# SchedulingApp – documentation

## Osnovno o aplikaciji

SchedulingApp web aplikacija predstavlja platformu za rezervaciju termina u sportskim objektima koji su oglaseni unutar aplikacije.Njena osnovna svrha jeste pruzanje mogucnosti korisnicima da lako, u svega par klika rezervisu termin koji zele u objektu koji takodje zele, da lako provere dostupnost termina, otkazu termin, izvrse placanje termina i budu obavesteni ukoliko je iz nekog razloga termin otkazan. Aplikacija podrzava **osnovne CRUD operacije (Create, Read, Update i Delete),** a osim toga podzane su i uloge **administratora** i **customera.** S tim u vezi, postoje i odredjene privilegije, odnosno ogranicenja po pitanju pristupa i mogucnosti

* **Administrator** ima mogucnost: **pristupa sadrzaju stranice, manipulaciju nad podacima** (sto ukljucuje dodavanje novog sportskog objekta, azuriranje termina, brisanje termina i objekata iz baze podataka). Takodje, Administrator ima uvid u sve rezervacije koje su korisnici izvrsili, pa shodno tome moze da izvrsi otkazivanje rezervacije ukoliko je to potrebno iz nekog razloga.
* **Customer (Korisnik):** ima mogucnost pretrage i filtriranja sportskih objekata I termina, rezervisanje termina, otkazivanje sopstvenih rezervacija, kao I vrsenje placanja putem integrisanih metoda.

**Placanje termina** implementirano je putem **Stripe** platforme, koja obuhvata **karticno placanje** na siguran nacin. Kada korisnik odabere termin koji zeli da rezervise, sistem ga void kroz process placanja gde korisnik unosi svoje podatke. Stripe obezbedjuje sigurnu komunikaciju i zastitu podataka koriscenjem enkripcije, a aplikacija dobija potvrdu o uspesnosti transakcije. Na osnovu uspesno izvrsenog placanja, korisniku se potvrdjuje rezervacija termina.

**Filtriranje termina i objekata** je omoguceno kako bi korisnici lakse pronasli zeljeni sportski objekat I termin. Aplikacija podrzava I napredne opcije filtriranja I pretrage, gde dostupni filteri obuhvataju:

* **Lokaciju objekata:** korisnik moze pretrazivati objekte po lokaciji
* **Datum termina:** omogucava selekciju termina za odredjeni dan

## Tehnologije koriscene u razvoju i arhitektura sistema

Aplikacija je razvijena koriscenjem sledecih tehnologija:

* **ASP .NET Web API (.NET 9):** koriscen je za implementaciju **backend** dela aplikacije. Web API obezbedjuje **RESTful** servis koji omogucava kreiranje, citanje, azuriranje i brisanje podataka iz baze (MSSQL). Takodje, impelementirane su i logika za autentifikaciju i autorizaciju korisnika (JWT), upravljanje rezervacijama i integracija sa eksternim servisima poput Stripe-a.
* **React:** koriscen za implementaciju **frontend** dela aplikacije. React omogucava brzu I dinamicnu izradu UI-a, gde je korisnicima omoguceno da u real – time-u pretrazuju I filtriraju objekte, rezervisu termine I izvrse placanje. Koriscen je takodje I **Redux Toolkit Query** za upravljanje stanjem I asinhronu komunikaciju sa Web API – jem.
* **MSSQL:** baza podataka u kojoj se cuvaju informacije o korisnicima, sportskim objektima, terminima i rezervacijama
* **Stripe:** integrisan za karticna placanja
* **Cloudinary:** koriscen za skladistenje slika sportskih objekata, profilnih slika korisnika i svih ostalih multimedijalnih fajlova

Prednosti ovakvog pristupa:

1. **Razdvojen backend I frontend:** omogucava nezavisno razvijanje I odrzavanje sistema. Bacnekd servis moze lako da se koristi I od strane drugih klijenata (npr. mobilne aplikacije), dok frontend ostaje fokusiran na korisnicko iskustvo.
2. **Skalabilnost:** koriscenjem ASP .NET Web API – ja I React – a omoguceno je da aplikacije lako raste I da se dodaju nove funkcionalnosti.
3. **Sigurnost:** autentifikacija I autorizacija su implementirani pomocu JWT tokena, a placanja se obavljaju putem Stripe – a koji obezbedjuje PCI DSS standard (**Payment Card Industry Data Security Standard** je bezbednosni standard koji definiše zahteve za bezbedno rukovanje podacima o platnim karticama). I sigurnu enkripciju podataka.
4. **Korisnicko iskustvo:** React omogucava responzivni dizajn I brz interfejs, dok filteri po lokaciji, datumu znatno olaksavaju process rezervacije.

**Arhitektura sistema**

Arhitektura **SchedulingApp** aplikacije zasnovana je na principu razdvojenih slojeva (**frontend**, **backend**, **baza podataka**) i integracije sa eksternim servisima.

**Pregled komponenti:**

1. **Frontend (React)**
   * Izgrađen u React-u.
   * Komunicira sa backend servisom preko REST API poziva.
   * Koristi **Redux Toolkit Query** za state management i asinhronu komunikaciju.
   * Omogućava korisnicima pretragu i filtriranje sportskih objekata (po lokaciji, datumu, sportu itd.), rezervaciju termina i online plaćanje.
2. **Backend (ASP.NET Web API – .NET 9)**
   * Implementiran u ASP.NET Web API.
   * Obavlja poslovnu logiku aplikacije: autentifikaciju i autorizaciju (JWT tokeni), CRUD operacije nad sportskim objektima i terminima, validaciju rezervacija.
   * Pruža RESTful endpoint-e za frontend.
   * Povezan je sa eksternim servisima: **Stripe** (plaćanja) i **Cloudinary** (upravljanje slikama).
3. **Baza podataka (MSSQL)**
   * Čuva sve podatke: korisnike, sportske objekte, termine, rezervacije, uloge.
   * Entity Framework Core koristi se za ORM (mapiranje objekata na relacije).
   * Migracije se koriste za verzionisanje baze.
4. **Eksterni servisi**
   * **Stripe**: obezbeđuje sigurno procesiranje kartičnih plaćanja. Backend komunicira sa Stripe API-jem, dok frontend prikazuje korisnički interfejs za unos kartičnih podataka.
   * **Cloudinary**: koristi se za čuvanje i distribuciju slika sportskih objekata. Backend se povezuje sa Cloudinary servisom za upload, dok frontend prikazuje slike.

A screenshot of a computer program

AI-generated content may be incorrect.

# Objasnjenje Backend dela

## Osnovno o backendu (RESTful servisima, itd)

**Backend** predstavlja “srce” sistema jer sadrzi **poslovnu logiku** I obezbedjuje **komunikaciju** izmedju klijentske aplikacije I baze podataka. On je razvijen u ASP .NET Core Web API (.NET 9) I organizovan je prema principima **ciste arhitekture** i **separacije slojeva.**

Glavne karakteristike backend – a:

1. **ASP .NET Core Web API:**
   * Implementira **RESTful servise** koji izlazu resurse (korisnici, sportski objekti, termini, rezervacije, korpa, itd)
   * Svaki endpoint ima jasno definisan **controller** i **metodu** koja obradjuje HTTP zahtev
2. **Entity Framework Core (ORM sloj):**
   * Omogucava rad sa bazom podataka kroz **model klase (entitete),** bez potrebe za rucnim pisanjem SQL upita
   * Koristi **migracije** za kreiranje I azuriranje seme baze podataka
   * Povezuje se sa **MSSQL Serverom,** gde se cuvaju svi podaci
3. **Autentifikacija I autorizacija**
   * Implementirana koriscenjem **ASP .NET Identity** i **JWT tokena**
   * Korisnici imaju uloge: **Admin** i **Customer**
   * Administrator ima dodatne privilegije (dodavanje, izmena, brisanje podataka, pregled svih rezervacija)
4. **Placanje (Stripe integracija)**
   * Backend komunicira sa **Stripe API – jem** za obradu transakcija
   * Sigurnost je obezbedjena jer se karticni podaci ne cuvaju u aplikaciji vec se prosledjuju Stripe servisu
   * Backend validira transakciju i azurira status rezervacije nakon uspesne uplate
5. **Cloudinary integracija:**
   * Koristi se za cuvanje slika sportskih objekata, kao I slika korisnika
   * Backend sadrzi servis za upload i cuvanje URL adresa koje se kasnije prikazuju na frontend – u
6. **Sigurnost i validacija:**
   * Koriscenjem **ModelState validacije** prilikom obrade ulaznih podataka
   * **Global error handling** I vracanje HTTP status kodova (200, 201, 400, 404, 500…)
   * Zasticeni endpoint – i (npr. rezervacija, placanje) zahtevaju **validan JWT token**

## RESTful Servisi

**RESTful servisi** predstavljaju osnovu za komunikaciju između klijentske aplikacije (frontend) i servera (backend). U ovoj aplikaciji, **ASP .NET Core Web API** koristi se za implementaciju REST arhitekture. RESTful API koristi standardne HTTP metode za rad sa resursima:

* **GET:** Dobavljanje podataka
* **POST:** Kreiranje novih podataka
* **PUT:** Ažuriranje postojećih podataka
* **DELETE:** Brisanje podataka

Ključne RESTful karakteristike projekta:

* **Resursi su jasno definisani:** Svaka ruta predstavlja jedan entitet
* **Stateless komunikacija:** Svaki HTTP zahtev je nezavistan – ne čuva se stanje između poziva
* **JSON format odgovora:** Podaci se razmenjuju u JSON formatu, što omogućava lako povezivanje sa frontend aplikacijama.
* **Autentifikacija i autorizacija:** RESTful API koristi JWT (JSON Web Token) za autentifikaciju korisnika i zaštitu ruta.
* **Korišćenje odgovarajućih HTTP kodova**

## Konfiguracioni fajlovi

Unutar aplikacije imam sledece konfiguracione fajlove:

1. **launchSettings.json:** definise profil pokretanja aplikacije tokom razvoja. Bitne stavke unutar njega su:
   * **applicationUrl:** portovi na kojima backend radi
   * **ASPNETCORE\_ENVIRONMENT:** odredjuje okruzenje (Development, Starting, Production)
   * **launchBrowser:** automatsko otvaranje Swagger UI u browser
2. **appsettings.json:** fajl koji cuva konfiguraciju aplikacije. Bitne stavke u njemu su:
   * **ConnectionStrings:** string za povezivanje sa MSSQL bazom podataka
   * **ApiSettings.Secret:** kljuc za potpisivanje JWT tokena
   * **CloudinarySettings:** podaci za Cloudinary integraciju
   * **Logging:** nivo logovanja

## Middleware i pipeline

* **Middleware** je niz komponenti kroz koje svaki HTTP zahtev prolazi. Redosled registracije middleware – a (odnosno **pipeline**) u Program.cs je bitan – menja ponasanje i sigurnost aplikacije. Svi middleware-i se povezuju u tzv. **pipeline** (lanac izvršenja). Svaki zahtev prolazi redom kroz middleware komponente, a svaka komponenta odlučuje da li da:
  + **prosledi zahtev sledećem middleware-u**, ili
  + **završi obradu i vrati odgovor**.
* **Middleware** je komponenta koja ucestvuje u obradi HTTP zahteva i odgovora. Svaka middleware komponenta moze da uradi nesto pre sledece komponente (loguje, validira token, itd). Moze da prosledi zahtev sledecoj komponenti u lancu ili da presence i odmah vrati odgovor. Svi zahtevi u middleware lancu prolaze kroz **pipeline** (predstavlja **lanac middleware – a**).
* **Pipeline** funkcionise kao niz delegata koji se pozivaju jedan za drugim. Redosled middleware -a je bitan jer se pipeline izvrsava redosledom kojim je middleware registrovan. Svaki HTTP zahtev koji ulazi u aplikaciju prolazi kroz pipeline, a odgovor se vraća kroz isti pipeline.
* *Kako Middleware radi?*

1. ***Ulazak u Pipeline:***

Kada HTTP zahtev dođe do servera, ulazi u pipeline i prolazi kroz niz middleware komponenti.

1. ***Sekvencijalno izvršavanje:***

Middleware komponente se izvršavaju redom kojim su dodate u pipeline-u. Svaka komponenta može odlučiti da: 1. Obradi zahtev i zaustavi dalji prolazak kroz pipeline i 2. Prosledi zahtev sledećoj komponenti u nizu

1. **Izlazak iz pipeline:**

Nakon što zahtev prođe kroz sve middleware komponente, dolazi do krajnje tačke (obično kontroler ili krajnja tačka koja vraća odgovor). Odlazni odgovor prolazi nazad kroz pipeline, gde svaka middleware komponenta može da izvrši dodatnu obradu odgovora pre nego što se on pošalje klijentu

* Middleware i Pipeline komponente u Program.cs

1. **app.UseSwagger i app.UseSwaggerUI**

Uključene su samo ako je aplikacija u razvojnom okruženju (Development)

**UseSwagger** omogućava generisanje Swagger specifikacije za API

**UseSwaggerUI** omogućava interfejs za pregled i testiranje API-ja putem Swagger UI-a

1. **app.UseHttpsRedirection**

Preusmerava sve HTTP zahteva na HTTPS

Povećava sigurnost aplikacije osiguravanjem da se koristi sigurni protocol

1. **app.UseCors**

Konfiguriše CORS (Cross – Origin Resource Sharing) politike koje omogućavaju ili ograničavaju zahteve sa drugih domena. U ovom slučaju omogućava sve zahteve (sva zaglavlja, metode i izvore).

1. **app.UseAuthentication**

Middleware komponenta za autentifikaciju koja obrađuje JWT tokene u zahtevu. Proverava JWT token i postavlja korisnički identitet ako je token validan.

1. **app.UseAuthorization**

Middleware komponenta za autorizaciju koja obezbeđuje da samo ovlašćeni korisnici mogu pristupiti određenim resursima. Proverava pravila autorizacije i uloge korisnika.

1. **app.MapControllers**

Mapira kontrolere kao krajnje tačke za obradu zahteva. Obezbeđuje da se zahtevi usmeravaju na odgovarajuće kontrolere i akcije.

* **Kako Middleware Radi u Pipeline-u**

1. **Zahtev dolazi na server:** Kada klijent (npr. web pretraživač) pošalje HTTP zahtev, zahtev ulazi u pipeline.
2. **Redirekcija na HTTPS:** app.UseHttpsRedirection proverava da li je zahtev na HTTPS, ako nije, preusmerava ga.
3. **CORS Pravila:** app.UseCors omogućava ili blokira zahteve sa različitih domena na osnovu postavljenih pravila.
4. **Autentifikacija:** app.UseAuthentication proverava JWT token u zaglavlju zahteva. Ako je token validan, identitet korisnika je postavljen.
5. **Autorizacija:** app.UseAuthorization proverava da li korisnik ima dozvolu za pristup traženom resursu.
6. **Mapiranje Kontrolera:** app.MapControllers usmerava zahtev na odgovarajući kontroler i akciju na osnovu URL-a.
7. **Swagger UI (samo u razvojnom okruženju):** Ako je aplikacija u razvojnom okruženju, app.UseSwagger i app.UseSwaggerUI omogućavaju dokumentaciju i testiranje API-ja putem Swagger UI-a.

**Odgovor:** Nakon što zahtev prođe kroz sve relevantne middleware komponente i stigne do odgovarajućeg kontrolera, obrađuje se i vraća odgovor klijentu.

## Dependency Injection

**Dependency Injection (DI)** je dizajn pattern koja omogucava objektima da **dobiju svoje zavisnosti (druge objekte) od spoljnog izvora,** umesto da ih sami kreiraju. Drugim recima, **DI** je nacin da **klase ne prave same svoje zavisnosti** (new), vec da **im te zavisnosti framework sam ubaci (injectuje).** To znaci da umesto da neka klasa direktno kreira objekte koje koristi, ona samo kaze “treba mi servis X”, a DI kontejner (ugradjen u ASP .NET Core) ce joj ga obezbediti.

Vrste lifetime – a u DI:

1. **Transient (AddTransient):**
   * Nova instanca servisa svaki put kada se zatrazi
   * Primer: lagane i stateless operacije (npr. helper servisi)
2. **Scoped (AddScoped):**
   * Jedna instanca po HTTP request
   * Primer: rad sa DbContext – svaka HTTP request ima svoj kontekst
3. **Singleton (AddSingleton):**
   * Jedna instanca za celu aplikaciju
   * Svi koji ga koriste dele isti Singleton
   * Primer: konfiguracija, cache, servisi koji ne zavise od request - a

Osnovni koncepti:

1. **Zavisnost:** To je objekat za koji klasa treba da funkcionise. Npr, ako klasa “Car” koristi klasu “Engine”, tada je “Engine” zavisnost za “Car”
2. **Inject (Injektovanje):** Proces pruzanja zavisnosti klasi. Moze se vrsiti putem: **konstruktora, set – era** ili **interfejsa.**

Vrste Dependency Injection -a:

1. **Constructor Injection: (najcesce koriscen)**

Zavisnosti se ubrizgavaju kroz konstruktor objekta. Ovo osigurava da su sve zavisnosti objekta dostupne u trenutku njegovog stvaranja.

1. **Property Injection:**

Omogucava ubrizgavanje zavisnosti kroz svojstva (properties) objekta. To omogucava da se zavisnosti postave nakon sto je objekat kreiran.

1. **Method Injection:**

Omogucava ubrizgavanje zavisnosti kroz metode. Moze se koristiti kada je odredjena zavisnost potrebna samo za odredjenu metodu.

1. **Interface Injection:**

Zahteva da klasa implementira interfejs koji omogucava ubrizgavanje zavisnosti. Ovde objekat mora pruziti metodu za postavljanje zavisnosti definisanih u interfejsu.

### 7.1 Primeri Dependency Injection – a u aplikaciji

**a) Constructor Injection u Auth Controller**

**A screen shot of a computer code

AI-generated content may be incorrect.**

Ovde ASP .NET Core automatski ubacuje:

* **ApplicationDbContexts:** registrovan je sa builder.services.AddDbContext<>()
* **Iconfiguration:** ugradjen servis framework – a
* **UserManager<ApplicationUser>** i **RoleManager<IdentityRole>:** registruje ih AddIdentity
* **CloudinaryService:** registruje se rucno sa builder.Services.AddSingleton<CloudinaryService>()

**b) Registracija servisa u Program.cs**

**A screen shot of a computer

AI-generated content may be incorrect.**

### 7.2 Dijagram toga Dependency Injection – a

A screenshot of a computer program

AI-generated content may be incorrect.

