Razvoj i testiranje klijentskih web-aplikacija u C#-u

Šimić, Josipa

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Josipa Šimić

RAZVOJ I TESTIRANJE KLIJENTSKIH WEB-APLIKACIJA U C#-U

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, Studeni, 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Klasična web-aplikacija najčešće prati klijent-poslužitelj arhitekturu. Klijent šalje zahtjeve poslužitelju kroz mrežu, dok poslužitelj obrađuje te zahtjeve, obavlja odgovarajuće funkcije, te vraća odgovor klijentu. Konkretno, poslužitelj je udaljeno računalo s pristupom mreži koje sprema i pokreće (engl. *hosts*) izvorni kod aplikacije. Klijent je računalo korisnika koji pristupa aplikaciji pomoću internetskog preglednika. Preglednik preuzima i izvršava klijentski dio koda aplikacije, odnosno statički sadržaj za oblikovanje web-stranica napisan u HTML-u i CSS-u, te dinamički sadržaj za korisničku interakciju, obično napisan u JavaScriptu.

JavaScript je dugo vremena bio jedini programski jezik koji se mogao izvršavati u pregledniku, pa posljedično i jedini jezik kojim se ostvarivala korisnička interakcija direktno na klijentu. Za korištenje bilo koje funkcionalnosti napisane u drugom programskom jeziku, klijent bi morao slati zahtjeve poslužitelju kako bi se izvršio odgovarajući kod. Ovo je često bio problem za aplikacije kojima su potrebne vrlo visoke performanse klijentskog koda. Kao interpretativni jezik visoke razine, JavaScript nije uspijevao biti dovoljno brz, unatoč raznim optimizacijama.

U tu svrhu napravljen je WebAssembly, programski jezik niske razine u formatu binarnog koda koji je izvršiv u pregledniku, i u koji se kompilirani jezici poput C-a mogu prevesti. Iako je WebAssembly napravljen u svrhe poboljšanja performansi na klijentskoj strani, donio je i druge važne promjene u razvoju web-aplikacija. Zahvaljujući mogućnostima ovog jezika, na tržište su počeli dolaziti alati koji omogućuju razvoj klijentskog dijela aplikacija u tipično poslužiteljskim jezicima.

Jedan takav alat je Blazor, razvojni okvir tvrtke Microsoft koji omogućava programerima pisanje klijentskog koda u C#-u, objektno orijentiranom jeziku visoke razine. U ovom radu proučit ćemo Blazor, ulogu WebAssemblyja u njegovoj implementaciji te alate i mogućnosti koje nudi. Zatim ćemo proći kroz proces razvoja web-aplikacije u Blazoru i proučiti opcije testiranja koda.

Poglavlje 1

Razvojni okvir Blazor

1.1 Platforma .NET

.NET je razvojna platforma otvorenog koda (engl. *open source*) tvrtke Microsoft koja se sastoji od alata, jezika i biblioteka za razvoj raznih vrsta aplikacija. Jezici podržani u .NET-u su jezici visoke razine: F#, Visual Basic te najčešće korišteni C#. Platforma nudi nekoliko implementacija:

- **.NET Framework** je originalna implementacija .NET-a koja podržava razvoj i pokretanje aplikacija unutar operacijskog sustava Windows. Ova implementacija sadrži okvire za razvoj web i desktop aplikacija, ali više nije preporučena za nove projekte jer se aktivno ne razvija.
- **.NET** (ranije poznat kao .NET Core) je najnovija unificirana implementacija .NET-a koja podržava razvoj višeplatformskih aplikacija, to jest, aplikacija koje se mogu razvijati i pokretati unutar različitih operacijskih sustava, poput Windowsa, macOS-a i Linuxa. Glavne prednosti su poboljšane performanse, modularnost i podrška za više platformi. Unutar .NET-a relevantno je istaknuti **ASP.NET**, skup razvojnih okvira koji se koriste za izradu web-aplikacija.
- **Xamarin** (integriran u moderni .NET) omogućava razvoj aplikacija za mobilne operacijske sustave, poput iOS-a i Androida, koristeći zajednički .NET kod.

Sve implementacije .NET-a uključuju usluge poput upravljanja dretvama (engl. *thread management*), oslobađanja memorije (engl. *garbage collection*), sigurnosti tipova (engl. *type-safety*), upravljanja iznimkama (engl. *exception handling*) i drugih funkcionalnosti koje pojednostavljuju proces pisanja koda.

Programeri također mogu koristiti razne alate i pakete koji pojednostavljuju implementaciju funkcionalnosti aplikacija te integraciju s vanjskim sustavima i bazama podataka. U .NET-u se programer ne mora baviti najnižim tehničkim detaljima razvoja, već se može fokusirati na višu razinu, pridajući veću pažnju arhitekturi i dizajnu aplikacije.

Razvojni okvir Blazor podržan je unutar .NET implementacije, kao dio ekosustava ASP.NET, od verzije 3.0, s velikim poboljšanjima u verziji 5.0 i novijima.

1.2 Osnovne značajke Blazora

Blazor je razvojni okvir unutar ASP.NET ekosustava koji omogućuje programerima izradu interaktivnih web-aplikacija koristeći C#. Cilj je olakšati razvoj na klijentskoj strani aplikacije programerima koji su upoznati s .NET-om.

Prototip Blazora predstavio je Steve Sanderson, softverski arhitekt tvrtke Microsoft, 2017. godine na konferenciji NDC Oslo. Njegova namjera bila je demonstrirati kako se C# može koristiti za razvoj web-aplikacija u pregledniku, koristeći WebAssembly i nove mogućnosti JavaScript sučelja. U tu svrhu razvio je jednostavni razvojni okvir koji podržava pokretanje aplikacija unutar preglednika. Ova demonstracija privukla je veliku pozornost i izazvala pozitivne reakcije, što je dovelo do razvoja novog Microsoftovog ASP.NET proizvoda, Blazora.

Za razvoj klijentskog dijela aplikacije Blazor koristi tzv. Razor sintaksu, sintaksu oznaka (engl. *markup syntax*) u ASP.NET-u koja omogućuje programeru povezivanje C# koda s događajima na HTML elementima (primjerice, klik gumba ili proširenje padajućeg izbornika).

```
<p>Current count: @currentCount</p>
<button id="counter-button" @onclick="IncrementCount">Increment</button>

@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Primjer 1.1: Razor sintaksa

Primjer 1.1 prikazuje osnovnu Razor sintaksu. HTML element `counter-button` je gumb čiji klik izvršava C# metodu `IncrementCount()`. U općenitom razvojnom okviru ASP.NET-a (npr. u okviru ASP.NET MVC), kad bi korisnik kliknuo `counter-button` gumb, inicirao bi se povratni zahtjev (engl. *postback*), odnosno zahtjev poslužitelju da odradi akciju povezanu s gumbom, to jest da izvrši metodu `IncrementCount()`. Jedna od posljedica slanja takvog zahtjeva je ponovno učitavanje cijele stranice na klijentu, budući da poslužitelj šalje potpuno novi HTML po završetku tražene akcije. U Blazoru, posebno u Blazor WebAssemblyju, za time više nema potrebe.

Blazor ima nekoliko modela: Blazor Server, Blazor Hybrid i Blazor WebAssembly.

Blazor Server programeru omogućuje korištenje Razor i C# sintakse u razvoju klijentske strane aplikacije. U aplikaciji razvijenoj unutar Blazor Servera, C# kod se i dalje izvršava na poslužitelju, ali koristeći SignalR za komunikaciju s klijentom. SignalR je ASP.NET biblioteka koja ostvaruje direktnu komunikaciju između klijenta i poslužitelja, na način da poslužitelj može slati poruke klijentu u stvarnom vremenu, a klijent može slati zahtjeve poslužitelju bez potrebe za ponovnim učitavanjem stranice. Ovaj model koristan je za razvoj aplikacija u kojima postoji veća potreba za poslužiteljskim resursima, poput baze podataka.

Blazor Hybrid napravljen je kao pomoćni okvir za korištenje Razor komponenti u razvoju desktop i mobilnih aplikacija, te za integraciju s web verzijama tih aplikacija.

Nas će najviše zanimati Blazor WebAssembly, model u kojem se C# kod aplikacije izvršava isključivo u pregledniku.

1.3 Blazor WebAssembly

Rad preglednika i WebAssembly

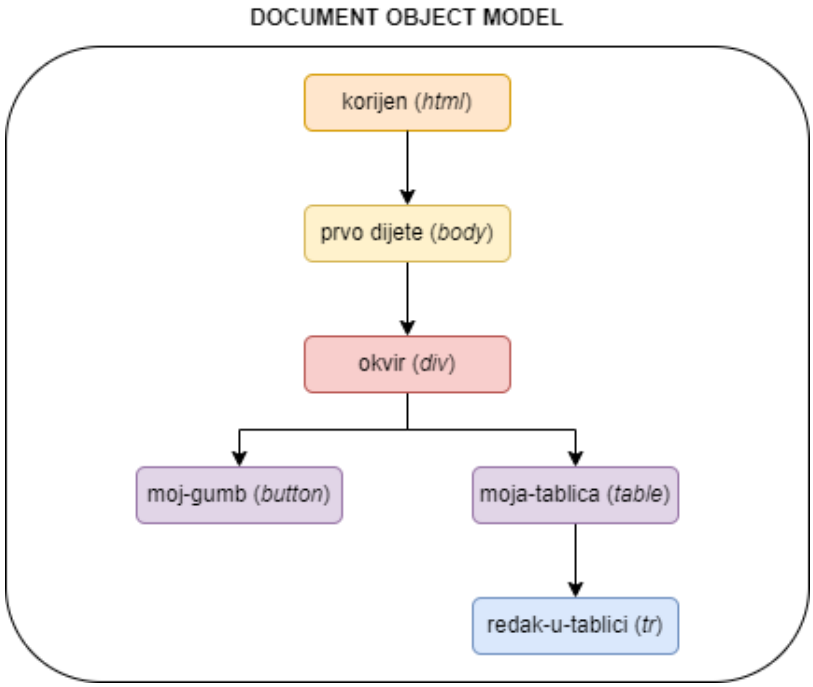
Pri pristupu web-aplikaciji preko njezinog URL-a, preglednik preko mreže dohvaća statički sadržaj, odnosno HTML i CSS datoteke, te dinamički sadržaj, odnosno JavaScript datoteke, koje čuva i šalje poslužitelj.

HTML kod se parsira u Document Object Model (DOM), strukturu unutar preglednika nalik stablu. Svaki HTML element postaje čvor u DOM stablu. Elementi mogu imati roditeljske, dječje i susjedne čvorove, ovisno o tome kako su ugniježđeni u HTML kodu.

```
<html>
    <body>
        <div id="okvir">
            <button id="moj-gumb"></button>
            <table id="moja-tablica">
                <tr id="redak-u-tablici"></tr>
            </table>
        </div>
    </body>
</html>
```

Primjer 1.2: HTML kod koji se parsira u pregledniku

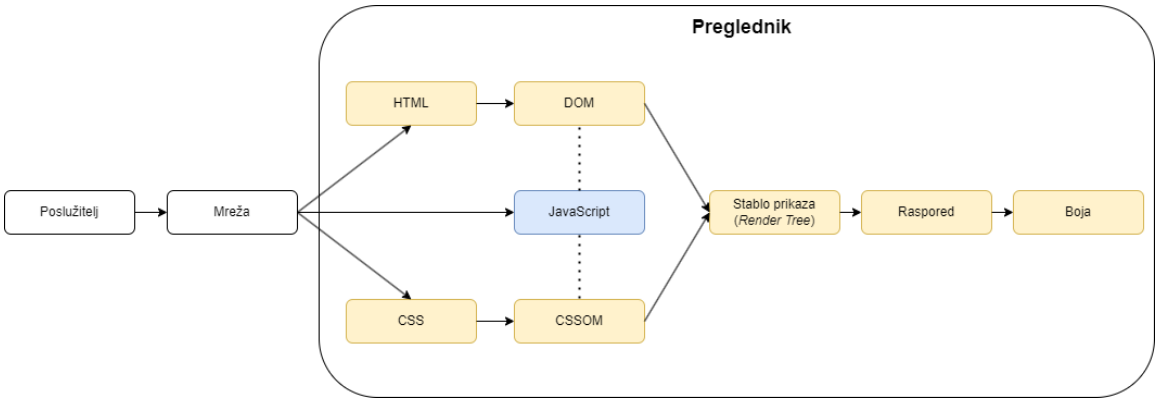
HTML kod u Primjeru 1.2 će se parsirati u DOM stablo ilustrirano na Slici 1.1.



Slika 1.1: DOM izgrađen iz parsiranog HTML koda

CSS kod se na sličan način parsira u CSS Object Model (CSSOM). Preglednik primjenjuje CSSOM pravila na odgovarajuće DOM čvorove. Na taj način preglednik izgrađuje

stablo prikaza (engl. *render tree*), hijerarhijski poredak elemenata koji će se zapravo prikazati korisniku. Primjerice, ako je na neki DOM čvor primijenjeno CSSOM pravilo "`display: none;`", taj čvor se neće nalaziti u stablu prikaza. U stablu prikaza primijenjena su sva CSSOM pravila osim dimenzija, pozicija i boje. Dimenzije i pozicije elemenata preglednik može odrediti tek nakon što odredi koji će se elementi prikazati na stranici, odnosno tek nakon što izgradi stablo prikaza. Određivanjem dimenzija i pozicije svakog čvora u stablu prikaza, preglednik izrađuje raspored elemenata na stranici. Naposljetku, kad su svi elementi na svojoj poziciji, preglednik može dodati boju. Opisani proces ilustriran je na Slici 1.2.



Slika 1.2: Rad preglednika

U isto vrijeme, preglednik učitava JavaScript sadržaje u JavaScript Engine, izoliranu okolinu koja interpretira i izvršava JavaScript kod. JavaScript Engine ima pristup sučeljima (API) koja oblikuju DOM i CSSOM. Primjerice, `document.getElementById` je funkcija u sklopu DOM sučelja koja vraća dio DOM-a obuhvaćen elementom s odgovarajućim identifikatorom. JavaScript Engine jedina je okolina unutar preglednika koja ima pristup spomenutim sučeljima.

Dakle, korisnička interakcija s aplikacijom ostvaruje se manipulacijom DOM-a (isključivo pomoću JavaScript Enginea) i komunikacijom s poslužiteljem preko mreže. Manipulacija DOM-a prikazuje odgovor korisniku na njegov zahtjev, recimo, mijenjanjem boje gumba s tekstom *Spremi* koji je korisnik kliknuo, dok je poslužitelj najčešće taj koji obavlja kompleksne zadatke inicirane klijentskim zahtjevom, poput ažuriranja i spremanja podataka koji su prikazani na dijelu stranice s kliknutim gumbom. Poslužiteljski kod je zato često napisan u jeziku pogodnijem za implementiranje poslovne logike aplikacije od interpretativnih jezika visoke razine kao što je JavaScript.

Ovakav rad web-aplikacija dugo je vremena bio dovoljno dobar. Međutim, tijekom posljednjeg desetljeća, sadržaji u web-aplikacijama postajali su znatno kompleksniji. Moderni koncepti poput 3D grafike, virtualne stvarnosti i sličnih, zahtijevali su visoke performanse. Obrađivanje korisničkih zahtjeva na udaljenom poslužitelju očito nije bila opcija, a JavaScript je, unatoč raznim optimizacijama, bio prespor da bi se potrebne funkcije odrađivale direktno na klijentu. Javila se potreba za korištenjem drugih jezika unutar preglednika – jezika niže razine i visokih performansi, poput C-a i Rusta.

U lipnju 2017. godine objavljena je prva standardizirana verzija WebAssemblyja, programskog jezika binarnog formata koji se može izvršavati u modernim preglednicima. WebAssembly nije namijenjen za direktno pisanje koda, već je nalik klasičnom asembleru; to je jezik vrlo niske razine koji se vrlo brzo izvršava i u koji se kompilirani jezici poput C-a i Rusta mogu prevesti.

WebAssembly se izvršava u izoliranoj i ograničenoj okolini (engl. *sandbox*) unutar preglednika, koja ima direktan pristup memoriji preglednika. Međutim, okolina u kojoj se WebAssembly izvršava nema direktan pristup sučeljima za manipulaciju DOM-om kojima JavaScript Engine ima pristup. Okolina je vrlo ograničena, jer je njezina glavna uloga omogućiti brzo izvršavanje kompleksnih operacija unutar preglednika. WebAssembly nije zamjena za JavaScript, već nadopuna. Za manipulaciju DOM-om, WebAssembly okolina može (i mora) komunicirati s JavaScript Engineom.

Unatoč ograničenjima WebAssemblyja u odnosu na JavaScript, ovaj jezik donio je značajne promjene u razvoju web-aplikacija, nudeći način kako izvršavati kod napisan u različitim jezicima unutar preglednika, što prije nije bilo moguće.

WebAssembly u ASP.NET-u

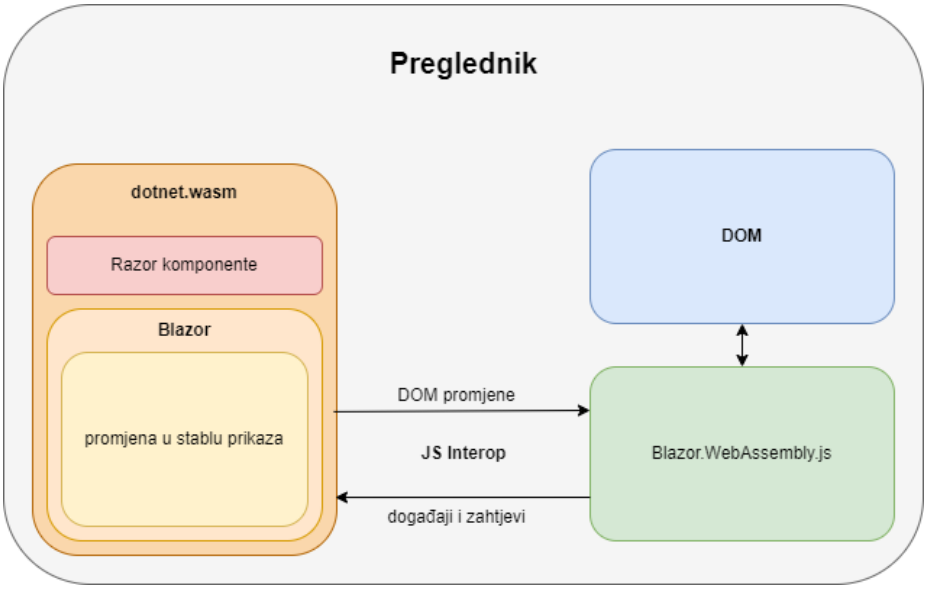
Razor sintaksa osmišljena je kao tehnologija za lakši razvoj klijentskog koda web-aplikacije unutar razvojnih okvira ASP.NET-a. Programer može jednostavnije izraditi HTML komponente i popratne poslužiteljske metode u C#-u, dok se mehanizmi ASP.NET-a u pozadini brinu za generiranje završnog HTML koda koji se šalje klijentu kao statička datoteka.

Međutim, prije pojave SignalR-a, s Razor tehnologijom nije bilo moguće izbjeći pisanje JavaScript koda u okvirima ASP.NET-a, pa čak ni u najjednostavnijim web-aplikacijama. Razlog tomu je upravo izrada cijelog HTML koda na poslužiteljskoj strani, zbog čega bi preglednik morao ponovno učitavati cijelu stranicu po završetku poslužiteljeve obrade zahtjeva. Ponovno učitavanje cijele stranice radi, recimo, promjene boje jednog gumba nema smisla i loša je praksa, budući da se može izbjeći korištenjem JavaScripta. Dolazak

SignalR-a velikim je dijelom riješio taj problem, no postavljalo se pitanje može li se komunikacija s poslužiteljem izbjeći u potpunosti pomoću WebAssemblyja.

C# nema direktan prevoditelj u WebAssembly, no to nije bio problem, budući da je odredište kompilacije C#-a (u .NET-u) Mono Runtime, okruženje za pokretanje i izvršavanje .NET aplikacija. Mono Runtime okruženje napisano je u C-u, koji se može prevesti u WebAssembly. Korištenjem Razor tehnologije, prevoditelja C-a u WebAssembly te pomoćnih JavaScript mehanizama, napravljen je Blazor WebAssembly, model Blazora koji izvršava čitav kod aplikacije u pregledniku.

Dinamičke biblioteke (DLL) koje sadrže kompilirani kod za izvršavanje Blazora te dinamičke biblioteke koje sadrže kompilirani kod aplikacije izvršavaju se u pregledniku u datoteci dotnet.wasm koja je učitana pomoću JavaScripta. Ta datoteka izvršava WebAssembly kod te, za svaku potrebnu promjenu u stablu prikaza, kontaktira JavaScript Engine pomoću JS Interopa, mehanizma za komunikaciju .NET koda i JavaScripta. JavaScript Engine izvršava pomoćni JavaScript kod iz datoteke Blazor.WebAssembly.js koji u konačnici ostvaruje promjene u DOM-u koje je Blazor poslao. Proces je ilustriran na Slici 1.3.



Slika 1.3: Rad Blazor WebAssembly aplikacije

U Blazor WebAssemblyju programer može razviti složeniju, potpuno klijentsku web-aplikaciju uz sve mogućnosti .NET-a, poput integriranja s vanjskim sučeljima i osiguranja kvalitete pomoću .NET okvira za pisanje testova. U daljnim poglavljima ćemo to i pokazati, kroz razvoj i testiranje konkretne aplikacije u Blazor WebAssemblyju.

Poglavlje 2

Alati za razvoj u Blazoru

Prije nego što počnemo razvijati konkretnu aplikaciju u Blazoru, upoznat ćemo se s osnovnim alatima koji će nam biti potrebni.

2.1 Programski jezik C#

C# je objektno orijentirani programski jezik visoke razine, razvijen kao dio .NET platforme. U osnovnoj sintaksi, C# sličan je C-u i Javi. Bitne značajke C#-a su:

- Oslobađanje resursa (engl. *garbage collection*)
  C# ima ugrađeni sustav upravljanja memorijom koji oslobađa prostor čim neki objekt prestane biti u uporabi, što olakšava pisanje memorijski sigurnog koda.

- Statičko tipiziranje (engl. *static typing*)
  C# je statički tipiziran jezik, što znači da se tip svake varijable određuje u periodu kompiliranja (engl. *compile time*), čime se smanjuju greške tijekom perioda izvršavanja (engl. *runtime*).

- Višedretvenost (engl. *multithreading*)
  C# podržava mehanizme upravljanja dretvama koji su jednostavni za korištenje i održavanje.

- Language Integrated Query (LINQ)
  C# uključuje biblioteku LINQ, alat koji olakšava dohvaćanje podataka iz raznih izvora, implementirajući jedinstvenu sintaksu za pisanje upita.

- Asinkrono programiranje
  C# podržava asinkrono programiranje, to jest, funkcionalnost izvršavanja opsežnijih zadataka (poput dohvaćanja podataka preko mreže) bez prekidanja glavnog tijeka aplikacije.

Proučimo neke bitne koncepte u ovom jeziku koji će nam biti potrebni za kasniji razvoj aplikacije.

`null` objekti i iznimke

U C#-u, član klase može biti metoda (engl. *method*), polje (engl. *field*) ili svojstvo (engl. *property*). Metoda predstavlja funkciju klase. Može primati parametre s kojima obavlja neki zadatak i može vratiti neku vrijednost po završetku. Metoda je strogo vezana za neku klasu, za razliku od generalne programske funkcije. Polje predstavlja određeni podatak o objektu koji se direktno sprema na neko mjesto u memoriji, bez prethodnih akcija. Svojstvo također predstavlja određeni podatak o objektu, ali je u srži metoda koja olakšava dohvat (engl. *get*) i spremanje (engl. *set*) tog podatka u pripadno, implicitno definirano polje. U Primjeru 2.1 prikazana je klasa `Dog`, koja sadrži polje `Birthday`, svojstvo `Age` i metodu `Bark()`.

