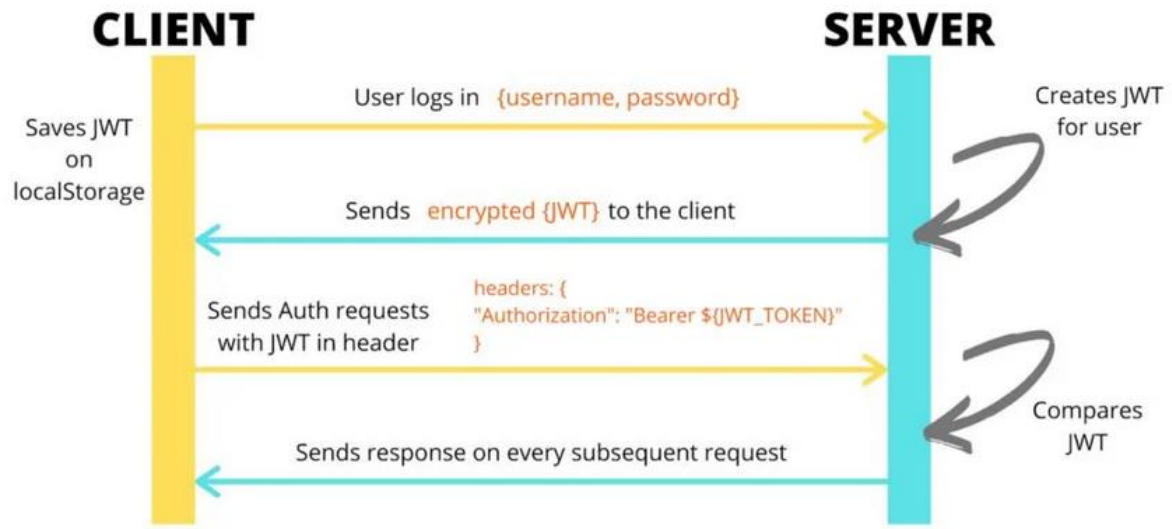
```
var passwordHash = BCrypt.Net.BCrypt.HashPassword(user.Password,
workFactor: 12);
```

Nakon što je Bcrypt iterirao onda mora i hashirati. Hashiranje se provodi koristeći Blowfish algoritam. Hashirana vrijednost često bude oko 60 znakova. Koristiti ću Bcrypt u ovom radu tako da ću pomoću njega Hashirati lozinku i spremiti je u bazu. Čitanje iz baze će biti napravljeno tako da kada korisnik pokuša se prijaviti na aplikaciju, tada on šalje sa POST request-om zahtjev koji hashira njegovu lozinku te uspoređuje vrijednost te lozinke te bilo koje hashirane lozinke u bazi. Ako je prijava uspješna aplikacija vraća statusni kod 200(OK). Prilikom registracije korisnika njegova lozinka se hashira i sprema u bazu.
Bcrypt se prvi put pojavljuje 1999 te je danas široko prihvaćen standard za osiguravanje sigurnost aplikacija.

3.6.2. JWT

JWT (eng. *JSON Web Token*) je token koji je danas standard za prijenos informacija u kompaktnom i zaštićenom načinu [9]. Informacije koje JWT šalje su digitalni potpisane pomoć HMAC algoritma ili pomoću kombinacije javnog i privatnog ključa. Naime HMAC (Hash-based Message Authentification Code) je u suštini kriptografski algoritam koji se koristi da osiguramo da sadržaj JWT-a se nije promijenio te da je JWT izdao ovlašteni izdavač. Prilikom stvaranja JWT-a trebali bi spremiti JWT u HTTP-only kolačić zbog veće sigurnosti protiv XSS napada, treba se staviti vrijeme isteka (expiration date), treba se koristiti HTTPS te ne smijemo spremati senzitivne informacije u JWT (poput lozinke).
JWT se najčešće koristi za autorizaciju korisnika te za izmjene informaciji između 2 strane (npr. API i korisnik). Autorizacija korisnika pomoću JWT-a u kontekstu ASP.NET Core Api-ja funkcionira tako da prijava mora biti POST putanja. Prilikom slanja zahtjeva korisnik upisuje svoje korisničko ime i lozinku te šalje. S obzirom da šalje na POST putanju onda ta ista vraća statusni kod. U slučaju da je prijava uspješna, poslužitelj vraća statusni kod OK(200), a u slučaju da je prijava ne uspješna poslužitelj vraća statusni kod 400 (loše formatiran zahtjev), 401 (prijava nije uspjela), 403 (pristup zabranjen), 404 (resurs nije pronađen), 500 (greška na

poslužitelju). Ako je prijava uspješna poslužitelj vraća statusni kod OK(200) te stvara JWT koji se sprema u HTTP-only kolačić te se kasnije koristi za autentifikaciju korisnika za daljnji pristup aplikaciji.



Slika 8: Prikaz logike JWT tokena (izvor: [5])

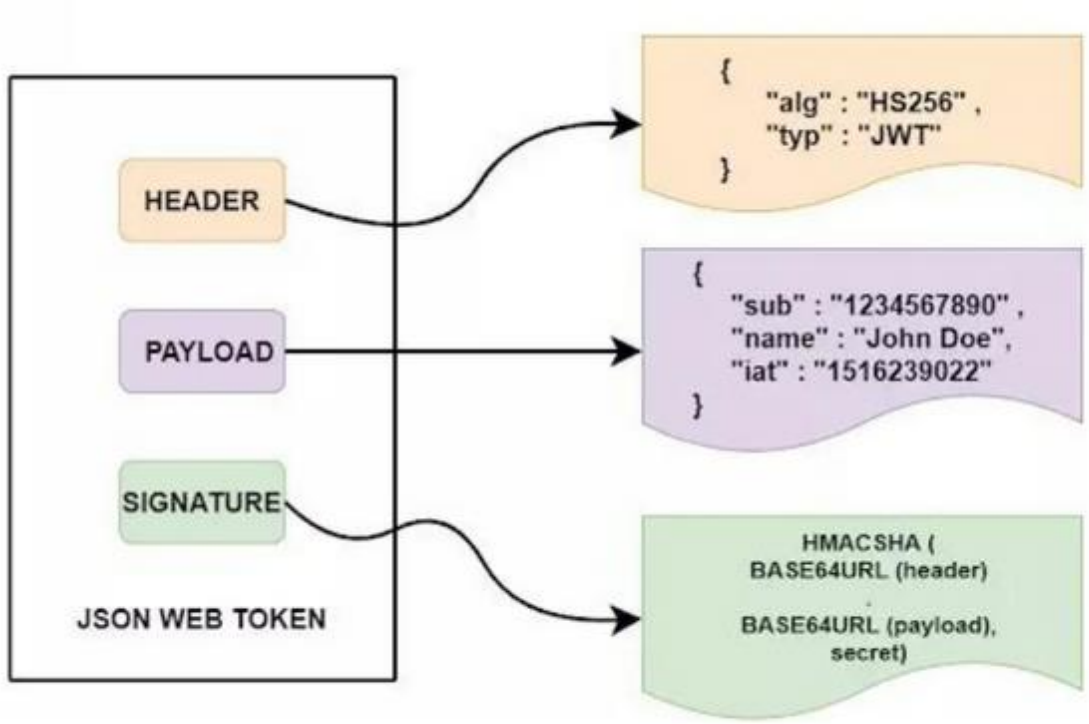
Autentifikacija pomoću JWT

Autentifikacija pomoću JWT tokena funkcionira na dosta jednostavan način. Prilikom prijave u aplikaciju korisnik unosi svoje korisničke podatke te ako ti podaci odgovaraju korisniku koji postoji u bazi podataka onda API stvara JWT token pomoću kojeg korisnik može dalje pristupati aplikaciji.

Autorizacija pomoću JWT

Autorizacija pomoću JWT tokena funkcionira tako da kod svakog kontrolera koji mora imati autorizirani pristup ima deklaraciju [Authorize]. [Authorize] atribut označava da pristup REST putanji imaju samo ovlašteni korisnici Ova ključna riječ označava da svaki kontroler mora primiti Authorization: Bearer JWT prilikom stvaranja upita. U slučaju da je token valjan korisnik može pristupiti podacima, u slučaju da nije vraća 401(Unauthorized). U slučaju mog završnog rada kao što sam već rekao moj JWT token se sprema u httpOnly kolačić te se čita i dekodira iz istoga.

Struktura JWT-a



Slika 9: Prikaz strukture JWT-a (izvor: [6])

Svaki JWT se sastoji od 3 dijela: glava, sadržaj i potpis [4]. Glava se sastoji od 2 dijela: Algoritam koji se koristi te tip tokena (u ovom slučaju JWT). Sadržaj se sastoji od payloada koji najčešće sadržava izjave, publiku te datum isteka. Signature se sastoji od hashirane vrijednosti glave i sadržaja [5]. Potpis koristimo da validiramo da JWT se nije mijenjao u prijenosu.

3.6.3. HTTP-only kolačić

Kolačići su način da se podaci mogu slati sa poslužiteljske strane na klijentsku. Prvi put se počinju koristiti 2002. godine. HTTP-only kolačić je običan kolačić koji ima HTTP-only atribute. HTTP-only atributi samo znače da se podacima može pristupiti pomoću HTTP protokola, što naravno znači da ne mogu biti modificirani sa klijentske strane. Jedna od glavnih prednosti korištenja HTTP-only kolačića je to što preventiraju XSS napade tako što ne dopuštaju Javascript-u da koristi kolačić. U završnom radu ću spremati vrijednost JWT-a u HTTP-only kolačić. HTTP-only kolačići stvaraju još jedan sloj sigurnosti za osjetljive podatke jer HTTP-only kolačić se postavlja I koristi na poslužiteljskoj strani.

3.6.4. Zaštita od CSRF napada

Cross-site request forgery (CSRF) je vrsta napada koja napadaču dopušta da vrši operacije nad aplikacijom za koju nema pristup [14]. Da se aplikacija zaštiti od CSRF napada treba definirati CORS policu sigurnosti gdje definiramo koje stranice imaju pristup API-ju te postavljanje vrijednosti tokena u HTTP-only kolačić. HTTP-only kolačić štiti od CSRF napada jer tom kolačiću ne može pristupiti Javascript napadača da izmjeni zadržaj kolačića. Još jedan od načina kako možemo zaštititi našu aplikaciju od CSRF napada je korištenjem CSRF tokena. Naime CSRF token je token koji se generira i šalje prilikom svakog generiranja zahtjeva prema API-ju. Token se provjerava prilikom svakog slanja da se utvrdi identitet vlasnika te da je token poslan sa naše web aplikacije. CSRF token bi se trebao spremati u httpOnly kolačić.