* **Registracija** (builder.Services...): specificiram kako i sa kojim lifetime – om servis zivi
* **DI Container:** cuva sve registrovane servise
* **Konstruktor kontrolera:** ASP .NET Core automatski ubacuje zavisnosti
* **Action metode:** koristim vec ubacene instance

## Projektni Obrasci u aplikaciji

U okviru aplikacije koristim sledece projektne obrasce (design patterns):

1. **Dependency Injection:**

 builder.Services.AddSingleton<CloudinaryService>();

 builder.Services.AddDbContext<ApplicationDbContexts>(...)

 builder.Services.AddIdentity<ApplicationUser, IdentityRole>()...

* ASP .NET DI kontejner registruje service sa odredjenim zivotnim vekom (Scoped, Singleton, Transient). Kontroleri, servisi ili hosted servisi ih dobijaju automatski kroz **constructor injection.** Koristim DI kroz sve kontrolere.

1. **Factory Pattern:** kreira kompleksne objekte bez da eksplicitno pozivam new

 WebApplication.CreateBuilder(args) → kreira *builder* koristeći **factory method**.

 AddDbContext<ApplicationDbContexts> – kreira instancu *DbContext-a* pomoću *factory-ja* koji konfiguriše EF Core.

1. **Options Pattern:**

* builder.Services.Configure<CloudinarySettings>(builder.Configuration.GetSection("CloudinarySettings"));
* Konfiguracija (npr. Cloudinary API kljucevi) se mapira na strongly typed klase (CloudinarySettings) koje se kasnije ubacuju kroz DI.

1. **Middleware (Chain of Responsibility Pattern):**

 app.UseHttpsRedirection();

 app.UseCors("AllowFrontend");

 app.UseAuthentication();

 app.UseAuthorization();

* Svaki middleware odlucuje da li ce:
  + Obraditi zahtev I zavrsiti pipeline
  + Proslediti zahtev sledecem middleware – u.

1. **Strategy Pattern:** moze se menjati algoritam validacije bez menjanja koda koji ga koristi

 AddAutentication().AddJwtBearer(…):JWT validacija za proveru tokena.

 **TokenValidationParameters:** definise razlicite strategije validacije (ValidateIssuer, ValidateAudience, IssuerSigninKey)

1. **Observer / Publisher – Subscribe**

** builder.Services.AddHostedService<TerminStatusUpdateService>();** -> background service reaguje na dogadjaje u aplikaciji (periodicni task ili event – driven)

** Identity:** sistem interno koriste event – e (npr. kada se korisnik registruje, loguje, logout-uje).

1. **Repository & Unit of Work (implicitno kroz EF Core):**

* **ApplicationDbContexts:** predstavlja Unit of Work
* **DbSet<Entity>:** unutar njega funkcionise kao Repository za svaku tabelu

1. **Builder Pattern**

* **WebApplication.CreateBuilder(args) +** kasnije **builder.Build()**
* Kreira se kompleksni objekat WebApplication korak po korak (dodaju se servisi, konfiguracija, middleware).

## Povezivanje sa bazom podataka i struktura baze

Za bazu podataka koriscen je **MSSQL**. Aplikacija je povezana sa SQL Server bazom podataka koja se zove SchedulingApp\_API. Unutar fajla **appsettings.json,** sekcija **ConnectionStrings** sadrzi konekcioni string za bazu podataka.

### 9.1 Kako je aplikacija povezana za bazom?

* Unutar fajla **appsettings.json** nalazi se konekcioni string **"DefaultDbConnection": "Server=.;Database=SchedulingApp\_API;...:** to ukazuje da se koristi SQL Server.

"ConnectionStrings": {

"DefaultDbConnection": "Server=.;Database=SchedulingApp\_API;TrustServerCertificate=True;Trusted\_Connection=True;"

},

DefaultDbConnection specificira **lokalni SQL Server (**"Server=.;), bazu pod imenom **SchedulingApp\_API,** I koristi Trusted\_Connection=True;" sto znaci da koristi integrisanu Windows autentifikaciju.

* Aplikacija je povezana sa bazom podataka koristeći **Entity Framework Core** koji koristi konekcione stringove iz appsettings.json za uspostavljanje veze sa bazom podataka. Projekat koristi **Entity Framework Core** kao **ORM (Object – Relational – Mapping)** alat za rad sa bazom podataka. EF Core omogućava da se baze i tabele definišu koristeći C# klase umesto ručnog pisanja SQL upita.

A screen shot of a computer code

AI-generated content may be incorrect.

* Unutar Program.cs sam registrovao **ApplicationDbContexts** kao **EF Core DbContext** i to predstavlja **middleware pipeline** u ASP .NET Core aplikaciji, omogucavajuci aplikaciji da komunicira sa SQL Server bazom podataka. Odnosno EF Core koristi taj connection string za sve DB operacije koje radi kroz ApplicationDbContext.
* **builder.Configuration.GetConnectionString("DefaultDbConnection")** preuzima string za povezivanje sa bazom podataka iz konfiguracionog fajla **appsettings.json.**
* **builder.Services.AddDbContext<ApplicationDbContexts>()** registruje ApplicationDbContexts klasu kao **DbContext** u Dependency Injection kontejneru. Svaku put kada aplikaciji treba pristup bazi podataka, ASP .NET Core će koristiti ovu instancu.
* **options.UseSqlServer(connectionString)** : povezuje EF Core sa SQL Serverom koristeći dati connection string. Ovo omogućava aplikaciji da koristi LINQ upite I EF Core migracije za interakciju sa bazom.
* **builder.Services.AddIdentity<ApplicationUser, IdentityRole>().AddEntityFrameworkStores<ApplicationDbContexts>():** ovim sam registrovao Identity. Ovo govori Identity sistemu da koristi moj DbContext za cuvanje korisnika/rola, itd. **Identity** predstavlja gotov sistem za Autentifikaciju i Autorizaciju.

### 9.2 Uloga ApplicationDbContext (DbContext)?

**DbContext** predstavlja sesiju sa bazom podataka i omogucava upite i cuvanje podataka. On je glavni posrednik ili „most“ izmedju C# objekata i baze podataka, tj. tabela. Omogucava:

1. **Mapiranje entiteta u tabele:**
   * Na osnovu DbSet<T> svojstava i klasa (Termin, SportskiObjekat, itd), EF Core gradi **model** (metapodatke) koji opisuje tabele, kolone, kljuceve i veze.
   * Atributi ( [Key], [ForeignKey], [Required], [NotMapped] ) u OnModelCreating uticu na mapiranje.
2. **Izvrsavanje upita i kreiranje IQueryable:**
   * DbSet<T> predstavlja kolekciju entiteta i omogucava LINQ upite
   * Upiti su transformisani u SQL od strane provajdera (UseSqlServer) tek kada ih enumeriram (ToListAsync(), FirstOrDefaultAsync()...)
3. **Pracenje promena (Change Tracking):**
   * DbContext prati stanje entiteta koje ucita ili mu se pridruze
   * **ChangeTracker:** komponenta DbContext – a koja prati entitete koje sam ucitao ili pridruzio kontekstu i njihovo stanje. Kljucne stvari: **Detached** (entitet nije pracen); **Unchanged** (ucitan, nije promenjen); **Added** (novo dodan INSERT); **Modified** (izmenjen UPDATE); **Deleted** (obelezen za brisanje DELETE).Radi tako sto kada **Find, FirstOrDefault, ToList** ili **Attach** vrati entitet, DbContext ga registruje u trackeru. Promene na svojsvtima detektuju se automatski (DetectChanges) ili eksplicitno pozivom ChangeTracker.DetectChanges(). Na SaveChanges() EF Core prelazi kroz tracker, pravi SQL naredbe i izvrsava ih u transakciji.
4. **Persistencije promena: SaveChanges() / SaveChangesAsync()**
   * Grupisane promene (insert/update/delete) Tracker pretvara u SQL instrukcije i izvrsava ih u jednoj transakciji (podrazumevano EF otvara i zatvara transakciju sa SaveChanges).
5. **Konfiguracija modela:**
   * **OnModelCreating(ModelBuilder builder):** omogucava preciznu konfiguraciju veza, indeksa, ogranicenja, mapiranja vrednosti i ponasanja brisanja (cascade).
6. **Registracija i zivotni ciklus DI:**
   * **AddDbContext<ApplicationDbContexts>(...):** registruje DbContext u DI containeru (po defaultu je scoped – jedan po HTTP zahtevu). Vazno: Ne injectuje se scoped DbContext u singleton bez kreiranja scope – a.
7. **Transakcije, Migracije i snapshot:**
   * **DbContext** sudeluje u migracijama (generisanje Migrations klase i ModelSnapshot) i podrzava eksplicitne ili implicitne transakcije.

**ApplicationDbContext** odnosno mapira C# klase (entitete) u tabele (preko **DbSet<T>** i konvencija/atributa). **ApplicatioDbContext klasa.** DbContext klasa zapravo omogućava komunikaciju između aplikacije i baze podataka i sadrži DbSet svojstva koja predstavljaju tabele u bazi podataka.

U ovom slučaju, AplicationDbContext nasleđuje **IdentityDbContext<ApplicationUser>,** što znači da će koristiti ASP .NET Core Identity sistem za upravljanje korisnicnima. **ApplicationUser** je klasa koja predstavlja korinički model i povezuje se sa **IdentityDbContext,** kako bi podržala funkcionalnosti kao što su **autentifikacija** i **autorizacija** i **upravljanje korisničkim nalozima**.

**DbContextOptions** parametar koji prima konstruktor klase ApplicationDbContext, sadrži konfiguraciju konteksta baze podataka, uključujući connectionString koji se koristi za povezivanje sa odgovarajućom bazom podataka.

**DbSet<T> ClassName:** ApplicationDbContext sadrži **DbSet<T>** za svaki entitet koji se mapira na tabelu u bazi podataka.**OnModelCreating metoda** se overriduje, kako bi se konfigurisalo mapiranje entiteta i dodali početni podaci (seed-ovanje tabela). U primeru se definišu početni podaci za entitet MenuItem kroz **HasData metod**. To se koristi za inicijalizaciju baze podataka sa početnim podacima, kada se baza prvi put kreira ili kada se koriste migracije.

### 9.3 Code First vs Database First pristup i Migracije

**Entity Framework Core** koristi u ovom slucaju **Code First pristup.**

* **Code First pristup** podrazumeva da se model baze podataka definise putem C# klasa u kodu
* **Migracije** se koriste za kreiranje i azuriranje baze podataka na osnovu tih modela. **Migracije** u ASP .NET Core aplikacijama sluze za **upravljanje promenama u strukturi b.p.** tokom vremena. Omogucavaju da se promene koje napravimo u modelima entiteta, primene na bazu podataka bez potrebe za rucnim editovanjem baze. Migracije koriste **EF Core** i omogucavaju automatsko generisanje SQL skripti koje primenjuju promene na bazi podataka.
* Migracije se u .NET aplikaciji kreiraju pomocu komande ***dotnet ef migrations add <migration\_name>.*** Ta komanda ce uporediti trenutnu bazu podataka sa modelima definisanim u DbContext – u i generisati skriptu koja implementira promene u bazi.
* Nakon sto se migracije kreiraju, moguce ih je primeniti na bazu podataka pomocu ***dotnet ed database update.*** Svaki **Property** postaje **Kolona** u bazi podataka.

### 9.4 Struktura baze podataka i modeli

Tip baze podataka koji se koristi u aplikaciji je **relacioni tip baze podataka.** Ovaj tip koristi se iz razloga jer postoje jasno definisane veze između različitih entiteta.Još jedan od razloga za korišćenje relacionog tipa baze podataka je to što omogućavaju efikasnu obradu složenih upita, uključujući **join – ove** između različitih tabela, **grupisanje** podataka, **filtriranje** i **sortiranje.** Takođe, omogućava i referencionalni integritet, odnosno da se kroz strane ključeve pravilno povežu podaci među tabelama.

#### 9.4.1 Modeli

Modeli koji se koriste u bazi podataka i koji predstavljaju entitete u bazi podataka:

#### 9.4.2 ApplicationUser Model

* **ApplicationUser:** proširuje **IdentityUser klasu** (klasa koja je namenjena za upravljanje korisnicima i autentifikaciju)i dodaje property Name. Koristi se za reprezentaciju korisnika u aplikaciji. Nasleđuje **IdentityUser**, što znači da već ima polja za: UserName, PasswordHash, Email, PhoneNumber, Roles itd.

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.3 SportskiObjekat Model

* **SportskiObjekti:** predstavlja entitet koji odgovara sportskim objektima. Veza: 1 SportskiObjekat – N Termina. Navigaciono polje **List<Termin> Termini** omogucava da lako dohvatimo sve termine za dati objekat.

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.4 Termini Model

* **Termini:** predstavlja entitet koji cuva termine u okviru sportskog objekta. Ima **strani kljuc SportskiObjekatId.** Navigacija **SportskiObjekat** omogucava EF Core – u da poveze termin sa objektom. UserId Property se koristi da bismo znali ko je rezervisao termin.

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.5 Korpa Model

* **Korpa:** predstavlja korpu gde se nalaze rezervacije koje korisnik rezervise (sportski objekti i termini u okviru tih sportskih objekata). Svaka korpa je povezana sa jednim korisnikom preko **UserId.** Veza: **1 Korpa – N stavki korpe (1:N).** [NotMapped] polja oznacava da se ti property ne mapiraju u bazi podataka.

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.6 StavkaKorpe Model

* **StavkaKorpe:** predstavlja jednu stavku u korisnikovoj korpi. Svaka stavka se odnosi na **jedan sportski objekat.** OdabraniTermini predstavlja listu termina koje korisnik zeli da rezervise. Veza: **1 StavkaKorpe – 1 SportskiObjekat (1:1).** Koristim **List<Termin> OdabrniTermini** jer su odabrani termini u okviru stavke korpe lista termina koje je korisnik odabrao za jedan sportski objekat. Kada korisnik bira termine, aplikacija ih obicno **dinamicki dodaje ili uklanja.** Potrebno je i da ih sortiram po redu odabira, pa je zato List<T> najzgodniji izbor.

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.7 RezervacijaHeader Model

* **RezervacijaHeader:** predstavlja glavne informacije o rezervaciji. Povezana je sa **ApplicationUser** preko id – a. Pored ostalih property – a sadrzi i listu **RezervacijaDetalji** koji predstavljaju pojedinacne stavke u rezervaciji. Veza: **1 RezervacijaHeader – N RezervacijaDetalji.** Koristim **IEnumerable** jer je to najopstiji tip kolekcije. Omogucava samo **citanje (iteraciju)** kroz kolekciju (sa foreach), ne podrzava dodavanje ili uklanjanje elemenata. Idealno je kada je potreban samo read – only pristup podacima. Koristim ga jer se RezervacijaDetalji u okviru RezervacijaHeader se obično samo **čitaju** i prikazuju korisniku (npr. lista termina koje korisnik rezerviše).

A screen shot of a computer program

AI-generated content may be incorrect.

#### 9.4.8 RezervacijaDetalji Model

* **RezervacijaDetalji:** predstavlja pojedinacne stavke u rezervaciji. Veza: **1 RezervacijaHeader – N RezervacijaDetalji (1:N); 1 RezervacijaDetalji – 1 Termin (1:1).**

A screen shot of a computer program

AI-generated content may be incorrect.

### 9.5 Relazije i veze izmedju tabela

**1)** **SportskiObjekat(1) – Termini(N):**

* Sportski objekat ima **List<Termin> Termini**
* **Termin** ima **SportskiObjekatId** + navigaciju SportskiObjekat
* **Jedan objekat moze imati vise termina (1:N)**

**2) StavkaKorpe(N) – SportskiObjekat(1):**

* **StavkaKorpe** sadrzi **SportskiObjekatId** i navigaciju SportskiObjekat
* Svaka stavka korpe odnosi se na jedan sportski objekat; jedan objekat moze biti u vise stavi korpe

**3) Korpa(1) – StavkaKorpe(N):**

* **Korpa** ima **ICollection<StavkaKorpe> StavkaKorpe**
* **StavkaKorpe** ima **KorpaId** (konvencionalni FK)
* Jedna korpa sadrzi vise stavki, svaka stavka pripada jednoj korpi

**4) StavkaKorpe – Termin:** definisano kao List<Termin> OdabraniTermini

* StavkaKorpe sadrzi listu termina, dok Termin nema navigaciju prema StavkaKorpe, ali ce EF Core to mapirati kao **many to many odnos** uz automatsku join tabelu.

**5) RezervacijaHeader(1) – RezervacijaDetalji(N)**

* **RezervacijaHeader** ima kolekciju **RezervacijaDetalji**
* **RezervacijaDetalji** ima **RezervacijaHeaderId**
* Jedan header sadrzi vise detalja rezervacije (1:N)

**6) RezervacijaDetalji(N) – Termin(1):**

* **RezervacijaDetalji** sadrzi **TerminId** i navigaciju **OdabraniTermini** (tipa Termin)
* Svaki rezervacija detalji entitet se vezuje na jedan termin (N:1). Jedan termin moze biti referenciran iz RezervacijaDetalji.

**7) RezervacijaHeader(N) – ApplicationUser(1):**

* **RezervacijaHeader** ima ApplicationUserId i navigaciju **User**
* Jedan korisnik moze imati vise rezervacija.

**8) Korpa – ApplicationUser:**

* **Korpa** ima **UserId** (bez navigacije). Prema konvenciji ovo je FK prema AspNetUsers.
* Jedan korisnik – jedna aktivna korpa

**9) Termin – UserId:**

* **Termin** sadrzi string? UserId, sto oznacava koji je korisnik rezervisao/zauzeo termin.

**A computer screen shot of several different colored boxes

AI-generated content may be incorrect.**

### 9.6 SQL Joinovi

**SQL Join** se koristi kada želimo da spojimo podatke iz više tabela. Entity Framework koristi **LINQ upite** (skup upita ugrađenih u C# jezik koji omogućava rad sa kolekcijama, bazom podataka, XML-om, itd),koji interno generišu SQL Join.

Vrste SQL Join – ova:

* **INNER JOIN:** Kada želimo podatke koji **postoje u obe tabele**
* **LEFT JOIN:** Kada želimo **sve podatke iz leve tabele i podatke iz desne ako postoje.**
* **RIGHT JOIN:** Kada želimo **sve podatke iz desne tabele i podatke iz leve tabele ako postoje.**
* **FULL OUTER JOIN:** Kada želimo **sve podatke iz obe tabele** bez obzira da li su povezani.

### 9.7. SQL Klauzule

**SQL klauzule** su ključni delovi SQL upita koji definišu način na koji se podaci filtriraju, sortiraju, grupišu i manipuliraju.

