Razvoj aplikacija za web pomoću okvira Blazor

Tropčić, Stefan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Stefan Tropčić

Razvoj aplikacija za web pomoću okvira Blazor

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Stefan Tropčić

Matični broj: 35918/07–R

Studij: Poslovni sustavi

RAZVOJ APLIKACIJA ZA WEB POMOĆU OKVIRA BLAZOR

ZAVRŠNI RAD

Mentor:

Marko Mijač, mag. inf.

Varaždin, rujan 2020.

Stefan Tropčić

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Razvojem tehnologije raste upotreba interneta te web aplikacija. Danas je pristup web aplikacijama moguć s raznih uređaja od računala do pametnih satova. Potreba za izradom velikog broja aplikacija za web u što kraćem vremenu dovodi do razvoja okvira koji olakšavaju i ubrzavaju razvoj. Jedan takav okvir je i Blazor unutar kojeg se, za razliku od većine trenutno dostupnih okvira za razvoj klijentskog dijela aplikacije za web, ne razvija pomoću JavaScripta nego C#-a. Ta činjenica omogućuje programeru ponovno korištenje poslovne logike i modela pisanih za poslužitelj na klijentu te mu pruža mogućnost samostalnog razvoja web aplikacije znanjem samo jednog jezika. Blazor ovisno o modelu posluživanja koristi različite tehnologije. Kod poslužiteljskog oblika koristi se komunikacija u stvarnom vremenu preko SignalR biblioteke, a kod klijentskog oblika mogućnosti WebAssembly-ja, otvorenog standarda implementiranog u svim trenutno aktivnim preglednicima. Kao i neki drugi okviri za razvoj jednostraničnih aplikacija (engl. *Single-page application* (SPA)), npr. React, Blazor koristi komponente koje se lakše razvijaju i testiraju, a u konačnici mogu realizirati i najkompleksnija klijentska sučelja. Uz opis trenutno popularnih tehnologija za web aplikacije i sam Blazor okvir, na kraju rada prezentirana je aplikacija za upoznavanje novih osoba implementirana pomoću Blazor okvira.

Ključne riječi: web; Blazor; JavaScript, WebAssembly, C#, SPA, .NET-Core.

Sadržaj

1. Uvod

Napretkom tehnologije internet je iz dana u dan dostupan sve većem broju ljudi, a uz razvoj nekih globalnih projekata kao Starlink (skup satelita koji pruža širokopojasni internet i na područjima koja do sada uopće nisu imala mogućnost pristupa internetu [1]), bit će dostupan cjelokupnoj svjetskoj populaciji. Najčešći oblik konzumiranja interneta uz e-mail, telefoniju i dijeljenje datoteka je World Wide Web čije aplikacije koristi svaki čovjek s pristupom računalu, tabletu, mobitelu, pametnom satu ali i ostalim pametnim uređajima. Upravo te aplikacije, odnosno njihov jednostavniji, brži i kvalitetniji razvoj, tema su ovog rada.

Kako programeri imaju zadatak razvoja aplikacija za web koje se mogu koristiti na širokom spektru uređaja, s različitom snagom samog hardwarea ali i brzinom interneta kojem ti uređaji imaju pristupa te za različite domene, od poslovnih, edukacijskih do zabavnih i medijskih, razvili su velik broj tehnologija da bi si olakšali određene probleme s kojima su se susreli. To je dovelo do toga da danas za sve postoji po nekoliko načina mogućeg rješavanja problema, od vrste programskog jezika do mnogobrojnih biblioteka i okvira unutar samih jezika. Sve to dolazi na naplatu u obliku potrebe za dugotrajnim obrazovanjem programera unutar akademske zajednice, ali i tijekom zaposlenja, kako bi stekao dovoljno znanja za samostalnu izgradnju jedne aplikacije za web. Uz to, širenjem interneta raste i potreba za izradom sve većeg broja aplikacija te se danas nalazimo u situaciji da na tržištu ne postoji dovoljno kompetentnih ljudi za taj posao.

Najvećim uzročnikom dugotrajnog obrazovanja programera koji razvijaju aplikacije za web, a posljedično i manjak kompetentnih radnika, smatram korištenje odvojenih tehnologija za razvoj klijentskog i poslužiteljskog dijela aplikacije. Naime, danas za razvoj poslužiteljskog dijela aplikacije imamo na izbor velik broj programskih jezika kao što su C#, PHP, Java, Rust, Python, Go, itd.. Na klijentskom dijelu ne postoji tako velik izbor programskih jezika za izradu interaktivnosti unutar aplikacija. Dapače, postoji samo jedan jezik, univerzalan za sve moderne preglednike, a to je JavaScript. To znači da se pri razvoju bilo koje aplikacije za web mora koristiti minimalno dva programska jezika. No što kada programer ne bi trebao učiti dva jezika, jedan za poslužiteljsku stranu aplikacije, a drugi za klijentsku stranu? Što kada bi se cijela aplikacija mogla napisati unutar jednog jezika, što bi zasigurno rezultiralo kraćim periodom edukacije i većim izborom stručnjaka?