3.6.5.Zaštita od XSS napada

Cross side scripting (XSS) napad je tip napada koji omogućava napadaču da da stavi svoje skripte u našu web aplikaciju [15]. Te skripte mogu dovesti do ne autoriziranih pristupa te curenja podataka. Za spriječavanje XSS napada često se koriste sljedeće metode:

- **Validacija inputa** – prije slanja bio kakvih informacija klijentski dio aplikacije bi trebao imati validaciju koja onemogućava slanje određenih znakova ili riječi.
- **httpOnly kolačić** – omogućava kolačiću da bude nedostupan sa klijentske strane.
- **Content Secure Policy (CSP)** – omogućava programerima da definiraju koje izvore podataka preglednik smije učitati na svojoj stranici. Korištenje CSP-a ograničava korištenja „inline“ skripti.

3.6.6.Usporedba ASP.NET Core sa Node.js

Node.js je razvojni okvir za razvijanje web aplikacija pomoću Javascript programskog jezika. Koristi programiranje pomoću događaja te ne-blokirajući model. Koristi se za izgradnju web-servera te API-ja. Iako je Iako je ASP.NET napravljen za isključivo Windows operacijske sustave, ASP.NET Core je stvoren kao jezik koji ima „crossplatform“ podršku. To znači da se oba programska okvira mogu izvršavati na više operacijskih sustava poput Linux, MacOS te Windows. Jedna od glavnih prednosti Node.js je to što je napisan u Javascript-u. To osigurava programeru da napravi cijelu aplikaciju pomoću Javascripta-a uključujući klijentsku I poslužiteljsku stranu. Sigurnost te skalabilnost su neke od glavnih prednosti ASP.NET Core programskog okvira. ASP.NET koristi uvijek HTTPS što označava šifriranu komunikaciju između poslužitelja I klijenta. U praksi oba programska okvira su dobra i korisna. Node.js se koristi češće za sustave koji funkcioniraju pomoću mikroservisa poput eBay-a, Netflix-a te

Uber-a, dok ASP.NET Core se koristi ipak više za velike distributirane sisteme poput Stack Overflow-a i Intel-a.

4. Korištenje klijentskih tehnologija sa ASP.NET Core

4.1. Angular

Klijentske tehnologije su skup alata pomoću kojih možemo kreirati interaktivne i dinamične web aplikacije. To su sve tehnologije koje su odgovorne za ono što korisnik vidi, te način na koji korisnik interaktira sa našom aplikacijom. Osnovni skup klijentskih tehnologija uključuje HTML (eng. *HyperText Markup Language*), CSS (eng. *Cascading Stype Sheets*) te Javascript. Kako su zahtjevi postajali sve kompleksniji, sa vremenom ove 3 osnovne tehnologije nisu više bile dovoljne. Zbog ovih razloga su stvoreni razni programski okviri koji olakšavaju proces stvaranja modernih i interaktivnih web aplikacija. Neki od najpopularnijih su: Vue.Js, React te Angular. Za ovaj rad sam odabrao Angular programski okvir zbog prethodnih iskustava te lakoće korištenja.

Angular je razvojni okvir napravljen od strane Google-a 2009 godine [19]. On je razvojni okvir otvorenog koda što znači da bilo tko može skinuti izvorni kod i mijenjati programski okvir po želji. Angular postoji u dvije verzije: AngularJS te Angular. Druga verzija zvana Angular 2, odnosno Angular je stvorena 2016 godine te je značajno drugačiji od originalnog AngularJS-a. Neke od glavnih promjena su:

- **Typescript** – od 2016 Angular koristi isključivo Typescript
- **Komponente** – razvoj pomoću komponenata. Komponenta je dio ponovno iskoristivog koda te cijeli kod funkcionira pomoću komponenata
- **CLI** (eng. *Command Line Interface*)

Angular je samo jedan od brojnih Javascript programskih okvira te često se uspoređuje sa React.js te Vue.js. Programski okvir Angular ima mogućnost stvaranja rutera koji omogućavaju jednostavnu navigaciju između različitnih komponenti unutar web aplikacije. Ruter ostvaruje SPA (Single Page Application) pristup izgradnje web aplikacija jer prilikom korištenja rutera web preglednik ne mora učitavati ponovo cijelu stranicu. Rute se definiraju u app.module.ts klasi.

4.2. Typescript

Typescript je “superset” Javascript programskog jezika. Za razliku od Javascript-a:

- TypeScript uvodi strože definiranje tipova podataka kao što su number, string, boolean, array, tuple, enum itd. Ovo pomaže programerima da uhvate greške tijekom razvoja jer TypeScript provjerava tipove prilikom kompilacije, što nije slučaj s čistim JavaScriptom.
- Typescript uvodi Dependency injection koji olakšava testiranje i smanjuje ovisnosti između različitih dijelova aplikacije.
- TypeScript uvodi enkapsulaciju, što omogućava skrivanje određenih podataka ili implementacija unutar klasa. Korištenjem public, private, protected i readonly modifikatora.
- TypeScript uvodi bolje debuggiranje sa pristupom boljim alatima koji pružaju detaljniji opis grešaka prilikom razvoja.

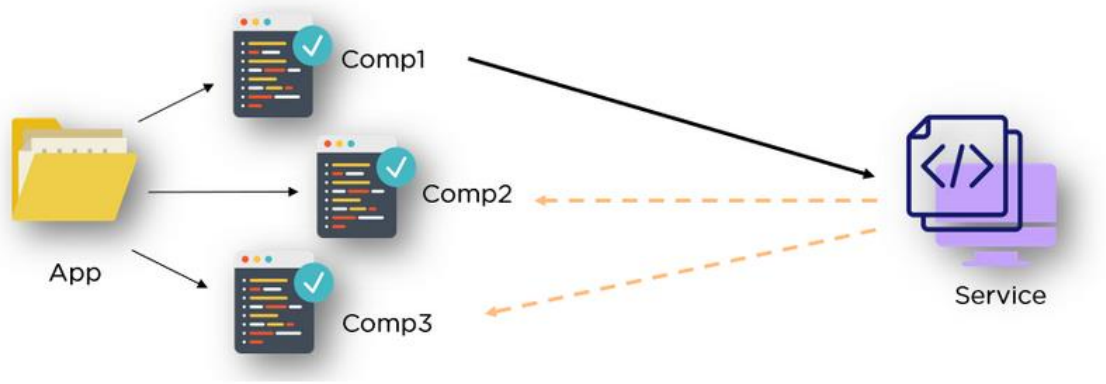
4.3. Komponente unutar Angular-a

Komponente u Angular-u su osnovni blok za izgradnju modernih I robusnih web aplikacija [19]. S obzirom da cijeli programski okvir Angular koristi “component-based” arhitekturu imalo bi smisla objasniti što je zapravo komponenta. Naime svaka Angular aplikacija se sastoji od više komponenata koji zajedno stvaraju jednu veću kohezivnu cijelinu [19]. Komponente su dizajnirane da budu ponovno iskoristive. Komponenta je samo klasa napisana pomoću Typescript-a koja ima dekorator @. Osnovna komponenta koja se generira prilikom stvaranja novog Angular projekta je komponenta zvana AppComponent. To je početna komponenta koja ima predložak cijele aplikacije. Svaka komponenta u Angular-u se sastoji od 3 dijela:

- **selector** - CSS selektor koji određuje kako će se komponenta koristiti u HTML-u. To je oznaka po kojoj Angular prepoznaje i umeće komponentu u DOM-u.
- **templateUrl** - putanja do HTML datoteke koja sadrži predložak (template) komponente. Predložak definira strukturu i sadržaj prikaza komponente.
- **styleUrl** - niz putanja do CSS datoteka koje sadrže stilove specifične za komponentu. Ovi stilovi se primjenjuju samo na elemente unutar predloška te komponente.

4.4. Servisi u Angular-u

Servisi u Angular-u su samo Typescript klase koje imaju @injectible dekorator koji samo označava da se taj određeni servis može "injektirati" u neku od komponenata koja treba taj servis [19]. Servisi većini vremena sadrže poslovnu logiku te jedan se servis može koristiti na više komponenata. U slučaju ovog završnog rada moji servisi služe za komunikaciju sa API-jem te dohvaćanje podataka.



Slika 10: Prikaz interakcije između komponenti i servisa u Anuglar-u (izvor: [2])