1. **SELECT:** Koristi se za selektovanje kolona iz tabele.
2. **FROM:** Definiše izvor tabele ili tabela iz kojih se podaci uzimaju.
3. **WHERE:** Filtrira redove na osnovu uslova pre nego što se izvrši bilo kakva agregacija ili grupisanje.
4. **GROUP BY:** Grupisanje redova sa istim vrednostima u jednu grupu. Često se koristi sa agregatnim funkcijama kao što su COUNT, SUM, AVG, itd.
5. **HAVING:** Filtrira rezultate nakon grupisanja pomoću GROUP BY. HAVING se koristi isključivo sa agregatnim funkcijama (kao što su SUM, COUNT, AVG, MAX, MIN) i grupisanim rezultatima.
6. **ORDER BY:** Sortira rezultate upita u rastućem ili opadajućem redosledu.
7. **JOIN:** Spaja redove iz dve ili više tabela na osnovu povezanih kolona.
8. **UNION / UNION ALL:** Kombinuje rezultate iz dva ili više SELECT upita. UNION uklanja duplikate, dok UNION ALL zadržava sve redove, uključujući duplikate.
9. **INTERSECT:** Vraća redove koji su prisutni u oba SELECT upita.
10. **EXCEPT / MINUS:** Vraća redove iz prvog SELECT upita koji nisu prisutni u drugom SELECT upitu.
11. **DISTINCT:** Uklanja duplikate iz rezultata SELECT upita.
12. **AS:** Koristi se za dodeljivanje aliasa tabelama ili kolonama u rezultatu upita.
13. **LIMIT / OFFSET:** Ograničava broj redova koje upit vraća. OFFSET se koristi za preskočene redove.
14. **INSERT INTO:** Dodaje nove redove u tabelu.
15. **UPDATE:** Menja postojeće redove u tabeli.
16. **DELETE:** Briše redove iz tabele.
17. **TRUNCATE TABLE:** Briše sve redove iz tabele, ali ne može se koristiti za brisanje samo određenih redova.
18. **ALTER TABLE:** Menja strukturu tabele, kao što su dodavanje ili brisanje kolona.
19. **DROP TABLE:** Briše tabelu i sve njene podatke.
20. **CREATE TABLE:** Kreira novu tabelu u bazi podataka.

### 9.8 Razlika izmedju SQL klauzula

1. **SELECT vs FROM:**

* **SELECT** odredjuje koje kolone zelim da vidim u rezultatu
* **FROM** odredjuje iz koje tabele se podaci citaju

1. **WHERE vs HAVING**

* **WHERE:** filtrira redove pre grupisanja, tj pre GROUP BY
* **HAVING:** filtrira redove nakon grupisanja tj nakon GROUP BY

1. **GROUP BY vs ORDER BY:**

* **GROUP BY:** grupisanje redova po odredjenim kolonama (radi sa agregatnim funkcijama npr, COUNT, SUM, AVG…)
* **ORDER BY:** sortiranje rezultata po jednoj ili vise kolona (rastuce ASC ili opadajuce DESC)

1. **JOIN vs UNION:**

* **JOIN:** kombinuje kolone iz vise povezanih tabela po redovima
* **UNION:** kombinuje rezultate iz vise upita po kolonama (vertikalno)

1. **IN vs EXISTS:**

* **IN:** proverava da li vrednost postoji u listi ili podupitu
* **EXISTS:** vraca TRUE ako podupit vrati barem jedan red

1. **ANY vs ALL:**

* **ANY:** poredi sa bilo kojom vrednoscu iz podupita
* **ALL:** poredi sa svim vrednostima iz podupita

### 9.9 DTO klase

**DTO (Data Transfer Object) klase** koriste se za prenos podataka izmedju razlicitih slojeva aplikacije, cesto izmedju klijentskog i serverskog sloja. Sluze kao **kontejneri za podatke koje treba preneti.** Cilj koriscenja DTO klasa je pojednostavljenje strukture podataka i smanjenje kolicine podataka koji se prenose izmedju slojeva aplikacije, sto moze poboljsati i sigurnost i performanse.

### 9.10 Kolekcije i interfejsi u .NET – u

1. **IEnumerable<T>:**

* Najosnovniji interfejs za kolekcije
* Omogucava samo **citanje** elemenata pomocu foreach. Nema **dodavanja, brisanja,** ni **pristupa po indeksu**
* Koristi se kada nam treba samo iteracija (read – only pristup)
* Vremenska kompleksnost: **O(n)** – za foreach prolazak

1. **ICollection<T>:**

* Nasledjuje IEnumerable<T> i dodaje metode **.Add(); .Remove(), .Count**
* Nema indeksiranje; Standard je za navigacione kolekcije u EF Core (npr. public ICollection<Termin> Termini { get; set; })
* Koristi se kada nam treba fleksibilnost i kada ne zavisim od konkretne implementacije
* Vremenska kompleksnost: **O(1) –** za Count, a za Add i Remove zavisi od implementacije

1. **IList<T>:**

* Nasledjuje ICollection<T> i dodaje **indeksiranje** (list[0])
* Podrzava **ubacivanje** i **brisanje** na odredjenom indeksu
* Koristi se kada nam treba **uredjena lista** sa pristupom po indeksu
* Vremenska kompleksnost: **O(n) – Insert(index, item); O(n) – RemoveAt(index); O(1) – this[index],** tj. Pristup po indeksu

1. **List<T>**

* Konkretna implementacija IList<T>; Najcesce koriscena kolekcija u praksi.
* Omogucava **brzo dodavanje, uklanjanje i indeksiranje**
* Vezivanje za konkretan tip (manje fleksibilno od ICollection<T>)
* Koristi se kada nam je potrebna iskljucivo lista i znamo da necemo menjati implementaciju
* Interno koristi **niz** (T[] ) kao osnovnu strukturu podataka, koji se dinamicki prosiruje kada se popuni
* Vremenska kompleksnost: **O(1) – Add(item) – amortizovano,** a ponekad resize = O(n), ali retko**; O(n) – Insert(index, item); O(n) – Remove(item)** (pretraga + pomeranje); **O(n) – RemoveAt(index); O(1) – Indexing(list[i]); O(1) – Contains(item); O(1) – Count;**

1. **IQueryable<T>:**

* Koristi se za **upite nad bazom podataka** (LINQ to SQL, EF Core)
* Ne izvrsava upit odmah – koristi **deferred execution (odlozeno izvrsavanje)**
* Npr. upit: IQueryable<User> users = db.Users.Where(u => u.IsActive); se izvrsava **tek kada se enumerise (npr. sa ToList() ).**
* Koristi se kada radim sa bazom i hocu LINQ upite prevesti u SQL upite
* Vremenska kompleksnost: **Ovo je deferred execution, kompleksnost zavisi od baze, a ne od C# implementacije**

1. **Dictionary<TKey, TValue>:**

* Kolekcija koja cuva podatke u **parovima kljuc – vrednost**
* Brz pristup vrednostima preko kljuca (dict[„key“] )
* Kljucevi moraju biti jedinstveni
* Koristi se kada mi treba **mapiranje**
* Vremenska kompleksnost: **O(1) – Add(key, value) –** procecno, a u najgorem slucaju O(n), ako se desi hash collision za sve; **O(1) – Remove(key); O(1) – ContainsKey(key); O(1) – dict[key] –** pristup po kljcu; **O(n) – Iteracija svih parova**

1. **HashSet<T>:**

* Kolekcija koja cuva samo **jedinstvene elemente**
* Brz pristup, koristi hashiranje
* Koristi se kada zelim da sprecim duplikate
* Vremenska kompleksnost: **O(1) – Add(item); O(1) -Remove(item); O(1) – Contains(item); O(n) - iteracija**

1. **Queue<T>:**

* Struktura **FIFO** (First In First Out)
* **Enqueue()** dodaje na kraj, **Dequeue()** skida sa pocetka
* Koristi se za redove cekanja, taskove koji se obradjuju redom
* Vremenska kompleksnost: **O(1) – Enqueue(item); O(1) – Dequeue(); O(1) – Peek(); O(n) – Iteracija;**

1. **Stack<T>:**

* Struktura **LIFO** (Last In First Out)
* **Push()** dodaje na vrh, **Pop()** skida poslednji ubaceni elemenat
* Koristi se za „undo“ operacije, parsiranje izraza, itd.
* Vremenska kompleksnost: **O(1) – Push(item); O(1) – Pop(); O(1) – Peek(); O(n) - iteracija**

1. **ObservableCollection<T>:**

* Kolekcija koja obavestava UI kada se promeni (dodavanje/brisanje elemenata)
* Koristi se u WPF, Blazor i MVVM aplikacijama za **data binding.**
* Vremenska kompleksnost: **O(1) – Add(item); O(n) – Remove(item); O(n) – Insert(index, item).**

## Controllers

**Controller (Kontroler)** je klasa u ASP .NET Core aplikaciji koja prima HTTP zahteve (GET, POST, PUT, DELETE...) i vraca HTTP odgovore.  
Obicno nasledjuje **ControllerBase** ili **Controller** (ako koristi View)  
Kontroler je deo MVC (Model – View – Controller) arhitekture i on sluzi kao posrednik izmedju Modela (baze podataka) i View – a (klijentskog interfejsa).

U REST API-ju kontroleri izlazu **endpointe** (npr /api/rezervacija/sveRezervacije).

### 10.1 Controller Base klasa

**ControllerBase** je osnovna klasa za ASP .NET Controller-e i omogućava klasama koje je nasleđuju da koriste funkcionalnosti kontrolera. Neke od njih su

**1. pristup HTTP zahtevima, odgovorima, modelima stanja** (**ModelState);**

**2. ControllerBase omogućava korišćenje atributa** kao što su: **[HttpGet], [HttpPost], [HttpPut] i [HttpDelete]** koij se koriste za mapiranje HTTP zahteva na metode kontrolera;

**3.** **Metode za vraćanje odgovora:** neke od njih su: Ok(), BadRequest(), NotFound(), Created().

### 10.2 HTTP Zahtevi (HTTP Requests)

**HTTP zahtev** je poruka koju klijent šalje serveru da bi tražio određene resurse, kao što su web stranice, slike, ili drugi podaci.

Osnovni delovi HTTP zahteva su:

1. **HTTP Metoda**: Definiše akciju koju klijent želi da izvrši.
2. **URL (Uniform Resource Locator)**: Adresa resursa koji se traži. Sastoji se od protokola, domena i putanje do resursa.
3. **HTTP Verzija**: Definiše verziju HTTP protokola koja se koristi (npr. HTTP/1.1, HTTP/2).
4. **Zaglavlja (Headers)**: Dodatne informacije koje klijent šalje serveru, kao što su tip sadržaja, autentifikacija, ili informacije o klijentu (User-Agent).
5. **Telo zahteva (Request Body)**: Podaci koji se šalju serveru (obično sa POST, PUT, ili PATCH metodama). Telo može sadržati JSON, XML, ili druge formate podataka.

### 10.3 HTTP Odgovori (HTTP Response)

**HTTP odgovor** je poruka koju server šalje nazad klijentu kao odgovor na zahtev. Odgovor sadrži informacije o resursu, statusu zahteva, i dodatnim podacima.

Osnovni delovi HTTP odgovora su:

1. **HTTP Verzija**: Verzija HTTP protokola koju koristi server (npr. HTTP/1.1, HTTP/2).
2. **Statusni Kod (Status Code)**: Numerički kod koji označava ishod zahteva. Statusni kodovi su podeljeni u pet kategorija:
   1. **1xx (Informacioni)**: Zahtev je primljen i procesira se.
   2. **2xx (Uspeh)**: Zahtev je uspešno obrađen (npr. 200 OK).
   3. **3xx (Preusmerenje)**: Dodatna akcija je potrebna (npr. 301 Moved Permanently).
   4. **4xx (Greška Klijenta)**: Problem sa zahtevom od strane klijenta (npr. 404 Not Found).
   5. **5xx (Greška Servera)**: Problem na strani servera (npr. 500 Internal Server Error).
3. **Zaglavlja (Headers)**: Dodatne informacije koje server šalje klijentu, kao što su tip sadržaja, dužina sadržaja, ili informacije o keširanju.
4. **Telo odgovora (Response Body)**: Sadržaj odgovora, kao što je HTML stranica, JSON podaci, ili binarni podaci (npr. slike, video). U zavisnosti od zahteva, telo može biti prazno (npr. za 204 No Content).

U okviru aplikacije koristim **ApiResponse** klasu kako bih standardizovao sve HTTP odgovore. To radim iz razloga da bih imao:

1. **Konzistentnost odgovora:** bez obzira da li vracam podatke, greske ili bilo koji drugi odgovor, svi ce imati isti format:

A screenshot of a computer code

AI-generated content may be incorrect.

1. **Lakse rukovanje greskama:** umesto da klijent parsira razlicite formate gresaka, uvek moze da proveri IsSuccess i procita ErrorMessages  
   A screen shot of a computer code

   AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### 10.4 HTTP Status kodovi

**HTTP status kodovi** su trocifreni brojevi koje server vraća kao odgovor na klijentov zahtev.

Kategorije status kodova:

1. **1xxx: Informacioni odgovori**

* **100 Continue:** Klijent može nastaviti sa slanjem tela zahteva
* **101 Switching protocols:** Server prihvata zahtev za promenu protokola
* **102 Processing:** Server je primio i obrađuje zahtev, ali još uvek nema odgovora.

1. **2xx: Uspešni odgovori:**

* **200 OK:** Zahtev je uspešno obrađen
* **201 Created:** Zahtev je uspešno obrađen i resurs je kreiran
* **202 Accepted:** Zahtev je prihvaćen za obradu ali nije završen
* **203 Non-Autoritative Information:** Vraća se meta informacija iz drugog izvora
* **204 No Content**: Zahtev je uspešno obrađen, ali nema sadržaja za povratak.
* **205 Reset Content**: Zahtev je uspešno obrađen, ali klijent treba da resetuje prikaz.
* **206 Partial Content**: Server vraća deo resursa kao odgovor na Range zahtev

1. **3xx: Preusmeravajući odgovori**

Ovi status kodovi označavaju da klijent treba da preduzme dodatne akcije kako bi dovršio zahtev.

* **300 Multiple Choices**: Više mogućih odgovora na zahtev.
* **301 Moved Permanently**: Resurs je trajno premješten na novi URL.
* **302 Found**: Resurs je privremeno premješten na novi URL.
* **303 See Other**: Klijent treba da koristi drugi URL za pristup resursu.
* **304 Not Modified**: Resurs nije modifikovan od poslednjeg zahteva.
* **305 Use Proxy**: Traženi resurs mora biti pristupljen preko proxy servera.
* **307 Temporary Redirect**: Resurs je privremeno premješten na novi URL.
* **308 Permanent Redirect**: Resurs je trajno premješten na novi URL.

1. **4xxx: Klijentske greške**

Ovi status kodovi označavaju da je došlo do greške na strani klijenta.

* **400 Bad Request**: Zahtev nije ispravan.
* **401 Unauthorized**: Potrebna je autentifikacija.
* **402 Payment Required**: Rezervisan za buduću upotrebu.
* **403 Forbidden**: Server odbija da obradi zahtev.
* **404 Not Found**: Traženi resurs nije pronađen.
* **405 Method Not Allowed**: HTTP metoda nije dozvoljena za resurs.
* **406 Not Acceptable**: Resurs nije prihvatljiv prema Accept zaglavljima.
* **407 Proxy Authentication Required**: Potrebna je autentifikacija preko proxy servera.
* **408 Request Timeout**: Server je istekao čekajući na zahtev.
* **409 Conflict**: Došlo je do konflikta sa trenutnim stanjem resursa.
* **410 Gone**: Resurs više nije dostupan i neće biti vraćen.
* **411 Length Required**: Nedostaje Content-Length zaglavlje.
* **412 Precondition Failed**: Preuslov iz zahteva nije ispunjen.
* **413 Payload Too Large**: Veličina zahteva je prevelika.
* **414 URI Too Long**: URI zahteva je predugačak.
* **415 Unsupported Media Type**: Medijski tip nije podržan.
* **416 Range Not Satisfiable**: Traženi opseg nije zadovoljavajući.
* **417 Expectation Failed**: Expect zaglavlje zahteva nije ispunjeno.
* **418 I'm a teapot**: Aprilska šala, označava da server nije teapot.
* **421 Misdirected Request**: Zahtev je usmeren ka serveru koji nije sposoban da ga obrađuje.
* **422 Unprocessable Entity**: Zahtev je dobro formatiran, ali se ne može obraditi zbog semantičkih grešaka.
* **423 Locked**: Resurs je zaključan.
* **424 Failed Dependency**: Zavisan zahtev nije uspeo.
* **425 Too Early**: Prethodni zahtevi su preuranjeni.
* **426 Upgrade Required**: Potrebno je preći na drugi protokol.
* **428 Precondition Required**: Preuslov zahteva je potreban.
* **429 Too Many Requests**: Previše zahteva u kratkom vremenskom periodu.
* **431 Request Header Fields Too Large**: Polja zahteva su prevelika.
* **451 Unavailable For Legal Reasons**: Resurs je nedostupan iz pravnih razloga.

1. **5xxx: Serverske greške**

Ovi status kodovi označavaju da je došlo do greške na strani servera.

* **500 Internal Server Error**: Došlo je do greške na serveru.
* **501 Not Implemented**: Server ne podržava funkcionalnost potrebnu za obradu zahteva.
* **502 Bad Gateway**: Primljen je neispravan odgovor od ulaznog servera.
* **503 Service Unavailable**: Server nije dostupan.
* **504 Gateway Timeout**: Server nije dobio pravovremeni odgovor od ulaznog servera.
* **505 HTTP Version Not Supported**: HTTP verzija nije podržana.
* **506 Variant Also Negotiates**: Pregovaranje sadržaja je uzrokovalo cirkularnu referencu.
* **507 Insufficient Storage**: Server nema dovoljno prostora za obradu zahteva.
* **508 Loop Detected**: Server je otkrio beskonačnu petlju prilikom obrade zahteva.
* **510 Not Extended**: Potrebno je proširenje zahteva.
* **511 Network Authentication Required**: Potrebna je mrežna autentifikacija.

### 10.5 HTTP Metode

**HTTP metode** su definisane u HTTP protokolu i koriste se za određivanje akcija koje treba preduzeti nad resursima na serveru.

1. **HTTP GET:**

* **Svrha**: Dohvatanje resursa sa servera.
* **Idempotentnost**: Idempotentna (ponovljeni zahtevi imaju isti efekat kao jedan).
* **Telo zahteva**: Nema (parametri se obično šalju u URL-u kao query string).
* **Telo odgovora**: Da (resurs koji je zatražen).

1. **HTTP POST:**

* **Svrha**: Slanje podataka serveru za kreiranje novog resursa.
* **Idempotentnost**: Nije idempotentna.
* **Telo zahteva**: Da (podaci koji se šalju serveru).
* **Telo odgovora**: Može biti (novokreirani resurs ili rezultat operacije).