```
public class Dog
{
    public DateTime Birthday; // Polje
    public int Age // Svojstvo
    {
        get
        {
            var age = DateTime.Now.Year - Birthday.Year;
            return DateTime.Now < Birthday.AddYears(age) ? --age : age;
        }
        set { }
    }
    public void Bark() // Metoda
    {
        Console.Write("Vau vau");
    }
}
```

Primjer 2.1: Klasa `Dog`

Svaka klasa ima najmanje jedan konstruktor. Ako sami ne definiramo nijedan konstruktor, klasa dobiva implicitni konstruktor koji ne prima nijedan parametar. Pozivanjem konstruktora klase, poput klase `Dog` iz Primjera 2.1, izrađujemo objekt. Drugim riječima, u

memoriju spremamo zadanu vrijednost polja `Birthday` (što je zapravo zadana vrijednost strukture `DateTime`, odnosno vrijednost `1.1.0001. 00:00:00`), podatke za izvršavanje metoda `get` i `set` svojstva `Age`, podatke za izvršavanje metode `Bark()`, te cijelom tom dijelu memorije pridružujemo jedinstvenu (objektnu) referencu. Ako samo deklariramo instancu klase `Dog`, bez uporabe konstruktora, dobivamo `null` objekt, to jest, prazan objekt. Prazan objekt nema spremljene zadane podatke u memoriji i ne dobiva objektnu reference, već praznu, to jest `null` referencu. Vidi Primjer 2.2.

```
Dog lassie = new Dog(); // lassie ima objektnu referencu

Dog marley; // marley nema objektnu referencu
```

Primjer 2.2: Instanciranje klase `Dog`

Zbog toga, u C#-u nije dozvoljeno dohvaćati članove objekta koji je reference `null`, jer njegovi članovi u memoriji ne postoje. Ako takvo što pokušamo, izbacit će se greška u periodu izvršavanja aplikacije (engl. *runtime error*), poznatija kao `null` iznimka (engl. *`null` exception*). Za svaki objekt koji nismo eksplicitno konstruirali (već, recimo, dohvatili iz nekog izvora), potrebno je redovito provjeravati ima li `null` referencu. Osim direktne provjere pomoću klasičnog `if` uvjeta, postoje i operatori koji mogu pomoći u kompaktnijem i preglednijem pisanju koda:

- Operator `?` primjenjuje se prije dohvata nekog člana objekta. Operator provjerava je li objekt `null` i ako nije, dohvaća traženi član. Ako objekt jest `null`, operator vraća nazad `null` bez pokušaja dohvata člana ili bacanja iznimke.
- Operator `??` koristi se kad se želi izbjeći potencijalno preuzimanje `null` reference s nekog objekta. Operator će provjeriti je li mu objekt slijeva `null` te ako nije, vratit će njegovu referencu. Ako objekt slijeva jest `null`, vratit će referencu onog objekta koji definiramo zdesna (naravno, uz uvjet da smo definirali objekt istog tipa kao što je objekt slijeva).

Različite načine ispitivanja `null` objekta možemo vidjeti na Primjeru 2.3.

```
Dog lassie = new Dog();
// Konzola ispisuje:  Vau vau
lassie.Bark();

Dog marley;
// Program ne ulazi u if-blok.  Nema iznimke.  Konzola ništa ne ispisuje.
if (marley != null)
{
    marley.Bark();
}
// Vraća se null vrijednost.  Nema iznimke.  Konzola ništa ne ispisuje.
marley?.Bark();
// Prepisuje se referenca od lassie.
Dog nonNullDog = marley ?? lassie;
// Baca se null iznimka.
marley.Bark();
```

Primjer 2.3: Ispitivanje `null` vrijednosti

`null` iznimka ne odnosi se samo na referentne tipove (objekte), već i na tzv. `null`-abilne vrijednosne tipove. Vrijednosni tipovi su standardni tipovi poput `int`, `char`, `double` i slično. Ne prenose se po referenci (osim ako ju eksplicitno ne proslijedimo), nego po vrijednosti, koja je uvijek zadana. `null`-abilni vrijednosni tipovi su vrijednosni tipovi koji mogu imati vrijednost `null`. Svaki vrijednosni tip može se pretvoriti u `null`-abilni vrijednosni tip, koristeći operator `?` na kraju oznake tipa. Recimo, tip `int?` je podatak kojem je inherentni tip (engl. *underlying type*) `int`, ali može imati i vrijednost `null` u memoriji. `null`-abilni tipovi su korisni ako ne želimo da nam član nekog objekta ima zadanu vrijednost, već da uopće nema vrijednost sve dok ju eksplicitno ne pridružimo. Primjer 2.4 pokazuje rad s `null`-abilnim vrijednosnim tipovima.

```
int? number;
if (number.HasValue) // Program ne ulazi u if-blok.
{
    number.Value++;
}
var copiedNumber = number?.Value ?? 1; // Prepisuje se vrijednost 1.
var incrementedNumber = number.Value + 1; // Baca se null iznimka.
```

Primjer 2.4: `null`-abilni vrijednosni tipovi

Lambda izrazi

Lambda izrazi u C#-u su sintaktički skraćeni način pisanja anonimnih funkcija koje se mogu proslijediti kao argumenti metodama. Lambda izrazi su posebno korisni za definiranje funkcija "u hodu", kad je izraz dovoljno jednostavan da nije potrebno definirati posebnu metodu. Primjer 2.5 pokazuje jednostavni lambda izraz, koji prima cijeli broj, te vraća njegovu vrijednost pomnoženu s 2.

```
// Izraz zdesna je lambda izraz, koji pridružujemo delegatu multiply.
Func<int, int> multiply = x => x * 2;

var product = multiply(5);

// Konzola ispisuje:  10
Console.Write(product);
```

Primjer 2.5: Lambda izraz

Biblioteka LINQ

Language Integrated Query (LINQ) biblioteka je alat u C# i .NET ekosustavu koji implementira pisanje upita unutar koda na konzistentan i čitljiv način. Biblioteka omogućava pretraživanje, filtriranje i transformaciju podataka neovisno o tome gdje se podaci nalaze; u memoriji, u bazi podataka ili XML dokumentima.

LINQ koristi funkcionalni stil programiranja za rad s podacima, to jest, implementira metode poput `Where`, `Select`, `OrderBy` i drugih, koje odgovaraju sintaksi dohvaćanja podataka iz nekog općenitog izvora. LINQ podržava sljedeće načine dohvata podataka:

- *LINQ to Objects* (rad s kolekcijama objekata u memoriji)
- *LINQ to SQL* (izvršavanje SQL upita na bazama podataka)
- *LINQ to XML* (rad s XML datotekama)

Primjer 2.6 prikazuje dohvat svih vrijednosti polja `Birthday`, iz liste objekata tipa `Dog`, kojima svojstvo `Age` ima vrijednost veću od 5.

```
var dogs = new List<Dog>
{
    new Dog
    {
        Birthday = DateTime.Now.AddYears(-7)
    },
    new Dog
    {
        Birthday = DateTime.Now.AddYears(-6)
    },
    new Dog
    {
        Birthday = DateTime.Now.AddYears(-3)
    }
};

// Rezultat je lista s vrijednošću polja Birthday
// prva dva objekta iz kolekcije dogs.
List<DateTime> olderDogsBirthdays = dogs.Where(dog => dog.Age > 5)
    .Select(dog => dog.Birthday)
    .ToList();
```

Primjer 2.6: Sintaksa biblioteke LINQ

Asinkrono programiranje

Asinkrono programiranje u C#-u omogućava izvršavanje dugotrajnih zadataka, poput pristupa bazi podataka, mrežnih zahtjeva ili čitanja i pisanja datoteka, bez blokiranja glavne dretve programa. Sintaksa asinkronog programiranja u C#-u sastoji se od sljedećih ključnih riječi

- ključna riječ `async` označava metodu koja može sadržavati asinkroni kod. Metoda označena ovom riječi mora biti `void` ili vraćati objekt tipa `Task` ili `Task<T>`.
- ključna riječ `await` označava mjesto u kodu gdje program treba čekati dovršetak neke asinkrone operacije, ali bez blokiranja glavne dretve. Drugim riječima, kod u asinkronoj metodi se zaustavlja dok se zadatak ne dovrši, a kontrola se privremeno vraća njezinom pozivatelju.

Primjer 2.7 pokazuje primjenu asinkronog programiranja.

```
static async Task Main()
{
    // Pozivamo asinkronu metodu koja simulira dugotrajan zadatak.
    var result = await GetDataAsync();
    Console.WriteLine(result);
}

static async Task<string> GetDataAsync()
{
    // Simuliraj dugotrajnu operaciju.
    await Task.Delay(3000);
    return "Podaci";
}
```

Primjer 2.7: Sintaksa asinkronog programiranja

Biblioteka Fluent Results

Fluent Results je biblioteka u .NET-u koja omogućuje jednostavno i pregledno upravljanje rezultatima operacija u kodu. Umjesto klasičnog vraćanja rezultata kroz bacanje iznimki, ova biblioteka implementira specijalizirane klase za prijenos informacija o uspješnosti operacija. Korištenje biblioteke prikazano je u Primjeru 2.8.

```
using FluentResults;
public Result<int> Divide(int dividend, int divisor)
{
    if (divisor == 0)
    {
        return Result.Fail<int>("Dijeljenje s nulom!");
    }
    var quotient = dividend / divisor;
    return Result.Ok(quotient);
}
public void Division()
{
    var result = Divide(10, 0);
    var consoleOutput = result.IsSuccess ? "Rezultat je: " + result.Value
        : result.Errors.First().Message;
    Console.WriteLine(consoleOutput); // Konzola ispisuje:  Dijeljenje s nulom!
}
```

Primjer 2.8: Korištenje biblioteke Fluent Results

2.2 Razor komponente

Razor komponente su alat u Blazoru koji omogućava lakšu i organiziraniju izgradnju klijentskog koda C# programerima. Koriste Razor sintaksu, ali ju i proširuju dodatnim funkcionalnostima, poput interakcije među komponentama i upravljanja stanjem. Zbog toga, Razor komponente podsjećaju na komponente koje se koriste u razvojnim okvirima poput Reacta, Angulara, Vue.js-a i sličnih.

Osnovne značajke

Razor komponenta definirana je kodom koji je napisan koristeći Razor sintaksu i koji je spremljen u datoteku s ekstenzijom `.razor`. Naziv Razor komponente uvijek je isti kao naziv pripadne datoteke. Komponenta se sastoji od HTML dijela u kojem se koristi ranije spomenuta sintaksa oznaka te od C# dijela koji je određen blokom `@code`. Komentari u HTML dijelu Razor komponente omeđeni su znakovima `@*` i `*@`.

Primjer 2.9 prikazuje Razor komponentu `TaskList` koja prikazuje, odnosno skriva elemente kolekcije `Tasks` ovisno o vrijednosti polja `TasksHidden`.

```
@if (TasksHidden)
{
    <p class="no-tasks">Zadaci su skriveni.</p>
}
else
{
    @foreach (var task in Tasks)
    {
        <div class='task-item'
            @* Dinamičko mijenjanje CSS-a pomoću C# koda.  *@
            style="color:@(task == "Important task!" ? "red" : "black");">
           @task
        </div>
    }
}
<button @onclick="ToggleTasksHidden">Klikni</button>
@code {
    private string[] Tasks = { "Task 1", "Task 2", "Important task!" };
    private bool TasksHidden = false;
    private void ToggleTasksHidden() => TasksHidden = !TasksHidden;
}
```

Primjer 2.9: Razor komponenta `TaskList`

Klikom na gumb *Klikni*, vrijednost polja se mijenja. U primjeru provjeravamo vrijednost polja `TasksHidden` pomoću `@if` uvjeta, to jest, klasičnog `if` uvjeta C#-a koji se koristi unutar HTML dijela Razor komponente. Slično, listu zadataka ispisujemo pomoću `@foreach` petlje. Općenito, prefiks `@` u HTML dijelu Razor komponente označava umetanje C# koda. U Primjeru 2.9 također možemo vidjeti umetanje C# koda u atributu `style` elementa `<div>` kako bismo mogli dinamički promijeniti boju elementa ovisno o varijabli `task`.

Svaka Razor komponenta može se koristiti unutar neke druge komponente. U Primjeru 2.10 možemo vidjeti kako bi komponenta `TaskOverview` koristila komponentu `TaskList`.

```
<h3>@Title</h3>
@* Umetanje komponente TaskList u komponentu TaskOverview.  *@
<TaskList />
@code {
    private string Title = "Pregled zadataka";
}
```

Primjer 2.10: Datoteka `TaskOverview.razor`

Dakle, prikaz komponente `TaskOverview` uključivao bi prikaz teksta *Pregled zadataka* te prikaz komponente `TaskList` odmah ispod. U ovom kontekstu komponenta `TaskOverview` je roditeljska komponenta, dok je komponenta `TaskList` podređena komponenta, odnosno dijete.

Upravljanje stanjem

Stanje Razor komponente je skup podataka koji određuju sadržaj i status komponente u bilo kojem trenutku. Radi se o, primjerice, privatnim poljima ili svojstvima u C# kodu, koja na neki način mijenjaju određeni HTML element. Čim dođe do promjene u stanju, komponenta se osvježava. U Primjeru 2.11, za upravljanje stanjem definiran je događaj `@bind` koji automatski povezuje korisnikov unos s C# varijablom `ime`. Posljedično, tekst u elementu `<p>` automatski se osvježava čim korisnik izmijeni unos u elementu `<input>`.