5. Praktična implementacija: Course Manager aplikacija

5.1. Uvod u projekt

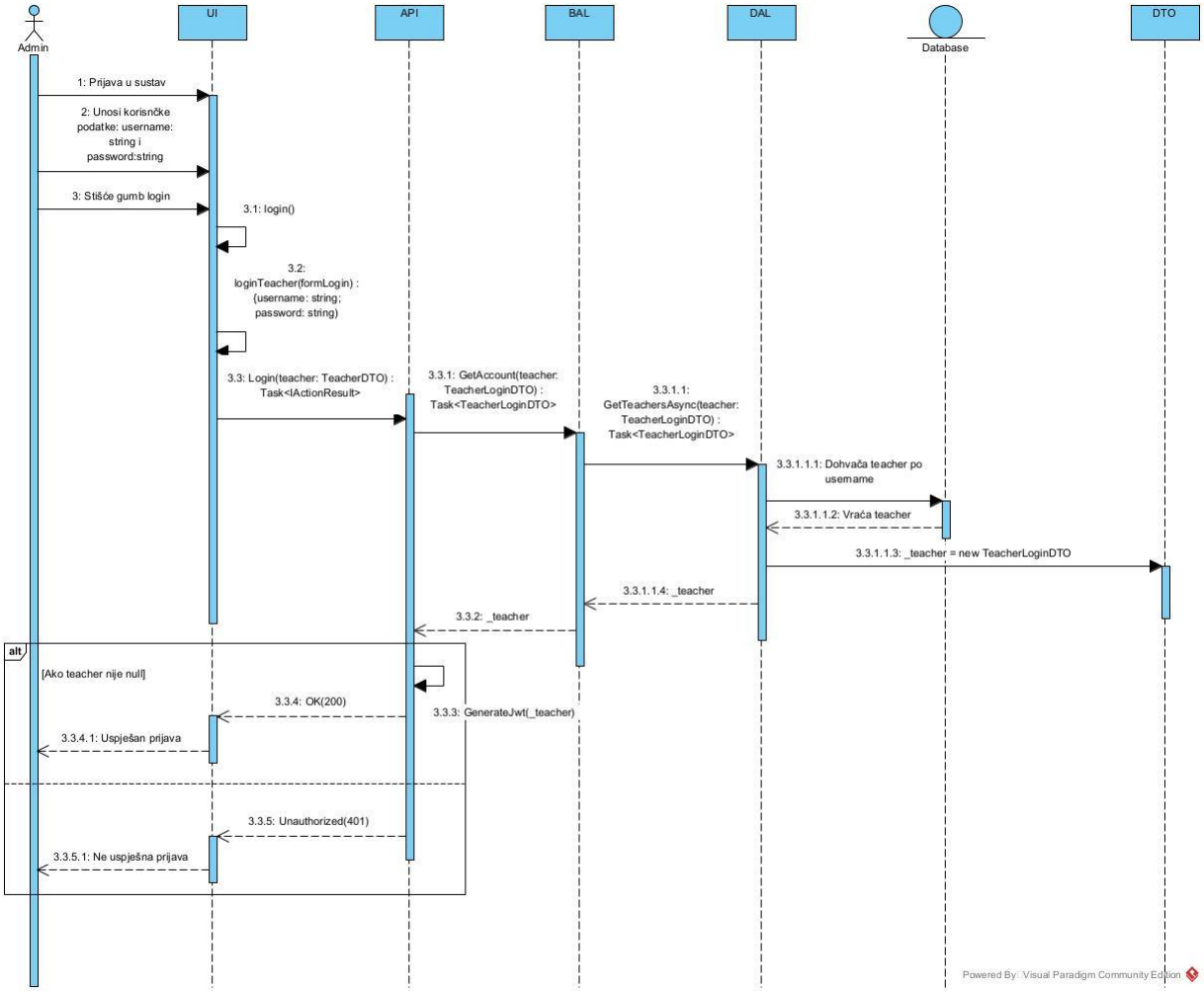
Za praktičnu implementaciju ovoga završnog rada odabrao sam višeslojnu arhitekturu. Kao što sam već i prije rekao višeslojna arhitektura će se sastojati od 5 slojeva: UI (eng. *User Interface*), API (eng. *Application Programming Interface*), BAL (eng. *Business Logic Layer*), DAL (eng. *Data Access Layer*) te DTO (eng. *Data Transfer Object*). Ovaj sam pristup odabrao jer smatram da znatno olakšava pristup izgradnji web aplikacija i povećava čitljivost koda. Za ovaj projekt neću implementirati čitavu bazu podataka nego već sljedeće tablice iz baze: Teachers, AcademicYears, StudyPrograms, CoursesInAcademicYears. Aplikacija je objašnjenja na primjeru StudyPrograms tablice iz baze. Ideja aplikacije je upravljanje kolegijima na fakultetu. Kao što sam već rekao nisam implementirao cijelu aplikacije nego module sa nekim HTTP metodama. Koristeći aplikaciju se bilo koji korisnik može prijaviti te registrirati. U slučaju da je korisnik admin onda može definirati nove akademske godine te konfigurirati kolegije u njima. Admin također može i kreirati novi sveučilišni smjer. Aplikacija

koristi JWT token kao način autorizacije korisnika. JWT token se sprema u httpOnly kolačić. Korisnik se može odjaviti.

5.1.1. Dizajn rješenja

U dizajnu rješenja ću opisati ponašanje aplikacije pomoću diagrama slijeda te diagrama klasa. Diagram slijeda (eng. *Sequance diagram*) su UML (eng. *Unified Model Language*) diagrami koji samo prikazu slijed događaju u nekom kontekstu [16]. Diagram slijeda je samo skupina životnih linija (eng. Lifelines) te njihova međusobna interakcija [16]. Za zahtjeve ovog završnog rada ja sam napravio 2 diagrama slijeda koji prikazuju interakciju između slojeva aplikacije. Diagrami su napravljeni pomoću Visual Paradigm programa.

U sljedećem primjeru ću objasniti rad aplikacije pomoću diagrama slijeda:

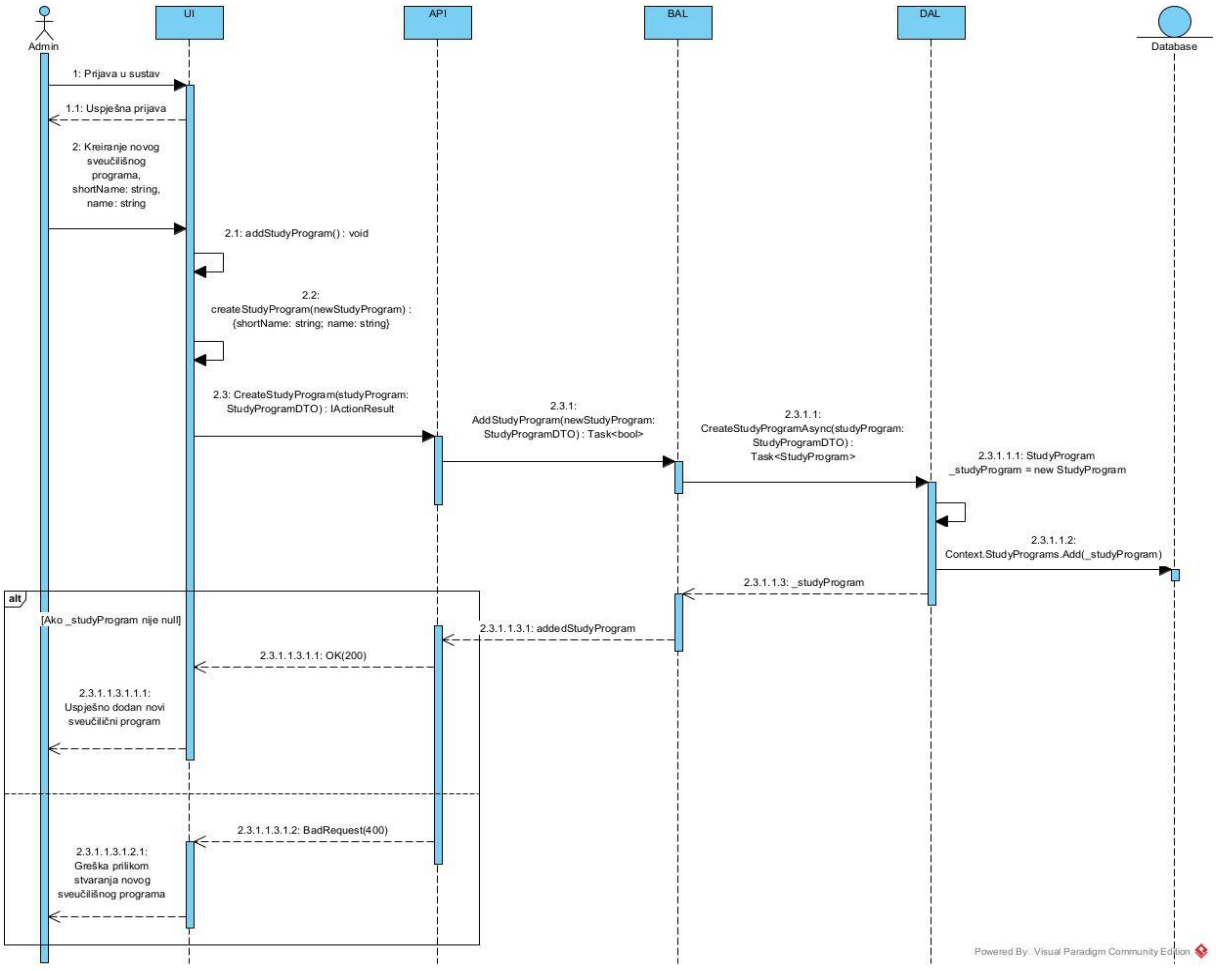


Slika 11: Diagram slijeda za prijavu (izvor: slika autora)

Za potrebe funkcionalnosti prijave korisnik prvo odlazi na web aplikaciju koja će mu prikazati formu za login. Korisnik tada unosi svoje korisničke podatke u ovom slučaju korisničko ime i

lozinku te šalje te podatke aplikaciji. Sada prvo što se događa UI projekt tj. Angular projekt prvo u svojoj kompononeti poziva metodu login() koja uzima podatke prijave te šalje ih svom pripadajućem servisu. U servisi UI projekt koristi metodu loginTeacher() te formatira JSON zahtjev na API te šalje POST upit API dijelu aplikacije. Prilikom slanja na API poziva se metoda Login() te ta metoda poziva metodu iz BAL sloja te ta metoda poziva metodu iz DAL sloja aplikacije. DAL sloja šalje LINQ upit na bazu da dohvati korisnika po njegovom korisničkom imenu te provjeri da li je lozinka jednaka. Nakon LINQ upita samo vraćamo podatke sve do API sloja preko DAL I BAL sloja. U API dijelu moramo napraviti provjeru da li dohvačeni korisnik postoji. U slučaju da postoji API sloj poziva metodu GenerateJwt() te generaira novi JWT token te ga sprema u httpOnly kolačić za daljnju autorizaciju te vraća statusni kod OK(200). U slučaju da je BAL sloj vratio null korisnika onda API sloj vraća statusni kod pomoću kojeg se vidi da je prijava netočna te se ne stvara JWT token te korisnik dobiva poruku neuspješna prijava.