1. **HTTP PUT:**

* **Svrha**: Ažuriranje postojećeg resursa ili kreiranje resursa ako ne postoji.
* **Idempotentnost**: Idempotentna.
* **Telo zahteva**: Da (podaci koji se ažuriraju).
* **Telo odgovora**: Može biti (ažurirani resurs ili rezultat operacije).

1. **HTTP DELETE:**

* **Svrha**: Brisanje resursa sa servera.
* **Idempotentnost**: Idempotentna.
* **Telo zahteva**: Nema (ponekad može imati, ali je retko).
* **Telo odgovora**: Može biti (potvrda brisanja ili rezultat operacije).

1. **HTTP PATCH:**

* **Svrha**: Ažuriranje dela resursa.
* **Idempotentnost**: Nije nužno idempotentna.
* **Telo zahteva**: Da (promene koje treba primeniti).
* **Telo odgovora**: Može biti (ažurirani resurs ili rezultat operacije).

### 10.6 HTTP zaglavlja (HTTP Headers)

**HTTP Zaglavlja** su ključno – vrednosni parovi koji se šalju u HTTP zahtevima i odgovorima kako bi pružili dodatne informacije o zahtevu, odgovoru ili samim podacima koji se prenose.

Mogu se koristiti za:

* **Identifikaciju klijenta i servera:** (npr. browser, server tip, API verzija)
* **Kontrolu keširanja**
* **Autorizaciju i autentifikaciju**
* **Definisanje formata i endpointa podataka**

Vrste HTTP zaglavlja:

1. **Zaglavlja zahteva: (Request Headers):**

Koriste se u HTTP zahtevima koje klijent šalje serveru. Primer može biti Authorization, koji sadrži kredencijale za autentifikaciju

1. **Zaglavlja odgovora (Response Headers)**

Koriste se u HTTP odgovorima koje server šalje klijentu. Primer je **Acess – Control: Allow – Origin:** Definiše koji domeni mogu pristupiti resursima (CORS),**Content – Type:** Označava tip podataka (text/html, application/json)

1. **Zaglavlja entiteta (Entity Headers):**

Opisuju sadržaj samog tela poruke. Primeri: **Content – Length:** dužina tela odgovora u bajtovima; **Content – Encoding:** Način enkodiranja; **Last – Modified:** Datum poslednje izmene resursa.

## Vrste ucitavanja podataka u EF Core

### 11.1 Eager Loading (nestripljivo ucitavanje)

* Podaci se ucitavaju odmah zajedno sa glavnim entitetom u **jednom upitu** koristeci **.Include()** i **.ThenInclude()**
* Ucitava povezane podatke **odmah zajedno sa glavnim objektom** u jednom SQL upitu. Brzi je ako podaci trebaju odmah, ali moze povuci vise podataka nego sto je potrebno.
* Primer:

A screen shot of a computer code

AI-generated content may be incorrect.

* Ovde se odmah ucitavaju RezervacijaDetalji i povezani OdabraniTermini

### 11.2 Explicit Loading (Eksplicitno ucitavanje)

* Rucno ucitavanje vezanih podataka kada mi treba
* Primer:



### 11.3 Lazy Loading (Lenjo ucitavanje)

* Podaci se ucitavaju **tek kada im se prvi put pristupi.**
* Radi samo ako je omogucen **Lazy Loading Proxies** paket i ako su properties virtual.
* Mana: moze da pravi vise upita ka bazi podataka.
* Prednost: dobar je ako ne znam da li ce mi ti podaci trebati

## Async metode

**Async vs Sync** metode:

* **Async metode:** metoda se izvrsava **neblokirajuce** – dok ceka bazu ili API, program moze da nastavi sa drugim zadacima
* **Sync metode:** metoda se izvrsava **blokirajuce** – dok ne zavrsi, program ne ide dalje

**Async / await**

* **async** je kljucna rec koja obelezava metodu da sadrzi asinhroni kod
* **await** se koristi da „saceka“ rezultat asinhrone operacije, ali bez blokiranja niti
* Primer:

A black background with white text

AI-generated content may be incorrect.

* Umesto da blokira nit dok se upit zavrsi, vraca kontrolu aplikaciji dok ne dobije rezultat

**Async void:**

* **async void** se ne koristi osim za event handlere
* Problem je sto se ne mogu hvatati greske ni pratiti zavrsetak metode
* Izuzeci baceni u okviru metode ne mogu biti uhvaceni od strane pozivanog try – catch bloka

**Primer:**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

* **async:** oznacava da se radi o asinhronoj metodi
* **Task:** zadatak, predstavlja operaciju koja se izvrsava asinhrono, slicno obecanju da ce se nesto zavrsiti kasnije.
* **ActionResult:** Rezultat akcije(tip koji predstavlja HTTP odgovor u Web API – ju, npr. Ok, BadRequest, NotFound, itd)
* **<ActionResult<ApiReponse>>:** rezultat akcije je u obliku ApiResponse (klasa za standardizovan odgovor).

## ModelState

**ModelState** je kolekcija koja sadrzi **stanje validacije** modela koji je ASP .NET Core binder mapirao iz HTTP zahteva.  
Kada dodje zahtev (npr. PUT sa JSON telom ili form-data), ASP .NET Core koristi **model binding** da popuni objekat.  
**ModelBinding** omogucava da ne moram rucno da citam HTTP zahteve i parsiram podatke, vec Framework to radi za nas. Tacnije to je proces gde framework automatski mapira dolazne HTTP podatke u C# objekte (parametre metoda ili svojstva klasa). **Binder** je komponenta koja vrsi ovu konverziju.

U procesu binding – a ASP .NET Core proverava:

* Da li su sva polja uspesno mapirana
* Da li su ispostovani **atributi validacije** ( [Required], [StringLength], itd)
* Ako bilo sta ne prodje validaciju => ModelState.IsValid == false.

Ukratko, **ModelState** sluzi kao **validacija inputa korisnika** i sprecava da nastavimo dalje ako korisnik posalje neispravan ili nepotpun objekat.

## Autentifikacija i Autorizacija korisnika – Opste

**Autentifikacija** je process provere identiteta korisnika. U aplikaciji, koristi se ASP .NET Identity za kreiranje i proveru korisničkih naloga.

**Autentifikacija** se implementira u metodi **Login** u **AuthController-u.** Korisničko ime i lozinka se proveravaju u bazi podataka uz pomoć **UserManager<ApplicationUser>.** Ukoliko su korisničko ime i lozinka ispravni, generiše se **JWT Token.**

## Autentifikacija

### 15.1 JWT Token

**JWT Token**(JSON Web Token) je standardni način za prenošenje informacija između stranica kao JSON objekat. JWT token sastoji se od 3 dela, koji su odvojeni **tačkama (.):**

1. **Zaglavlje:** sadrži informacije o vrsti tokena i algoritmu potpisa(HMAC SHA256 ili RSA).
2. **Payload (Telo):** Sadrži klijentske podatke ili claims, poput uloge, imena, ID-a.
3. **Potpis:** Koristi se za proveru integriteta tokena

**JWT token** se na serveru najčešće šalje putem **HTTP zahlavlja (Header) (**ključno – vrednosni parovi koji se šalju u HTTP zahtevima i odgovorima kako bi pružili dodatne informacije o zahtevu, odgovoru ili samim podacima koji se prenose), obično u **Autorization zaglavlju,** koristeći Bearer šemu. JWT Token može biti poslat i **kao deo URL-a**, **kao query parametar**. Ovo nije preporučeno iz sigurnostnih razloga. JWT token još može biti i smešten u HTTP cookie i slan automatski sa svakim zahtevom.

JWT Token se može slati i preko:

1. **Query Parametara:**

Može biti poslat kao deo URL-a, ali to nije preporučljivo zbog bezbednostih razloga.

1. **Telo Zahteva:**

JWT Token može biti poslat u telu POST zahteva, obično kao deo JSON objekta.

1. **Postavlja se kao HTTP – only cookie:** tek nakon što se korisnik prijavi token se postavlja kao http – only cookie odnosno uključuje se automatski u zaglavlje kolačića prilikom svakog sledećeg zahteva ka serveru.

Za pravljenje JWT tokena koristi se **ClaimsIdentity** koji sadrži **Claims** o korisniku, poput ID- a, email-a i uloge. Token se generiše koristeći **JWTSecurityTokenHandler** i **SigningCredentials.**

Kako funkcioniše JWT tokem?

1. **Kreiranje:** Kada korisnik uspešno unese svoje podatke za prijavu, server kreira JWT koji sadrži osnovne informacije o korisniku. Token je zatim potpisan koristeći tajni ključ servera i vraća se klijentu.
2. **Slanje:** Klijent čuva token obično u lokalnom skladištu (localStorage) ili kolačićima (cookies). Svaki put kada klijent pravi zahtev ka serveru, šalje token u HTTP zaglavlju (najčešće u Autorization zaglavlju kao Bearer <token>)
3. **Verifikacija:** Server na kraju prima token i verifikuje njegov integritet i validnost koristeći tajni ključ. Ako je token validan, server može ekstraktovati korisničke informacije iz payload-a i koristiti ih za autorizaciju i pristup resursima

Kada **token istekne,** korisnik mora ponovo da se prijavi da bi dobio novi token. Medjutim ako zelim da dobijem novi token automatski, potrebno je implementirati Refresh token sistem, gde bih u bazi imao polja RefreshToken(string tip) i RefreshTokenExpired(DateTime tip). I implementirao bih **endpoint** za **refresh tokena.**

### 15.2 Kako se generise JWT Token i kako se salje korisniku

**Token** se generise u metodi Login (AuthController.cs):

A computer screen shot of a program code

AI-generated content may be incorrect.

Ovde se:

* Dodaju **Claimovi –** predstavljaju potpis korisnika tj osnovne informacije o korisniku, oni se cuvaju u JWT tokenu i predstavljaju zapravo atribute korisnika(ID, ime, email, rola, broj telefona)
* Podesava se **vreme isteka tokena** (1h)
* Potpisuje se **pomocu tajong kljuca** (ApiSettings:Secret)
* Prilikom uspesnog logovanja token se generise i **postavlja se kao HTTP Only cookie** (Response.Cookies.Append(„jwt“, jwtToken, cookieOptions);
* Takodje se token vraca u telu odogovora kao deo objekta LoginResponseDTO (svojstvo Token).

U okviru **Program.cs** (podesavanje autentifikacije i validacije tokena):

* JWT autentifikacija je konfigurisana koriscenjem **AddAutentication** i **AddJwtBearer.**
* Token se validira pomocu **tajnog kljuca** (ApiSettings:Secret iz konfiguracije)
* Podeseni su parametri validacije: proverava se potpis (ValidateIssuerSigninKey = true), ali se ne proveravaju issuer i audience (ValidateIssuer = false; ValidateAudience = false).
* Swagger je podesen tako da ocekuje JWT token u **HTTP header-u** pod imenom “Authorization” sa semom “Bearer”.

A computer screen shot of a program

AI-generated content may be incorrect.

### 15.3 JWT Bearer

**AddAuthentication** postavlja šemu autentifikacije za aplikaciju. U ovom slučaju koristi se **JWT Bearer autentifikacija.** Znači, aplikacija će očekivati JWT token u HTTP headerima prilikom svakog poziva na zaštićene rute. **DefaultAuthenticateScheme** i **DefaultChallengeScheme** postavljaju da je standardni način autentifikacije putem Bearer tokena, što znači da će aplikacija u svakom zahtevu čekati JWT token.

**AddJwtBearer** konfiguriše detalje za verifikaciju I obradu JWT tokena.

* **RequireHttpsMetadata = false**: Ova opcija znači da nije obavezno koristiti HTTPS za metapodatke o JWT-u (što je korisno za razvoj i testiranje, ali u produkciji treba da bude true).
* **SaveToken = true**: Ova opcija omogućava da se token sačuva u **HttpContext** i bude dostupan tokom životnog ciklusa korisničkog zahteva. Međutim, ovo ne znači da se token čuva u bazi podataka. Samo se privremeno **čuva u memoriji servera** tokom trajanja HTTP zahteva. **Token se ne čuva trajno** na backendu. Ova opcija je korisna ako želiš da imaš pristup tokenu unutar aplikacije dok je zahtev aktivan (npr. možeš ga koristiti za dalja proveravanja ili za logovanje korisnika).
* **TokenValidationParameters**: Ovde se postavljaju parametri za validaciju tokena:
* **ValidateIssuerSigningKey = true**: Ovaj parametar omogućava da se proveri **da li je ključ koji je korišćen za potpisivanje tokena validan**.
* **IssuerSigningKey**: Ključ koji se koristi za validaciju potpisa tokena. U ovom slučaju koristi se **simetrični ključ** (key), koji je string koji je pretvoren u bajtove.
* **ValidateIssuer = false** i **ValidateAudience = false**: Ove opcije onemogućavaju validaciju **issuer-a** i **audience-a** u JWT-u. To znači da server neće proveravati ko je izdao token ili za koga je token izdat. Ako želiš da proveravaš te informacije, moraš ih postaviti na true.

Ovo znači da kada se koristi SaveToken = true, token se **privremeno čuva u HttpContext** **tokom trajanja HTTP zahteva** (samo za trenutnu sesiju). Token se ne upisuje u bazu podataka niti u neki trajni storage.

## Autorizacija

**Autorizacija** je proces dodeljivanja prava korisnicima na osnovu njihove uloge ili identiteta. Nakon uspesne autentifikacije korisnika, njemu se dodeljuju uloge, koje se kasnije koriste za kontrolu pristupa razlicitim delovima aplikacije. Implementira se pomocu **JWT Tokena** koji se salje svakim zahtevom nakon prijave. Na serverskoj strani JWT token se proverava ukljucivanjem **Authorize atributa** na odredjene metode kontrolera. Autorizacija se izvodi na osnovu informacija sadrzanih u JWT tokenu, kao sto je uloga korisnika.

## Login Metoda

**Login metoda** obrađuje HTTP POST zahtev za prijavu korisnika. HTTP POST metoda se koristi iz par razloga: 1) **Sigurnost:** korišćenjem HTTP POST zahteva, podaci kao što su korisničko ime i lozinka, šalju se kroz telo zahteva, što ih čini manje vidljivim na URL-u (putanji) zahteva, 2) **Skladištenje osetljivih podataka:** korišćenjem HTTP POST zahteva, lozinke (koje predstavljaju osetljive podatke) šalju se u telu zahteva, koje nije vidljivo u URL-u i neće se zadržavati u logovima servera.

Funkcionisanje Login metode:

1. Prvo se proverava da li korisnik sa datim korisničkim imenom i lozinkom, postoji u bazi podataka
2. Ako korisnik ne postoji, ili je lozinka neispravna, vraća se odgovor sa greškom
3. Ako je prijava uspešna, **generiše se JWT Token** koji sadrži informacije o korisniku i potpisuje se tajnim ključem
4. Token se šalje korisniku u odgovoru i na kraju se vraća odgovor o uspešnosti operacije prijave
5. Frontend prima token i čuva ga u localStorage.
6. Token se zatim koristi za autentifikaciju
7. Kada korisnik klikne na „Logout“, token se briše iz localStorage i korisnik se odjavljuje.

Korišćene su security metode u vezi sa autentifikacijom i autorizacijom korisnika. Konkretno neke od njih su:

1. **Hashing lozinke:** Metod: **UserManager<ApplicationUser>.CreateAsync()** u Register Metodi. Kada se korisnik registruje, lozinka se **hashuje** pomoću **UserManager<ApplicationUser>.CreateAsync()** čime se osigurava da se lozinke ne čuvaju u plain – text formatu. Metod **PasswordHasher<ApplicationUser>.HashPassword()**  u **UpdateUserDetails()** metodi omogućava da se prilikom ažuriranja korisnika, nova lozinka takođe hashuje pre nego što se sačuva u bazi podataka.
2. **Provera lozinke pri prijavi:** Metod **UserManager<ApplicationUser>.ChechPasswordAsync()** u **Login()** metodi proverava da li unesena lozinka odgovara hashovanoj lozinci korisnika iz baze podataka.
3. **Generisanje JWT tokena za autentifikaciju:** Metod **JwtSecurityTokenHandler.CreateToken()** u metodi **Login()** omogućava da kada se korisnik prijavi uspešno, generiše se JWT Token koji sadrži korisničke podatke I korisničku ulogu (ClaimTypes.Role); metod **SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)** omogućava da se JWT token potpisuje **sigurnim ključem** kako bi se osiguralo da nije izmenjen.

Da bih osigurao da samo **autorizovani** (ulogovani) korisnici mogu pristupiti određenim metodama, koristim **[Authorize] atribut** nad određenim metodama.

## Register metoda

**Register metoda** obrađuje HTTP POST zahtev za registraciju novog korisnika

Funkcionisanje Register metode:

1. Najpre se proverava da li korisničko ime već postoji u bazi i ako je to slučaj, vraća se odgovor sa odgovarajućom greškom
2. Ako korisničko ime ne postoji, **kreira se novi ApplicationUser objekat** sa podacima iz zahteva za registraciju
3. Zatim se koristi **UserManager** da se novi korisnik kreira u bazi
4. Ako je kreiranje uspešno, **proverava se da li postoje uloge u bazi,** i ukoliko ne postoje, kreiraju se. Novom korisniku se dodaje uloga „Admin“ ili „Customer“ na osnovu informacija iz zahteva
5. Na kraju se vraća odgovor koji ukazuje na to da li je operacija uspešna ili ne

## Dodeljivanje rola korisniku

U ASP .NET Core Identity role se dodeljuju korisnicima pomocu **UserManager<ApplicationUser>** i njegovog metoda **AddToRoleAsync.**U aplikaciji se to postize u metodi **Register:**

A screen shot of a computer program

AI-generated content may be incorrect.

* Prvo se proverava **da li role postoje u bazi.** Ako „Admin“ ili „Customer“ ne postoje u bazi (tabela AspNetRoles), one se prvo kreiraju
* Zatim se **dodeljuju role korisniku.** Kada se korisnik registruje, proverava se tip role koji je poslat iz forme (register.Role). Ako je Admin, dodaje se u rolu Admin, u suprotnom dodaje se u rolu Customer.
* Zatim se role cuvaju u **JWT Tokenu kao Claim.** To se radi iz razloga jer kada se korisnik prijavi, potrebno je procitati njegove role iz JWT Tokena.