```
<input type="text" @bind="name" placeholder="Input name" />
<p>Hello, @name!</p>
@code {
    private string name = "";
}
```

Primjer 2.11: Upravljanje stanjem

Dijeljenje podataka među komponentama

Razor komponente mogu prihvaćati parametre pomoću atributa `[Parameter]`, čime se omogućuje prosljeđivanje podataka iz roditeljske komponente u podređenu. Definirajmo komponentu `ChildComponent` kao u Primjeru 2.12. Komponenta prihvaća parametar `DisplayText`.

```
@* ChildComponent.razor *@
<p>@DisplayText</p>
@code {
    [Parameter]
    public string DisplayText { get; set; }
}
```

Primjer 2.12: Komponenta `ChildComponent` koja prihvaća parametar

Tada bi komponenta koja koristi komponentu `ChildComponent`, mogla proslijediti vrijednost `DisplayText` parametra, na način prikazan u Primjeru 2.13.

```
@* ParentComponent.razor *@
<ChildComponent DisplayText="@TextFromParent" />
@code {
    private string TextFromParent = "Hello from parent!";
}
```

Primjer 2.13: Prosljeđivanje parametra komponenti `ChildComponent`

Komunikacija među komponentama

Osim prosljeđivanja podataka putem parametara, moguće je koristiti događaje za komunikaciju među komponentama. Podređena komponenta može obavijestiti roditeljsku komponentu o nekom događaju korištenjem Blazor strukture `EventCallback`.

```
@* ChildComponent.razor *@
<button @onclick="OnClick">Click me!</button>
@code {
    [Parameter]
    public EventCallback OnClickEvent { get; set; } // Proslijeđena akcija.
    // Obavijest roditelju da izvrši akciju.
    private async Task OnClick() => await OnClickEvent.InvokeAsync();
}
```

Primjer 2.14: Komponenta `ChildComponent` koja prihvaća događaj

U Primjeru 2.14, komponenta `ChildComponent` prima akciju za `onclick` događaj na gumbu, te obavještava roditeljsku komponentu da obavi proslijeđenu akciju čim se gumb klikne. U Primjeru 2.15 vidimo prosljeđivanje akcije u roditeljskoj komponenti.

```
@* ParentComponent.razor *@

@* Roditelj prosljeđuje željenu akciju.  *@
<ChildComponent OnClickEvent="@OnClickAction" />

@code {
    private void OnClickAction()
    {
        Console.WriteLine("Child component event!");
    }
}
```

Primjer 2.15: Komponenta `ParentComponent` prosljeđuje događaj

Primijetimo da se proslijeđena akcija izvršava asinkrono kako se korisnička interakcija s komponentom `ChildComponent` (i aplikacijom generalno) ne bi zaustavila. Općenito je dobra praksa izvršavati roditeljske akcije asinkrono, iako u našem primjeru akcija nije dugotrajna. Roditeljskih akcija koje se pozivaju od strane podređene komponente često bude nekoliko i mogu se izvršavati jedna za drugom. Takve situacije mogu dovesti do onemogućavanja korisničke interakcije, bez obzira na trajanje izoliranih akcija. Iz tog razloga je potrebno ostaviti kontrolu dretve podređenoj komponenti i pustiti roditeljsku komponentu da se ažurira u pozadini.

Metode životnog ciklusa

Razor komponenta u svojoj suštini je klasa u C#-u koja nasljeđuje Blazorovu baznu klasu `ComponentBase`. Klasa `ComponentBase` ima definirane metode `OnInitialized()`, `OnParameterSet()` i `OnAfterRender()`, koje se pozivaju u trenucima inicijalizacije, postavljanja parametra i izrađenog prikaza komponente, redom.

Ove metode se nazivaju metode životnog ciklusa Razor komponente (engl. *component lifecycle methods*). Metode su virtualne, što znači da ih svaka nasljedna klasa može nadjačati (engl. *override*). Ovo nam omogućava dodavanje proizvoljnih akcija u ključnim trenucima životnog ciklusa Razor komponente.

Primjer 2.16 pokazuje korištenje metode `OnInitialized()` koja će ispisati potvrdu u konzoli svaki put kad se komponenta inicijalizira.

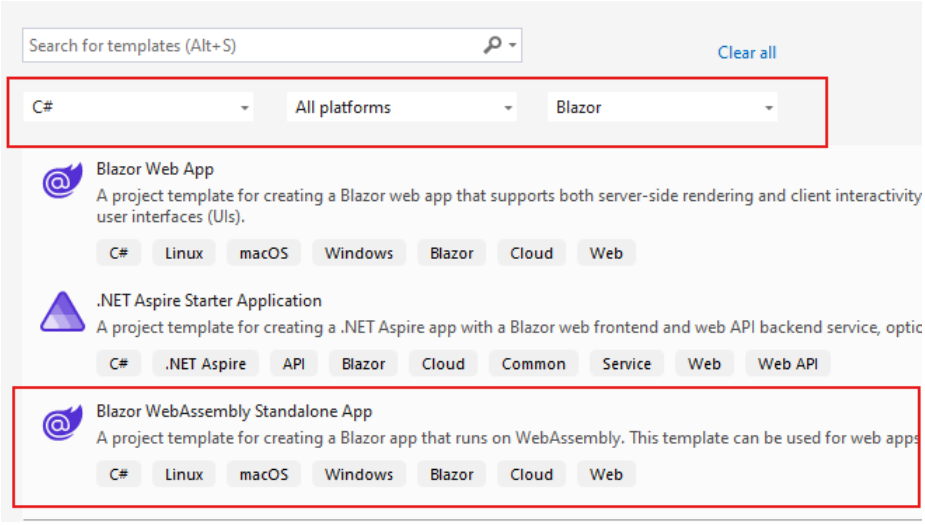
```
@* MyComponent.razor *@
@code {
    protected override void OnInitialized()
    {
        Console.WriteLine("Component is initialized!");
    }
}
```

Primjer 2.16: Metoda životnog ciklusa `OnInitialized()`

2.3 Izrada Blazor projekta u okruženju Visual Studio

Platforma .NET integrirana je u razvojno okruženje Visual Studio koje ćemo koristiti za razvoj aplikacije. Unutar okruženja potrebno je instalirati paket alata *ASP.NET And Web Development* kako bi projektni obrasci Blazora bili dostupni.

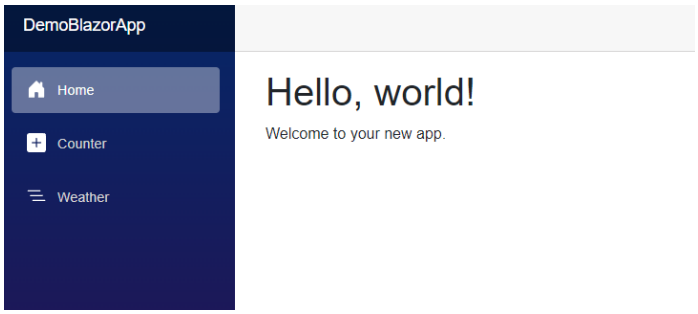
Pokretanjem aplikacije Visual Studio nude se opcije otvaranja postojećeg ili izrade novog projekta. Biramo opciju *Create a new project*. Zatim tražimo željeni razvojni okvir. Parametre pretrage namjestimo kao na Slici 2.1, birajući jezik C# i okvir Blazor. Među rezultatima pretrage, biramo *Blazor WebAssembly Standalone App* kao željeni predložak projekta.



Slika 2.1: Pretraga projekta

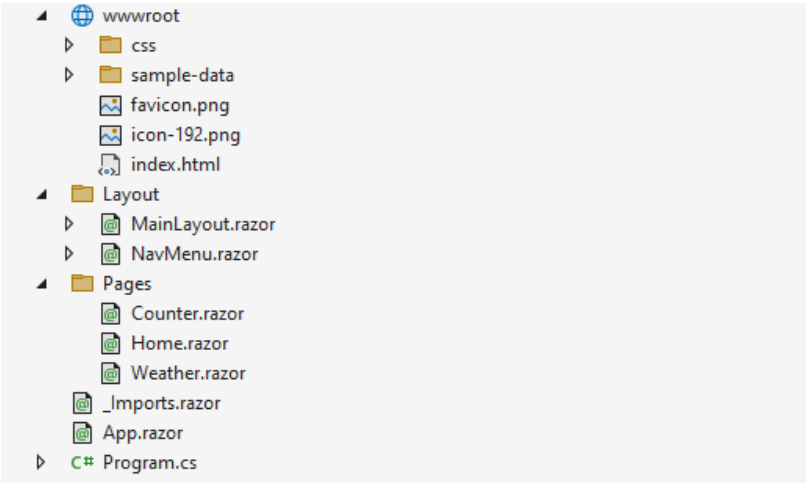
Iduće dvije stranice nude opcije konfiguracije projekta. Dovoljno je samo unijeti ime projekta na prvoj stranici i ostale opcije ostaviti kakve jesu. Klikom gumba *Create* na posljednoj stranici dobivamo novi projekt u Blazor WebAssemblyju.

Rješenje (engl. *solution*), odnosno strukturu koja organizira i povezuje dijelove projekta, možemo odmah izgraditi i pokrenuti klikom gumba *Start Without Debugging* ili `Ctrl + F5` prečacem na tipkovnici. U pregledniku se pokreće demo aplikacija, prikazana na Slici 2.2.



Slika 2.2: Demo aplikacija

Povratkom u Visual Studio i pregledom sadržaja u Solution Exploreru, istaknut ćemo neke mape i datoteke. Istaknuti sadržaj prikazan je na Slici 2.3.



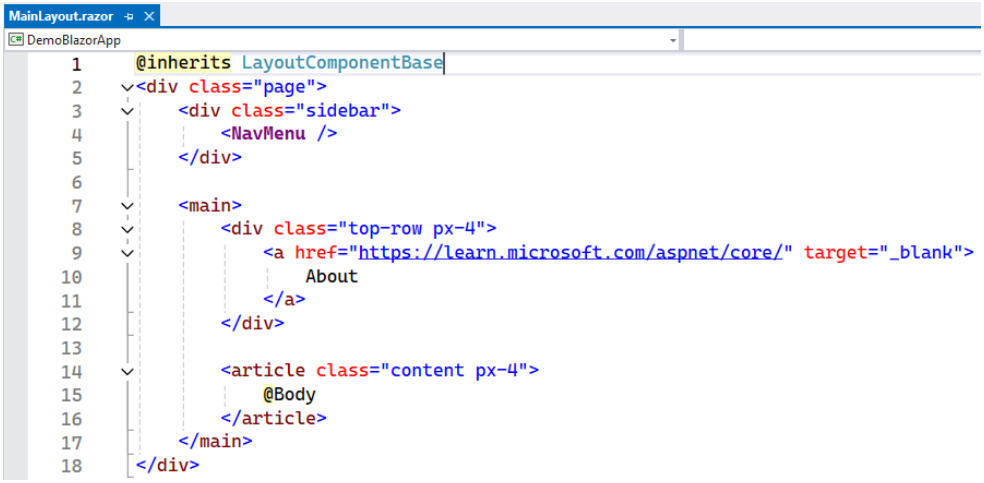
Slika 2.3: Sadržaj rješenja projekta

- Mapa `wwwroot` sadrži statičke resurse aplikacije koji će se učitati u pregledniku, poput HTML i CSS datoteka, JavaScript datoteka, slika i slično. Sadrži datoteku `index.html`, ulaznu točku preglednika u aplikaciju. U datoteci `index.html` nalazi se osnovni HTML kod te se učitava modul WebAssemblyja.

- Datoteka `Program.cs` početna je točka aplikacije u kojoj se konfigurira njezino pokretanje. U ovoj datoteci se također postavlja umetanje zavisnosti (engl. *dependency injection*), softverski obrazac dizajna koji ćemo detaljnije opisati kasnije.

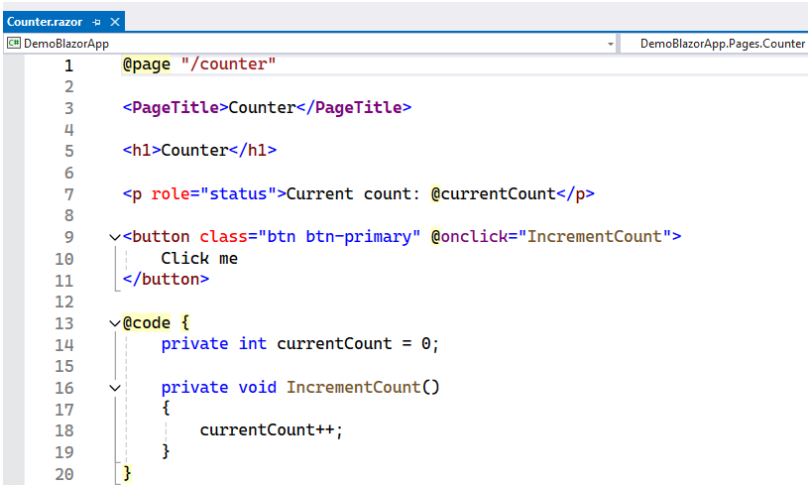
- Datoteka `App.razor` implementira Razor komponentu `App`, korijen Razor komponenti aplikacije. `App` je osnovna Razor komponenta u kojoj se postavljaju početne komponente te u kojoj se nalazi komponenta `Router` za navigaciju među stranicama.

- Mapa `Layout` sadrži Razor komponente koje definiraju osnovni izgled aplikacije (npr. `header`, `footer` i `body`). U prikazanoj demo aplikaciji na Slici 2.2, vidljivi su izgrađeni sadržaji ove mape. Komponenta `MainLayout` sadrži komponentu `NavMenu` (ljubičasti izbornik na lijevoj strani) te sadržaje stranice koju je korisnik otvorio na mjestu elementa `@Body` u datoteci `MainLayout.razor` (Slika 2.4).



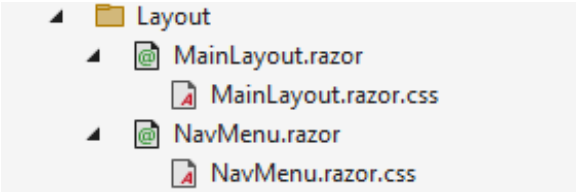
Slika 2.4: Sadržaj datoteke `MainLayout.razor`

- Mapa `Pages` sadrži Razor komponente koje definiraju stranice aplikacije. Stranica je definirana ključnom riječi `@page` na početku datoteke te željenim navigacijskim nastavkom, odnosno rutom (Slika 2.5). Početna stranica koja se otvara automatski pri pristupu aplikaciji ima prazni navigacijski nastavak ”/”. U rješenju demo aplikacije, to je komponenta `Home`.



Slika 2.5: Datoteka `Counter.razor` definira stranicu `/counter`

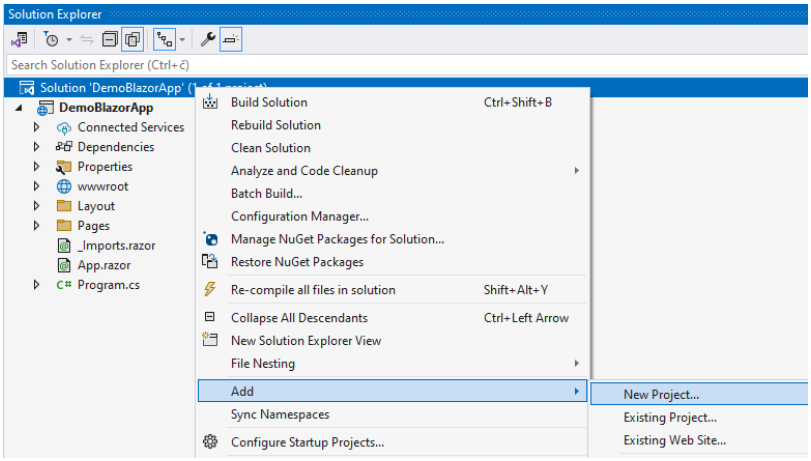
Osim u globalnom CSS-u koji se nalazi u mapi `wwwroot`, stil Razor komponenti možemo definirati u tzv. izoliranom CSS-u. Izolirana CSS datoteka je datoteka čiji se CSS primjenjuje samo na HTML elemente u datoteci pripadne Razor komponente. U demo aplikaciji, komponente `MainLayout` i `NavMenu` imaju definiran izolirani CSS (Slika 2.6). Ako se Razor komponenta zove `Component.razor`, tada se njezina izolirana CSS datoteka mora zvati `Component.razor.css`.



Slika 2.6: Izolirani CSS Razor komponenti

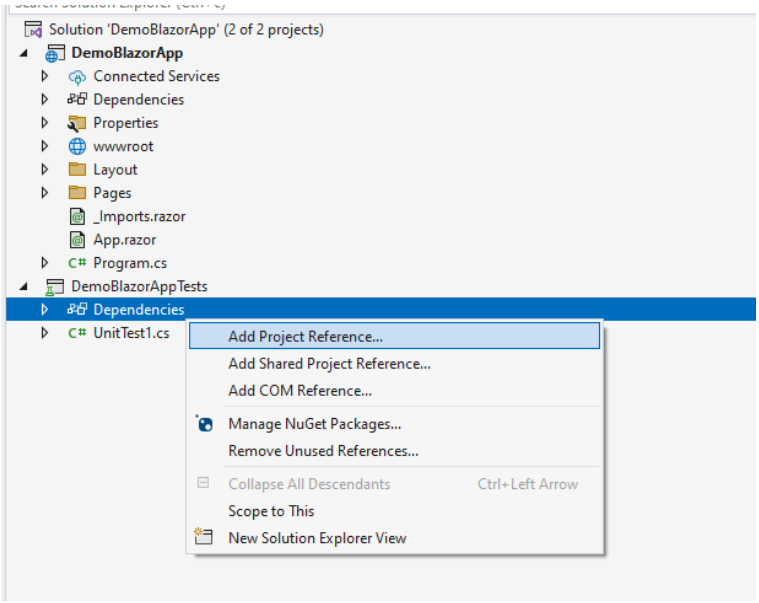
U rješenju demo aplikacije imamo jedan projekt, odnosno glavni projekt koji sadrži implementaciju funkcionalnosti aplikacije. Često se uz glavni projekt javlja potreba za sporednim projektima, odnosno komponentama rješenja koje ne implementiraju aplikaciju direktno, ali sudjeluju u njezinom životnom ciklusu na neki način. To su, recimo, projekti s prilagođenim bibliotekama ili projekti u kojima se implementiraju testovi.

Novi projekt dodajemo desnim klikom na rješenje te odabirom opcije *Add*. U izborniku opcije *Add* biramo *New Project...* (Slika 2.7).



Slika 2.7: Dodavanje novog projekta u rješenje

Ponovno nam se otvara pretraga projektnih obrazaca koju smo vidjeli na Slici 2.1. Biramo željeni obrazac, upisujemo ime projekta, biramo verziju .NET-a i završavamo. Na Slici 2.8 dodan je novi projekt tipa *xUnit Test Project* koji smo nazvali `DemoBlazorAppTests`. Na istoj slici prikazana je opcija kojom dodajemo referencu drugog projekta kako bismo mogli koristiti njegov kod.



Slika 2.8: Dodavanje reference drugog projekta

Poglavlje 3

Razvoj aplikacije u Blazor WebAssemblyju

Izvorni kod aplikacije koju razvijamo u ovom poglavlju može se naći u sljedećem repozitoriju: https://github.com/josipasimic/SchemaPal.

3.1 Struktura aplikacije

Sadržaj aplikacije

Razvit ćemo aplikaciju za izradu relacijskih shema baza podataka. Relacijska shema je reprezentacija svih dijelova baze podataka s pripadnim tehničkim svojstvima potrebnim za implementaciju u specifičnom DBMS-u (Database Management System) za relacijske baze podataka. DBMS u našem slučaju je Microsoft SQL Server. U našoj aplikaciji, shema će uključivati tablice s njihovim imenima, stupcima (atributima) i vezama s drugim tablicama. Za svaki stupac u tablici prikazat ćemo ime, tip podatka i vrstu ključa (ako stupac predstavlja neki ključ). Naposljetku, uključit ćemo i indekse svake tablice, s pripadnim tipom i stupcima na kojima je indeks dodan.

Razlikovat ćemo i implementirati sljedeće funkcionalne cjeline:

- **Biranje opcija prijave** uključuje stranicu za izbor načina korištenja aplikacije (prijavljeni korisnik ili gost), te stranice za prijavu i registraciju. Aplikacija automatski preusmjerava korisnika na ovu stranicu svaki put kad se započne nova sesija (prvi pristup aplikaciji, odjava iz aplikacije ili istekla prijava).
- **Glavni izbornik** je zadana stranica koja se automatski prikazuje korisniku s aktivnom sesijom. Unutar ove stranice korisnik može izabrati željenu funkcionalnost za

izradu shema: izrada nove sheme, izrada sheme iz predloška, uvoz sheme iz JSON datoteke i, u slučaju da je korisnik prijavljen (tj. ne koristi aplikaciju kao gost), otvaranje i uređivanje ranije spremljene sheme.

- **Platno za izradu shema** je stranica koja se otvara čim korisnik izabere neku od funkcionalnosti izrade shema u glavnom izborniku. Sastoji se od proširivog izbornika s lijeve strane u kojem korisnik ima organizirani pregled sadržaja sheme i gdje može dodavati, brisati i ažurirati podatke svih dijelova sheme, te od platna na većem dijelu stranice na kojem korisnik može premještati tablice, crtati veze i izvesti shemu u željenom formatu.

- **Korisnički račun (API integracija)** je skup funkcionalnosti koje zahtijevaju komunikaciju s poslužiteljskom aplikacijom. Poslužiteljska aplikacija implementira prijavu i registraciju korisnika te spremanje, dohvat i brisanje shema prijavljenih korisnika. Navedene funkcionalnosti aplikacija izlaže kroz API (Application Programming Interface).

Organizacija koda

Organizirat ćemo kod po tipovima klasa ili datoteka, a potom po funkcionalnim cjelinama. Imenički prostori (engl. *namespace*) pratit će organizaciju datoteka u rješenju. Testni projekt pratit će organizaciju glavnog projekta.

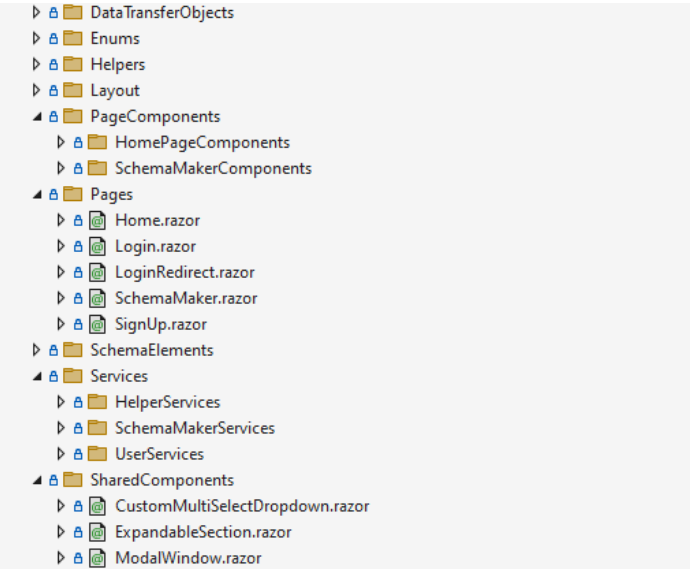
Razor komponente koje implementiraju stranice nalazit će se u mapi `Pages`. Sve kompleksnije stranice koristit će podređene Razor komponente koje će se nalaziti u mapi `PageComponents`. Ako neka stranica koristi više podređenih komponenti, unutar mape `PageComponents` napravit ćemo posebnu mapu samo za njezine komponente. Podređene komponente koje nisu specifične samo za jednu stranicu, već se mogu koristiti na općenitoj Razor komponenti, bit će spremljene u mapi `SharedComponents`.

Skoro sve metode vezane za događaje unutar komponenti pozivat će pripadne servise (klase) za izvršavanje potrebnih akcija. Primjerice, gumb za dodavanje nove tablice na platno pozivat će metodu `CreateNewTable` iz sučelja `ISchemaObjectFactory`. Klasa `SchemaObjectFactory` implementirat će navedeno sučelje te će imati jedinstvenu odgovornost upravljanja elementima unutar sheme. Dakle, razdvojit ćemo domensku logiku u specijalizirane klase s jedinstvenim odgovornostima. Također ćemo apstrahirati implementacije tih klasa definiranjem odgovarajućih sučelja koja će se koristiti u Razor komponentama (i općenito ostatku koda). Drugim riječima, razvijat ćemo aplikaciju prateći SOLID principe [8] objektno orijentiranog programiranja.

Sve servise, odnosno sučelja i pripadne implementacije, spremat ćemo u mapu `Services`. Unutar te mape razdvojit ćemo ih po funkcionalnim domenama. Primjerice, servise koji služe specifično za izradu shema ćemo spremati u novu mapu unutar mape `Services`. Napravit ćemo i mapu za klase koje predstavljaju elemente shema. Konkretno, u toj mapi nalazit će se klase poput `Table`, `Column`, `Relationship` i drugih.

Uz to ćemo koristiti i neke pomoćne klase za čuvanje i prijenos podataka među Razor komponentama koje ćemo spremati u mapu `Helpers`. Izradit ćemo i mapu `Enums` u koju ćemo spremati sve enumeracije različitih svojstava u aplikaciji. Naposljetku, dodat ćemo i mapu `DataTransferObjects` u kojoj će se nalaziti klase s najosnovnijim podacima za prijenos na API.

Organizacija koda prikazana je na Slici 3.1.



Slika 3.1: Organizacija sadržaja u projektu

Umetanje zavisnosti

Umetanje zavisnosti (engl. *dependency injection*) u .NET-u je obrazac dizajna kojim odvajamo određene odgovornosti od nižih dijelova softvera. Temelji se na mehanizmu prijenosa (umetanja) potrebnih servisa (zavisnosti) određenoj klasi, bez da ih ta klasa mora izrađivati ili dohvaćati samostalno. Korištenje ovog obrasca značajno olakšava pisanje tes-

tova i općenito održavanje aplikacije.

Obrazac primjenjuje princip inverzije kontrole, što znači da klasa više ne kontrolira stvaranje njoj potrebnih zavisnosti. Umjesto toga, zavisnosti joj se prenose iz drugog dijela programa, obično iz IoC (Inversion of Control) kontejnera, alata za upravljanje zavisnostima. Platforma .NET ima ugrađeni IoC kontejner koji pomaže pri registraciji i rukovanju zavisnostima unutar aplikacije. Zavisnosti se najčešće registriraju koristeći tri osnovne metode:

- Metoda `AddSingleton` registrira klasu koja se instancira samo jednom i koristi s jedinstvenom referencom tijekom cijelog životnog ciklusa aplikacije. Ovo su najčešće klase koje nemaju nikakva polja ili svojstva (engl. *stateless class*) pa dijeljenje reference ne utječe na objekt.
- Metoda `AddScoped` registrira klasu koja se instancira za pojedini opseg rada aplikacije. Ovo su najčešće klase čije stanje, odnosno referencu želimo očuvati na razini korisničkih sesija.
- Metoda `AddTransient` registrira klasu koja se instancira svaki put kada se zatraži.

Zavisnosti se registriraju u početnoj datoteci programa, što je u našem slučaju `Program.cs`. Registriranje servisa u našoj aplikaciji može se vidjeti u Primjeru 3.1. Registriramo ih s njihovim sučeljima kako bismo mogli koristiti ta sučelja u ostatku koda te kako bi IoC kontejner znao koju je implementaciju sučelja potrebno instancirati i proslijediti.