Kao rješenje navedenog problema vidim Blazor, web okvir koji nam omogućuje pisanje klijentskog i poslužiteljskog dijela aplikacija za web u jednom jeziku (C#), a posljedično ponovno korištenje veće količine koda, korištenje poznatih biblioteka i rad u poznatoj okolini. U početnom dijelu rada predstavit ću trenutno potrebne jezike i okvire za razvoj jedne

aplikacije. To uključuje HTML i CSS kao osnovne strukturne jedinice, JavaScript jezik koji se koristi u preglednicima od sredine 90-ih, te okvire kao React, Angular i Vue koji dominiraju klijentskom sferom. Tema središnjeg dijela je Blazor. Započet ću s opisivanjem tehnologija WebAssembly i SignalR na kojima se Blazor bazira, a nakon toga slijedi prikaz njegovih osnovnih značajki, odnosno sve što je potrebno za izradu jedne aplikacije koristeći Blazor. Na posljetku slijedi primjer, jednostavne aplikacije za upoznavanje novih osoba, korištenjem samo C# na poslužitelju i klijentu.

2. Tradicionalne tehnologije za razvoj aplikacija za web

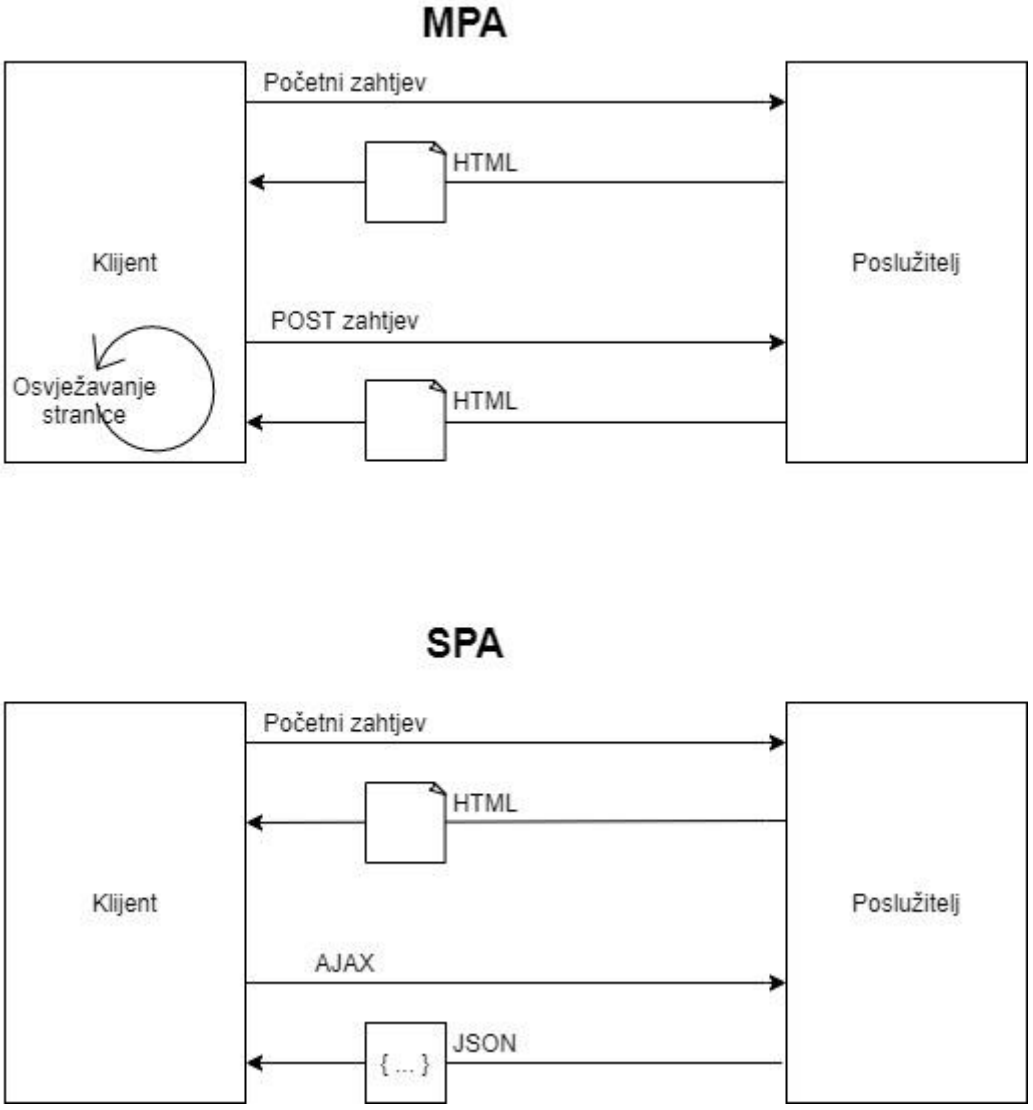
2.1. Aplikacije za web

Aplikacije za web su aplikacije koje se poslužuju s udaljenog poslužitelja i korisnici ih konzumiraju preko interneta. Njihov kod je najčešće pisan u nekoliko jezika: HTML, CSS i JavaScript na klijentu, dok je na poslužiteljskoj strani moguće odabrati između više tehnologija među kojima su najpopularniji Java, PHP, C#, Python i Go. Tradicionalne web stranice su višestranične (engl. *Multi-page Application* (MPA)) te komuniciraju s poslužiteljem sinkrono. Sinkrona komunikacija u ovom slučaju znači da inicijalno klijent (preglednik) pošalje zahtjev poslužitelju, na koji poslužitelj nakon obrade prosljeđuje odgovor natrag klijentu. U ovom obliku komunikacije taj odgovor je nova web stranica, to jest novi HTML dokument. Da bi se taj novi dokument prikazao korisniku nakon što preglednik dobije odgovor, dolazi do osvježavanja pri čemu se gubi na kvaliteti korisničkog iskustva (engl. *user experience* (UX)). Kako bi se spriječilo osvježavanje stranice i postiglo što ugodnije iskustvo korištenja aplikacija za web, nastala je asinkrona komunikacija.