# React

**React** je JavaScript biblioteka za kreiranje korisnickih interfejsa baziranim na komponentama. Smatra se **slojem prikaza** u aplikaciji. Podeljen je na dva glavna API – ja:

**1. React Component API:** Ovaj API su delovi stranice koje prikazuje React DOM

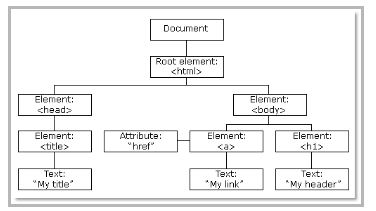
**2.** **React DOM:** Ovo je API koji se koristi za izvrsavanje renderovanja (proces prikazivanja sadrzaja i elemenata web stranice u browseru) na web stranici. **DOM** predstavlja HTML u browseru nakon sto je renderovan. **DOM API** sluzi da JS moze da promeni sadrzaj na stranici.

React omogucava **deklarativno programiranje** (opisujem sta se prikazuje, ne kako se prikazuje), kao i jednosmerni tok podataka.

**JSX** je sintaksna ekstenzija JavaScript-a. Koristi se sa React-om kako bi opisao izgled UI-a. Pomocu JSX-a mozemo pisati HTML strukture unutar istog fajla koji sadrzi i JavaScript kod.

**Web browseri** ne mogu direktno da citaju JSX. Oni su napravljeni da citaju samo regularne JavaScript objekte, a JSX to nije. Da bi web browser mogao da procita JSX fajl, on mora biti transformisan u regularni JavaScript objekat. Za tu transformaciju koristi se **Babel.**

**DOM** (Document Object Model) predstavlja HTML dokument kao logicku strukturu u obliku stabla. Svaka grana tog stabla se zavrsava cvorom, koji sadrzi objekte.



**React** odrzava laku, internu verziju pravom DOM-a u memoriji, poznatu kao **Virtual DOM.** Kada se promeni stanje nekog objekta, **virtual DOM** menja samo taj objekat u stvarnom DOM-u, umesto da azurira sve objekte. To omogucava efikasnije i brze azuriranje interfejsa.

## Sta je Event u React-u?

**Event (dogadjaj)** je radnja koju korisnik ili sistem moze da pokrene, kao sto je pritisak na taster ili klik misem. U Reactu se nazivi dogadjaja pisu koristeci **camelCase stil** (npr. onClick), za razliku od malih slova u HTMLu. Kada koristimo JSX, prosledjujemo funkciju kao obradjivac dogadjaja **(event handler)** umesto stringa kao u HTMLu.

## Syntethic Events?

**Syntethic events** (sinteticki dogadjaji) kombinuju ponasanje razlicitih native (ugradjenih) dogadjaja u jedan API, obezbedjujuci doslednost izmedju razlicitih browser-a. Aplikacija se ponasa dosledno bez obzira na to u kom browseru se pokrece.

* Ovo je **preventDefault** deo sintetickog dogadjaja:

![A computer code with text

AI-generated content may be incorrect.](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4RDuRXhpZgAATU0AKgAAAAgABAE7AAIAAAAMAAAISodpAAQAAAABAAAIVpydAAEAAAAYAAAQzuocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEVzaG5hIFZlcm1hAAAFkAMAAgAAABQAABCkkAQAAgAAABQAABC4kpEAAgAAAAM0MgAAkpIAAgAAAAM0MgAA6hwABwAACAwAAAiYAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAxOToxMjozMSAxNDowMToyMgAyMDE5OjEyOjMxIDE0OjAxOjIyAAAARQBzAGgAbgBhACAAVgBlAHIAbQBhAAAA/+ELHmh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8APD94cGFja2V0IGJlZ2luPSfvu78nIGlkPSdXNU0wTXBDZWhpSHpyZVN6TlRjemtjOWQnPz4NCjx4OnhtcG1ldGEgeG1sbnM6eD0iYWRvYmU6bnM6bWV0YS8iPjxyZGY6UkRGIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iLz48cmRmOkRlc2NyaXB0aW9uIHJkZjphYm91dD0idXVpZDpmYWY1YmRkNS1iYTNkLTExZGEtYWQzMS1kMzNkNzUxODJmMWIiIHhtbG5zOnhtcD0iaHR0cDovL25zLmFkb2JlLmNvbS94YXAvMS4wLyI+PHhtcDpDcmVhdGVEYXRlPjIwMTktMTItMzFUMTQ6MDE6MjIuNDIxPC94bXA6Q3JlYXRlRGF0ZT48L3JkZjpEZXNjcmlwdGlvbj48cmRmOkRlc2NyaXB0aW9uIHJkZjphYm91dD0idXVpZDpmYWY1YmRkNS1iYTNkLTExZGEtYWQzMS1kMzNkNzUxODJmMWIiIHhtbG5zOmRjPSJodHRwOi8vcHVybC5vcmcvZGMvZWxlbWVudHMvMS4xLyI+PGRjOmNyZWF0b3I+PHJkZjpTZXEgeG1sbnM6cmRmPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5LzAyLzIyLXJkZi1zeW50YXgtbnMjIj48cmRmOmxpPkVzaG5hIFZlcm1hPC9yZGY6bGk+PC9yZGY6U2VxPg0KCQkJPC9kYzpjcmVhdG9yPjwvcmRmOkRlc2NyaXB0aW9uPjwvcmRmOlJERj48L3g6eG1wbWV0YT4NCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgPD94cGFja2V0IGVuZD0ndyc/Pv/bAEMABwUFBgUEBwYFBggHBwgKEQsKCQkKFQ8QDBEYFRoZGBUYFxseJyEbHSUdFxgiLiIlKCkrLCsaIC8zLyoyJyorKv/bAEMBBwgICgkKFAsLFCocGBwqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKv/AABEIAOEBWQMBIgACEQEDEQH/xAAfAAABBQEBAQEBAQAAAAAAAAAAAQIDBAUGBwgJCgv/xAC1EAACAQMDAgQDBQUEBAAAAX0BAgMABBEFEiExQQYTUWEHInEUMoGRoQgjQrHBFVLR8CQzYnKCCQoWFxgZGiUmJygpKjQ1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4eLj5OXm5+jp6vHy8/T19vf4+fr/xAAfAQADAQEBAQEBAQEBAAAAAAAAAQIDBAUGBwgJCgv/xAC1EQACAQIEBAMEBwUEBAABAncAAQIDEQQFITEGEkFRB2FxEyIygQgUQpGhscEJIzNS8BVictEKFiQ04SXxFxgZGiYnKCkqNTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqCg4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2dri4+Tl5ufo6ery8/T19vf4+fr/2gAMAwEAAhEDEQA/APo2eZbe2lmcErGhcgdcAZrE/wCEw0//AJ43P/fK/wDxVauqf8ge8/64P/6Caq+Gv+Retv8Agf8A6GaAKn/CYaf/AM8bn/vlf/iqP+Ew0/8A543P/fK//FVtFFe4beobCjGRnuad5EX/ADyT/vkUAYf/AAmGn/8APG5/75X/AOKo/wCEw0//AJ43P/fK/wDxVbnkRf8APJP++RR5EX/PJP8AvkUAYf8AwmGn/wDPG5/75X/4qj/hMNP/AOeNz/3yv/xVbnkRf88k/wC+RR5EX/PJP++RQBh/8Jhp/wDzxuf++V/+Ko/4TDT/APnjc/8AfK//ABVbnkRf88k/75FHkRf88k/75FAGH/wmGn/88bn/AL5X/wCKo/4TDT/+eNz/AN8r/wDFVueRF/zyT/vkUeRF/wA8k/75FAGH/wAJhp//ADxuf++V/wDiqP8AhMNP/wCeNz/3yv8A8VW55EX/ADyT/vkUeRF/zyT/AL5FAGH/AMJhp/8Azxuf++V/+Ko/4TDT/wDnjc/98r/8VW55EX/PJP8AvkUeRF/zyT/vkUAYf/CYaf8A88bn/vlf/iqP+Ew0/wD543P/AHyv/wAVW55EX/PJP++RR5EX/PJP++RQBh/8Jhp//PG5/wC+V/8AiqP+Ew0//njc/wDfK/8AxVbnkRf88k/75FHkRf8APJP++RQBh/8ACYaf/wA8bn/vlf8A4qj/AITDT/8Anjc/98r/APFVueRF/wA8k/75FHkRf88k/wC+RQBh/wDCYaf/AM8bn/vlf/iqP+Ew0/8A543P/fK//FVueRF/zyT/AL5FHkRf88k/75FAGH/wmGn/APPG5/75X/4qj/hMNP8A+eNz/wB8r/8AFVueRF/zyT/vkUeRF/zyT/vkUAYf/CYaf/zxuf8Avlf/AIqj/hMNP/543P8A3yv/AMVW55EX/PJP++RR5EX/ADyT/vkUAYf/AAmGn/8APG5/75X/AOKo/wCEw0//AJ43P/fK/wDxVbnkRf8APJP++RR5EX/PJP8AvkUAYf8AwmGn/wDPG5/75X/4qj/hMNP/AOeNz/3yv/xVbnkRf88k/wC+RR5EX/PJP++RQBh/8Jhp/wDzxuf++V/+Ko/4TDT/APnjc/8AfK//ABVbnkRf88k/75FHkRf88k/75FAGH/wmGn/88bn/AL5X/wCKo/4TDT/+eNz/AN8r/wDFVueRF/zyT/vkUeRF/wA8k/75FAGH/wAJhp//ADxuf++V/wDiqP8AhMNP/wCeNz/3yv8A8VW55EX/ADyT/vkUeRF/zyT/AL5FAGH/AMJhp/8Azxuf++V/+Ko/4TDT/wDnjc/98r/8VW55EX/PJP8AvkUeRF/zyT/vkUAYf/CYaf8A88bn/vlf/iqP+Ew0/wD543P/AHyv/wAVW55EX/PJP++RVe5YW7xmJVUtkHA+lAGfB4qsbi5ihSK4DSOEBKrjJOPWtusDxB/yGNG/67/+zJW/QBV1T/kD3n/XB/8A0E1V8Nf8i9bf8D/9DNWtU/5A95/1wf8A9BNVfDX/ACL1t/wP/wBDNAFXxBrf9lXdpbxyLFLeOEWR4mkCABiThfoB+NaY1K0FlJdNcL5ULFJJMEBWBwR+dVtQspbnWdNmiAKW03mSZPQGORf5kVk3ul6r/Z+p6bbWsckd1M88dwZQBh33FSOuRz7VuowkkjmlKpGUna66fcbVzrum2czxXN2qPGwVxgnYSMjOBwMHqeKmvNTs9PjV7udUD/dABYt9AMk9RVAaXN9o1x2RSL3b5RyOQIgvPpyDUMVhf6eLG5htxdyRWKW0sHmhdrDHKk8eueewpcsO4+eor6f1f8Sa58UadbX9lbmQut4jukkaMwG0gY4B7n8MVaTVLaGwa5uruJo/NdA6qRkhiNoHJJGCOOuKqXNtfNe6dfraRtJAJUkgSQcByvOTjpt5+tVoNJvrS1tpkjWWa2uriXyDIAHWR2IIPTdgjr6mnywsv67k89RSf+Xp/wAE1RrWnG1S4+1x+U7iMMcj5sZ2kdQcdjUtjqNrqULS2UwlVW2twQVPoQeRWI+kXt1cx3k0KxM97FM8HmBhGiIwz6bsnnHoK07Sylh1/ULplAhuI4VQg9Su/PH4iplGCWjLjOo3qtCbUr02Nk0qIJJWISKMtje56DNFhfpe6al22Ihg+YrH/VsOGB+hBH4VS1TSrrVNSgIvbiygtl3o9uU3PIcjkMrDAHt39qrR+H7uCHUrMahPcwXqeYJZyodJehHyqBtIC9B13dc0KMOXV6icqnPotP6/4Yut4i037PcSRXAkNvG0hUKcsF6kcfMORyM9abpuu2+rNZyWs6qs8LuYHjYOcbMkE44G4Z9cj0oiXULqCW2ubCK1TyWj8zzQ2SQANoA6deuOg4qlZaXfG70triHyEs7Ke1dhICSWMW1hj12N9Krlhb/g+RLlUuu3o+6NWHWdPuLz7LDdI02cBecMRnIB6E8HgelNi13TJ7pLeK7RpZGZVGDgsCQRnGM8HjrxWZDpuoPDY6fNbpFDZTRyG7WYFpAhyMDGctgBs44LUg0S6Hh+3tRGgmj1KO5b5h9wXIkJz67c/wAqOSHcPaVe34fh/wAE17e/M+r3lkYwBbLGwfP3twP+FJNqUFrcXH2q7iSKFYyylSDHuOASfQms+RdTsfEN7c22mm7huI4grLMiYKg54J96r32k32oNfStbrGbpbTEbODjZJucHHHT86FGN9Xpp+g3UnbRa6/rb9DSm120Om6hcWUizSWUTyNGQVOQpI6jocdauRXsTrEHdVkki8zZnoMcn6VlavpNzd3V5Jbou2bS5bZeQMux4H/16ba6JdWy3ELTGaO9ttksrt+8ikCbfl/2cdAOh570uWFtw5qilaxfh1Ce6s3ura23R8GEM2GkXucdvarVpdR3trHcQ52OMjIwR7Gs7T9Tjg0Um6jkils1Ec0W0lgwHGPXPYiptDtpLfSlFwrJJK7zMjHJTexbb+GcVMopJlwk21r0JLe/a61K6t4kAjtWEcjk8lioYYHphqu1j6Qps9R1CzcOFMwkgZyWLqUXJ3HJOGyOT2rYpTST0KpttahRRRUGgUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFU9Q6xfU/0q5VPUOsX1P9KAMzxB/wAhjRv+u/8A7Mlb9YHiD/kMaN/13/8AZkrfoAq6p/yB7z/rg/8A6Caq+Gv+Retv+B/+hmrWqf8AIHvP+uD/APoJqr4a/wCRetv+B/8AoZoA0R/x8P8A7q/zNPpg/wCPh/8AdX+Zp9AFbUL6PTrJrmZWYAqoVerMzBVA9ySBU0EjSwI8kTRMwyY2IJX2OOKxfF9lFf6LFHND5oF7bEDHT98gP6E1Va4i0fUNVtZBJGs0Sm0RVLbwI8EL75HStowUo3W//DHPKo4zs9v+H/yOnorkdO09dRuLCLUEkkh/saDcjMQC2T196WN5m0PTxqbTNYpLPHck7txVWZU3Y5K4HJPtQ6XS4lXbV7f1p/mdRPcxW5jEzhPNcRpn+Jjk4/Q0scjPJKrRMgRgFY4w4wDkfnjn0rktUstJutKspIIXksLe+V3L7yqpsIJXP8PTpxUuJhDqwtfN+y/boc+Xu3eR5EWdmOf8mn7NWF7Zp7f1Y6yiuKuxG2n3w0AyjTzAgJTfgS+aPuZ9t2ce1bFraLp3iqOKzDpb3NnJLMu4lWkV0AbnocM31/Ck6dluVGs29v6ZrXF5DazW0UxIa5l8qPAzltrN/JTU9YniG2F3eaJG6s0f24l9pI48iXr7VTh8zTrHWraGF3t4LhUhjYkhEZELH1KgsxwD2wKSgmlb+tbDdRqTTWn/AALnT0V57tuFtruzt7hZbUyWDxvZwvEgLXQDbcsx6AZwa3ha3Gn6jqttoy7N1iJYEckp55LgH9BmqlSt1/rT/MiNdy6f1r/kbtxeQ2stvHMSGuZPKjwM5baW/kppLK9iv4XlgztSWSI5GPmRip/UGuTiXT31TRDpouTMt1m58wNkHyX/ANZn+KiSG8/sMeXKlvbjUrtrgzQPIpUzSYyqspI/HFP2S/r5iVeV27f1p/mdddXcFlAZrqQRoCBk9yegHqapR6/ZvDNIy3ERijaUpNCyMyqMkjI5rCsoZYY4bwT/AG6zguA3l21o8aIuzG5FZmJAyDwfX3rTvtYs9T0bUIrBnuD9lly6Rnap2Hgn156UvZpabj9s2r7eRqwXH2hgVhcRtEsiynGDnPHrkcH8aW8u4rCylurgkRRLuYgZ4rmkF8t1eizDiQaLa+WDnG7fNnHvjH6VBfLpbaDqI0hbh7k2r+avz4LfL9/PG7689fen7JcwOs1G/qdkCCAe2M1Fb3UV0rtA25VbbuxwT7etc+xvzbapHcKV1E2zNbkZ8ry8cBfcHg9+/TFSaoJZ/DNkdFmjitg8ZkPlNIPKxyNqspPO3Iz0BqPZ+ZXtXa6Ru3N1FaRCSclUyAWxkL7n0HvUoIIBHIPQ1zfh1UigvJHvreey2D5YrV4YUGDkje7Z98YrR8PCT/hHbXdgZQmPg8Jk7OP93bSlBRKhUcrablyO8hmupIIiWeLhyBwp64z64I4qesfw9kDUUlIa4W9YTuo2q77E5VSTgY28ZPTrWxUyVnYuEnKN2FFFFSWFFFFABRRRQAUUUUAFFFFABRRRQAVT1DrF9T/SrlU9Q6xfU/0oAzPEH/IY0b/rv/7Mlb9YHiD/AJDGjf8AXf8A9mSt+gCrqn/IHvP+uD/+gmqvhr/kXrb/AIH/AOhmrWqf8ge8/wCuD/8AoJqr4a/5F62/4H/6GaANEf8AHw/+6v8AM0+mD/j4f/dX+Zp9ABRRRQAUVXv5J4dNuZbOMS3CRM0SE4DMAcD8TXOaReaj9rtpbq8s/JlQvMDfmQsAvVU8tduD154q4wck2ZSqKMlF9Tq6K5ue4N14mvrKbVJLWBbeGSNY3CHJLZIb8BVaO+v7u5sbaK+byWvJ4Rcpg+fGqZzxxkHK5/2c1fsn3IddLp5fjY62iuT1K51UatdQpNDDDbIhhaW/MJYbeXI8tt3ORyf4elWZtTvtPC/aJEme+t0+zmMgoJ+AVBOMg5DD/dal7J6B7eN3psdHRXJXs+rrqUtsZ4US3ijaN5L4wFzt+ZyPLbcM5GM9h6022fUr6PVJJ9TdHtBG8f2ZgUJ8lWPUfMpJzin7J2vcX1hXtZnXF1DhCwDMCQM8nHX+dUW13SkvWs31G2W5VtrRGUbgeuMVhXCHUdb8PXs91NbedYSs3lSbF3Hyjj8cnjvj2qxp8+nxjXBqUluIvtkhkEpGNu1c5Bp+zSV3/Wtg9s27LT/hrnSVBeXsFhbme6YqgIA2qWJJ6AAZJPsK5uSbU7bQdKhhKqszMrvNMYCqcmNS21iDjHbORVO7hvL2KxivL4AR6lGiG0uzKyfu2yGbavPfp3ojR11egpYjTRanaxuJI1dc4YAjcCD+R6U6uR8Q6jeWn2uewmZRpkXzma68sO20MPk2HfnI5yOc1Jca1LBa+JTPd+VLAFe2VmwVU26EFR6b9/45pexbV1/W3+Y3iIptPp/wf8jpPtcH2qS38xfNjjWV1/uqSQD+at+VPaaNITK7qsYGdxPGPrXK3KNdahrCl3DyaDb/ADIcNndcU7TYQE0KzeeS4tnieVmll3fvFVNsf0wXOP8AZp+zXf8Aq1wVZt7f1ex06TxSOUjkVmUAkA8gHpTba0gtFdbaMRq7FyF6ZPU1n6yPJuNOuYB/pP2pIgAcb0bhwfUBctj/AGa0bcAQ4ExmGT8xOe/SsmrK6Nk7ys+gl1aQXsPlXUYkj3BijdCRyM+tTdKKKm5Vle5EtrClw86Rqsrj5mH8X1qWiigLWCiiigYUUUUAFFFFABRRRQAUUUUAFFFFABVPUOsX1P8ASrlU9Q6xfU/0oAzPEH/IY0b/AK7/APsyVv1geIP+Qxo3/Xf/ANmSt+gCrqn/ACB7z/rg/wD6Caq+Gv8AkXrb/gf/AKGatap/yB7z/rg//oJqr4a/5F62/wCB/wDoZoA0R/x8P/ur/M0+mD/j4f8A3V/mafQAUUUUAFQJY2kbyvHawq83+tYRgF/r6/jRf3QsdOubsxtIIImlKIMs20E4A9eKzVudSs7GXUb2aG5hW3aYwQRHdkLnCHPzVcYtrQzlKKdmiy2jWkt7cz3MSTrcJGhilQMo2ZxwfrV3yYsofLTMf3PlHy/T0rGgvNSt7ixa9mhnjviF2QxkeUdu7IOeV46n2qlpWsapLp+h6heywSR6ksavEke3YzLncDn26VfJJq9/6/pGaqQTtb+v6Z0c9nbXTI1zbxTGM5QyIG2n1GelSMiMVLKp2nK5HQ+1Y0eoXkesmPUGNtE8xigQxZSUdVIcHgkdiBz61Auq6ibFNXLR/ZXcD7J5R3hS+Oufve1Lkl3H7WPY25ba0vtpnhguPKf5d6B9jD0z0NSCGIb8RoPM+/8AKPm4xz68cVy0Oo3Gn6ZP9jt3mln1eeIbACVzIxzgkDtjr1IrX0W61G489NStpIhHt8uSVUVpMjn5VZgMY/WiUGluEKkZNK2rLzW1pdQIjQwTRRkFFKBlUjpj0xTDpWnm8+1Gxtjc5z53krvz67sZrn9IurldJ0+xsmWKW5kuCZnTcqBXPbuxJGB6A+lWZdZvLaGS3naITpdR2xumXbGAwyGIz17Y9SKbhJOyZKqwaTa/q1zelhjniaKeNZI2GGR1BBHuDTEtLaKKOKOCJI4zlEVAAh9QO1c+NavrK81KG4nhvlstPNyPITDMw3cEAn0HFadppimxtzLKZZWZJppT/wAtWA/QZ6DtScHFasuNRTeiLk1jaXEolntYZZApUO8YY4PUZPaiWxtLiRZJrWGV1UqrPGCQD1APpWFdQ3VjqA0qxkMdvqTO6vuy1ufvSBfrkkehNNjur230+4nsPKgsdNd4EtWiO51j+U854JwSPUEetPkfRke0V2nH1OjWGNZC6xoHKhCwUZKjOB9Bk8e5qrcaRZT2whWFYAsnmo0ICFH/ALwx3/nWcutXDW+vyfKv2Bh5IK9AbeOTn/gTGtizlaexglfG6SNWOPUjNS1KOpopQnp/W5XjtLW41AXvm/aJYAYl+cMsTdGwB0bse/arUEEdvF5cQwuSfxNYMes3P2q1TEYWbV5bR8L1RY5WH45ReanfUr921hLOJZpLSWNIkGASCqluvBPJIHHpTcJbERqQtdL+rXNuiuS1m8u7jw0ywagyXUN/apKxg2Mu6ZBgrn3B44I4710U08mnaPLcXRa6kt4Wkfy0wZNoJwF9eOlJ02kXGqm2rdL/ANfcWmZUQs7BVUZJJwAKas8TzNEsqNIqhmQMMgHocehwfyrm9WGqP4T1Ke7uraSKXTpmMccZGwmI/dbPI+tTSaxNZy3yiNXW2srd4wF5LOWXk+nAP50/ZtrQn2yT1Vv6f+R0JIAyTge9Fcxr66hB4dvHv7uGaIqrYSLaU+deBzyOtWl12WTT9RvYYsrZgotsR+83AA5YdsgggehzR7N2ug9slKzVv6f+Ru0EgYyevSuXTW9Yto3nurCd4PKeQs8ca7CACoG12LDr2pLoX66/4dN9cwzCS7kdRGm0r/o0vA55HPWn7J9WL266J9PxZ1NFUdXvGtLL9y5W4mby4FVAzO+CcAEgE4BPJHSqugandXxu4L6Jo5rV1Ul1ClsrnkAkD86jkfLzGjqJT5TYpizRNM0SyIZEALIGG5QehI/A1lWH/I16v/1zg/k1VdW1i7s7jUlthFugW18ssv8Az0k2nP4U1Tbdl5fiS6qUeZ+f4X/yOiJABJ4A6miuY1W6vYrXWNPvpY5w+mTXEbomwqACCpHfqOau6Lqhv5fJiHlxW8MeVkXDyBl4cDsvUZ7kGm6bUeYFWTlym1RVWSxjmu5ZphvEkIi2EcAZJP55H5VU0qS4b7RbeaWWzujFvkXJdNgYAH23AZ/2TUct1cvmadmjVqnqHWL6n+lXKp6h1i+p/pUlmZ4g/wCQxo3/AF3/APZkrfrA8Qf8hjRv+u//ALMlb9AFXVP+QPef9cH/APQTVXw1/wAi9bf8D/8AQzVrVP8AkD3n/XB//QTVXw1/yL1t/wAD/wDQzQBoj/j4f/dX+Zp9MH/Hw/8Aur/M0+gAorO10TppM9xZOyXMCGSP5yFJHZh0I+tXbe4jurdJ4SSjjKkqVP5Gqtpcnm97lJCARg8g1QtND06xn821tgjAbVG5isY9FUnCjjoAKv0Uk2tgcU3dooWeh6bYXHnWlqsbgbU+YkRj0QE4Qey4qSPS7KK0tLaOALDZ7fIXcfk2jA788etW6KfNJ7sShFbIprpVml59qWNhJnIHmNsBznITO0HnrjNRjQtNF99rFqvm7t+Nx2b853bM7d2f4sZ960KKOaXcOSPYqnTLM2stuYFMU0jSOpJ5ZjknPY554plto9jZqoghIKyCTczszMwG0EsSSeDjkmrtFHNLuPkje9ijJo2ny2K2j2w8lGLoAxDIxzllYHIPJ5B70h0uODTWtNPjhjVjlllXeJPXdnkk+pzV+ijml3FyR7GTpWjfY5JJZorWMvGIlhtY9saIMnA+uataZYNptr9lEzSwRnEAf70adlJ747HrirlFDm3uEacY2sRSWsMtzDcSJmWHPltk/LkYNVJ9C025vDdTWqtKxDN8xCuRgAsoOGIwMEg4wK0KKSk1sxuMXujPvNB03ULhp7u1DyPH5bkOyh154YA4bGTjOcZ4q9HGsUKxxDCooVR6AdKdRQ5NqzBRindIyLfQom06e21MJP59y9wwQldjFsjaRyCPUc1Zh0XT7eGaKG3CrOwaT5mJZgAA2Sc54HPWr1FU5yfUlU4LoUl0iyWxe0MReGT7/mSM7MexLElsjsc8VZggjtrdIIt2xBgb3LHHuTkn8akoqXJvcpRitkZa+G9JRJEWzXZJG0RTe21EYYKqucICOPlxVv8As+0MkztArGaJYZN3IdBnCkHjHzH86s0U3KT3YlCC2RmJ4d0tFkUWu4SDB3yO2BkHC5PyjIHAwOKuGytjdNcGFfNePynbH3kzkA+uMn8z61PRQ5Se7GoRWyM+20PTrWVpIrYFmVl/eOz7VOMqoYnaDgcDA4ptv4f0y1uIpoLbEkLl4iZGbyyVK4XJ4GGIwOOelaVFHPLuL2cOyILuzgvoPJuULLnIIYqyn1DDBB9xTLLTbTT2la0i2NMQZGLFi5AwCSSSTjvVqild2sPlje9tTMvPD2nX1411Ok4mdQrNFdSxbgOmQrAVINFsPLZDAWDrGrF5GYsIzlMknJwav0U+eW1xezhe9kVrjTrW6eR7iEO0kLW7nJGY26rQ+n2sjxu0I3RxmJWBIIQjlc+nAqzRS5mPlj2ImiZbUxWzCNgu1CRkL6fWqlrpMdtBBGJZHaOUzO7HmRznJP5nitCijmdrA4pu4VT1DrF9T/SrlU9Q6xfU/wBKRRmeIP8AkMaN/wBd/wD2ZK36wPEH/IY0b/rv/wCzJW/QBV1T/kD3n/XB/wD0E1V8Nf8AIvW3/A//AEM1a1T/AJA95/1wf/0E1V8Nf8i9bf8AA/8A0M0AaI/4+H/3V/mafTB/x8P/ALq/zNPoAoaxZS6lp0llGVRJxslcnlV74Hc1fqjqmrQ6V9k86OSQ3U/kRiMZO7Yzj/0Aj6kVVuvElvaXAt3iYzhVaWMyxqYwfXcwyfYZNaKMmkkZOcIybb1Niiueg8RzXevxW9pZyTWMtqswmUqNuWIJOWzgYxjGc1fh1y3ns9LuVjkC6mVEQIGVyhf5ufQdqHTkgjVhLZmlRWIfFFuLO5uha3Jhtp2gdgo++JfLwOeeefp+VW7LVhdX0tpNbTWs6LvVJcfOmcbgQT37daThJK7Q1Vg3ZM0KKpaleSWyQxWwU3FxJ5ce7oOMlj64AzircSskSrI5kYDBcgDcfXiptpcvmu7DqKKKQwooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKp6h1i+p/pVyqeodYvqf6UAZniD/kMaN/13/9mSt+sDxB/wAhjRv+u/8A7Mlb9AFXVP8AkD3n/XB//QTVXw1/yL1t/wAD/wDQzVrVP+QPef8AXB//AEE1V8Nf8i9bf8D/APQzQBoj/j4f/dX+Zp9MH/Hw/wDur/M0+gClf6f9uudPl8zZ9iufPxtzv/dumPb7+c+1VdQ0Rrm9N1Zy28UsgCy+faiYMB6cjB9+R7Vr0VSnJbEOnGW5lDR5Yr+2ube6VTHCIJg8IPmoCTxggKck84I9qqWnhu5t206NtSD2mmy7reLyMNt2MgVm3c43dcDpznrXQEgAknAHU1Ti1OC4u/IthJNgfNIi/u19i3TPsM1SnO2hDp001cpnQM6LLYfaf9Zdtc+Z5fTM/m7cZ/DP4+1XDYZ1tdQ8zpbmDy9vqwbOfw6U+LUrKe7ktYbqF54/vxq4LL9RUGo6tFpt1aRz7VjuC4aVn2hAq5zRebdgtTir+gmr2s80cFxZKr3NpJ5iI3G8Ywy57Egnmr8bmSJXZGQsASrdR7VAuo2T2sdyl1CYJCFSQONrE9gaSXVLCCBJpryFIpPuOzgBu3FL3mrWKvFNyuWqKw9T8RQ22r6fp1vd2kct2jSl5jkbAVGAARyxbj/dNWx4h0c3QthqdqZy2zy/NG7d6Y9aPZytewe1hdq+xo0VQj1Npb4W6QfL5roX39Ao5OMepAxWZdeN9HtprmJrqEtbyxxf60fOWIBx/u55+hoVObdkhSrU4q7djoqKzotV8+8EMMIdDIU8wPxgKGJ6e4H41cuLmG0haa6lSGNeru2AKlxa0LUk1dEtFVV1Oxa1S5W7hMMjBUk3jaxPQZpE1WxlsJL2C5jmt4wdzxNuAx1HHf2o5X2Dmj3LdFZMGpapJIkkmjbLSTG1hcgzKD3ZMAAfRifanWmqvd4y0MGL6W22uSTIE3fd/wBr5c9+AarkZKqRZqUVVbU7Fb0WbXcIuTyIi43flUWm6rHfafFcyhYDLI0aqWzkhiOPypcrtcrnje1y/RWcfEGkLdfZjqVqJ9+zy/NG7dnGMetSvf41VLCGPfJ5fmyMzYCJnHHHJz2o5Zdhc8ejLlFZena/aXrGKSWKG48+aJYTICzCORkyB77c1bm1KytrpLa4uoY5pPuRu4DN9BQ4yTtYFOLV0yzRVOXV9PguDBNewJKHEZRpACGIDBceuCD+NSXd/aWKq17cxQKxwpkcLk0uV9h80e5YorFh8QGf7EUhULc38toTvzgIJDuH18v9a0TqVkLz7IbuH7RjPlbxu646U3CSFGpGSumWaKyX1W+nncaVpq3METlJJZrjytxHXYNp3fjtGe9JLrMgnmhdEsylh9qL3J4iOWHz4OMDGTg/jT5JC9rE16Kq3Oo2likZvrqGAvgLvcLuPtmoodVjm1K6t8KIreKOXzt/DBt3/wATS5Xa5XPFO1y/RVI6iq6jBbOo23KloJFbO7AycjHHH1pJdWt7a8NvdrJbjIEcsi/u5OB0YcDrjBweDxjmjlYuePcvUUAgjIOQe9FSWFU9Q6xfU/0q5VPUOsX1P9KAMzxB/wAhjRv+u/8A7Mlb9YHiD/kMaN/13/8AZkrfoAq6p/yB7z/rg/8A6Caq+Gv+Retv+B/+hmrWqf8AIHvP+uD/APoJqr4a/wCRetv+B/8AoZoA0R/x8P8A7q/zNPpg/wCPh/8AdX+Zp9ABRRRQA2WNJoXilUMjqVZT3B6iqNjpjadIqWty/wBjVcC3cbtnptPUD2rQopptKxLim7nJRaZqsmqafPcQXHmwXO64lM6eSw2MCY0BzjJHBwfrWzqVjJdaxpUyxB4reSRpCSPlymBwfetSitHUbdzONGMU18zlptCuZftsRtgYJdZjuUXcoBiEce44zx8wbjqeT3qxrOlTtqsd/ax3cyiAQGG0nWJk+YnI3EAg5557DrV+51uOK7e2trS6vpYiBMLZARFkZAYsQM4wcDJwRxyKcusRy3FtDBFIXubeSaMSKY8bCoIIIyDlx2q+aejsZ8lLVX/q/wDwTNsNGmtdU0mQW+yG3tLlJN0okKO7xMozwT91ugwMfSro0q8F4Jjq8xj37vK8mPGM9M7c1pozGJWlUIxAyuc4PpmoFvon1SSwAbzY4UmJxxtZmUfjlDUOcmaKnCOn9bf8AsFeuOD64rkh4Z1drhGfV5tr3ZlkJiiOFCnaenXOOK66qOq6xaaMls98zKlzcLbqwXIVmBIJ9Bx17UqcpJ2iFWEJK83t5l0KB259cVmaxazyyWlzbwfavsshdrfcF35GMjPG4dskdetT3mrWtlqFlZTFvPvXKRKq56KWJJ7DAq7kZxnmpV4tMt8s043OXk0i8vJhdPZCBZr2CZrUup2KinLNjjccjIBPQVbm0aW41LWMIsUF5bwCNxjBkUvkkDn+59R9K3aKv2sjNUI/18/8zJgvtXaRLeXSCjLgPcG4TyT6lcHefoVH1qjDpN6lxZs0OBHrFxcud44jZZQrde5ZeOvNdJRS9pbZDdJPdtmBZW15ZYsZNKW5XzjJ9saRNh+fO4j728DnhccDmq2j6PqOnSWtxMgm+eRJIGYfuVZyQ6HpnnkdSPpg9RRT9q7PzF7GN077GU2lXhvPO/teYR793leTHjGemduajj/0TxZKJCdlzbgo8hz8wY5RT9OcVs0Yz1qed9S/ZrdHMf2JcjQniW2UXJ1VrkfMudhuS+7Of7n49vameI9N1O/+3xQwXEvmQkWrQzpHGp29Hyck7s4IGOnSuqoq1VknczeHi48tzmW0W5e08Rh7ZWlvSptySpL4t40HOePnU9cdM1Ymtby11GC9SxOoA2iW7RCRA0TAklhuIBBzzzn5Rwa3qKXtH1K9jFbf1rf9Tl9P0e+gg0tZLZYzBqlxcSKjghI2E20j1++vHXmoV8P3Ecxs5IL2eFpvN877Yqw/fzyv3tw68DHHUV11FP20r3J+rwtb+v60MOA6jo+60h0x72BpWaKWGZF2AnPzhyD1J+7n6VU1nS9Sv4dTYW6ebc6M9sqpICDKd/ygnHqOTgV09FJVGne2pToprlbdjDmt7uz1VruOwbUEmhSLajoGhx1++QNp6nBz7VRl0LUJr6e4jjitlEFv5dsGBikZN2YyQAdvIGcDscdq6qihVWhOjF7sxLwm+1bTIApieMGWYI43wfLwpI7E8e9WrzTH1GSRL24LWbcC3Rdu4YGdx6nnPTHBrRxRU870sX7Na36jY41iiWONdqIAqgdhTqKKg0CqeodYvqf6VcqnqHWL6n+lAGZ4g/5DGjf9d/8A2ZK36wPEH/IY0b/rv/7Mlb9AFXVP+QPef9cH/wDQTVXw1/yL1t/wP/0M1a1T/kD3n/XB/wD0E1V8Nf8AIvW3/A//AEM0AaI/4+H/AN1f5mn0wf8AHw/+6v8AM0+gAooooAKKKa0iK6ozqGb7qk8n6UAOoqCa7WG6gg2lnmJxj+EAck1JLNHBGZJ5FjRerOwAH4mnZiujFt7ptEmu4Ly1upEluXminhgaUOHO7B2glSOnPGAOag1RLi9vYLhbOdA2l3ilWXJVmMW1TjIycHj2NdIDkZHIorRTs721MnSvHlvocpLaqv2E6zYzXdgtiiLEsDSiOXPJZBkkkbcHHG1umajks799TnbSbeW2hOnQqBNkM2JZSYw3OCQfwyK6+in7VkewXf8Az+8gsShsYTFC8CbBiN1wy+xFZ3iCz+2zaTE0Bnh+2nzl27l2GGUHd7cgc+tWLjRLS6uGmlkvQzdRHfzov4KrgD8qs2dlFYw+VA0zLnP76d5T+bkmpUkndGjjKS5Xt/Xkc++hyWeoaVIXku3S8wZSpJjiEUgUE+gyOT1JqhdaXctqlwLiS4NzJcFoZorHeyoWGAJc4UAYyCR0PBrtqKpVpLUzeHi1YKKKKxOkKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACqeodYvqf6VcqnqHWL6n+lAGZ4g/wCQxo3/AF3/APZkrfrA8Qf8hjRv+u//ALMlb9AFXVP+QPef9cH/APQTVXw1/wAi9bf8D/8AQzVrVP8AkD3n/XB//QTVXw1/yL1t/wAD/wDQzQBoj/j4f/dX+Zp9MH/Hw/8Aur/M0+gAopGdV27mC7jgZPU+lLQAyZXeF1ify3ZSFcDO044OKyNKsp7G8P221WedlwdQDbi/sc8r9BxW1TZY1mieOTJV1KtgkcH3HSqUrKxEoptPsYYuN9rea1saRXXyrfy+vlZxuB9CTuz6AGs3RL2WLUZbPzRqFs1qZgkV091k7wvDv9TkZPSuuSNI41jRQqKNoUDgD0pVRU+6oX6CtPaKzVjJ0m2ncx9H0y5tLpphiztGXC2CNvVTgcjPC4wRtXjv1rZoorOUnJ3ZtGKgrIKKKKkoKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACqeodYvqf6VcqnqHWL6n+lAGZ4g/wCQxo3/AF3/APZkrfrA8Qf8hjRv+u//ALMlb9AFXVP+QPef9cH/APQTVXw1/wAi9bf8D/8AQzVrVP8AkD3n/XB//QTVXw1/yL1t/wAD/wDQzQBoj/j4f/dX+Zp9MH/Hw/8Aur/M0+gCC9toru0eKYfKRkHup7EehFVNFnnutPt7ieUktDgxsoySCRvz7j8Ks39mb61NuZpIUcjeY+GZe657Z9RzSw2UMEqvEu3ZEIlUdFUdqu65bGbT57liiiioNAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKp6h1i+p/pVyqeodYvqf6UAZniD/kMaN/13/9mSt+sDxB/wAhjRv+u/8A7Mlb9AFXVP8AkD3n/XB//QTVXw1/yL1t/wAD/wDQzVrVP+QPef8AXB//AEE1V8Nf8i9bf8D/APQzQBFrbNbSQ3sTFWikiD7cktGWO5QB1J9OtbCMHQMMgEZGQQfyNUL2xa+voN7qsMDpMVA+ZmUkjn0rQqm1yoiKfM30CiiipLCiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKp6h1i+p/pVyqeodYvqf6UAZniD/kMaN/13/wDZkrfrA8Qf8hjRv+u//syVv0AVdU/5A95/1wf/ANBNVfDX/IvW3/A//QzV68ha4sLiFCA0kTICemSMVjWen67Y2iW8E1l5aZxu3E8nPp70AbRdEuG3sq5UYycdzTvPi/56p/30Kx5LbxBKm15rHH0b/Cof7M1v/ntZf+Pf4UAb3nxf89U/76FHnxf89U/76FYP9ma3/wA9rL/x7/Cj+zNb/wCe1l/49/hQBvefF/z1T/voUefF/wA9U/76FYP9ma3/AM9rL/x7/Cj+zNb/AOe1l/49/hQBvefF/wA9U/76FHnxf89U/wC+hWD/AGZrf/Pay/8AHv8ACj+zNb/57WX/AI9/hQBvefF/z1T/AL6FHnxf89U/76FYP9ma3/z2sv8Ax7/Cj+zNb/57WX/j3+FAG958X/PVP++hR58X/PVP++hWD/Zmt/8APay/8e/wo/szW/8AntZf+Pf4UAb3nxf89U/76FHnxf8APVP++hWD/Zmt/wDPay/8e/wo/szW/wDntZf+Pf4UAb3nxf8APVP++hR58X/PVP8AvoVg/wBma3/z2sv/AB7/AAo/szW/+e1l/wCPf4UAb3nxf89U/wC+hR58X/PVP++hWD/Zmt/89rL/AMe/wo/szW/+e1l/49/hQBvefF/z1T/voUefF/z1T/voVg/2Zrf/AD2sv/Hv8KP7M1v/AJ7WX/j3+FAG958X/PVP++hR58X/AD1T/voVg/2Zrf8Az2sv/Hv8KP7M1v8A57WX/j3+FAG958X/AD1T/voUefF/z1T/AL6FYP8AZmt/89rL/wAe/wAKP7M1v/ntZf8Aj3+FAG958X/PVP8AvoUefF/z1T/voVg/2Zrf/Pay/wDHv8KP7M1v/ntZf+Pf4UAb3nxf89U/76FHnxf89U/76FYP9ma3/wA9rL/x7/Cj+zNb/wCe1l/49/hQBvefF/z1T/voUefF/wA9U/76FYP9ma3/AM9rL/x7/Cj+zNb/AOe1l/49/hQBvefF/wA9U/76FHnxf89U/wC+hWD/AGZrf/Pay/8AHv8ACj+zNb/57WX/AI9/hQBvefF/z1T/AL6FHnxf89U/76FYP9ma3/z2sv8Ax7/Cj+zNb/57WX/j3+FAG958X/PVP++hR58X/PVP++hWD/Zmt/8APay/8e/wo/szW/8AntZf+Pf4UAb3nxf89U/76FHnxf8APVP++hWD/Zmt/wDPay/8e/wo/szW/wDntZf+Pf4UAb3nxf8APVP++hVS9kR2i2MrYPODmsz+zNb/AOe1l/49/hQNN1wHPnWX/j3+FADvEH/IY0b/AK7/APsyVv1gSaXq93f2c17LalbaUPiMsDjIJ7e1b9ABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQB//Z)