```
using SchemaPal.Services;
// ...
var builder = WebAssemblyHostBuilder.CreateDefault(args);
// ...
builder.Services.AddSingleton<IPositionService, PositionService>();
// ...
builder.Services.AddScoped<IUserSessionService, UserSessionService>();
// ...
builder.Services.AddTransient<ISchemaObjectFactory, SchemaObjectFactory>();
```

Primjer 3.1: Registriranje zavisnosti u datoteci `Program.cs`

Nakon što smo registrirali sve servise, odnosno zavisnosti koje će nam biti potrebne, umećemo ih u odgovarajuće klase. Zavisnosti se najčešće umeću u obične klase na dva načina: kroz konstruktor ili kroz svojstvo. Mi ćemo koristiti konstruktor. U Primjeru 3.2, klasi `PositionService` potreban je servis `CoordinatesCalculator` za računanje koordinata elemenata, stoga umećemo potrebno sučelje u konstruktor.

```
public class PositionService : IPositionService
{
    private readonly ICoordinatesCalculator _coordinatesCalculator;
    public PositionService(ICoordinatesCalculator coordinatesCalculator)
    {
        _coordinatesCalculator = coordinatesCalculator;
    }
    // ...
 }
```

Primjer 3.2: Umetanje zavisnosti u običnu klasu pomoću konstruktora

Zavisnosti možemo umetnuti i u Razor komponente koristeći direktivu `@inject`. Primjer 3.3 pokazuje Razor komponentu `TableComponent` koja prikazuje tablice na platnu za izradu shema, za što joj je potreban servis `StyleService` koji vraća očekivane CSS značajke pojedinih dijelova tablice.

```
@* SchemaMakerComponents/CanvasComponents/TableComponent.razor *@
@inject IStyleService _styleService
@* ...  *@
<div class="connection-point"
   @* ...  *@
   @onmouseover="() => SetConnectionPointColorOnMouseOver(connectionPoint.Id)">
</div>
@* ...  *@
@code {
   private async Task SetConnectionPointColorOnMouseOver(/* ... */)
   {
       _styleService.SetConnectionPointsColor(/* ... */);
   }
   // ...
}
```

Primjer 3.3: Umetanje i korištenje zavisnosti u komponenti `TableComponent`

3.2 Jednostavne funkcionalnosti

Koristeći opisane alate, principe i obrasce implementirat ćemo svaku ranije opisanu funkcionalnu cjelinu aplikacije, počevši s onim jednostavnima. Jednostavne cjeline uključuju stranice s opcijama prijave i glavni izbornik. Razvoj počinjemo od stranica, koje potom razlažemo na komponente te čije funkcije implementiramo u odgovarajućim servisima.

Biranje opcija prijave

U ovaj skup funkcionalnosti uključujemo tri stranice: stranicu za opcije prijave, stranicu za prijavu i stranicu za registraciju. Sve tri stranice bit će poprilično jednostavne, bez podređenih komponenti.

Stranica za opcije prijave sadržavat će tri gumba koji će služiti za preusmjeravanje korisnika na druge stranice. Za preusmjeravanje korisnika na neku stranicu koristimo Blazorov ugrađeni servis `NavigationManager`. Implementacija stranice prikazana je u Primjeru 3.4. Klik gumba *Nastavi kao gost* preusmjerava korisnika na početnu stranicu (glavni izbornik) i započinje sesiju gosta.

```
@* LoginRedirect.razor *@
@inject NavigationManager _navigationManager
@inject IUserSessionService _userSessionService
<div class="main-content">
    <div class="auth-options">
        <button class="btn-primary" @onclick="Login">Prijavi se</button>
        <button class="btn-secondary" @onclick="SignUp">Registriraj se</button>
        <button class="btn-outline-dark" @onclick="ContinueAsGuest">Nastavi kao
            gost</button>
    </div>
</div>
@code {
    private void Login() => _navigationManager.NavigateTo("/login");
    private void SignUp() => _navigationManager.NavigateTo("/signup");
    private async void ContinueAsGuest()
    {
        await _userSessionService.StartGuestUserSession();
        _navigationManager.NavigateTo("/");
    }
}
```

Primjer 3.4: Opcije prijave i `NavigationManager`

U Blazor WebAssemblyju, za upravljanje sesijama preporuča se instalacija biblioteke SessionStorage koja sadrži `SessionStorageService`, servis za upravljanje podacima o korisničkoj sesiji unutar prostora za pohranu podataka u pregledniku. Upravljanje podacima sesije gosta pomoću ovog servisa prikazano je u Primjeru 3.5.

U svrhe korištenja servisa `SessionStorageService` na jedinstvenom mjestu u kodu, implementirat ćemo pomoćni servis `UserSessionService` koji se brine za kreiranje i prekid svih vrsta korisničkih sesija u aplikaciji.

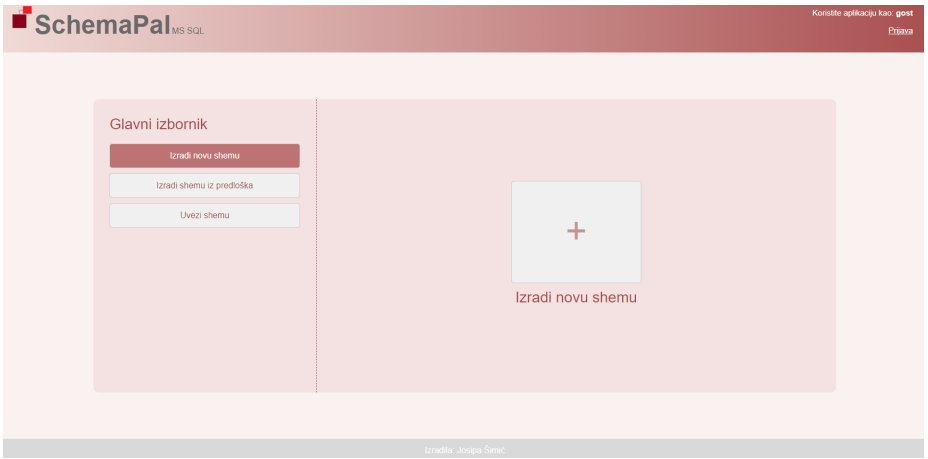
```
using Blazored.SessionStorage;
public class UserSessionService : IUserSessionService
{
    private readonly ISessionStorageService _sessionStorage;
    public UserSessionService(ISessionStorageService sessionStorageService) {
        _sessionStorage = sessionStorageService;
    }
    public async Task StartGuestUserSession()
    {
        await _sessionStorage.SetItemAsync("isLoggedIn", false);
        await _sessionStorage.SetItemAsync("isGuest", true);
    }
}
```

Primjer 3.5: Korištenje servisa `SessionStorageService`

Stranice za prijavu i registraciju korisnika vrlo su slične. Osim gumba, sadrže standardna tekstualna polja za unos potrebnih podataka te pozivaju API za autentifikaciju korisnika, o kojem ćemo više reći kasnije.

Glavni izbornik

Glavni izbornik je zadana stranica aplikacije. S lijeve strane sadrži gumbe kojima korisnik može izabrati željene funkcionalnosti, dok s desne strane prikazuje podređene komponente koje implementiraju ili preusmjeravaju na korištenje odgovarajućih funkcionalnosti. Izgled glavnog izbornika prikazan je na Slici 3.2.



Slika 3.2: Glavni izbornik

Implementacija glavnog izbornika prikazana je u Primjeru 3.6. Klikom odgovarajućeg elementa <li>, korisniku se u desnom dijelu izbornika prikazuje željena funkcionalnost: gumb za izradu nove sheme, predlošci za izradu shema ili dijalog za uvoz sheme iz JSON datoteke. Zadnje dvije funkcionalnosti odvojene su u posebne komponente.

```
@page "/"
@using SchemaPal.Enums
<div class="home-grid">
    <div class="sidebar">
        <h3>Glavni izbornik</h3>
        <ul class="nav-buttons">
            <li @onclick="ShowNewSchema">Izradi novu shemu</li>
            <li @onclick="ShowTemplateSchemas">Izradi shemu iz predloska</li>
            <li @onclick="ShowImportSchema">Uvezi shemu</li>
        </ul>
    </div>
    <div class="main-content">
        @if (currentView == HomePageViewType.NewSchema) {
            <button @onclick="OpenSchemaMakerCanvas">+</button>
            <h3>Izradi novu shemu</h3>
        }
        else if (currentView == HomePageViewType.TemplateSchemas) {
            <TemplateSchemas /> // Podređena komponenta za prikaz predložaka.
        }
        else if (currentView == HomePageViewType.ImportSchema) {
            <ImportSchema /> // Podređena komponenta za input file dijalog.
        }
    </div>
</div>
@code {
    private HomePageViewType currentView = HomePageViewType.NewSchema;
    private void ShowNewSchema() => currentView = HomePageViewType.NewSchema;
    private void ShowTemplateSchemas() => currentView =
        HomePageViewType.TemplateSchemas;
    private void ShowImportSchema() => currentView =
        HomePageViewType.ImportSchema;
    // ...
}
```

Primjer 3.6: Implementacija glavnog izbornika u datoteci `Home.razor`

Podređena komponenta za uvoz sheme iz JSON datoteke sadrži Blazorov dijalog `InputFile` kojim korisnik može učitati JSON datoteku. Sadržaj datoteke se čita i šalje u `SchemaInjectionService`, servis za prijenos dohvaćene sheme na stranicu za izradu. Ako je prijenos uspio, korisnika

se preusmjerava na stranicu. Inače se korisniku prikazuje odgovarajuća poruka. Implementacija je prikazana u Primjeru 3.7.