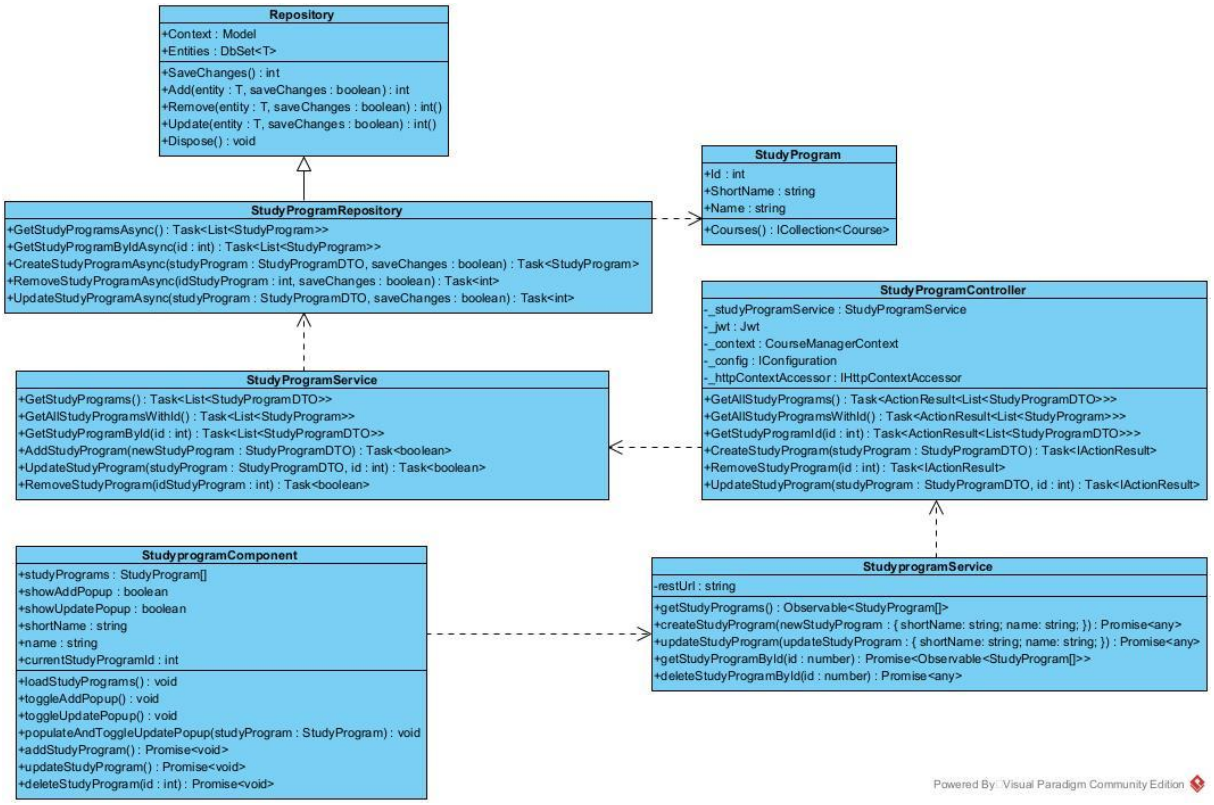
U sljedećem diagramu slijeda vidimo proces kada korisnik želi stvoriti novi sveučilišni program:



Slika 12: Diagram slijeda kreiranje novog sveučilišnog programa (izvor: slika autora)

Za početak procesa kreiranja novog sveučilišnog programa korisnik mora biti prijavljen i autoriziran. Kada korisnik ispuni te uvijete onda odlazi na dio aplikacije gdje se nalazi sad sadržaj kreiranja novog sveučilišnog programa. Iako u bazi podataka piše da svaki sveučilišni program mora imati id, ne mora se postaviti jer je autoincrement. Korisnik šalje API sloju kratko ime (eng. *shortName*) i puno ime (eng. *name*). Prilikom slanja tih podataka API sloju poziva se addStudyProgram() metoda koja čita vrijednosti unosa iz HTML-a komponente pomoću [ngModel] te poziva metodu iz servisa zvanu createStudyProgram(). Ova metoda služi da poziva metode i REST putanje API sloja. API sloj za kreiranje novog sveučilišnog programa očekuje primiti shortName i name. Ova metoda šalje POST zahtjev na API sloj. Na API sloju se vrši autorizacija pomoću JWT tokena te kontrola pristupa. API sloj poziva metodu iz BAL sloja zvanu CreateStudyProgram() koja poziva metodu iz DAL sloja gdje se pomoću konteksta i LINQ-a stvara novi red u tablici sveučilišnih programa. Slojevi DAL i BAL vraćaju novokreirani sveučilišni program te API sloj provjerava da li je novokreirani sveučilišni program različit od null. U slučaju da je API vraća UI sloju statusni kod OK(200), ali u slučaju da nije API vraća statusni kod BadRequest(400).

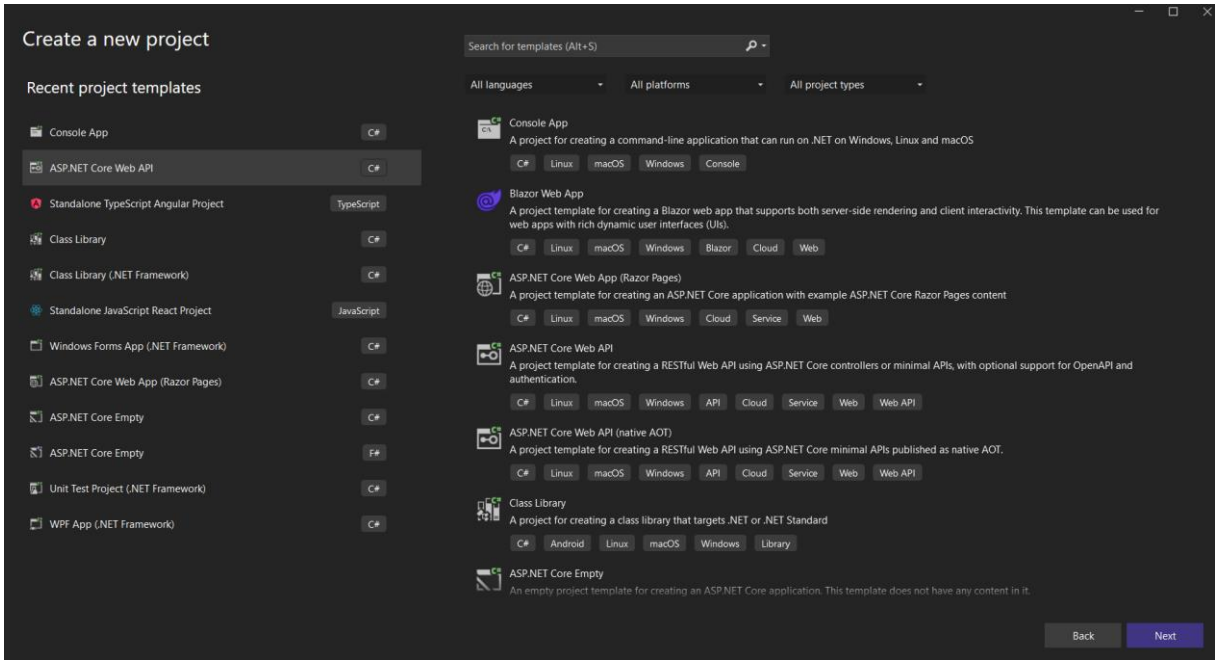
Sljedeća slika prikazuje interakciju klasa na primjeru funkcionalnosti upravljanja sveučilišnim programima.



Slika 13: Diagram klasa sveučilišnih programa (izvor: slika autora)

5.2. Stvaranje projekta

Za stvaranje projekta treba nam Visual Studio razvojnu okruženje. Za ovaj konkretan primjer koristiti će se 3 Class Library projekta, 1 ASP.NET Core Web API te Standalone TypeScript Angular Project.



Slika 14: Kreiranje novog ASP.NET Core web API projekta (izvor: slika autora)

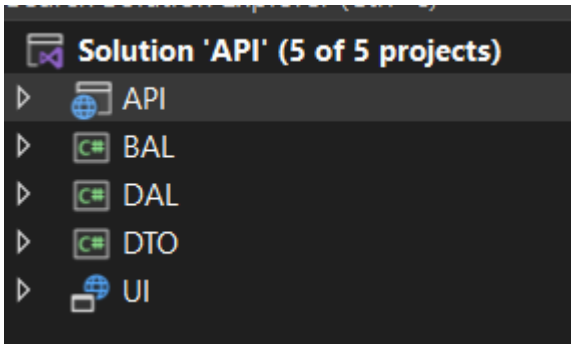
Za instaliranje Standalone Typescript Angular Projekta na računalu prvo moramo instalirati node.js, typescript te Angular CLI. Angular CLI se koristi za stvaranje novih projekata, generiranje koda, testiranje, stvaranje novih komponenti i servisa.

```
npm install -g @angular/cli
```

Pomoću ove komande se preuzima Angular CLI te nakon toga za provjeru je potrebno napraviti

```
ng version
```

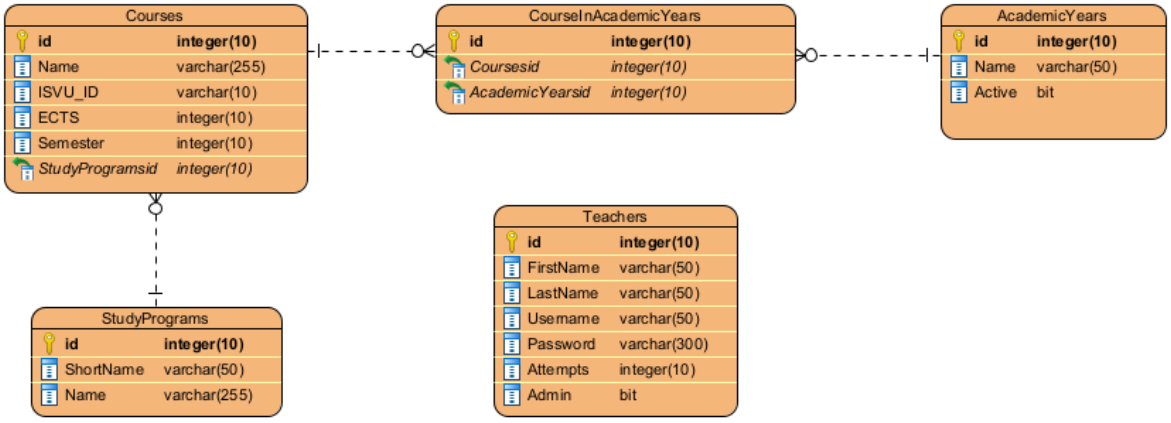
pomoću kojega možemo provjeriti da li je Angular uopće instaliran te koja je verzija.
Nakon svih dodanih projekata aplikacija ima sljedeću strukturu.