## Razlika izmedju Elementa i Komponente?

**React Element:** obican JavaScript objekat koji opisuje **sta zelimo da se pirkaze na korisnickom interfejsu.** Predstavlja DOM cvor (npr. <div>, <button>, itd) ili komponentu u odredjenom trenutku. Elementi su **nepromenljivi (immutable)**: kada se jednom naprave njihova svojstva se ne mogu menjati. Umesto toga, kreiraju se novi elementi da bi se prikazale promene. Mogu biti ugnjezdeni jedni u druge putem props-a.

**Komponenta:** je funkcija ili klasa koja **vraca element** (ili stablo elemenata) da bi opisala deo korisnickog interfejsa. Moze primati **ulazne vrednosti (props)** i imati **svoje stanje (state).**

## Sta su Controlled (kontrolisane) komponente u Reactu?

**Controlled components (kontrolisane komponente)** su React komponente koje u potpunosti **upravljaju stanjem formi** (npr. <input>, <textarea>, <select>), pomocu **React state-a.**

To znaci da forma **ne upravlja svojim internim stanjem sama,** vec React postaje **jedini izvor za podatke iz forme.**

Koraci za implementaciju controlled component:

1. Inicijalizuj state pomocu **useState** (funkcionalne komponente) ili u **constructor-u** (klasne komponente).

2. Postavi vrednost forme na odgovarajucu state promenljivu (value = {someState})

3. Napravi **onChange handler** koji azurira state pomocu **setState/useState.**

4. Povezi **onChange handler** sa elementima forme

Primer:

A screen shot of a computer code

AI-generated content may be incorrect.

U ovakvim komponentama **DOM ne sadrzi podatke** – vec su svi podaci smesteni u **React state.**

Primer:

A screen shot of a computer code

AI-generated content may be incorrect.

Ovde je **DOM taj koji upravlja vrednostima forme,** a **React samo pristupa podacima kada je to potrebno** (putem ref).

## Sta su uncontrolled components(nekontrolisane komponente)?

**Uncontrolled components (nekontrolisane komponente)** su form elementi (npr. <input>, <textarea>, <select>) koji **upravljaju svojim stanjem samostalno, putem DOM-a,** a ne preko React-a.

React u ovom slucaju samo **pristupa vrednostima pomocu ref-a,** kada je to potrebno.

Ovo ponasanje je slicno tradicionalnom HTML-u.

Koraci za implementaciju uncontrolled components:

1. Napravi referencu pomocu **useRef()** (funkcionalna komponenta) ili **React.createRef()** (klasna komponenta).
2. Povezi taj ref sa elementom forme.
3. Pristupi vrednosti forme kroz **ref.current.value** prilikom dogadjaja **(onSubmit).**

## Sta se nalazi u okviru React komponente?

U okviru React komponente nalaze se sledece oblasti:

1. **Data:** Podaci koji dolaze odnekud i koje komponenta renderuje.
2. **Lifecycle:** Sastoji se od metoda ili Hooks – a koje implementiramo da bismo odgovorili na faze ulaska i izlaska komponente u React procesu renderovanja, koje se desavaju tokom vremena.
3. **Events:** kod koji pisemo za odgovaranje na interakcije korisnika
4. **JSX:** predstavlja sintaksu React komponenata koja se koristi za opisivanje strukture korisnickog interfejsa.

## React komponente (Funkcionalne vs Klasne)?

**Komponente** predstavljaju osnovne gradivne blokove svake React aplikacije. One su zapravo recnik JSX oznacavanja. Mogu biti samostalne ili ugnjezdene u druge komponente (tzv. parent/child struktura).

Postoje dve vrste komponenata:

1. **Funkcionalne (Funkcijske) komponente:** Definisu se kao JS funkcije i mogu da primaju **props(parametre)** i vracaju **JSX.** Mogu koristiti **hook-ove.** Lakse su za testiranje i refaktorisanje, kao i za odrzavanje. Koriste se u vecini slucajeva.
2. **Klasne komponente:** Definisu se kao ES6 klase koje produzavaju React.Component i imaju **render()** (**render** je funkcija koja prihvata JSX kao argument i renderuje ga u DOM cvoru koji je prosledjen funkciji React.createRoot() ), metod koji vraca React elemente odnosno JSX.

## Kako funkcionise component lifecycle?

Kod **class components:**

* **componentDidMount:** nakon prvog renderovanja
* **componentDidUpdate:** nakon azuriranja
* **componentWillUnmount:** pre uklanjanja komponente

Kod **funkcionalnih komponenti:**

* **useEffect** se koristi

## Stanje (State) u Reactu?

**Stanje (state):** je dinamicki deo React komponente. Moze se deklarisati pocetno stanje komponente koje se menja tokom vremena. Stanja se koriste za **upravljanje promenama dinamickih podataka** tokom vremena. Stanja se obicno koriste u klasnim komponentama, ali mogu se koristiti i u funkcionalnim uz pomoc **hookova.**

1. **Stanje u klasnim komponentama**

* Definiše se u konstruktoru
* Promene stanja vrše se metodom setState

1. **Stanje u funkcionalnim komponentama**

* Koristi se **useState hook:** on omogućava korišćenje state – a unutar funkcionalnih komponenata. useState prihvata jedan argument koji predstavlja početno stanje i vraća niz sa dva elemenata: **1. Trenutno stanje** i **2. Funkcija koja se koristi za ažuriranje tog stanja.**
* Funkcionalna komponenta nema stanje i metode „životnog ciklusa“

**Čista funkcija** je funkcija bez neželjenih efekata, odnosno kada je pozvana pomoću određenih skupova argumenata, ona uvek generiše isti izlaz. To je karakteristično i za funkcionalne komponente.

**Pocetno stanje komponente** nije obavezno postaviti, ali ako komponenta koristi stanje, trebalo bi ga postaviti, zato sto ako komponenta ocekuje da ce odredjena svojstva stanja biti postavljena, a ona nisu, onda ce komponenta „otkazati“ ili ce se prikazivati nesto neocekivano. Pocetno stanje komponente uvek treba da bude objekat sa jednim ili vise svojstava. Kada postavljamo stanje komponente, mi zapravo **spajamo stanje komponente sa objektom koji prosledjujemo pozivu setState().**

## Svojstva (Props)?

* **Props (svojstva):** se koriste za prosledjivanje podataka u React komponentama. Umesto pozivanja metode sa novim stanjem kao argumentom, svojstva se prosledjuju samo kada se komponenta render- uje, odnosno prosledjuju se vrednosti svojstava JSX elementima.
* **Props (Svojstva) su read –** **only** i ne mogu se menjati u komponenti koja ih prima. Razlikuju se od **stanja** po tome sto se **ne menjaju nakon pocetnog renderovanja komponente.** Ako je vrednost svojstva promenjena, a mi zelimo da ponovo render – ujemo komponentu, moramo prvo da prikazemo JSX koji je koriscen za njeno renderovanje.
* Glavna svrha props-a:
  + **Prosledjivanje podataka** komponentama
  + **Pokretanje promena stanja**
  + **Pritup preko this.props.nesto** u klasnim komponentama

## Context?

* **Context** se koristi za podatke koji treba da stignu do bilo koje komponente u nasoj aplikaiciji. Postoje dva kljucna koncepta Context – a:

**1. Context Provider:** kreira podatke i osigurava da su dostupni u svim React komponentama

**2.** **Context Consumer:** komponenta koja koristi ove podatke unutar konteksta

## Razlika izmedju TypeScript i JavaScript?

**JavaScript** je dinamicki tipiziran jezik (nema provere tipova pri kompajliranju, sve se proverava tek u runetime-u). Ne zahteva tipove(string, number, boolean...) prilikom definicije promenljivih ili funkcija.

**TypeScript** je superset JS-a. Sadrzi sve sto ima JS + ima dodat **static type checking.** Kompajlira se u JS, dakle browser vidi samo JS. Omogucava **tipove.** Sprecava gomilu bagova pre nego sto se kod pokrene.   
U aplikaciji koju sam razvijao koristio sam **React + TS.**

# React Hooks

**Hooks** su specijalne funkcije koje omogucavaju koriscenje state – a i drugih React funkcionalnosti u funkcionalnim komponentama. Drugim recima, **Hooks** su API koji omogucava funkcionalnim komponentama da se „zakace“ u React funkciji.

## useState Hook?

**useState():** omogucava da funkcionalne React komponente imaju stanje. Prihvata jedan argument koji predstavlja pocetno stanje i vraca niz sa dva elementa: **1. Trenutno stanje i 2. Funkcija koja se koristi za azuriranje tog stanja.**



**value –** trenutna vrednost stanja; **setValue:** funkcija koja azurira stanje

Koristi se tipicno za brojace, toggle – ove i jednostavna stanja.

## useEffect Hook?

**useEffect:** koristi se za pokretanje „sporednih efekata“ u funkcionalnoj komponenti. Sporedni efekti mogu ukljucivati: **dodavanje podataka, fetch podataka sa servera, pretplate na dogadjaje (subscribe), manipulisanje DOM – om** ili bilo koju drugu operaciju koja nije direktno vezana za render – ovanje komponente.

Koristi se za **izvrsavanje odredjenih operacija nakon svakog renderovanja komponente ili nakon promene odredjenih promenljivih (zavisnosti).**

A computer code with text

AI-generated content may be incorrect.

* **Funkcija sporednog efekta (efekat):**
* Funkcija unutar useEffect se izvrsava nakon svakog renderovanja komponente (osim prilikom prvog renderovanja, ako je naveden prazan niz zavisnosti [dependencies]).
* **Cleanup funkcija(opciono):**
* Opcioni je deo, ona se vraca iz funkcije sporednog efekta. Izvrsava se pre sledeceg pokretanja sporednog efekta ili pre unmount-a komponente. Koristi se za ciscenje resursa.
* **Dependencies (Zavisnosti):**
* Ako su navedene zavisnosti, useEffect ce se opet pokrenuti samo ako se neka od zavisnosti promeni od poslednjeg renderovanja.
* Ako nema zavisnosti ( [] ), useEffect se pokrece samo jednom nakon prvog renderovanja.

## useContext Hook?

**useContext:** omogucava pristup vrednostima iz konteksta bez potrebe za **props drilling-om (**kada prosledjujemo props-e kroz vise komponenti da bi stigli do one kojoj su potrebni. Resava se pomocu **Context API** i **Redux-a)**. Koristi se kada imamo **globalne podatke** koje koriste mnoge komponente (npr. Tema, jezik, autentifikacija).

function Username() {

const user = React.useContext(UserContext);  
 return (  
 <p>  
 Logged in as <strong> {user.name} </strong>  
 </p>  
 );  
}

U ovom primeru, vrednost konteksta user je stanje koje postavlja komponenta UserProvider kada odgovori na API poziv. To znači da se vrednost konteksta user ažurira pomoću Hooka **useContext()** kad god se promeni vrednost user.

**useContext()** može se koristiti kada nam je potreban pristup globalnim podacima, bez obzira koliko je komponenta duboko ugnježdena u JSX označavanju.

## useReducer Hook?

**useReducer:** koristi se za slozenije logike state-a, koje ukljucuju vise podstanja ili akcija. Predstavlja **alternativu za useState** kada je stanje slozenije. Koristi se kada stanje zavisi od prethodnog ili kada imamo vise akcija koje menjaju stanje.

Funkcija prihvata dva argumenta: **1. Funkciju reduktora koja azurira stanje i 2. Pocetno stanje komponente.** Povratna vrednost useReducer je **niz,** sa stanjem kao prvim elementom i dispatcher funkcijom kao drugim elementom.

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Argument state** je trenutno stanje komponente, a **argument action** se prosledjuje funkciji dispatch() – omogucava slanje akcija koje opisiju promene u stanju aplikacije. Vrednost **action.type** koristi se da utvrdi sta treba da se radi.

## useRef Hook?

**useRef:** koristi se kada zelimo da dobijemo referencu na DOM element. Omogucava kreiranje mutabilnog objekta koij drzi vrednost koja ne uzrokuje re-renderovanje kada se menja

**A screen shot of a computer program

AI-generated content may be incorrect.**

Koristi se tipicno za: **fokusiranje elemenata, cuvanje prethodnih vrednosti itd.**

## Sta je memoizacija u Reactu?

**Memoizacija** je tehnika optimizacije koja cuva efekat funkcija koje zahtevaju mnogo resursa i ponovo koristi taj rezultat kada se pojave isti ulazi.