```
@inject NavigationManager _navigationManager
@inject ISchemaInjectionService _schemaInjectionService
<h3 class="import-font">Uvezi shemu iz <i>SchemaPal</i> JSON datoteke:</h3>
<InputFile OnChange="OnInputFileChange" accept=".json" />
<p class="message">@Message</p>
@code {
    private string Message = string.Empty;
    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        var file = e.GetMultipleFiles(maximumFileCount: 1).FirstOrDefault();
        using var stream = file.OpenReadStream();
        using var reader = new StreamReader(stream);
        var jsonData = await reader.ReadToEndAsync();
        ImportSchemaPalJson(jsonData);
    }
    private void ImportSchemaPalJson(string jsonData)
    {
        // Pročitani sadržaj iz JSON datoteke šalje se u servis za prijenos na
            stranicu za izradu.
        var result = _schemaInjectionService.PushSchemaFromJsonImport(jsonData);
        if (result.IsFailed)
        {
            Message = result.Errors.First().Message;
            return;
        }
        // Servis za prijenos je uspješno primio shemu, stoga korisnika
            preusmjeravamo na stranicu za izradu.
        _navigationManager.NavigateTo("/schema-maker");
    }
}
```

Primjer 3.7: Podređena komponenta `ImportSchema`

U Primjeru 3.8, metoda `PushSchemaFromJsonImport` pokušava deserijalizirati JSON sadržaj u `DatabaseSchema` objekt te, ovisno o uspjehu operacije, vraća odgovarajući objekt tipa `Result`. Ako je deserijalizacija uspješna, metoda sprema deserijalizirani objekt u polje klase. Objekt će biti tamo sve dok ga stranica za izradu shema ne preuzme.

```
public class SchemaInjectionService : ISchemaInjectionService
{
    private DatabaseSchema InjectedSchema { get; set; } = null;
    //...
    public Result PushSchemaFromJsonImport(string jsonSchema)
    {
        DatabaseSchema databaseSchema;
        try
        {
            databaseSchema = _jsonConverter.Deserialize<DatabaseSchema>(jsonSchema);
        }
        catch (Exception ex)
        {
            return Result.Fail(ex.Message);
        }
        this.InjectedSchema = databaseSchema;
        return Result.Ok();
    }
    // ...
}
```

Primjer 3.8: Implementacija metode za prijenos sheme iz JSON datoteke na platno

Stranica za izradu shema preuzima shemu u trenutku svoje inicijalizacije, pozivajući metodu `PopSchema()` u svojoj metodi životnog ciklusa `OnInitialized()`. Implementacija metode `PopSchema()` može se vidjeti u Primjeru 3.9.

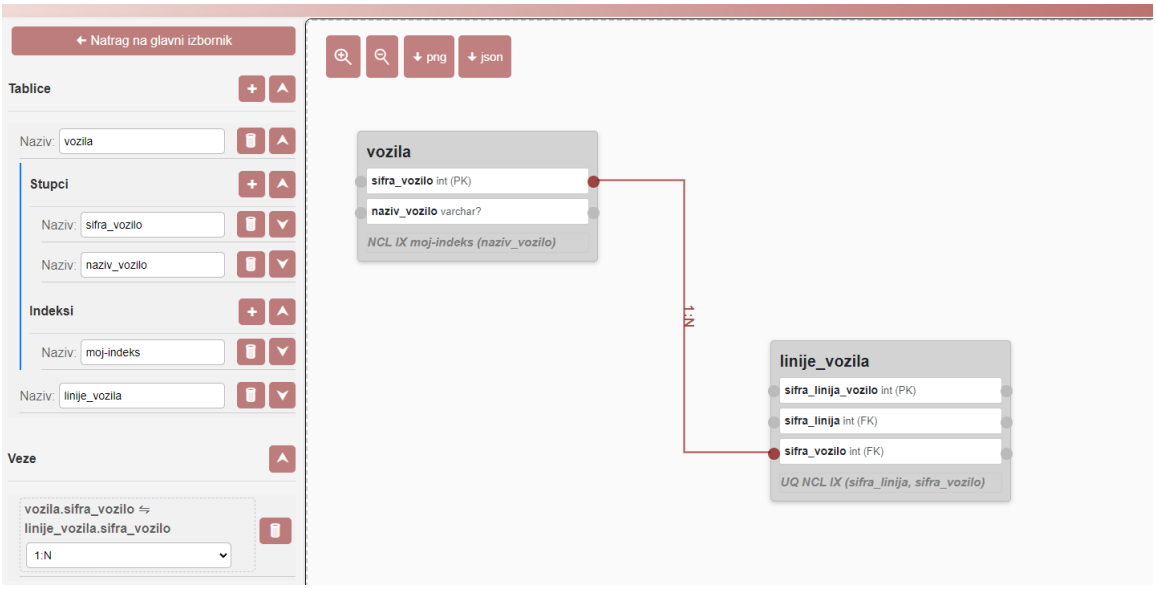
```
public class SchemaInjectionService : ISchemaInjectionService
{
    private DatabaseSchema InjectedSchema { get; set; } = null;
    // ...
    public DatabaseSchema PopSchema()
    {
        var injectedSchema = InjectedSchema;
        InjectedSchema = null;
        return injectedSchema;
    }
}
```

Primjer 3.9: Implementacija metode za preuzimanje uvezene sheme

Ovo je općeniti mehanizam prijenosa bilo koje postojeće sheme na platno. Iz tog razloga, sve podređene komponente glavnog izbornika vrlo su slično implementirane.

3.3 Složena funkcionalnost

Stranica za izradu sheme najkompleksnija je stranica u aplikaciji. Sastoji se od nekoliko slojeva Razor komponenti i koristi većinu servisa implementiranih u aplikaciji. Očekivani izgled može se vidjeti na Slici 3.3. Lijevo na stranici nalazi se proširivi izbornik u kojem korisnik dodaje, briše i ažurira elemente sheme. Desno na stranici je platno na kojem korisnik pomiče tablice i crta veze. Kad bi se korisnik pokušao vratiti nazad na glavni izbornik, iskočio bi mu prozor koji bi ga tražio potvrdu prije negoli omogući povratak.



Slika 3.3: Stranica za izradu sheme

Prikaz i podjela elemenata na stranici

Svi elementi na stranici dio su glavnog objekta stranice koji je tipa `DatabaseSchema` (mapa `SchemaElements`). Konkretno, objekt tipa `DatabaseSchema` sadrži liste objekata tipa `Table`, `Relationship` i slično. Klasa `Table` definira polja za spremanje svojih koordinata na platnu. Ovakva reprezentacija domenskih entiteta omogućuje nam da prikazujemo elemente i iscrtavamo shemu na stranici koristeći standardnu Razor sintaksu.

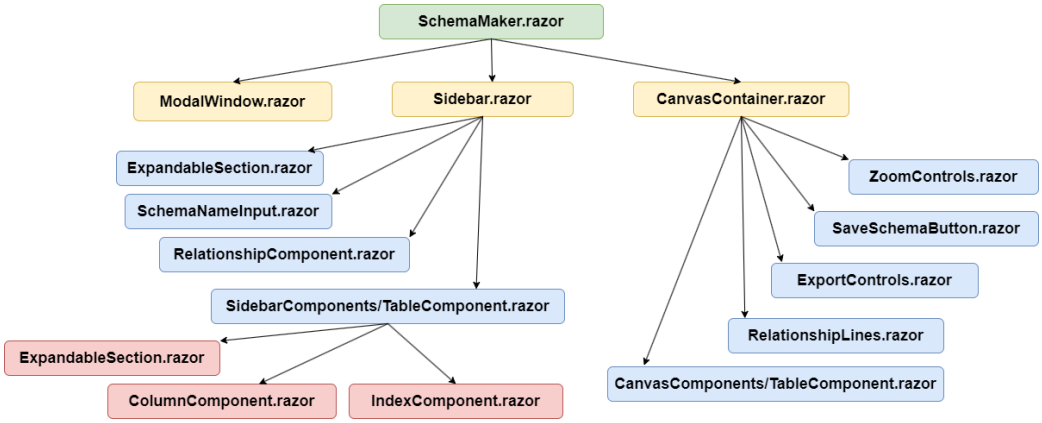
Primjerice, sve tablice na platnu ćemo prikazivati pomoću petlje `@foreach` u kojoj ćemo kreirati element `<div>` s dinamički postavljenim `style` svojstvima pozicije (Primjer 3.10). Pomoću metoda upravljanja stanjem, čim se dogodi promjena na nekom objektu tipa `Table`,

moguće je automatski obnoviti prikaz svih asociranih Razor komponenti. Dakle, za ostvarivanje promjena u stablu prikazu, dovoljno je promijeniti objekt u klasičnom C# kodu. Na ovome se temelje sve funkcionalnosti koje ćemo implementirati. Premještanje tablice na platnu će se svesti na promjenu koordinati u C# objektu tipa `Table`, dodavanje novog stupca će se svesti na dodavanje objekta tipa `Column` u odgovarajuću C# listu itd.

```
@* Komponenta platna *@
@foreach (var table in DatabaseSchema.Tables)
{
    <div class="table" draggable="true"
        @* Postavljanje pozicije tablice na platnu *@
        style="top: @table.TopPx; left: @table.LeftPx;">
        <strong>@table.Name</strong>
        @foreach (var column in table.Columns)
        {
            <div class="column">@column.Name</div>
        }
    </div>
}
```

Primjer 3.10: Pojednostavljeni kod za iscrtavanje tablice na platnu

Stranica ima puno sadržaja, stoga je krucijalno da ju razložimo na smislene komponente. Izradit ćemo hijerarhiju komponenti prateći funkcionalne entitete koje vidimo na stranici. Počinjemo od komponente `SchemaMaker` koja sadrži komponente `Sidebar` (lijevi izbornik), `CanvasContainer` (platno) i `ModalWindow` (iskočni prozor). `Sidebar` sadrži komponente `TableComponent` i `RelationshipComponent`, i tako nastavljamo dok ne dođemo do najosnovnije komponente. Dijagram je prikazan na Slici 3.4.



Slika 3.4: Razor komponente stranice za izradu sheme

Komponenta `Sidebar`

Ključne funkcionalnosti lijevog izbornika su:

- Upravljanje tablicama, što uključuje dodavanje nove tablice, mijenjanje naziva, dodavanje stupaca, dodavanje indeksa i brisanje tablice.
- Upravljanje stupcima, što uključuje ažuriranje svojstava stupaca (naziv, tip podatka, tip ključa, `null` vrijednosti) i brisanje stupaca.
- Upravljanje indeksima, što uključuje ažuriranje svojstava indeksa (naziv, tip indeksa, tip ključa, stupci indeksa) i brisanje indeksa.
- Upravljanje vezama, što uključuje ažuriranje kardinaliteta i brisanje veza.

Prođimo kroz funkcionalnost dodavanja nove tablice.

Primjer 3.11 prikazuje metodu za dodavanje nove tablice u komponenti `SchemaMaker`. Metoda poziva servis `SchemaObjectFactory` zadužen za kreiranje i brisanje elemenata sheme. Na kraju se cijela komponenta u potpunosti ažurira i prikazuje korisniku jer, budući da se kreirala nova tablica, potrebno je ponovno prikazati i lijevi izbornik i platno. Za ažuriranje komponente koristimo metodu za upravljanje stanjem `StateHasChanged`. Uočimo kako je metoda proslijeđena u podređenu komponentu `Sidebar`.

```
@page "/schema-maker"
@inject ISchemaObjectFactory _schemaObjectFactory
<Sidebar @* ... *@
    AddNewTable="AddNewTable">
</Sidebar>
@* ... *@
@code {
    // ...
    private async Task AddNewTable()
    {
        var newTableId = _schemaObjectFactory.CreateNewTable(this.DatabaseSchema);
        await InvokeAsync(StateHasChanged);
    }
}
```

Primjer 3.11: Dodavanje nove tablice u komponenti `SchemaMaker`

U Primjeru 3.12 vidimo kako komponenta `Sidebar` prosljeđuje radnju koju je dobila od komponente `SchemaMaker` podređenoj komponenti `ExpandableSection`.

```
@* Sidebar.razor *@
<ExpandableSection Title="Tablice" OnAddNewItem="AddNewTable">
    @foreach (var table in Tables)
    {
        <TableComponent Table="@table"/>
    }
</ExpandableSection>
@* ... *@
@code {
    // ...
    [Parameter] public List<Table> Tables { get; set; }
    [Parameter] public Func<Task> AddNewTable { get; set; }
}
```

Primjer 3.12: Prosljeđivanje radnje `AddNewTable` iz komponente `Sidebar`

Komponenta `ExpandableSection` sažima i proširuje definirani sadržaj (u ovom slučaju listu tablica) te uz to sadrži gumb za dodavanje novog elementa. Na klik tog gumba, komponenta obavještava roditeljske komponente da se proslijeđena radnja mora izvršiti (Primjer 3.13).

```
@* ExpandableSection.razor *@
<div class="expandable-row">
    <span>@Title</span>
    <button class="btn large-bold-font"
        @onclick="() => OnAddNewItem()"> @* Pozivanje roditeljske radnje. *@
        +
    </button>
@* ... *@
</div>
@* ... *@
@code {
    // ...
    [Parameter] public string Title { get; set; }
    [Parameter] public Func<Task> OnAddNewItem { get; set; }
    // ...
}
```

Primjer 3.13: Pozivanje radnje `AddNewTable` iz komponente `ExpandableSection`

Pogledajmo implementaciju metode `CreateNewTable` u klasi `SchemaObjectFactory`. Metoda konstruira novi objekt tipa `Table` kojem dodjeljuje početno ime, primarni ključ, početnu poziciju na platnu i identifikator pridružen iz polja `_tableId` (Primjer 3.14).

```
public class SchemaObjectFactory : ISchemaObjectFactory
{
    private int _tableId = 1;
    private int _columnId = 1;
    // ...
    public int CreateNewTable(DatabaseSchema databaseSchema)
    {
        var newTable = new Table
        {
            Id = _tableId,
            Name = "tablica_" + _tableId,
            Columns = new[] { CreatePrimaryKeyColumn(_columnId, _tableId) },
            CoordinateX = SchemaMakerConstants.TableStartingCoordinateX,
            CoordinateY = SchemaMakerConstants.TableStartingCoordinateY
        };
        databaseSchema.Tables.Add(newTable);
        // Inkrementiramo ID polja za buduće elemente.
        _tableId++;
        _columnId++;
        return newTable.Id;
    }
}
```

Primjer 3.14: Implementacija metode `CreateNewTable` u klasi `SchemaObjectFactory`

Analogno su implementirane i ostale funkcionalnosti lijevog izbornika. Razor komponenta `SchemaMaker` definira odgovarajuću akciju te se ta akcija prosljeđuje sve do podređene komponente unutar koje se asocirani događaj okida.

Međutim, mogli bismo se upitati zašto se akcije definiraju u komponenti `SchemaMaker` pa prosljeđuju? Odnosno, zašto jednostavno nismo definirali akcije direktno u podređenim komponentama? Razlog tomu je primjenjivanje principa labave sprege (engl. *loose coupling*). Ako akcija podređene komponente utječe na roditeljsku, akcija bi se trebala definirati u roditeljskoj komponenti. U našem slučaju, dodavanje i brisanje nove tablice ili stupca mijenja cijelu stranicu, a ne samo lijevi izbornik, budući da se nova tablica ili stupac moraju pojaviti i na platnu. Ovim pristupom izbjegavamo blisku ovisnost glavne komponente o njezinim podređenim komponentama (engl. *tight coupling*), što dugoročno čini kod lakšim za proširivanje, refaktoriranje i generalno održavanje.

Komponenta `CanvasContainer`

Ključne funkcionalnosti platna su:

- Crtanje i prikaz veza pomoću točaka spajanja
- Premještanje (engl. *drag and drop*) tablica po platnu
- Izvoz sheme u PNG ili JSON formatu

Svaka tablica na platnu, za svaki stupac, ima po jednu točku spajanja sa svake strane (Slika 3.5).

Slika 3.5: Točke spajanja na tablici unutar platna

Točka spajanja je element koji se prikazuje povrh elementa tablice unutar Razor komponente `CanvasContainer/TableComponent`. Svaka točka spajanja jedinstveno je određena tablicom, stupcem i stranom na kojoj se nalazi. Klasa `ConnectionPoint` predstavlja točku spajanja u kodu. Klasa sadrži identifikacijska svojstva točke te koordinate za prikaz na platnu. Sve točke spajanja na shemi spremljene su u listu unutar objekta `DatabaseSchema`.

Klikom na točku spajanja započinje proces kreiranja veze. Ponovnim klikom na neku drugu točku spajanja proces završava. Proces se odvija kroz tri komponente. Komponenta `TableComponent`, u kojoj se okida događaj klika na točku, namješta parametre kreiranja veze uz pomoć klase `RelationshipCreationHelper` koja sprema podatke o tome koja je točka kliknuta i je li kliknuta prvi ili drugi put, odnosno, je li kreiranje veze počelo ili završilo. Primjer 3.15 prikazuje opisanu radnju.

```
@* CanvasContainer/TableComponent.razor *@
@foreach (var connectionPoint in TableConnectionPoints))
{
    <div class="connection-point"
        @onclick="() => ConnectionPointClick(connectionPoint.Id)">
    </div>
}
@* ...  *@
@code {
    [Parameter]
    public RelationshipCreationHelper RelationshipHelper { get; set; }
    [Parameter]
    public EventCallback OnConnectionPointClicked { get; set; }
    // ...
    private async Task ConnectionPointClick(string connectionPointId)
    {
        if (this.RelationshipHelper.CreationMode == CreationMode.None)
        {
            this.RelationshipHelper.CreationMode = CreationMode.StartCreation;
            this.RelationshipHelper.StartingPointId = connectionPointId;
        }
        else if (this.RelationshipHelper.CreationMode ==
            CreationMode.StartCreation)
        {
            this.RelationshipHelper.CreationMode = CreationMode.CloseCreation;
            this.RelationshipHelper.EndingPointId = connectionPointId;
        }
        // Obavještavamo roditelja da obavi proslijeđenu akciju.
        await this.OnConnectionPointClicked.InvokeAsync();
        // ...
    }
}
```

Primjer 3.15: Radnja na događaj klika točke spajanja u `TableComponent`

U Primjeru 3.15 također vidimo kako komponenta `TableComponent` obavještava roditeljsku komponentu `CanvasContainer` da izvrši proslijeđenu radnju za događaj klika na točku spajanja. Proslijeđena radnja prikazana je u Primjeru 3.16.