Slika 15: Inicijalna struktura projekta (izvor: slika autora)

5.3. Baza podataka

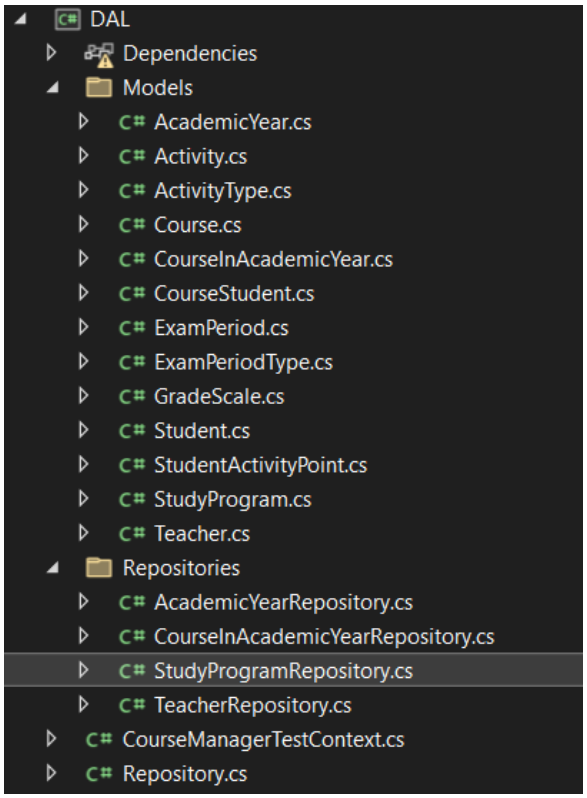
Baza podataka je napravljena u MySQL- u te ima sljedeću strukturu.



Slika 16: ERA model (izvor: slika autora)

5.4. DAL (eng. Data Access Layer)

U ovom poglavlju ću objasniti kako sam implementirao klase repozitorija. Struktura DAL projekta je sljedeća:



Slika 17: Struktura DAL projekta (izvor: slika autora)

Kada govorimo o standardnim i dobrim praksama u razvoju web aplikacija, često govorimo o višeslojnosti aplikacije. Tako je jedan od osnovnih slojeva sloj za pristup podacima (eng. Data Access Layer - DAL). Osnovna funkcija DAL-a je da pruži sloj aplikacije koji je odgovoran za komunikaciju izvorom podataka naše aplikacije, primjerice s bazom podataka. Ovaj pristup arhitekturi aplikacije osigurava da ukoliko dođe to promjene u izvoru podataka potrebno je promijeniti samo jedan sloj, odnosno jednu ovisnost što omogućava veću modularnost i bržu izgradnju aplikacije. Na slici 11 vidljiva je struktura DAL sloja u projektu pisanom za ovaj završni rad. Kako je vidljivo na slici u direktoriju "Models" nalazi se Scaffold baze podataka koji je napravljen koristeći Entity Framework Core, repozitoriji te kontekst za pristup bazi. U osnovi DAL obavežno sadrži informacije o modelima i repozitorijima. Modeli predstavljaju u ovom slučaju klase podataka koji se nalaze u bazi podataka, a repozitoriji su svojevrsna mjesta koja sadrže logiku za rad s podacima. Tako se primjerice u repozitorijima definiraju operacije za kreaciju, čitanje, ažuriranje i brisanje podataka (CRUD).

Glavna funkcija DAL projekta te svih repozitorija unutar njega je komuniciranje sa bazom. DAL projekt se sastoji od modela gdje se nalaze scaffold klasa iz baze pomoću Entity Framework Core-a klase CourseManagerTestContext. CourseManagerTestContext klasa služi za komunikaciju sa bazom podataka te sadrži konekciju za bazu, DbSet atributima koji odgovaraju tablicima u bazi te implementira IDisposable sučelje koje ubrzava performanse

aplikacije tako da miče kontekst kada se nekoristi. IDisposable sučelje omogućava životni ciklus CourseManagerTestContext klase.
Primjer modela (u ovom slučaju StudyProgram klasa)

```
public partial class StudyProgram
{
    public int Id { get; set; }

    public string ShortName { get; set; } = null!;

    public string Name { get; set; } = null!;

    public virtual ICollection<Course> Courses { get; set; } = new
List<Course>();
}
```

Vidimo ovdje da svaki StudyProgram ima svoj Id, ShortName, Name te je povezan na tablicu Courses 1:N. Svaki model je replike tablice iz baze.

Primjer repozitorija (u ovom slučaju StudyProgramRepository)

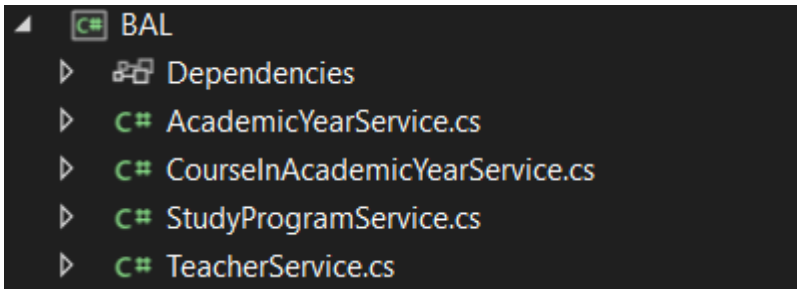
```
public class StudyProgramRepository : Repository<StudyProgram> {
     public StudyProgramRepository() : base(new CourseManagerTestContext())
{

     }
     public async Task<List<StudyProgram>> GetStudyProgramsAsync() {
         var query = from e in Entities
                     select e;
         return await query.ToListAsync();
     }

     public async Task<List<StudyProgram>> GetStudyProgramByIdAsync(int id)
{
         var query = from e in Entities
                     where e.Id == id
                     select e;
         return await query.ToListAsync();
     }
```

Prvo što se može uočiti je da ovaj repoziroij nasljeđuje generičnu klasu Repozitorij tipa StudyProgram te inicijalizira novi kontekst na bazu. Kontekst se mora inicijalizirati da se mogu raditi LINQ upiti nad bazom. U ovom primjeru repozitorija su napravljene 2 metode. GetStudyProgramsAsync te GetStudyProgramByIdAsync. Cilj metode GetStudyProgramsAsync je da dohvati sve vrste sveučilišnih programa. Upit na bazu je

napravljen pomoću LINQ-a te vraćamo listu svih sveučilišnih programa. U drugoj metodi GetStudyProgramByIdAsync metoda prima id te isto pomoću LINQ upita provjerava da li entitet koji je dohvaćen ima Id kao id koji je proslijeđen metodi. Kao što se vidi iz koda obe metode su asinkrone. Asinkrone metode označavaju se sa ključnom riječi **async.** Svaka asinkrona metoda mora vraćati **Task.** Asinkrone metode su izinmno korisne prilikom stvaranja upita nad bazom jer te metode mogu dugo trajati. Korištenje asinkronih metoda značajno poboljšava performanse aplikacije.

5.5. BAL (eng. Business Logic Layer)

BAL projekt sadrži svu poslovnu logiku aplikacije. Ima referencu na DAL projekt da može manipulirati sa podacima koje DAL projekt vraća. BAL (eng Business Logic Layer) je iznimno koristan sloj koji „uzima“ podatke iz DAL sloja te manipulira snjima. Manipulacija podatka iz DAL sloja se npr. događa pomoću DTO-a. Iako nije implementirano u ovom projektu BAL projekt često sadrži validaciju podataka. Naime BAL manipulira podacima da bi prezentacijski sloj, u ovom slučaju API mogao raditi snjima. Prednosti korištenja BAL sloja su odvajanje odgovornosti, ponovna uporaba koda, poboljšana sigurnost te smanjenje programske logike u API sloju- Korištenje BAL sloja je esencijalno za pravilu implementaciju višeslojne arhitekture.



Slika 18: Struktura BAL projekta (izvor: slika autora)

Na sljedećem isječku koda se vidi primjer StudyProgramService klase:

```
public class StudyProgramService {

    public async Task<List<StudyProgramDTO>> GetStudyPrograms() {

        using (var repo = new StudyProgramRepository()) {

            var studyPrograms = await repo.GetStudyProgramsAsync();

                return studyPrograms.Select(studyPrograms => new
StudyProgramDTO {
                    Id = studyPrograms.Id,
                    ShortName = studyPrograms.ShortName,
                    Name = studyPrograms.Name,
```

```
                }).ToList();
            }
        }
```

Cilj GetStudyPrograms metode je vraćanje svih sveučilišnih programama u obliku StudyProgramDTO klase. Metoda funkcionira na način da prvo dohvati sve sveučilišne programe pomoću metode iz repozitorija te preslikava rezultat repozitorija na DTO klasu. Korišten je DTO zato što metoda GetStudyProgramsAsync vraća i listu svih kolegija a to nam trenutno ne treba.

5.6. API (eng. Application Programming Interface)

API sloj u kontekstu višeslojne arhitekture predstavlja prezentacijski sloj. On je ulazna točka za klijentski dio aplikacije te sam po sebi ne sadrži nikakovu poslovnu logiku (BAL sloj sadrži). Uvijek se nalazi iznad DAL i BAL sloja te nema direktnu komunikaciju sa DAL slojem. Ovaj sloj funkcionira tako da podatke izlaže na RESTful putanje (GET, POST, PUT, DELETE) te obrađuje zahtjeve. Kao sloj za REST putanje on vraća statusne kodove u kojima se mogu nalaziti podaci, ali i ne moraju (npr. GET HTTP zahtjev mora vratiti neke podatke dok POST HTTP zahtjev vraća samo statusni kod). Nakon dohvaćanja podataka pomoću BAL sloja, API sloj vrši autorizaciju, autentifikaciju te manipuliranje tokenima i/ili kolačićima po potrebi te vraća odgovor u jedan od dva sljedeća formata: JSON ili XML. Korištenje API sloja u višeslojnoj arhitekturi ima naravno svoje prednosti kao što su: odvajanje odgovornosti što znači da je kod bolje organiziran te API sloj ne sadrži poslovnu logiku, sva autentifikacija i autorizacija se nalazi na jednom mjestu, olakšano dodavanje novih REST putanja jer nove REST putanje ne utječu na postojeću poslovnu logiku BAL sloja. API sloj sadrži ASP.NET Core Web API projekt unutar sebe te tu se nalaze svi kontroleri, Program.cs, Jwt.cs klase te appsetting.json. API projekt ima samo referencu na BAL projekt te prikazuje podatke u JSON formatu na REST putanjama.