**React.memo** i **useMemo** sprecavaju nepotrebna ponovna renderovanja i proracune tako sto kesiraju rezultate na osnovu zavisnosti.

## useMemo Hook?

**useMemo:** memorise rezultat funkcije da bi se izbegli skupi preracuni pri svakom renderu.

****

Koristi se u situacijama: **kada je funkcija skupa (**npr. Filtriranje neke velike liste), **kada rezultat zavisi od inputa.**

## useCallback Hook?

**useCallback:** se koristi da memorise funkciju i sprecava kreiranje nove reference pri svakom renderu. Npr. On memorizuje funkcije i ako se ona vise puta pozivaju sa istim parametrima onda funkcija ne preracunava svaki put ponovo rezultat. Sprecava re-renderovanje komponenti koje zavise od tih funkcija.

**A computer screen shot of a computer code

AI-generated content may be incorrect.**

Koristi se u situacijama kada: **funkciju prosledjujemo child komponenti koja koristi React.memo** ili **kada zelimo stabilnu referencu funkcije.**

# React Router Hooks

## useNavigate Hook?

**useNavigate:** je hook iz React Router biblioteke koji omogucava navigaciju izmedju razlicitih ruta u React aplikaciji. Umesto koriscenja tradicionalnih linkova za navigaciju, **useNavigate** omogucava navigaciju putem funkcija unutar komponenti.

## useParams Hook?

**useParams:** je hook iz React Router biblioteke kojiomogucava pristup parametrima URL-a, sto je korisno kada zelim oda dohvatimo i koristimo dinamicke delove URL-a u nasim komponentama.

  
Slika: primer postavljanja rute

UserDetails komponenta će odgovarati ruti /userDetails/userDetailsUpdate/:id, gde je id parametar URL – a



U komponenti **UserDetails** koristim **useParams** da pristupim parametrima URL – a

## useLocation Hook?

**useLocation:** je hook iz React Router bibliotekekoji se koristi za dobijanje informacija o trenutnoj lokaciji rute.

## useSearchParams?

**useSearchParams:** je hook iz React Router biblioteke koji se koristi za rad sa **query parametrima u URL – u** (npr. ?page=2&sort=asc).

**A screenshot of a computer code

AI-generated content may be incorrect.**

Vraca **1. searchParams:** objekat slican kao URLSearchParams, koji se koristi za citanje parametara iz URL-a; **2. setSearchParams:** funkcija za promenu parametara u URL-u.

Koristi se tipicno za **cuvanje stanja u URL-u (paginacija, filtriranje, sortiranje), kada zelim da korisnik moze deliti link sa trenutnim stanjem.**

# Custom Hookovi

**Custom Hookovi** su funkcije koje koriste osnovne **React Hookove (useEffect, useState...)** i omogucavaju ponovno koriscenje logike kroz vise komponenti. **Nisu novi tip hook-a,** vec samo nacin da se izdvoji ponovljena logika.



Primer koriscenja custom hook-a. Ove hook-ove generise RTK Query na osnovu mojih API definicija.

# Redux

**Redux** je biblioteka za upravljanje stanjem aplikacije u JavaScript aplikacijama, najcesce koriscena uz React. Omogucava predvidljivo upravljanje podacima pomocu centralnog **Store-a,** kroz koji se sve promene stanja vrse u jasno definisanim koracima (akcije -> reducer -> novo stanje).

## Kljucne komponente Redux arhitekture?

### Store – Centralno skladiste stanja:

* + Predstavlja **jedinstveni objekat** koji cuva celo stanje aplikacije.
  + Kreira se pomocu **configureStore()** iz @reduxjs/toolkit
  + Svaka komponenta moze citati stanje iz store-a ili slati akcije za njegovo menjanje

**A screen shot of a computer code

AI-generated content may be incorrect.**

### Actions:

* + Objekti koji opisuju sta se desilo, tj. objekti koji opisuju dogadjaje koji treba da promene stanje, odnosno neku promenu u aplikaciji.
  + Svaka akcija je obican JS objekat sa **obaveznim type poljem** (type – definise vrstu akcije), a moze sadrzati i dodatne podatke poput **payload** koje reducer koristi da izmeni stanje.

**A screen shot of a computer code

AI-generated content may be incorrect.**

### Reducers:

* + Funkcije koje definisu kako se stanje aplikacije menja kao odgovor na akcije.
  + Prima **state** i **action** i **vraca novo stanje.**
  + Moraju biti ciste funkcije (bez sporednog efekta)
  + U Redux Toolkit-u, reducere pisemo unutar **createSlice**

**A computer screen shot of a code

AI-generated content may be incorrect.**

### Slice**:**

* + Objekat koji kombinuje **state + reducer + akcije**
  + **createSlice()** pojednostavljuje rad sa Redux-om
  + Definise pocetno stanje, reducer-e i akcije i automatski kreira **action creators**

**A screenshot of a computer code

AI-generated content may be incorrect.**

### Dispatch:

* + Metod koji se koristi za slanje akcija na store, tj. F-ja koja salje akcije ka reduceru.
  + **dispatch() f-ja** se koristi za pokretanje akcija koje menjaju stanje.
  + U React komponentama koristimo **useDispatch() hook.**

**A screenshot of a computer code

AI-generated content may be incorrect.**

### Selectors:

* + Funkcija za citanje iz Store-a, tj. Funkcije koje omogucavaju citanje (selektovanje) specificnih delova stanja iz Store-a, tj. F-ja koja preuzima podatke iz stanja
  + Omogucavaju kompozabilan, efikasan i izolovan pristup stanju
  + U React komponentama koristimo **useSelector()** hook

**A computer code with text

AI-generated content may be incorrect.**

### Middleware:

* + Funkcije izmedju dispatch i reducera
  + Funkcije koje omogucavaju modifikaciju akcija pre nego sto one stignu do reducer-a.
  + Omogucavaju izvodjenje dodatnih operacija nad akcijama, kao sto su
    - Logovanje
    - Asinhrone operacije (npr. api pozivi)
    - Validacije
  + U Redux Toolkit-u najpoznatiji middleware je **RTK Query**
  + **Primer: Integracija sa API-jem**

**A screen shot of a computer code

AI-generated content may be incorrect.**

## Redux Toolkit?

**Redux Toolkit:** je biblioteka za koriscenje Redux-a.

Predstavlja napredni alat za rad sa API zahtevima

Automatski kreira slice, akcije, reducere, kesiranje, loading/error stanja

Olaksava upravljanje podacima dobijenim sa servera

Bitne funkcije iz Redux Toolkit-a:

* **createSlice:** 
  + funkcija koja pojednostavljuje kreiranje Redux reducer-a
  + **spaja** akcije (actions), kreatore, akcija (action creators), i reducere u jedno mesto
  + omogucava cist, jasan i koncizan kod bez potrebe za pisanjem switch izraza

A screenshot of a computer code

AI-generated content may be incorrect.

* + Sta sve vraca **createSlice:**
    - **reducer:** koji ide u **configureStore**
    - **actions:** automatski generisane funkcije (increment, decrement, itd
* **useSelector:**
  + hook koji omogucava React komponentama da pristupe stanju iz Redux Store-a. Prima funkciju selektora koja izvlaci deo stanja iz store-a i vraca taj deo kao rezultat
  + koristi se za **citanje vrednosti iz Redux Store-a** unutar React komponente

A computer code with text

AI-generated content may be incorrect.

* + **state:** globalni state
  + **state.counter.value:** konkretan deo koji se cita (definisano u createSlice
* **useDispatch:**
  + Koristi se za dobijanje reference na dispatch funkciju. Omogucava komponentama da salju akcije koje mogu promeniti stanje u Redux store-u
  + Koristi se za **slanje (dispatch-ovanje) akcija** ka store-u, sto pokrece promenu stanja

**A screen shot of a computer code

AI-generated content may be incorrect.**

* + - Klikom na dugme poziva se **dispatch(increment)**
    - To pokrece reducer u createSlice koji menja stat

**Veza izmedju createSlice, useSelector i useDispatch**

* Sve tri koriste se zajedno u React komponentama:

**A computer code with text

AI-generated content may be incorrect.**

## Kako saljem HTTP Request (tj. kako okidam API poziv)?

Koristim **RTK Query** iz Redux Toolkita za **definisanje API endpointa**.  
**API endpoints** definisani su mi u src/apis/  
U komponentama koristim hookove, poput:



I kada zelim da posaljem zahtev, pozivam funkciju **updateKorpa:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

Proces slanja HTTP Requesta:

1. Kada pozovem funkciju **updateKorpa,** RTK Query salje HTTP POST zahtev na /shoppingCart sa prosledjenim podacima.
2. RTK Query automatski handluje loading, succes i error state.
3. RTK Query automatski upisuje podatke u Redux store, kesira ih i omogucava refetch kada je potrebno.
4. Kada server odgovori, rezultat dobijam kao **promise** (mogu koristiti .then ili await). Na osnovu odgovora mogu prikazati notifikaciju, azurirati UI ili preusmeriti korisnika.

## Kako saljem HTTP Request (na primeru Login)

**authApi** file (RTK Query setup):  
Ovde koristim **Redux Toolkit Query** da napravim centralizovan API servis, koji se brine o:

* **Kreiranju HTTP Requesta** (fetchBaseQuery)
* **Cachiranju rezultata** (da ne vucem iste podatke vise puta)
* **Invalidaciji cache – a** kada se podaci promene
* **Generisanju hook – ova** koje mogu da koristim direktno u komponentama (useLoginUserMutation, useGetCurrentUserQuery, itd

Primer endpointa:

A screen shot of a computer program

AI-generated content may be incorrect.

Za **login** definisem **mutation** jer login menja stanje aplikacije.

Zatim u Login.tsx komponenti pozivam:



* **loginUser** je funkcija kojom okidam API poziv (HTTP Request)
* Ona automatski salje HTTP Request na <https://localhost:7210/api/auth/login> sa telom (body) koje joj prosledim
* Vraca **Promise (Rezultat neke asinhrone operacije)** koji sadrzi **data** ili **error**

Kako izgleda API poziv u praksi:

A screenshot of a computer program

AI-generated content may be incorrect.

* **loginUser** poziva backend (POST /api/auth/login) sa podacima koje sam mu prosledio kroz input
* Ako backend vrati status kod 200 OK, response.data sadrzi JSON koji je stigao
* Ako backend vrati gresku (401, 400...), onda response.error sadrzi detalje o gresci

Sta se dalje radi u okviru **handleSubmit:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

* Cekam odgovor **await loginUser(...)**
* Ako response.data postoji → login je uspesan:
  + Vadim token iz odgovora
  + Dekodiram ga pomocu jwtDecode
  + Upisujem token u **localStorage**
  + U Redux Store upisujem podatke o korisniku (**dispatch(setLoggedInUser(…)**)
  + Radim navigaciju na pocetnu stranicu (“/”)
* Ako response.error postoji → prijava je neuspesna
  + Uzimam errorMessages[0] sa backenda
  + Prikazujem pomocu toastNotify poruku o gresci

Umesto da koristim **fetch** ili **axios rucno**, RTK Query sve to radi za mene. Na meni je samo da **koristim hook i prosledim podatke.**

A screenshot of a computer program

AI-generated content may be incorrect.

Iza kulisa RTK Query radi sledece:

A screen shot of a computer code

AI-generated content may be incorrect.

## Kako funkcionise sistem dodavanja u Korpu?

U aplikaciji modelujem **Korpu** gde:

* Jedan korisnik aplikacije (**ApplicationUser**) ima jednu korpu (**Korpa**) (1:1 odnos)
* Korpa moze sadrzati vise stavki korpe (**StavkaKorpe**) (1: vise odnos)
* Svaka stavka korpe (**StavkaKorpe**) je povezana sa sportskim objektom (**SportskiObjekat**) i listom odabranih termina (**OdabraniTermini**).
* Cena za stavku korpe (**CenaZaObjekat**) racuna se na osnovu cene po satu sportskog objekta **(CenaPoSatu),** ukupnog trajanja ne-preklapajucih termina i broja ucesnika **(Kolicina)**

Endpoint **DodajIliAzurirajKorpuSaTerminima** na backendu upravlja dodavanjem ili azuriranjem stavke korpe sa odabranim terminima, dok **Redux slice** upravlja stanjem korpe na frontendu.

### 48.1: Backend DodajIliAzurirajKorpuSaTerminima endpoint:

* Poziva se putem HTTP POST zahteva na /api/korpa/dodajIliAzurirajKorpuSaTerminima sa parametrima: userId (ID korisnika); sportskiObjekatId (ID sportskog objekta); brojUcesnika; terminIds (Lista ID-eva odabranih termina)

Kako funkcionise korak po korak:

1. **Pronalazi se korisnikova korpa:**

A black screen with white text

AI-generated content may be incorrect.

Endpoint proverava da li korisnik ima korpu (tj. da li je vec dodao stavku u korpi), ucitava stavke korpe i povezane termine koristeci **eager loading.**

Ako korpa ne postoji, nova ce se kreirati kasnije

1. **Provera sportskog objekta**

A screen shot of a computer code

AI-generated content may be incorrect.

Proverava da li sportski objekat postoji u bazi podataka

Ako ne postoji, vraca BadRequest odgovor sa porukom o gresci

1. **Provera termina:**

A screen shot of a computer program

AI-generated content may be incorrect.

Pronalazi sve termine koji odgovaraju prosledjenim terminIds i pripadaju sportskom objektu.

Proverava da li svi trazeni termini postoje i pripadaju objektu. Ako ne, vraca BadRequest.

1. **Obelezava odabrane termine kao zauzete:**

A screen shot of a computer program

AI-generated content may be incorrect.

Prolazi kroz sve termine i one koji imaju status ”Slobodan“ oznacava kao „Rezervisan“ i dodeljuje im userId, da bismo znali koji korisnik je rezervisao koji termin.

1. **Kreiranje nove korpe ako je potrebno (tj. ako korisnik prethodno nema nijednu stavku u korpi)**

A screen shot of a computer code

AI-generated content may be incorrect.

Ako korpa ne postoji, kreira se nova i dodeljuje joj se userId i cuva se u bazi.

1. **Pronalazenje ili kreiranje stavke korpe:**

A screen shot of a computer program

AI-generated content may be incorrect.

Proverava da li stavka korpe za dati sportski objekat vec postoji.

Ako ne postoji, **kreira se nova stavka:**

* + Racuna se ukupno vreme ne-preklapajucih temrina koristeci funkciju **IzracunajUkupnoVremeBezPreklapanja**
  + Izracunava se cena: CenaZaObjekat = CenaPoSatu \* UkupnoSati \* BrojUcesnika
  + Postavlja svojstva stavke, ukljucujuci listu odabranih termina
  + Dodaje se stavka u bazu

1. **Azuriranje postojece stavke korpe**

A computer screen shot of a program code

AI-generated content may be incorrect.

Ako stavka vec postoji, azurira je:

* + Dodaje samo nove termine (koji nisu vec u OdabraniTermini) kako bi se izbeglo dupliranje
  + Ponovo ucitava sve termine za stavku iz baze radi tacnosti
  + Ponovo racuna cenu na osnovu azurirane liste termina i broja ucesnika
  + Azurira broj ucesnika **(Kolicina)** ako je prosledjen brojUcesnika veci nego sto je bio
  + Cuva azuriranu stavku u bazi

1. **Racunanje ukupnog vremena trajanja termina bez preklapanja**

A screen shot of a computer program

AI-generated content may be incorrect.

Funkcija prolazi kroz svaki terimn i racuna njegovo trajanje oduzimajuci **VremePocetka** od **VremeZavrsetka**

Sabira trajanje za ukupno vreme.

Napomena: podrazumeva se na nema preklapanja termina, jer se ne proverava preklapanje

1. **Cuvanje promena i povratni odgovor:**



Na kraju u funkciji **DodajIliAzurirajKorpuSaTerminima** cuvam sve promene u bazi podataka i vracam uspesan odgovor. (200 status code – OK).

### 48.2: FrontEnd: Redux (shoppingCartSlice)

Redux slice (**shoppingCartSlice**) upravlja stanjem korpe na frontendu, koje se azurira na osnovu odgovora sa API-ja

1. **Pocetno stanje:**

A black background with blue and pink text

AI-generated content may be incorrect.

Korpa pocinje sa praznom listom stavki korpe (stavkaKorpe: [])

1. **Relevantne akcije:**

A screen shot of a computer code

AI-generated content may be incorrect.

#### setShoppingCart:

* Azurira celu korpu sa podacima iz API-ja (npr. nakon poziva GetKorpa ili DodajIliAzurirajKorpuSaTerminima)
* Payload obicno sadrzi stavke korpe sa povezanim sportskim objektima i terminima

**A computer screen with text on it

AI-generated content may be incorrect.**

#### setTeminForObjekat:

* Azurira listu termina za odredjenu stavku korpe na osnovu sportskiObjekatId
* Ova akcija se poziva nakon odgovora **DodajIliAzurirajKorpuSaTerminima** da bi se termini azurirali na frontendu

A screen shot of a computer code

AI-generated content may be incorrect.

#### azurirajCenu:

* Azurira cenu (**cenaZaSportskiObjekat**) za stavku korpe na osnovu sportskiObjekatId
* Koristi se za odrazavanje nove cene nakon dodavanja ili azuriranja termina

### 48.3 Integracija sa Backendom

Mutacija **azurirajKorpuSaTerminima** salje POST zahtev na api/korpa/dodajIliAzurirajKorpuSaTerminima (RTK Query mutacija)

Backend validira podatke, azurira korpu i termine, racuna cenu i cuva podatke

Frontend azurira Redux store i UI na osnovu odgovora, koristeci akcije poput **setTerminForObjekat** i **azurirajCenu.**

## Repository/Unit of Work

**Repository** je projektni obrazac koji pruza **apstrakciju nad persistencijom** sto znaci prakticno da skrivam detalje implementacije (SQL upite, EF Core metode, konekcije) iza sloja koji mi nudi jednostavne metode za rad (Add, Get, Update, Remove). **Repository** mi u projektu predstavlja **genericki repozitorijum** koji enkapsulira DbSet<T>, sto znaci da ga ne vidim spolja vec ga samo vidi Repository. Svi koji koriste Repository rade sa njegovim metodama (GetAsync, AddAsync), a ne direktno sa DbSet<T>.

**Unit of Work** je projektni obrazac koji grupise vise repozitorijuma i izlaze **SaveAsync()** tako da vise promena mogu sacuvati jednim pozivom (jedna transakcija/jedna SaveChanges() )

U ovoj aplikaciji

* **DbSet<T>** prakticno predstavlja **Repository** jer je kolekcija koja predstavlja tabelu i nudi CRUD operacije.
* **DbContext** je **Unit of Work** jer prati izmene nad entitetima i nudi SaveChanges() kao jednu transakciju.