```
@* CanvasContainer.razor *@
@foreach (var table in DatabaseSchema.Tables)
{
    <TableComponent @* ...  *@
        RelationshipHelper="RelationshipHelper"
        OnConnectionPointClicked="DrawRelationship" />
}
@* ...  *@
@code {
    [Parameter]
    public RelationshipCreationHelper RelationshipHelper { get; set; }
    [Parameter] public EventCallback OnRelationshipDrawing { get; set; }
    // ...
    private async Task DrawRelationship()
    {
        // Pozivanje akcije iz SchemaMaker komponente za kreiranje veze.
        await this.OnRelationshipDrawing.InvokeAsync();
        // Nakon što se veza kreirala, bojamo odgovarajuće točke i pozicioniramo
            vezu na platnu.
        if (this.RelationshipHelper.CreationMode == CreationMode.StartCreation)
        {
            _styleService.SetConnectionPointsColor(/* ... */);
        }
        else if (this.RelationshipHelper.CreationMode ==
            CreationMode.CloseCreation)
        {
            _positionService.UpdateRelationshipPositions(/* ... */);
            _styleService.SetConnectionPointsColor(/* ... */);
        }
    }
}
```

Primjer 3.16: Radnja za crtanje veze u komponenti `CanvasContainer`

Radnja `DrawRelationship` na samom početku obavještava komponentu `SchemaMaker` da izvrši svoju proslijeđenu radnju koja kreira vezu (Primjer 3.17).

```
@* SchemaMaker.razor *@
@inject ISchemaObjectFactory _schemaObjectFactory
@* ... *@
<CanvasContainer @* ... *@
    RelationshipHelper="RelationshipHelper"
    OnRelationshipDrawing="AddNewRelationship">
</CanvasContainer>
@* ... *@
@code {
    private RelationshipCreationHelper RelationshipHelper = new();
    // ...
    private void AddNewRelationship()
    {
        if (this.RelationshipHelper.CreationMode == CreationMode.StartCreation)
        {
            var startId = this.RelationshipHelper.StartingPointId;
            var relationship = _schemaObjectFactory.CreateNewRelationship(startId);
            this.RelationshipHelper.CurrentRelationship = relationship;
            return;
        }
        if (this.RelationshipHelper.CreationMode == CreationMode.CloseCreation)
        {
            _schemaObjectFactory.CloseNewRelationship(this.DatabaseSchema,
                this.RelationshipHelper.CurrentRelationship,
                this.RelationshipHelper.EndingPointId);
        }
    }
}
```

Primjer 3.17: Radnja za kreiranje nove veze u komponenti `SchemaMaker`

Na kraju cijelog procesa stanje komponente `SchemaMaker` se mijenja, stoga se izrađuje novi prikaz stranice. Primijetimo da su u ovom procesu sve radnje striktno vezane za komponentu u kojoj se nalaze, to jest, nijedna komponenta ne upravlja prikazom ili podacima izvan nje same. `TableComponent` ne ažurira boje točaka jer je to van njezina opsega; točke spajanja i pozicije veza su u opsegu platna. Isto tako, kreiranje nove veze treba ažurirati cijelu stranicu, stoga je van opsega platna te se mora obaviti na razini stranice.

Metode za kreiranje nove veze u klasi `SchemaObjectFactory` i dalje donekle prate klasičan proces dodavanja novog elementa u shemu, koji smo vidjeli na primjeru dodavanja tablice, ali je podijeljen u dva dijela. `Relationship` objekt kreira se u `CreateNewRelationship` metodi, ali dobiva ID i dodaje se u shemu tek u metodi `CloseNewRelationship`.

Premještanje tablice po platnu funkcionira slično crtanju veze, u smislu da više komponenti sudjeluje u procesu. Ipak, proces je jednostavniji jer se ne odvija na razini stranice - sam premještaj tablice na platnu ne mijenja sadržaj lijevog izbornika. Za implementaciju funkcionalnosti koristi se standardni HTML Drag And Drop API[3], u kojem se događaj `dragstart` okida u komponenti `TableComponent`, dok se `drop` okida u komponenti `CanvasContainer`. `TableComponent` sprema svoje stare koordinate, a `CanvasContainer` računa i ažurira nove koordinate tablice te se ponovno prikazuje.

Funkcionalnosti izvoza sheme jedine su funkcionalnosti za koje moramo eksplicitno koristiti JavaScript. Blazor nema ugrađeni mehanizam za izvoz datoteka iz preglednika, stoga moramo koristiti JS Interop kako bismo ostvarili izvoz pomoću JavaScripta. Izvoz sheme inicira se klikom odgovarajućeg gumba na platnu. Servis koji se poziva na klik gumba naziva se `ExportService`. U Primjeru 3.18 prikazana je metoda u klasi `ExportService` koja izvozi shemu u JSON datoteci. Metoda asinkrono poziva JavaScript funkciju `saveAsFile` pomoću JS Interopa. Funkcija `saveAsFile` nalazi se u mapi `wwwroot/script` i koristi Blob[2] sučelje preglednika za kreiranje i izvoz datoteke.

```
using Microsoft.JSInterop;
public class ExportService : IExportService
{
    // ...
    public async Task ExportSchemaAsJson(DatabaseSchema databaseSchema)
    {
        // Serijaliziramo shemu.
        var databaseSchemaInJsonFormat = _jsonConverter.Serialize(databaseSchema);
        // Asinkrono pozivamo JavaScript funkciju koja će spremiti i izvesti JSON
            format sheme.
        await _jsRuntime.InvokeVoidAsync("saveAsFile",
            "schema.json",
            databaseSchemaInJsonFormat);
    }
}
```

Primjer 3.18: Pozivanje JavaScript funkcije iz C# metode `ExportSchemaAsJson`

Iako nam je cilj koristiti C# za razvoj ove aplikacije, svejedno je potrebno upoznati mehanizme komunikacije s JavaScript Engineom u ASP.NET-u. Blazor nam omogućuje da implementiramo kompleksan klijentski kod u C#-u. Razvoj funkcionalnosti poput izrade dijagrama ili premještanja elemenata po platnu u bilo kojem drugom ASP.NET okviru ne bi bio moguć bez JavaScripta. Ipak, uvijek ćemo nailaziti na specifična ograničenja iz već spomenutog razloga - WebAssembly ne mijenja JavaScript, već ga nadopunjuje.

3.4 API integracija

Želimo omogućiti korisnicima registraciju u sustav i spremanje podataka, no budući da koristimo WebAssembly model Blazora, ovakve funkcionalnosti ne možemo implementirati direktno u aplikaciji jer se njezin kod nalazi i izvršava u pregledniku. Potrebna nam je poslužiteljska aplikacija kojoj možemo pristupiti kroz programsko sučelje (API).

Poslužiteljska aplikacija treba nuditi sljedeće (ugrubo opisane) API funkcionalnosti:

- `Authentication/Register`
  Implementira POST metodu koja provjerava jesu li dobiveni podaci za registraciju uspješno uneseni, i ako jesu, sprema novog korisnika. Dobiveni podaci uključuju korisničko ime, izvornu (nekriptiranu) lozinku i potvrdu lozinke.

- `Authentication/Login`
  Implementira POST metodu koja provjerava postoje li uneseni korisnički podaci i jesu li točni. Ako jesu, metoda vraća pristupni token. Uneseni korisnički podaci uključuju korisničko ime i izvornu lozinku.

- `DatabaseSchemas/SaveOrUpdateSchema`
  Implementira POST metodu koja sprema novu shemu ili ažurira postojeću. Metoda prima objekt s identifikatorom, imenom i JSON reprezentacijom sheme (serijalizirani `DatabaseSchema` objekt).

- `DatabaseSchemas/GetSchemasForUser`
  Implementira GET metodu koja dohvaća osnovne informacije o svim shemama prijavljenog korisnika. Metoda vraća listu objekata s identifikatorom, imenom i zadnjim datumom ažuriranja sheme.

- `DatabaseSchemas/GetSchema`
  Implementira GET metodu koja dohvaća neku shemu prijavljenog korisnika. Prima identifikator sheme. Vraća objekt s identifikatorom, imenom i JSON reprezentacijom sheme.

- `DatabaseSchemas/DeleteSchema`
  Implementira DELETE metodu koja briše neku shemu prijavljenog korisnika. Metoda prima identifikator sheme za brisanje.

Mi se nećemo baviti razvojem programskog sučelja na poslužiteljskoj strani, već će nam fokus biti integracija s ovakvim sučeljem. Aplikacija može biti napisana u bilo kojem jeziku i okviru, sve dok nudi gore opisano sučelje. Primjer aplikacije koja implementira opisani API može se naći u repozitoriju: https://github.com/josipasimic/SchemaPalWebApi.

Kako bismo koristili API neke aplikacije u Blazoru, prvo trebamo registrirati `HttpClient` objekt s baznom adresom te aplikacije. `HttpClient` je klasa u ASP.NET-u koja olakšava HTTP komunikaciju s drugim aplikacijama. Konfiguracija novog HTTP klijenta je prikazana u Primjeru 3.19.

```
builder.Services.AddHttpClient(name: "SchemaPalApi",
    configureClient: client =>
    {
        client.BaseAddress = new Uri("https://my-api-host/api/"); // Demo adresa
    });
```

Primjer 3.19: Registriranje `HttpClient` objekta u datoteci `Program.cs`

Sada možemo napraviti servis koji će služiti isključivo za pozivanje API metoda. Dodajemo klasu `SchemaPalApiService` s odgovarajućim sučeljem `ISchemaPalApiService` (Primjer 3.20). Primijetimo kako svaka metoda sučelja odgovara jednoj API metodi.

```
public interface ISchemaPalApiService
{
    Task<Result> RegisterUser(UserRegistration userRegistration);
    Task<Result> LoginUser(UserLogin userLogin);
    Task<Result<Guid>> SaveDatabaseSchema(ExtendedSchemaRecord schemaRecord);
    Task<Result<List<ShortSchemaRecord>>> GetDatabaseSchemasForLoggedInUser();
    Task<Result<ExtendedSchemaRecord>> GetDatabaseSchema(Guid id);
    Task<Result> DeleteDatabaseSchema(Guid id);
}
```

Primjer 3.20: Sučelje `ISchemaPalApiService`

U konstruktoru klase `SchemaPalApiService` kreiramo objekt tipa `HttpClient` s odgovarajućim imenom (Primjer 3.21).

```
public class SchemaPalApiService : ISchemaPalApiService
{
    private readonly HttpClient _httpClient;
    public SchemaPalApiService(IHttpClientFactory httpClientFactory)
    {
        _httpClient = httpClientFactory.CreateClient("SchemaPalApi");
    }
}
```

Primjer 3.21: Kreiranje HTTP klijenta u klasi `SchemaPalApiService`

Budući da će se u metodama odvijati mrežna komunikacija, implementirat ćemo ih kao asinkrone. Primjer 3.22 prikazuje implementaciju metode `LoginUser`.

```
public class SchemaPalApiService : ISchemaPalApiService
{
    private string _accessToken = string.Empty;
    public async Task<Result> LoginUser(UserLogin userLogin)
    {
        var response = await _httpClient.PostAsJsonAsync("Authentication/login",
            userLogin);
        if (!response.IsSuccessStatusCode)
        {
            return Result.Fail("Prijava nije uspjela!");
        }
        var accessToken = await response.Content.ReadFromJsonAsync<AccessToken>();
        _accessToken = accessToken.Token; // Spremamo token za buduće zahtjeve.
        return Result.Ok();
    }
}
```

Primjer 3.22: Implementacija metode `LoginUser` u klasi `SchemaPalApiService`

`_httpClient` kreira zahtjev koristeći registriranu baznu adresu i proslijeđeni nastavak (ime kontrolera i željena API metoda). HTTP metoda specificirana je u imenu pozvane metode nad `_httpClient` objektom; u primjeru je to POST. U zahtjevu šaljemo instancu klase `UserLogin` (mapa `DataTransferObjects`) koja sprema korisničko ime i izvornu lozinku. Odgovor se materijalizira u objekt tipa `HttpResponseMessage` koji sadrži informacije o uspjehu tražene operacije i eventualni povratni sadržaj. Povratni sadržaj može se deserijalizirati u neki objekt pomoću metode `ReadFromJsonAsync` iz primjera. Klasa `AccessToken` sadrži jedinstveni string koji služi za autorizaciju korisnika u budućim zahtjevima. U svakom idućem zahtjevu ćemo taj string morati dodati u zaglavlje, kao što je prikazano u Primjeru 3.23.

```
_httpClient.DefaultRequestHeaders.Authorization = new
    AuthenticationHeaderValue("Bearer", _accessToken);
```

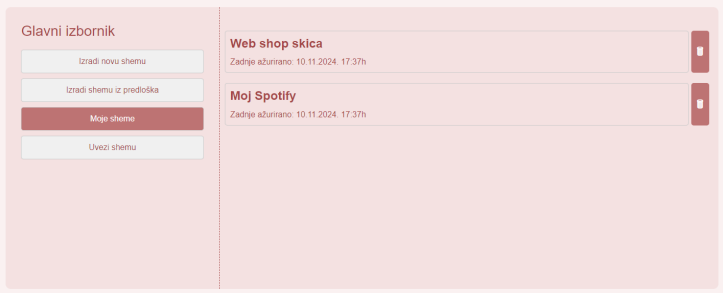
Primjer 3.23: Dodavanje pristupnog tokena u zaglavlje budućih zahtjeva

Sad kad smo integrirani na API, u Razor komponenti `Login` možemo na klik gumba *Prijavi se* pozvati metodu `LoginUser`, obraditi njezin odgovor te, po uspjehu, započeti korisničku sesiju. Upravljanje korisničkom sesijom implementirano je u servisu `UserSessionService`.

Na vrlo sličan način se implementiraju i preostale metode. Primjerice, ako želimo prikazati sve korisnikove sheme u glavnom izborniku, dodat ćemo mu novu podređenu komponentu `MySchemas`. Tu komponentu ćemo sakriti u načinu korištenja kao gost. Zatim ćemo implementirati API metodu za dohvat osnovnih informacija o svim shemama korisnika, `GetDatabaseSchemasForLoggedInUser`. Pratit ćemo format metode `LoginUser`, uz prethodno dodavanje pristupnog tokena (Primjer 3.23). U trenutku inicijalizacije komponente `MySchemas`, pozvat ćemo implementiranu metodu i prikazati dobiveni rezultat korisniku (Primjer 3.24 i Slika 3.6).

```
@foreach (var schema in this.UserSchemas)
{
    <div class="schema-name">
        <h4><strong>@schema.Name</strong></h4>
        <p>Zadnje azurirano: @schema.LastSaved</p>
    </div>
}
@code {
    private List<ShortSchemaRecord> UserSchemas = new();
    protected override async Task OnInitializedAsync()
    {
        var res = await _schemaPalApiService.GetDatabaseSchemasForLoggedInUser();
        if (res.IsSuccess)
        {
            UserSchemas = res.Value ?? new();
        } // Obrađujemo dalje neuspjeli rezultat...
    }
}
```

Primjer 3.24: Dohvat korisnikovih shema u komponenti `MySchemas`



Slika 3.6: Opcija *Moje sheme* u glavnom izborniku

Poglavlje 4

Testiranje aplikacije u Blazor WebAssemblyju

4.1 Osnove testiranja

Načini testiranja

Softversko testiranje je proces provjere rada softvera čija je svrha osigurati ispunjenost korisničkih zahtjeva te kvalitetu, pouzdanost i sigurnost uporabe. Razlikujemo dva načina izvedbe testiranja: ručno testiranje i automatizacija.

Ručno testiranje zahtijeva dediciranu osobu, odnosno testera, čiji zadatak je ispitivati softverski sustav iz perspektive korisnika. Tester ručno provjerava funkcionalnosti sustava kako bi utvrdio ispravnost rada. Ručni testovi su sporiji, ali su ključni za poboljšanje krajnjeg korisničkog iskustva (engl. *user experience*).

S druge strane, automatizacija uključuje korištenje softverskih alata i skripti čijim se izvršavanjem testira izvorni kod. Automatizirani testovi ne mogu pokriti jako specifične slučajeve, ali su brzi i ponovno upotrebljivi. Zato se automatizirani testovi najčešće koriste za ispitivanje najosnovnijih funkcija sustava. Za optimalno očuvanje kvalitete sustava, potrebno je kombinirati oba načina testiranja.

Iako se testiranjem softvera većinski bave QA (Quality Assurance) inženjeri, određeni dio automatiziranih testova pišu programeri, budući da je testiranje osnova sustava direktno vezano uz proces razvoja. Primjerice, u tehnici razvoja temeljenog na testiranju (engl. *test driven development*), programeri pišu testove i prije nego što dovrše pisanje koda kako bi se osigurala ispravnost najosnovnijih funkcija već u fazi razvoja. Iz tog razloga, većina

razvojnih okvira nudi jako dobru podršku za pisanje testova.

Razvojni okviri u ASP.NET-u nude razne alate i pakete za pisanje automatiziranih testova. Ipak, zbog stroge razdvojenosti koda na poslužiteljski i klijentski, klasična aplikacija u ASP.NET-u zahtijeva dodatno korištenje vanjskih alata kako bi se cijeli izvorni kod pokrio testovima. Iz tog razloga, klijentski kod često ostane zanemaren u procesu automatiziranog testiranja aplikacije. Programeri u ASP.NET-u daju fokus poslužiteljskom kodu koji se može testirati u C#-u, dok se testiranje klijentske strane obavlja većinom ili isključivo ručno.

Ovo više nije slučaj za razvoj aplikacija u Blazoru WebAssemblyju, u kojem je izvorni kod zapravo potpuno klijentski. Blazor, a posebno Blazor WebAssembly, nam nudi mogućnost testiranja klijentskog koda pomoću standarnih alata ASP.NET-a.

Razine testiranja

Ovisno o opsegu funkcionalnosti koje testiraju, softverski testovi podijeljeni su na sljedeće razine:

- Jedinični testovi (engl. *unit tests*) se provode na najmanjim dijelovima aplikacije, obično na funkcijama, metodama ili klasama. Cilj je osigurati da svaki pojedinačni dio koda radi ispravno i prema očekivanjima.

- Integracijski testovi (engl. *integration tests*) provjeravaju kako različiti dijelovi aplikacije funkcioniraju zajedno. Nakon što se pojedinačne komponente istestiraju, integracijsko testiranje fokusira se na interakciju među njima.

- Sistemski testovi (engl. *system tests*) provode se na sustavu kao cjelini. Ova razina testiranja uključuje testiranje kompletne aplikacije s fokusom na funkcionalnost, performanse i kompatibilnost prema definiranim specifikacijama.

- Testovi s kraja na kraj (engl. *end-to-end tests*) provjeravaju sve korake kroz koje bi korisnik mogao proći u relevantnim scenarijima korištenja aplikacije.

Sistemski testovi i testovi s kraja na kraj obično su u domeni QA inženjera. Ne ispituju kod na nižoj razini, već scenarije i funkcionalne cjeline. Većinski se izvode ručno, no mogu se i automatizirati uz pomoć raznih alata koji su najčešće van razvojnog okvira same aplikacije. Općenito se samo najosnovniji i najčešći scenariji automatiziraju, dok se ostali scenariji testiraju ručno, ponekad i uz suradnju s krajnjim korisnicima.

Jedinični i integracijski testovi obično su u domeni rada programera jer testiraju kod na nižoj razini. Samim time ih je teško testirati ručno, a relativno lako automatizirati. Gotovo uvijek se pišu u istom jeziku kao i izvorni kod, radi čega su i alati za njihovu implementaciju najčešće dobro podržani unutar razvojnog okvira aplikacije.

Mi ćemo se fokusirati na testove nižih razina u procesu testiranja naše aplikacije. Proučit ćemo standardne alate koje ASP.NET nudi za testiranje C# koda te dodatne alate koji su podržani u Blazoru specifično za testiranje Razor komponenti. Vidjet ćemo kako nam principi i obrasci programiranja koje smo primijenili tijekom razvoja omogućuju pisanje nezavisnih testova te pojednostavljuju cjelokupni proces.

4.2 Alati za testiranje u ASP.NET-u

Okvir za testiranje xUnit

xUnit je okvir za testiranje u ekosustavu .NET koji pojednostavljuje pisanje i automatizaciju testova u aplikacijama napisanima u C#-u. U xUnitu razlikujemo pojedinačne testove i parametrizirane testove. Pojedinačni testovi su testovi koji ne primaju ulazne parametre, odnosno, izvršavaju se samo za jedan testni slučaj. Ovi testovi označavaju se atributom `[Fact]`. Neka nam je zadana klasa `Calculator` iz Primjera 4.1.