Primjer StudyProgram kontrolera:

```
  [Route("api/[controller]")]
    [ApiController]
    public class StudyProgramController : ControllerBase {

        StudyProgramService _studyProgramService = new
StudyProgramService();
        private readonly Jwt _jwt;
        private readonly CourseManagerTestContext _context;
        private IConfiguration _config;
        private readonly IHttpContextAccessor _httpContextAccessor;
```

```
        public StudyProgramController(StudyProgramService
studyProgramService, IConfiguration configuration, IHttpContextAccessor
httpContextAccessor, CourseManagerTestContext context) {
            _studyProgramService = studyProgramService;
            _config = configuration;
            _context = context;
            _httpContextAccessor = httpContextAccessor;
            _jwt = new Jwt(_config, _httpContextAccessor, _context);
        }

        [Authorize]
        [HttpGet]
        public async Task<ActionResult<List<StudyProgramDTO>>>
GetAllAcademicYears() {
            var claims = _jwt.DecodeToken();
            string isAdmin = claims[1];
            if (isAdmin == "admin") {
                var courses = await
_studyProgramService.GetStudyPrograms();
                if (courses != null) {
                    return Ok(courses);
                } else {
                    return BadRequest();
                }
            }
            return Forbid();
        }
```

Na početku vidimo na se koristi [Route] atribut koji stvara HTTP putanju na taj resurs sa imenom api/StudyProgram iako se naš kontroler zove StudyProgramController [Route] atribut miče sufiks Controller. Atribut [ApiController] znači da se automatski svaki put vrši validacija model te kontroler će automatski vratiti statusni kod 400 Bad Request ako je zahtjev netočan. API kontoler pomoću Dependency Injection-a dobiva potrebnu konfiguraciju, kontekt, httpContextAccessor te stvara novu instancu klase Jwt. Kao što je već prije objašnjeno svaka metoda kontrolera mora imati svoj tip HTTP zahtjeva na rest putanji. HTTP zahtjevi mogu biti [HttpGet], [HttpDelete], [HttpPut] te [HttpPost]. Cilj ove metode je dohvatiti sve sveučilišne programe. S obzirom da ova metoda ima deklaraciju [Authorize] onda se svaki upit šalje sa autorizacijskim zaglavljem u kojem se nalazi Jwt. Prvo što metoda radi je pokušava dekodirati Jwt token ako postoji, pomoću metode DecodeJwt() koja će biti objašnjena kasnije. Metoda DecodeJwt() vraća listu svih prava koje Jwt token ima. U ovom slučaju provjerava da li korisnik koji radi zahtjev je admin. U slučaju da je korisnik admin onda dohvaća sve sveučilišne programe pomoću BAL sloja projekta te metode GetStudyPrograms(). U slučaju da metoda

vraća neke sveučilišne programe onda metoda vraća statusni kod 200 te sve dohvaćene sveučilišne programe. Ako zbog nekog razloga metoda nije vratila niti jedan sveučiišni program onda metoda vraća BadRequest odnosno ako korisnik nije admin onda vraća statusni kod 403 Forbidden.

5.6.1. Implementacija Jwt-a

Kao što sam već rekao JWT (eng. Json Web Token) služi za autorizaciju te siguran prijenos podataka. Token je sam po sebi efikasan. Token se sprema u httpOnly kolačić koji je već opisan. Cilj Jwt tokena u ovom završnom radu je autorizacija korisnika. Prvo što se mora napraviti za implementaciju Jwt tokena u aplikaciju je definirati ga u appsetting.json dokumentu. Za ispravno korištenje Jwt treba dodati sljedeće paktete u projekt: Microsoft.AspNetCore.Authentication.Cookies te Microsoft.AspNetCore.Authentication.JwtBearer

```
"Jwt": {
  "Issuer": "https://localhost:12000",
  "Audience": "https://localhost:12000",
  "Key": "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
  "ExpirationMinutes": 60
}
```

Prvo što se mora definirati tko je izdavač tokena (eng. Issuer). Izdavač tokena u ovom slučaju je postavljen na *https://localhost:12000* te taj izdavač potpisuje naš Jwt. Publika (eng. Audience) definira za koga je ovaj Jwt token namijenjen. Ključ (eng. Key) je tajni ključ te nitko nebi smio imati pristup ključu. Služi za potpisivanje i verifikaciju tokena te zadnji dio je vrijeme isteka u minutama (eng. ExpirationMinutes) koje je ovdje definirano kao 60 minuta.

Sljedeći korak je definirati strukturu Jwt-a u Program.cs klasi:

```
builder.Services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultSignInScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
})
```

Prvo što moramo postaviti je model autentifikacije. U ovom isječku koda vidimo da postavljamo Jwt Bearer Token kao glavnu sigurnosnu značajku autentifikacije. Moramo dodati kolačiće te DefaultChallangeScheme označava Jwt autentifikaciju za login.

```
.AddCookie(options => options.Cookie.Name = "token")
```

U ovoj liniji koda samo označavamo da koristimo kolačić te da je njegovo ime token

```
.AddJwtBearer(options => {
    options.TokenValidationParameters = new TokenValidationParameters {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"
]))
    };
```

Ovdje definiramo detalje strukture Jwt-a. Postavljamo vrijednosti izdavača i publike da su potrebni te ih čitamo if konfiguracije tj. iz appsetting.json datoteke. SymmetricSecurityKey dohvaća tajni ključ koji je postavljen u appsetting.json datototeci u ovom slučaju: `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`

```
    options.Events = new JwtBearerEvents {
        OnMessageReceived = context => {
            context.Token = context.Request.Cookies["token"];
            return Task.CompletedTask;
        }
    };
});
```

Ovaj isječak koda samo označava da prilikom svakog zahtjeva na API da se Jwt mora čitati iz kolačića zvanog token.

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

Moramo postaviti API da koristi autentifikaciju i autorizaciju.

UseAuthentication() označava da li postoji neki model autentifikacije definiran u Program.cs klasi i ako postoji da se koristi.

UseAuthorization() označava provjeru da li korisnik ima pravo pristupa nekom pristupu. Dobar primjer je korištenje [Authorize] deklaracije. U slučaju da neki kontroler ima [Authorize] deklaraciju UseAuthorization onda provjerava model autorizacije u Program.cs klasi.

Implementacija Jwt klase:

```
        private IConfiguration _config;
        private readonly IHttpContextAccessor _httpContextAccessor;
        private readonly CourseManagerTestContext _context;

        public Jwt(IConfiguration configuration, IHttpContextAccessor
httpContextAccessor, CourseManagerTestContext context) {
            _config = configuration;
            _httpContextAccessor = httpContextAccessor;
            _context = context;
        }
```

Na početku Jwt klase je definirano da ću koristiti konfiguraciju tj. appsetting.json, treba mi I httpContextAccessor da mogu postaviti i kreirati novi kolačić te kontekst na bazu.

Prvi dio GenerateJwt(TeacherLoginDTO teacher) izgleda ovako:

```
public string GenerateJWT(TeacherLoginDTO teacher) {
            var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
            var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);
            var audience = _config["Jwt:Audience"];
            var issuer = _config["Jwt:Issuer"];
            TimeZoneInfo croatiaTimeZone =
TimeZoneInfo.FindSystemTimeZoneById("Central Europe Standard Time");
            DateTime expiresLocalTime =
TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow,
croatiaTimeZone).AddMinutes(30);

            string username = teacher.Username;
            bool admin = GetUserAdmin(username);
            string adminValue = "nonAdmin";
            if(admin) {
                adminValue = "admin";
            }
```

Ovdje kao i u Program.cs klasi moramo deklarirati izdavača, publiku, tajni ključ te minute do isteka (u ovom slučaju 30 minuta). Nakon toga dohvaćam korisničko ime nastavnika iz DTO-a koji je proslijeđen te dohvaćam da li je nastavnik admin pomoću GetUserAdmin(username) metode. Ta metoda će biti objašnjena kasnije. U slučaju da je korisnik admin onda postavljam vrijednost adminValue varijable na admin.

Sljedeći dio koda ove metode se odnosi kreiranje izjava za token:

```
      var jwt_description = new SecurityTokenDescriptor {
          Subject = new ClaimsIdentity(new[] {new Claim("username",
username),
                                                new Claim("admin",
adminValue)
                                               }),
          Expires = expiresLocalTime,
          Audience = audience,
          Issuer = issuer,
          SigningCredentials = credentials
      };
```

Izjave se deklariraju pomoću korištenja SecurityTokenDescriptor-a. Naš Jwt token će imati samo 2 izjave koje su: korisničko ime nastavnika te admin vrijednost nastavnika. Prilikom stvaranja ovoga tokena proslijeđujemo varijable isteka, publike, izdavača i tajnog ključa.

Sljedeći isječak koda stvara token:

```
            var token = new
JwtSecurityTokenHandler().CreateToken(jwt_description);
            var encryptedToken = new
JwtSecurityTokenHandler().WriteToken(token);
```

Prvo stvaramo novi token pomoću CreateToken() metode te pomoću WriteToken() metode ga serijaliziramo. Varijabla encryptedToken sada ima glavu, sadržaj i potpis.