```
public class Calculator // Klasa koju testiramo
{
    public int Add(int a, int b) => a + b;
}
```

Primjer 4.1: Klasa `Calculator`

Pojedinačni test za metodu `Add` u klasi `Calculator` prikazan je u Primjeru 4.2.

```
using Xunit;
public class CalculatorTests // Testna klasa
{
    [Fact] // Pojedinačni test za metodu Add
    public void GivenCalculator_WhenAddingPositiveNumbers_ThenReturnExpectedSum()
    {
        int result = new Calculator().Add(2, 3);
        Assert.Equal(5, result);
    }
}
```

Primjer 4.2: Pojedinačni test za metodu `Add`

S druge strane, parametrizirani testovi mogu se izvršavati za više slučajeva koji su određeni proslijeđenim parametrima. Parametrizirane testove označavamo atributom `[Theory]`. Primjer parametriziranog testa za metodu `Add` iz Primjera 4.1. prikazan je u Primjeru 4.3. Test se dodaje u istu testnu klasu prikazanu u Primjeru 4.2.

```
[Theory] // Parametrizirani test za metodu Add
[InlineData(2, 3, 5)]
[InlineData(0, 0, 0)]
public void GivenAddends_WhenPerformingAddition_ThenReturnExpectedSum(
    int a, int b, int expected)
{
    int result = _calculator.Add(a, b);
    Assert.Equal(expected, result);
}
```

Primjer 4.3: Parametrizirani test za metodu `Add`

Osim vrijednosnih tipova koji se predaju kroz atribut `[InlineData]`, testnoj metodi mogu se predati i objekti kroz atribut `[MemberData]` pomoću klase `TheoryData`. Ova klasa omotava parametre testne metode u novi objekt pogodan za korištenje unutar atributa `[MemberData]`. Testni parametri mogu biti i vrijednosni i referentni te ih može biti nekoliko.

Implementirajmo klasu `Calculator` na način da, umjesto dva broja tipa `int`, metoda `Add` prima jedan objekt klase `Addends` (Primjer 4.4).

```
public class Addends // Pomoćna klasa za prijenos parametara
{
    public int First { get; set; }
    public int Second { get; set; }
}
public class Calculator // Klasa koju testiramo
{
    public int Add(Addends addends) => addends.First + addends.Second;
}
```

Primjer 4.4: Alternativna implementacija klase `Calculator`

Tada bismo novu verziju metode `Add` mogli istestirati na način prikazan u Primjeru 4.5. U primjeru smo omotali objekt tipa `Addends` i broj tipa `int`, za dva različita testna slučaja, u objekt tipa `TheoryData<Addends, int>` te smo na taj način omogućili atributu `[MemberData]` prosljeđivanje oba dva slučaja u testnu metodu.

```
[Theory]
[MemberData(nameof(TestCases))]
public void GivenAddends_WhenPerformingAddition_ThenReturnExpectedSum(
    Addends addends, int expectedSum)
{
    int result = _calculator.Add(addends.First, addends.Second);
    Assert.Equal(expectedSum, result);
}
public static TheoryData<Addends, int> TestCases()
{
    return new TheoryData<Addends, int>
    {
        { new Addends { First = 2, Second = 3 }, 5 },
        { new Addends { First = 0, Second = 0 }, 0 }
    };
}
```

Primjer 4.5: Parametrizirani test s objektima za metodu `Add`

Biblioteka Moq

Biblioteka Moq je alat u .NET-u koji se koristi za oponašanje objekata (engl. *object mocking*) u testovima. Biblioteka omogućuje programerima da simuliraju alternativno ponašanje zavisnosti klase koja se testira kako bi mogli testirati klasu neovisno i izolirano od umetnutih servisa. Drugim riječima, programerima je omogućeno izbjeći testiranje vanjskih metoda koje se pozivaju ili koriste u metodi koja se testira.

Definirajmo jednostavno sučelje `IUserService` u Primjeru 4.6.

```
public interface IUserService
{
    User GetUser(int id);
}
```

Primjer 4.6: Sučelje `IUserService`

Nećemo pokazivati neku konkretnu implementaciju sučelja, ali ćemo pretpostavljati da implementacija postoji i da je registrirana kao zavisnost u datoteci za pokretanje aplikacije. Promotrimo klasu `UserController` iz Primjera 4.7. Klasa koristi servis `IUserService`.

```
public class UserController
{
    private readonly IUserService _userService;
    public UserController(IUserService userService)
    {
        _userService = userService;
    }
    public string GetUserName(int id)
    {
        User user = _userService.GetUser(id);
        return user?.Name ?? "User not found";
    }
}
```

Primjer 4.7: Klasa `UserController` sa zavisnošću `IUserService`

Želimo jedinično istestirati metodu `GetUserName`. Osnovna zadaća metode je izvući vrijednost svojstva `Name` iz varijable `user`. Ne želimo uz to testirati proces dohvaćanja objekta koji je spremljen u varijabli jer je taj proces van osnovnog opsega metode. Pomoću biblioteke Moq možemo oponašati servis `IUserService`, tako da namjestimo proizvoljnu implementaciju metode `GetUser`. Primjer 4.8 prikazuje korištenje biblioteke Moq tijekom testiranja metode `GetUserName`.

```
using Moq;
using Xunit;
public class UserControllerTests
{
    [Fact]
    public void GivenUser_WhenGettingName_ThenReturnExpectedValue()
    {
        // Oponašanje servisa IUserService.
        // Kad metoda GetUserName primi ID jednak 1, vraća korisnika John Doe.
        var mockUserService = new Mock<IUserService>();
        mockUserService.Setup(us => us.GetUser(1))
            .Returns(new User { Name = "John Doe" });
        // Prosljeđujemo instancu oponašanog servisa u klasu koju testiramo.
        var controller = new UserController(mockUserService.Object);
        var result = controller.GetUserName(1);
        Assert.Equal("John Doe", result);
    }
}
```

Primjer 4.8: Oponašanje servisa `IUserService`

Bitno je napomenuti da se u Moqu mogu oponašati samo sučelja i apstraktne klase. Konkretne klase spadaju u niže dijelove softvera, odnosno jednom kad ih definiramo, ne možemo ih mijenjati bez direktne modifikacije njihovog koda. Sučelja su apstrakcije čije se implementacije lako mogu zamjenjivati unutar koda.

Biblioteka bUnit

bUnit je biblioteka za testiranje u ASP.NET-u koja olakšava izradu testova za Razor komponente u Blazoru. Biblioteka omogućuje testiranje interakcije, ponašanja i izlaznih rezultata Razor komponenti bez potrebe za pokretanjem cijele aplikacije u pregledniku. Bibilioteka se integrira s većinom standardnih okvira za testiranje u .NET-u, poput xUnita, MSTesta i drugih. Mi ćemo koristiti ovu biblioteku unutar okvira xUnit.

Neka nam je zadana Razor komponenta `HelloWorld` kao u Primjeru 4.9.

```
<h1>Hello, world!</h1>
```

Primjer 4.9: Komponenta `HelloWorld`

Tada bismo mogli napisati test koji provjerava je li sadržaj HTML-a unutar komponente očekivan. Takav test možemo vidjeti u Primjeru 4.10.

```
@using Bunit
@using Xunit
@inherits TestContext
@code {
    [Fact]
    public void GivenHelloWorld_WhenRendering_ThenMarkupIsExpected()
    {
        var component = Render(@<HelloWorld/>);
        component.MarkupMatches(@<h1>Hello world from Blazor</h1>);
    }
}
```

Primjer 4.10: Testiranje HTML sadržaja komponente `HelloWorld`

Svaka testna komponenta nasljeđuje ili koristi klasu `TestContext` iz biblioteke bUnit. `TestContext` stvara okruženje za testiranje Blazor komponenti. Klasa omogućuje simulaciju rada Blazor aplikacije i pruža funkcionalnosti za kreiranje i upravljanje komponentama unutar izoliranog testnog okruženja. U Primjeru 4.10 pozivamo metodu `Render` iz klase `TestContext` koja simulira prikaz željene komponente. Metoda vraća objekt tipa `IRenderedFragment`, sučelje koje predstavlja izrađeni prikaz komponente.

U bUnitu možemo testirati korisničku interakciju. Pretpostavimo da imamo komponentu `Counter`, prikazanu u Primjeru 1.1. Komponenta sadrži gumb čiji klik inkrementira brojač koji se ispisuje korisniku. Primjer 4.11 prikazuje testiranje korisničke interakcije s komponentom `Counter`. Test simulira klik gumba te provjerava je li se vrijednost brojača točno ažurirala.

```
@using Bunit
@using Xunit
@inherits TestContext
@code {
    [Fact]
    public void GivenHelloWorld_WhenRendering_ThenMarkupIsExpected()
    {
        var component = RenderComponent(@<Counter/>);
        // Provjera prije klika.
        component.Markup.Should().Contain("Current count: 0");
        component.Find("button").Click();
        component.Markup.Should().Contain("Current count: 1");
    }
}
```

Primjer 4.11: Testiranje funkcije gumba u komponenti `Counter`

U klasi `TestContext` možemo registrirati zavisnosti, što nam omogućava oponašanje umetnutih servisa komponenti. Recimo da imamo komponentu `WeatherComponent` koja koristi servis `IWeatherService`. U bUnitu možemo oponašati servis `IWeatherService` tako da ga registriramo u klasi `TestContext`. Primjer 4.12 pokazuje proces.

```
@using Moq
@code {
    [Fact]
    public void WeatherComponentDisplaysForecast()
    {
        using var testContext = new TestContext();
        var mockWeatherService = new Mock<IWeatherService>();
        mockWeatherService.Setup(s => s.GetForecastAsync()).ReturnsAsync("Sunny");
        // Registriramo mock servis u bUnit kontekstu
        testContext.Services.AddSingleton(mockWeatherService.Object);
        var component = testContext.RenderComponent<WeatherComponent>();
        Assert.Contains("Weather: Sunny", component.Markup); // Provjera
    }
}
```

Primjer 4.12: Oponašanje i registriranje servisa u bUnitu

4.3 Izrada testova

Aplikacija koju smo razvili u Poglavlju 3 potpuno je klijentska uz korištenje jedne API integracije. U klasičnoj web-aplikaciji sustavne komponente uključuju izvorni kod, bazu podataka, datotečne sustave i slično. Klijentske aplikacije su u tom smislu ograničene, budući da se izvršavaju isključivo u preglednicima. Naša aplikacija naizgled ima samo jednu sustavnu komponentu, a to je izvorni kod. Samim time, većina testova će biti jedinične razine. Ipak, imamo integraciju na API s kojom ćemo moći demonstrirati osnovnu ideju integracijskih testova.

Zahvaljujući principima po kojima smo razvijali aplikaciju, jedinični testovi za sve klase pratit će isti format i pravila implementacije. Vrste klasa koje testiramo su registrirani servisi sa sučeljima i Razor komponente. Svi drugi dijelovi aplikacije ne obavljaju nikakve konkretne funkcije, već služe kao prijenosnici podataka, stoga ih nema potrebe testirati.

Ako proučimo osnovnu strukturu testova koje smo dosad pokazali u primjerima, vidjet ćemo da se sastoji od tri glavna dijela: namještanje potrebnih podataka i elemenata, izvršavanje ključne akcije te nametanje očekivanog rezultata. Ovaj obrazac naziva se AAA (Arrange-Act-Assert) obrazac za pisanje testova. Obrazac ćemo primjenjivati u izradi testova te ćemo sukladno tome testovima davati imena u formatu `Given_When_Then`.

Za izradu jediničnih testova napravit ćemo novi projekt u rješenju aplikacije te ga nazvati `SchemaPal.UnitTests`. U projekt ćemo dodavati testove prateći organizaciju koda kakvu smo definirali u glavnom projektu. Primjerice, ako testiramo klasu `SchemaObjectFactory` koja se nalazi u mapi `SchemaPal/Services/SchemaMakerServices`, tada će se pripadna testna klasa nalaziti u `SchemaPal.UnitTests/Services/SchemaMakerServices`. Isto vrijedi za testove Razor komponenti.

Jedinično testiranje servisa

SOLID principi i umetanje zavisnosti korisni su iz više razloga, ali jedan od istaknutijih je olakšano testiranje. Funkcionalnosti aplikacije smo podijelili na klase s jedinstvenim odgovornostima. Klase smo apstrahirali sučeljima. Ta sučelja smo umetnuli odgovarajućim klasama kao zavisnosti iz IoC kontejnera. Ako metoda ima kompleksniju zadaću, zbog principa jedinstvene odgovornosti, vjerojatno koristi vanjski servis za određene operacije. Servis je sučelje koje možemo oponašati. Dakle, testiranje opsežnijih zadataka možemo odvojiti na dijelove različitih odgovornosti te ih izolirati. Na ovaj način smo omogućili pisanje jednostavnih, brzih i neovisnih jediničnih testova za svaki servis u aplikaciji.

Demonstracije radi, testirat ćemo osnovne metode u klasi `SchemaObjectFactory`. Testove pišemo u okviru xUnit. Testne metode dodajemo u klasu `SchemaObjectFactoryTests` koja se nalazi u pripadnoj datoteci tipa `.cs`.

- Metoda `CreateNewTable` dodaje novu tablicu sa stupcem i dvjema točkama spajanja. Samim time, metodu možemo testirati kroz tri slučaja: provjera podataka u novoj tablici, u novom stupcu te na točkama spajanja. U Primjeru 4.13 prikazan je test za treći slučaj.

```
[Fact]
public void GivenDatabaseSchema_WhenCreatingNewTable_ThenAddConnectionPoints()
{
    // Given (Arrange)
    var mockCoordinatesCalculator = new Mock<ICoordinatesCalculator>();
    mockCoordinatesCalculator
        .Setup(calculator =>
            calculator.CalculateConnectionPointsX(It.IsAny<List<ConnectionPoint>>()))
        .Returns( // Namještamo x-koordinate točaka
            new Dictionary<string, double> { ["1_1_1"] = 2, ["1_1_2"] = 3 });
    mockCoordinatesCalculator.Setup(calculator =>
        calculator.CalculateConnectionPointY(expectedColumnId,
            It.IsAny<Table>()))
        .Returns(77); // Namještamo y-koordinatu točaka
    var schemaObjectFactory = new
        SchemaObjectFactory(mockCoordinatesCalculator.Object);
    var databaseSchema = new DatabaseSchema();
    // When (Act)
    schemaObjectFactory.CreateNewTable(databaseSchema);
    // Then (Assert)
    Assert.Contains(databaseSchema.ConnectionPoints, cp =>
        cp.UniqueIdentifier == "1_1_1" &&
        cp.ConnectionPointLeftCoordinate == 2 &&
        cp.ConnectionPointTopCoordinate == 77);
    Assert.Contains(databaseSchema.ConnectionPoints, cp =>
        cp.UniqueIdentifier == "1_1_2" &&
        cp.ConnectionPointLeftCoordinate == 3 &&
        cp.ConnectionPointTopCoordinate == 77);
}
```

Primjer 4.13: Testni slučaj za metodu `CreateNewTable`

Identifikator točke spajanja određen je formatom *tableId_columnId_tableSide* i obzirom na njega oponašamo i potvrđujemo podatke. Oponašamo računanje koordinata jer je ta operacija van odgovornosti klase koju testiramo. Računanje koordinata je odgovornost klase `CoordinatesCalculator` te će se u sklopu nje i istestirati.

- Metoda `CloseNewRelationship` služi za zatvaranje nove veze nakon što akcija crtanja na platnu završi. Veza se pri zatvaranju može i ne mora dodati u shemu. Veza se smije dodati u shemu samo ako spaja dva stupca iz različitih tablica koji još nisu spojeni. U Primjeru 4.14 testiramo različite slučajeve zatvaranja veza.

```
[Theory]
[MemberData(nameof(TestCases))]
public void GivenDatabaseSchema_WhenClosingNewRelationship_ThenCloseCorrectly(
    Relationship newRelationship,
    ConnectionPoint endingConnectionPoint,
    int newRelationshipsCount)
{
    // Given
    var schemaObjectFactory = new SchemaObjectFactory(null);
    var databaseSchema = new DatabaseSchema();
    databaseSchema.Relationships.Add(new Relationship
    {
        SourceTableId = 1, SourceColumnId = 10,
        DestinationTableId = 2, DestinationColumnId = 20
    });
    // When
    schemaObjectFactory.CloseNewRelationship(databaseSchema, newRelationship,
        endingConnectionPoint, "mock_start_point");
    // Then
    Assert.Equal(1 + newRelationshipsCount, databaseSchema.Relationships.Count);
}
public static TheoryData<Relationship, ConnectionPoint, int> TestCases()
{
   return new TheoryData<Relationship, ConnectionPoint, int> {
      { new Relationship { SourceTableId = 1, SourceColumnId = 10 }, new
          ConnectionPoint(3, 30, TableSide.Left), 1 },
      { new Relationship { SourceTableId = 1, SourceColumnId = 10 }, new
          ConnectionPoint(1, 11, TableSide.Right), 0 },
      { new Relationship { SourceTableId = 1, SourceColumnId = 10 }, new
          ConnectionPoint(2, 20, TableSide.Right), 0 },
      { new Relationship { SourceTableId = 2, SourceColumnId = 20 }, new
          ConnectionPoint(1, 10, TableSide.Right), 0 }
   };
}
```

Primjer 4.14: Testni slučajevi za metodu `CloseNewRelationship`

U testu simuliramo shemu s jednom vezom. Zatim, kroz parametre omotane u `TheoryData` dodajemo drugu vezu u četiri različita slučaja. Prvi slučaj je valjan, to jest veza se smije

dodati u shemu, dok su preostala tri slučaja nevaljana (pokušavamo spojiti stupce iz iste tablice ili stupce za koje već postoji veza). Metoda `CloseNewRelationship` ne poziva nijednu metodu servisa `CoordinatesCalculator`, stoga ga nismo morali oponašati.

Jedinično testiranje Razor komponenti

Poslovnu logiku aplikacije odvojili smo od sloja Razor komponenti te implementirali i istestirali unutar sloja servisa. Zato se u testiranju komponenti možemo fokusirati na testiranje korisničke interakcije i metoda životnih ciklusa, dok servise koji se pozivaju oponašamo.

Prođimo glavne primjere testova za Razor komponente u našoj aplikaciji. Pišemo standardne xUnit testove koristeći biblioteku bUnit. Svaka testna klasa nasljeđivat će klasu `TestContext`. Točnije, napravit ćemo baznu klasu koja će nasljeđivati `TestContext` te koja će u konstruktoru registrirati sve servise koje registriramo u aplikaciji, kako ne bismo morali to raditi ručno za svaki test. Bazna klasa `ComponentTestsBase` prikazana je u Primjeru 4.15.

```
@inherits TestContext
@code
{
    protected Mock<IPositionService> PositionServiceMock = new();
    // Na isti način definiramo preostale singleton servise...
    protected Mock<ISchemaPalApiService> SchemaPalApiServiceMock = new();
    // Na isti način definiramo preostale scoped servise...
    protected Mock<ISchemaObjectFactory> SchemaObjectFactoryMock = new();
    public ComponentTestsBase()
    {
        Services.AddSingleton(PositionServiceMock.Object);
        // ...
        Services.AddScoped(_ => SchemaPalApiServiceMock.Object);
        // ...
        Services.AddTransient(_ => SchemaObjectFactoryMock.Object);
    }
    public void Dispose()
    {
        Services.Clear();
    }
}
```

Primjer 4.15: Klasa `ComponentTestsBase`

Svaka testna klasa nasljeđivat će baznu klasu iz Primjera 4.15. Na taj način u svakoj testnoj klasi možemo dohvaćati i oponašati samo potrebne servise. Promotrimo komponentu `MySchemas` čija je inicijalizacija prikazana u Primjeru 3.24. Komponenta u trenutku inicijalizacije dohvaća sheme prijavljenog korisnika pozivom na API te ih prikazuje korisniku (Slika 3.6). Istestirat ćemo njezinu inicijalizaciju tako da oponašamo API metodu koja dohvaća sheme te provjerimo prikazuju li se osnovne informacije o primljenim shemama uspješno. Test je prikazan u Primjeru 4.16.

```
@inherits ComponentTestsBase
@code {
    [Fact]
    public async Task
        GivenSchemasFromApi_WhenRenderingComponent_ThenDisplayCorrectData()
    {
        // Given
        var firstSchemaLastSaved = DateTime.UtcNow;
        var secondSchemaLastSaved = DateTime.UtcNow.AddDays(-1);
        var testSchemas = new List<ShortSchemaRecord>
        {
            new ShortSchemaRecord { Id = Guid.NewGuid(), Name = "Schema 1",
                LastSaved = firstSchemaLastSaved },
            new ShortSchemaRecord { Id = Guid.NewGuid(), Name = "Schema 2",
                LastSaved = secondSchemaLastSaved }
        };
        // Oponašamo registrirani servis iz ComponentTestsBase klase.
        SchemaPalApiServiceMock.Setup(service =>
            service.GetDatabaseSchemasForLoggedInUser())
            .ReturnsAsync(Result.Ok(testSchemas));
        // When
        var component = RenderComponent<MySchemas>();
        // Then
        var schemaElements = component.FindAll(".schema-item");
        Assert.Equal(2, schemaElements.Count);
        Assert.Contains("Schema 1", schemaElements[0].TextContent);
        Assert.Contains("Zadnje azurirano: " + firstSchemaLastSaved,
            schemaElements[0].TextContent);
        Assert.Contains("Schema 2", schemaElements[1].TextContent);
        Assert.Contains("Zadnje azurirano: " + secondSchemaLastSaved,
            schemaElements[1].TextContent);
    }
}
```

Primjer 4.16: Testiranje inicijalizacije komponente `MySchemas`

Pogledajmo primjer testiranja korisničke interakcije. Komponenta `TableComponent` iz Primjera 3.15 na događaj klika točke spajanja namješta podatke o točki u pomoćni objekt za kreiranje nove veze u shemi. Simulirajmo taj događaj u testu i provjerimo jesu li se podaci točno namjestili. Testna metoda implementirana je u Primjeru 4.17.

```
[Fact]
public void GivenConnectionPoint_WhenClicked_ThenSetRelationshipCreationData()
{
    // Given
    var mockTable = new Table { Id = 1 };
    var mockConnectionPoint = new ConnectionPoint(mockTable.Id, columnId: 10,
        TableSide.Left);
    var mockSchema = new DatabaseSchema
    {
        Tables = new List<Table> { mockTable },
        ConnectionPoints = new List<ConnectionPoint> { mockConnectionPoint }
    };
    var relationshipHelper = new RelationshipCreationHelper { CreationMode =
        RelationshipCreationMode.None };
    // Prikazujemo komponentu s parametrima koje joj inače
    // prosljeđuje roditeljska komponenta CanvasContainer.
    var component = RenderComponent<TableComponent>(parameters => parameters
        .Add(p => p.DatabaseSchema, mockSchema)
        .Add(p => p.Table, mockTable)
        .Add(p => p.RelationshipCreationHelper, relationshipHelper));
    // When
    var connectionPointElement = component.Find(".connection-point");
    connectionPointElement.Click();
    // Then
    Assert.Equal(RelationshipCreationMode.StartCreation,
        relationshipHelper.CreationMode);
    Assert.Equal(mockConnectionPoint.UniqueIdentifier,
        relationshipHelper.StartingConnectionPointId);
}
```

Primjer 4.17: Testiranje klika točke spajanja u komponenti `TableComponent`

Komponenta `TableComponent` prihvaća parametre `RelationshipCreationHelper`, `Table` i `DatabaseSchema` od svoje roditeljske komponente. Mi ih u testu oponašamo pomoću opcije prosljeđivanja parametara u metodi `RenderComponent` klase `TestContext`.

Na sličan način bismo mogli oponašati parametre tipa `EventCallback` ili `Func<T>`, odnosno akcije proslijeđene od roditeljske komponente koje se trebaju izvršiti za neke događaje

u podređenoj komponenti. Komponenta `ExpandableSection` iz Primjera 3.13 predstavlja proširivi izbornik koji prikazuje ili sažima određene stavke (primjerice, tablice u lijevom izborniku stranice za izradu shema). Komponenta prima akciju koja se mora obaviti na stranici u trenutku klika gumba za dodavanje nove stavke. Možemo istestirati izvršava li se primljena akcija u trenutku klika gumba. Test je prikazan u Primjeru 4.18.

```
@using SchemaPal.SharedComponents
@inherits ComponentTestsBase
@code {
    [Fact]
    public void GivenAddNewItemAction_WhenAddNewItemClicked_ThenActionIsInvoked()
    {
        // Given
        var wasOnAddNewItemCalled = false;
        Func<Task> onAddNewItem = () =>
        {
            wasOnAddNewItemCalled = true;
            return Task.CompletedTask;
        };
        // Akciju prosljeđujemo kroz odgovarajući parametar.
        var component = RenderComponent<ExpandableSection>(parameters => parameters
            .Add(p => p.Title, "Demo naslov")
            .Add(p => p.OnAddNewItem, onAddNewItem)
        );
        // When
        var dodajButton = component.Find("button[title=Dodaj]");
        dodajButton.Click();
        // Then
        Assert.True(wasOnAddNewItemCalled);
    }
}
```

Primjer 4.18: Testiranje poziva roditeljske akcije u komponenti `ExpandableSection`

Integracijsko testiranje

U .NET-u pojam integracijskog testa može imati različite interpretacije. S jedne strane, možemo reći da je integracijski test svaki test u kojem zavisnosti klase ne oponašamo, već pozivamo preko registrirane reference u aplikacijskom kodu. Takav test predstavlja direktni prelazak na razinu iznad jedinične, stoga u teoriji ima i najviše smisla. Ipak, u praksi se takva interpretacija rijetko koristi. Promatrati komunikaciju dva servisa unutar, recimo, istog imeničkog prostora kao integraciju nema previše smisla. Oba dva servisa su dio is-

tog projekta i istog dijela koda. Zajedno se grade, registriraju i izvršavaju u aplikaciji. Za klasični integracijski test trebamo imati odvojene sustavne komponente, odnosno neovisne dijelove aplikacije koji u određenim slučajevima uspostavljaju komunikaciju. To su, na primjer, izvorni kod i baza podataka ili dva odvojena projekta u rješenju aplikacije.

U klijentskim web-aplikacijama integracija najčešće podrazumijeva mrežnu komunikaciju sa specifičnim pomoćnim sustavima, poput poslužiteljske aplikacije koju smo ukratko opisali u Poglavlju 3. Dakle, kao primjer integracijskog testa u našoj aplikaciji, možemo napraviti test koji radi pravi HTTP poziv na API neke instance naše poslužiteljske aplikacije. Naravno, instanca koju pozivamo treba biti testna, odnosno različita od instance koju koristi naša klijentska aplikacija.

Za početak izrađujemo novi projekt u rješenju aplikacije, `SchemaPal.IntegrationTests`. Za razliku od projekta za jedinične testove, ovaj projekt neće pratiti organizaciju glavnog projekta, već ćemo testove raspodjeljivati po vrstama integracije. Dakle, testove za našu integraciju ćemo dodavati u mapu `Api/SchemaPalApi`. Kao okvir ćemo i dalje koristiti xUnit. Nakon što smo napravili projekt, radimo baznu testnu klasu u kojoj ćemo instancirati HTTP klijenta s testnom adresom. Primjer 4.19 prikazuje opisanu klasu.

```
public class SchemaPalApiTestsBase
{
    protected HttpClient _httpClient;
    protected readonly ISchemaPalApiService _schemaPalApiService;
    public SchemaPalApiTestsBase()
    {
        _httpClient = new HttpClient();
        // Instanciramo HTTP klijenta na odgovarajućem URL-u i predajemo
        // ga u naš API servis pomoću oponašanja servisa IHttpClientFactory.
        _httpClient.BaseAddress = new Uri("https://api-test-instance/");
        var mockHttpClientFactory = new Mock<IHttpClientFactory>();
        mockHttpClientFactory.Setup(factory =>
            factory.CreateClient("SchemaPalApi")).Returns(_httpClient);
        _schemaPalApiService = new
            SchemaPalApiService(mockHttpClientFactory.Object);
    }
    public void Dispose()
    {
        _httpClient.Dispose();
    }
}
```

Primjer 4.19: Bazna klasa za testiranje API integracije

Sve testne klase nasljeđivat će klasu iz Primjera 4.19. Izradit ćemo novu testnu klasu za svaku API metodu koju koristimo. Testnu klasu za metodu `Authentication/Login` možemo vidjeti u Primjeru 4.20.

```
using FluentAssertions;
using SchemaPal.DataTransferObjects;
namespace SchemaPal.IntegrationTests.Api.SchemaPalApi.Authentication
{
    public class LoginTests : SchemaPalApiTestsBase
    {
        [Fact]
        public async Task GivenValidUserData_WhenLoggingIn_ThenResultIsSuccess()
        {
            // Given
            var username = Guid.NewGuid().ToString();
            var password = "Password1122";
            await _schemaPalApiService.RegisterUser(new UserRegistration
            {
                Username = username,
                Password = password,
                PasswordConfirmation = password
            });
            // When
            var loginResult = await _schemaPalApiService.LoginUser(new UserLogin
            {
                Username = username,
                Password = password
            });
            // Then
            Assert.True(loginResult.IsSuccess);
            Assert.NotNull(loginResult.Value);
            Assert.NotEmpty(loginResult.Value.Token);
        }
    }
}
```

Primjer 4.20: Integracijsko testiranje metode `LoginUser`

Primijetimo da smo korisnika prvo morali registrirati u sustav prije nego što smo inicirali proces prijave. To je česta pojava u integracijskim testovima. Kako bi testovi bili maksimalno neovisni od instance API integracije, moraju implementirati sve potrebne korake da bi se ključna akcija testa mogla izvršiti. U Primjeru 4.20 moramo na bilo kojoj testnoj instanci osigurati da korisnik s danim imenom i lozinkom već postoji u poslužiteljskoj aplikaciji prije nego što provjerimo proces prijave, što možemo ostvariti jedino na način da

eksplicitno unesemo novog korisnika na početku testa. Ne možemo, i ne bismo ni trebali, oslanjati se na statičke podatke ili implementaciju poslužiteljske aplikacije koju testiramo.

Na isti način testiramo i ostale API metode. Primjer 4.21 prikazuje testnu metodu koja provjerava je li nova korisnikova shema uspješno spremljena.

```
public class SaveDatabaseSchemaTests : SchemaPalApiTestsBase
{
    [Fact]
    public async Task GivenNewUserSchema_WhenSaving_ThenResultIsSuccess()
    {
        // Given
        // Pomoćna metoda u nasljeđenoj klasi koja prijavljuje korisnika.
        await LoginUser(Guid.NewGuid().ToString(), "Password12345!",
            shouldRegister: true);
        var databaseSchema = new DatabaseSchema
        {
            Tables = new List<Table>
            {
                new Table
                {
                    Id = 1,
                    Name = "table1",
                    Columns = new List<Column> { new Column(234, "c234", 1) }
                }
            }
        };
        var schemaRecord = new ExtendedSchemaRecord
        {
            Name = "Test schema",
            SchemaJsonFormat = JsonConvert.SerializeObject(databaseSchema)
        };
        // When
        var res = await _schemaPalApiService.SaveDatabaseSchema(schemaRecord);
        // Then
        Assert.True(res.IsSuccess);
        Assert.NotEqual(Guid.Empty, res.Value); // res.Value je novi ID sheme.
    }
}
```

Primjer 4.21: Integracijsko testiranje metode `SaveDatabaseSchema`

Bibliografija

[1] bUnit, *Getting started with bUnit*, 2024, https://bunit.dev/docs/getting-started/index.html, posjećeno u studenom 2024.

[2] MDN Web Docs, *Blob*, 2024, https://developer.mozilla.org/en-US/docs/Web/API/Blob, posjećeno u listopadu 2024.

[3] _____, *HTML Drag and Drop API*, 2024, https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API, posjećeno u listopadu 2024.

[4] _____, *Populating the page: how browsers work*, 2024, https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work, posjećeno u listopadu 2024.

[5] Jimmy Engström, *Web Development with Blazor*, Packt Publishing Ltd., 2021.

[6] Sebastian Gingter, *Understanding and Controlling the Blazor WebAssembly Startup Process*, 2023, https://www.thinktecture.com/blazor/understanding-and-controlling-the-blazor-webassembly-startup-process/, posjećeno u listopadu 2024.

[7] Vladimir Khorikov, *Unit Testing Principles, Practices, and Patterns*, Manning, 2020.

[8] Microsoft Learn, *C# Best Practices : Dangers of Violating SOLID Principles in C#*, 2015, https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp, posjećeno u listopadu 2024.

[9] _____, *ASP.NET Core Blazor*, 2024, https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0, posjećeno u listopadu 2024.

[10] _____, *.NET dependency injection*, 2024, https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection, posjećeno u listopadu 2024.

[11] _____, *Unit testing C# in .NET using dotnet test and xUnit*, 2024, https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test, posjećeno u studenom 2024.

[12] Robert Manger, *Baze podataka*, Element, 2014.

[13] Elliot Moule, *Using Moq: A Simple Guide to Mocking for .NET*, 2024, https://www.codemag.com/Article/2305041/Using-Moq-A-Simple-Guide-to-Mocking-for-.NET, posjećeno u studenom 2024.

Sažetak

U ovom radu proučili smo Blazor, alat za razvoj interaktivnih web-aplikacija u C#-u. Blazor je razvojni okvir koji olakšava pisanje klijentskog koda programerima upoznatima s .NET-om. Programeri u Blazoru umjesto JavaScripta koriste programski jezik C# za razvoj funkcionalnosti na klijentskoj strani aplikacija. Posebno smo proučili Blazor WebAssembly, model Blazora u kojem se aplikacija u cijelosti izvršava unutar internetskog preglednika. Ovo je ostvareno pomoću WebAssemblyja, programskog jezika niske razine koji se može izvršavati u pregledniku.

U Blazor WebAssemblyju razvili smo složeniju web-aplikaciju. Korisničku interakciju s aplikacijom implementirali smo pomoću Razor komponenti, alata u Blazoru koji omogućava povezivanje C# koda s događajima HTML elemenata. Aplikaciju smo na nižim razinama istestirali koristeći standardne alate za pisanje testova u .NET-u uz integriranje alata bUnit, biblioteke u Blazoru koja omogućuje pisanje jediničnih testova za Razor komponente.

Summary

In this thesis we studied Blazor, a framework for building interactive web applications using C#. Blazor simplifies writing front-end code for developers familiar with .NET. In Blazor, developers use C# instead of JavaScript to develop client-side functionalities within web applications. We specifically examined Blazor WebAssembly, a Blazor model that enables application code to execute directly in the browser. This functionality is made possible through WebAssembly, a low-level language that modern web browsers can natively run.

With Blazor WebAssembly we built a more complex web application. User interactions were implemented using Razor components, a Blazor feature that enables linking C# code to HTML element events. We implemented lower-level testing using standard .NET testing tools with integrated bUnit, a Blazor library that supports writing unit tests for Razor components.

Životopis

Rođena sam 1. svibnja 2000. godine u Zagrebu. Odrasla sam u Zaprešiću, gdje sam pohađala osnovnu školu i opću gimnaziju. U srpnju 2018. godine upisujem preddiplomski studij Matematika na Prirodoslovno-matematičkom fakultetu u Zagrebu. Preddiplomski studij završavam u srpnju 2022. godine. Iste godine upisujem diplomski studij Računarstvo i matematika. U ožujku 2022. godine počinjem raditi kao .NET programer u softverskoj tvrtki Lemax.