Te u zadnjem isječku koda samo kreiramo novi kolačić te zapisujemo vrijednost Jwt-a u isti.

```
        _httpContextAccessor.HttpContext.Response.Cookies.Append("token",
encryptedToken,
                new CookieOptions {
                    Expires = expiresLocalTime,
                    HttpOnly = true,
                    Secure = true,
                    IsEssential = true,
                    SameSite = SameSiteMode.None
                });
```

Tablica 5: Objašnjenja opcija za httpOnly kolačić (izvor: [8])

Opcija za kolačić	Opis
Expires	postavlja vrijednost kada će ovaj kolačić prestati vrijediti.
httpOnly	označava da li je kolačić httpOnly u ovom slučaju da.
isEssential	označava da li je kolačić bitan za rad aplikacije. U ovom slučaju da jer korisnik ne može reći da ova aplikacija neće koristiti taj kolačić
SameSite	Može biti: • Lax (SameSiteMode.Lax) – označava da će se kolačići slati sa svim GET zahtjevima koji su napravljeni od neke druge stranice. • Strict (SameSiteMode.Strict) - Opcija koja označava da će se kolačići poslati isključivo na istu stranicu. To znači ako je zahtjev sa nekog drugog URL-a koji nije trenutni da se neće kolačić poslati. Omogućavanje ove opcije povećava sigurnost od CSRF napada. • None (SameSiteMode.None) – Kolačići će se poslati uvijek te CORS je dopušten.
Secure	ako je true onda se kolačić prenosi preko HTTPS protokola. U ovom slučaju je true.

```
            var response = new { token = encryptedToken, username =
teacher.Username };
            return JsonSerializer.Serialize(response);
```

Ovaj dio metode samo vraća serijalizirani token.

Sljedeća metoda je DecodeToken() koja čita sadržaj tokena te pomoću toga vrši dodatnu autorizaciju. To znači da nekim resursima se neće moći pristupiti ako korisnik nije admin.

```
        public List<string> DecodeToken() {
            var cookie =
_httpContextAccessor.HttpContext.Request.Cookies["token"];
            var handler = new JwtSecurityTokenHandler();
            var jwt = handler.ReadToken(cookie);
            var jwtS = jwt as JwtSecurityToken;
```

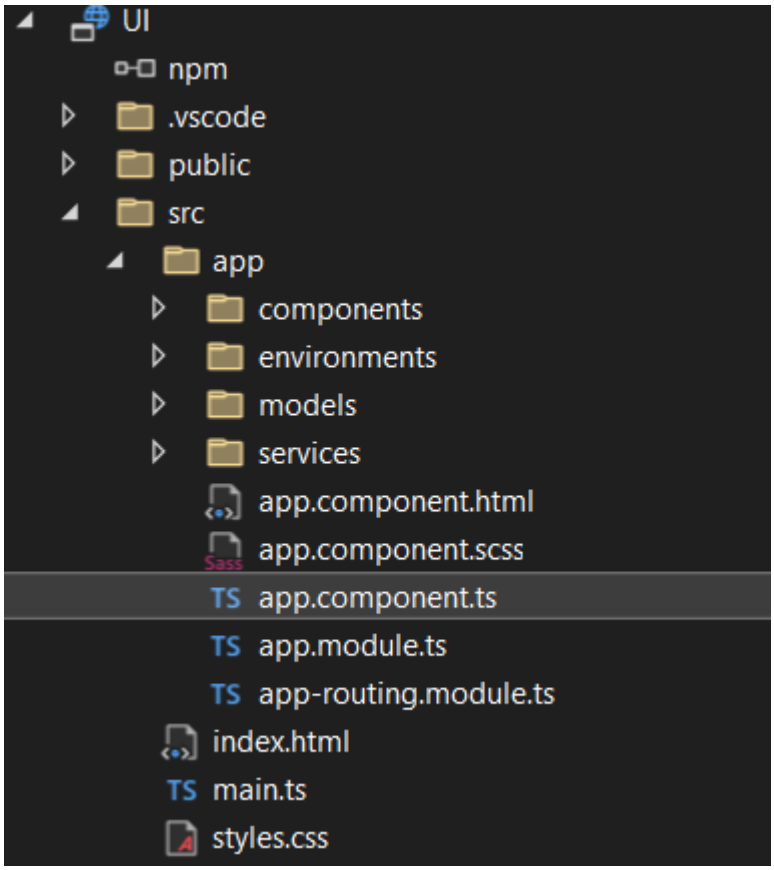
Prvo što se može vidjeti da je metoda DecodeToken() tipa List<string> te će vratiti 2 izjave koje postoje u tokenu. Prvo što se mora napraviti u metodi je dohvatiti kolačić iz HttpContext. Hvatamo kolačić imena token. Nakon uspješnog dohvaćanja kolačića moramo pročitati sadržaj istoga odnosno vrijednost Jwt tokena unutar kolačića. Kreiramo novu varijablu jwtS da možemo kasnije čitati izjave tokena.
Zadnji dio metode samo čita izjave te vraća iste kao listu:

```
            var jti1 = jwtS.Claims.First(claim => claim.Type ==
"username").Value;
            var jti2 = jwtS.Claims.First(claim => claim.Type ==
"admin").Value;
            var listOfClaims = new List<string> {
                jti1,
                jti2
            };
            return listOfClaims;
        }
```

U ovom konkretnom slučaju vraćamo listu sa 2 izjave: korisničko ime i da li je korisnik admin.

5.7. UI (eng. User Interface)

Kao što je već prije navedeno UI sloj se sastoji od Angular-a te prikazivanja podataka API-ja na klijentskoj strani. Kao što sam već rekao Angular programski okvir funkcionira na modelu komponenata. Svaka komponenta se sastoji od svoje typescript datoteke, html datoteke te css datoteke. Kod na UI projektu je raspoređen na sljedeći način:

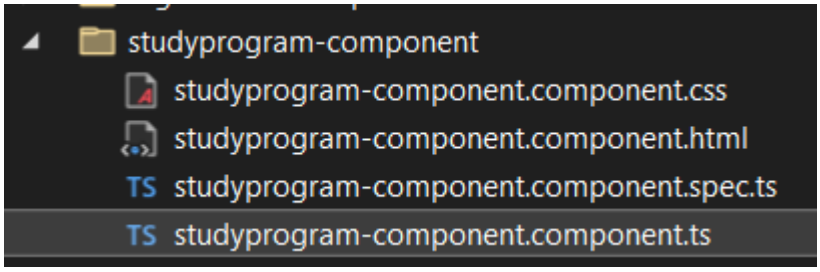


Slika 19: Struktura UI projekta (izvor: slika autora)

Komponente se nalaze u mapi components. Environments mapa je mapa koja samo sadrži environment datotektu u kojoj je definirana URL adresa mog API-ja. U mapi models se nalaze sva sučelja pomoću kojih radim upite na API te u servisima se nalazi sva poslovna logika, odnosno si pozivi na API itd.

5.7.1. Komponente

Na ovom specifičnom primjeru ću objasniti kako funkcionira komponenta za vrste sveučilišnih programa.
Ovako izgleda sadržaj mape studyprograms:



Slika 20: Struktura jedne Angular komponente (izvor: slika autora)

5.7.1.1. HTML

```
<div class="header-container">
  <h1>Study programs</h1>
  <button class="add-button" (click)="toggleAddPopup()">Add a new study
program</button>
</div>

<div *ngIf="showAddPopup" class="popup">
  <form (submit)="addStudyProgram()">
    <label for="shortName">Short name:</label>
    <input type="text" id="shortName" name="shortName"
[(ngModel)]="shortName">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" [(ngModel)]="name">
    <button type="submit">Add</button>
  </form>
</div>

<div *ngIf="showUpdatePopup" class="popup">
  <form (submit)="updateStudyProgram()">
    <label for="shortName">Short name:</label>
    <input type="text" id="shortName" name="shortName"
[(ngModel)]="shortName">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" [(ngModel)]="name">
    <button type="submit">Update</button>
  </form>
</div>
```

Ovdje vidimo da imamo jednostavnu HTML datoteku koja za početak ima naslov Study Programs te gumb pomoću kojega možemo dodati ili ažurirati određeni studijski program. [ngModel] označava samo označava da ovaj input ima dvosmjerno povezivanje na odgovarajuću komponentu. To samo znači da ako se podaci promijene u HTML-u onda će se promijeniti i u typescript datoteci i obrnuto.

```
<table>
  <thead>
    <tr>
      <th>Short name</th>
      <th>Name</th>
      <th>Update</th>
      <th>Delete</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr *ngFor="let studyProgram of studyPrograms">
      <td>{{ studyProgram.shortName }}</td>
      <td>{{ studyProgram.name }}</td>
      <td><button
(click)="populateAndToggleUpdatePopup(studyProgram)">Update</button></td>
      <td><button
(click)="deleteStudyProgram(studyProgram.id)">Delete</button></td>
    </tr>
  </tbody>
</table>
```

Drugi dio HTML datoteke stvara tablicu koja ima stupce Short name, Name te gumbove Update i Delete. *ngFor označava iteraciju sveučilišnog programa u svim sveučilišnim programima te za svaki sveučilišni program u svakom redu prikazuje odgovarajući Short Name i Name. Gumbovi Update i Delete su spojeni na komponentu sa metodama populateAndToggleUpdatePopup() te deleteStudyProgram.

5.7.1.2. Typescript

```
export class StudyprogramComponent implements OnInit {
  studyPrograms: StudyProgram[] = [];
  showAddPopup: boolean = false;
  showUpdatePopup: boolean = false;
  shortName: string = '';
  name: string = '';
  currentStudyProgramId: number | null = null;

  constructor(
    private studyProgramService: StudyprogramServiceService,
    private router: Router
  ) { }
```

Na početku komponente sam naslijedio OnInit te on će biti opisan kasnije. Prvo stvaram nove varijable koje odgovaraju [ngModel] dijelu iz HTML-a. Vidimo da studyPrograms je tipa StudyProgram koje je samo sučelje definirano u mapi models. Mapa models će biti opisana kasnije. Pomoću konstruktora ostvarujemo Dependency Injection.

```
  ngOnInit() {
    this.loadStudyPrograms();
  }

  loadStudyPrograms() {
    this.studyProgramService.getStudyPrograms().subscribe(
      (data: StudyProgram[]) => {
        this.studyPrograms = data;
      },
      error => {
```

```
        console.error("Error fetching study programs", error);
      }
    );
  }
```

ngOnInit je jedna od metoda životnog ciklusa aplikacije. On označava da će kod koji je postavljen unutar ngOnInit biti izvršen čim je izgradnja komponente završena. U ovom primjeru ngOnInit sadži metodu koja učitaje sve sveučilišne programe. Metoda loadStudyPrograms() poziva servis koji poziva API. Servis će biti objašnjen kasnije.

```
  async addStudyProgram() {
    const newStudyProgram = {
      shortName: this.shortName,
      name: this.name
    };
    try {
      const response = await
this.studyProgramService.createStudyProgram(newStudyProgram);
      if (response) {
        console.log("Added new study program");
        this.loadStudyPrograms();
      }
    } catch (error) {
      console.error("Error during creation", error);
    }
    this.showAddPopup = false;
  }

 async deleteStudyProgram(id: number) {
   try {
     const response = await
this.studyProgramService.deleteStudyProgramById(id);
     this.loadStudyPrograms();
   } catch (error) {
     console.error('Error deleting study program:', error);
   }
 }
```

Metode za stvaranje novog sveučilišnog programa te brisanje sveučilišnog programa isto pozivaju servis.

5.7.1.3. Servis

Servisi u kontekstu mog završnog rada sadržavaju logiku za pozivanje API metoda preko njihovih URL-ova.

```
export class StudyprogramService {
  private restUrl = `https://${environment.restAPI}/StudyProgram`;

  constructor(private http: HttpClient) { }


  getStudyPrograms(): Observable<StudyProgram[]> {
    const options = { withCredentials: true };
    return this.http.get<StudyProgram[]>(this.restUrl, options);
  }
```

Prvo što se događa u kodu je pozivanje URL REST putanje pomoću environment-a. Svaki servis mora inicijalizirati HttpClient jer će koristiti HTTP metode. U primjeru metode getStudyPrograms() koja je tip StudyProgram šaljemo GET upit na REST API te ga šaljemo sa opcijama koje su withCredentials = true. withCredentials označava da prilikom svakog slanja šaljemo u kolačiće u glavi zahtjeva. Kao što je i prije opisano šaljemo samo jedan kolačić koji u sebi sadržava Jwt token. Ova metoda vraća odgovor API-ja.

Sljedeća metoda kreira novu sveučilišnu godinu:

```
async createStudyProgram(newStudyProgram: { shortName: string; name:
string; }) {
   const options = { withCredentials: true };
   const response = await this.http.post(this.restUrl, newStudyProgram, {
observe: 'response', ...options }).toPromise();
   return response;
 }
```

Kao što se već vidi metoda za dohvaćanje svih sveučilišnih godina i ova metoda su praktično identične. Jedina razlika je što jedna koristi Observable a druga toPromise().

Razlika Observable sa toPromise

Glavna razlika je to što Observable je dio RxJS-a (eng. Reactive Extensions for Javascript) te on može vratiti jednu vrijednost, nijednu ili više te može raditi sa nizovima podataka pomoću .map ili .filter funkcija. S druge strane toPromise() uvijek vraća jednu vrijednost ili grešku te nema podršku za izvršavanje operacija koje manipuliraju sa podacima kao što Observable ima. Zbog ovih razloga koristim Observable kod GET metoda a toPromise() kod svih drugih jer jedino GET metoda vraća listu podataka.

5.7.1.4. Modeli

Modeli u Angular-u služe kao predložak podataka koje primamo/šaljemo. Na njih možemo gledati kao preslika očekivanog JSON formata od API-ja.

```
export interface StudyProgram {
  id: number;
  shortName: string;
  name: string;
}
```

Zaključak

U ovom završnom radu obradio sam najvažnije aspekte razvoja web aplikacija koristeći ASP.NET Core. Kroz uvodni dio opisana je povijest i osnovni koncepti ASP.NET Core, uključujući ASP.NET MVC i API razvoj. Također smo se osvrnuli na tehnologije i alate neophodne za razvoj modernih web aplikacija, kao što su Entity Framework Core, LINQ, te sigurnosne protokole poput JWT i OAuth 2.0.

Jedan od glavnih ciljeva rada bio je pružiti detaljan pregled razvoja RESTful API-ja koristeći ASP.NET Core. Dodatno, integracija s Angular-om omogućila je demonstraciju kako klijentske tehnologije mogu učinkovito komunicirati s poslužiteljskom stranom aplikacije. Cijela integracija programskog rješenja provedena je uporabom višeslojne (eng. nlayer) arhitekture. Implementacija je provedena kroz pet slojeva: UI (eng. *User Interface*), API (eng. Application Programming Interface), BAL (eng. Business Logic Layer), DAL (eng. Data Access Layer) te DTO (eng. Data Transfer Object). UI sloj je napravljen kao samostalan Angular projekt, API je ostvaren kroz ASP.NET Core web API tehnologiju, dok su BAL, DAL i DTO slojevi ostvareni kao Class Library projekti.

Praktična implementacija ovog završnog rada omogućila mi je primjenu teorijskih znanja na stvarnom projektu. U praktičnom dijelu prikazani su procesi autentifikacije i autorizacije te implementacija JWT-a i kolačića. Pomoću ovih procesa osigurana je sigurnost aplikacije protiv napada.

Objašnjena je i višeslojna arhitektura te obrazloženo zašto je odabrana kao preferirani način izgradnje ove web aplikacije. Na samom kraju rada prikazani su dijagrami slijeda koji prikazuju tok (eng. flow) aplikacije. Dijagrami slijeda obrađeni su na primjerima prijave i stvaranja novog sveučilišnog programa.

U konačnici, ASP.NET Core programski okvir se pokazao kao fleksibilan, snažan i moderan alat za razvoj suvremenih web aplikacija. Mogućnost integracije s nekim od najpopularnijih klijentskih programskih okvira te jednostavna i intuitivna sintaksa samo su neki od razloga zašto smatram da je ASP.NET Core jedan od najboljih programskih okvira za izgradnju modernih web aplikacija.

Popis literature

[1] Jeslur Rahman, „ Understand the Web/REST API: Asp.Net Core Web API in C# “, 2024.[Na internetu]. Dostupno na: https://medium.com/@jeslurrahman/understand-the-web-rest-api-asp-net-core-web-api-in-c-8236e2bcb0f1. [Pristupljeno 11-svibnja-2024]

[2] Chinmayee Deshpande, „ Introduction To Angular Service and Its Features“, 2024.[Na internetu]. Dostupno na: https://www.simplilearn.com/tutorials/angular-tutorial/angular-service [Pristupljeno 19-svibnja-2024]

[3] Entity Framework Tutorial, „ How Entity Framework Works?“, 2024.[Na internetu]. Dostupno na: https://www.entityframeworktutorial.net/entityframework6/how-entity-framework-works.aspx [Pristupljeno 14-svibnja-2024]

[4] Extio Technology, „ Understanding JSON Web Tokens (JWT): A Secure Approach to Web Authentication 2023.[Na internetu]. Dostupno na: https://medium.com/@extio/understanding-json-web-tokens-jwt-a-secure-approach-to-web-authentication-f551e8d66deb [Pristupljeno 19-svibnja-2024]

[5] miniOrange, „ What is JWT (JSON Web Token)? How does JWT Authentication work? 2023.[Na internetu]. Dostupno na: https://www.miniorange.com/blog/what-is-jwt-json-web-token-how-does-jwt-authentication-work/ [Pristupljeno 19-svibnja-2024]

[6] Akash Sharma, „ Understanding Request And Response Model: A General Overview ” 2023.[Na internetu]. Dostupno na: https://medium.com/@imakashsharma135/understanding-request-and-response-model-a-general-overview-831c24d2288 [Pristupljeno 25-svibnja-2024]

[7] Lokesh Gupta, „ HTTP Status Codes “ 2023.[Na internetu]. Dostupno na: https://restfulapi.net/http-status-codes/ [Pristupljeno 1-lipnja-2024]

[8] mdn Web Docs „ Using HTTP cookies “ 2024.[Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies [Pristupljeno 23-svibnja-2024]

[9] Marinko Spasojevic, Vladimir Pecanac „ Ultimate ASP.NET Core web API “ 2019.[Na internetu]. Dostupno na: https://dl.ebooksworld.ir/books/Ultimate.ASP.NET.Core.Web.API.Marinko.Spasojevic.Vladimir.Pecanac.CodeMaze.EBooksWorld.ir.pdf [Pristupljeno 23-svibnja-2024]

[10] Lokesh Gupta „ What is REST? “ 2023.[Na internetu]. Dostupno na: https://restfulapi.net/ Pristupljeno 29-svibnja-2024]

[11] ByteScout „ ASP.NET: History, Purpose, Versions“ 2023.[Na internetu]. https://bytescout.com/blog/2014/01/aspnet-history-purpose-versions.html [Pristupljeno 1-svibnja-2024]

[12] Orçun Yılmaz „ The DTO Pattern (Data Transfer Objects) “ 2023.[Na internetu]. https://medium.com/@orcunyilmazoy/the-dto-pattern-data-transfer-objects-8146b262636e [Pristupljeno 8-svibnja-2024]

[13] Dan Arias „ Hashing in Action: Understanding bcrypt “ 2021 .[Na internetu]. https://auth0.com/blog/hashing-in-action-understanding-bcrypt/ [Pristupljeno 13-svibnja-2024]

[14] Acunetix „ CSRF Attacks: Anatomy, Prevention, and XSRF Tokens “ 2022 .[Na internetu]. https://www.acunetix.com/websitesecurity/csrf-attacks/ [Pristupljeno 20-svibnja-2024]

[15] PortSwigger „ Cross-site scripting “ 2021 .[Na internetu]. https://portswigger.net/web-security/cross-site-scripting Pristupljeno 20-svibnja-2024]

[16] IBM „ Sequence diagrams “ 2021 .[Na internetu]. https://www.ibm.com/docs/en/rsas/7.5.0?topic=uml-sequence-diagrams [Pristupljeno 24-svibnja-2024]

[17] Mehmet Ozkaya „ Layered (N-Layer) Architecture“ 2021 .[Na internetu]. https://medium.com/design-microservices-architecture-with-patterns/layered-n-layer-architecture-e15ffdb7fa42 [Pristupljeno 22-svibnja-2024]

[18] Jaysri Saravanan „ What is Component in Angular? “ 2023 .[Na internetu]. https://medium.com/@jaysrisaravanan/what-is-component-in-angular-344762207a86 [Pristupljeno 29-svibnja-2024]

[19] Zanfina Svirca „ Everything you need to know about MVC architecture“ 2020 .[Na internetu]. https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1 [Pristupljeno 23-svibnja-2024]

Popis slika

Popis tablica