Asinkrona komunikacija postala je moguća uz tehnologiju Asinkronog JavaScripta i XML-a (engl. *Asynchronous JavaScript And XML* (AJAX)). On nam omogućuje da klijent pošalje zahtjev i primi odgovor u pozadini aplikacije, istodobno pružajući interaktivnost aplikacije. Nakon što klijent primi podatke oni se bez osvježavanja cijele stranice prikazuju klijentu. To u konačnici rezultira kvalitetnijim korisničkim iskustvom te su mnoge stolne aplikacije prepisane na web.[2]

Dolaskom AJAX-a u igru nastaju nove jednostranične web aplikacije (engl. *Single-page application* (SPA)) kako bi se prijenos podataka između klijenta i poslužitelja sveo na minimum. Kod njih nikada ne dolazi do potpunog osvježavanja web stranica. Aplikacija pri početnom učitavanju dohvaća osnovni HTML, CSS i JavaScript kod potreban za rad. Ovisno o korisnikovim interakcijama aplikacija preko AJAX-a s poslužitelja dohvaća potrebne podatke. Uz to izgrađena je od komponenti koje ovisno o korisnikovoj interakciji zajedno s pristiglim podacima dinamički izmjenjuju stranicu bez osvježavanja. Jednostranične aplikacije često postaju vrlo kompleksne upravo jer podatke treba obraditi na klijentskoj strani uz pomoć JavaScripta pa su nastali okviri i biblioteke kako bi se olakšao njihov razvoj. Moderne aplikacije za web često spajaju dva svijeta (neki moduli su izgrađeni na modelu MPA, a drugi na SPA) te čine skladnu cjelinu.[3]

Na Slici 1 prikazana je razlika između višestraničnih i jednostraničnih aplikacija. Kod višestraničnih aplikacija element komunikacije je HTML dokument, dok se kod jednostranične s poslužitelja na klijent ne šalje HTML dokument nego podaci. Ti podaci su danas najčešće u



Slika 1: MPA vs. SPA (vlastita izrada prema [2])

JavaScript objektnoj notaciji (engl. *JavaScript Object Notation* (JSON)), formatu za koji JavaScript ima ugrađeni parser pa je rad s tim oblikom podataka vrlo jednostavan.[3]

Rastom pametnih uređaja raznih oblika sve više se spominju i progresivne web aplikacije (engl. *Progressive Web Application* (PWA)). To su aplikacije za web koje su:

(1) **sposobne** (mogu iskoristiti značajke do sada dostupne samo nativnim aplikacijama)

(2) **pouzdane** (funkcioniraju s nativnim performansama i neovisno o internet povezanosti)

(3) **moguće ih je instalirati** (mogu se pokrenuti i izvan preglednika).

Nastale su kako bi spojile najbolje od dva svijeta (nativni i web): nativni svijet pruža bolju integraciju sa samim uređajem, a time i bolje korisničko iskustvo, dok web pruža veću dostupnost.[4]

2.2. HTML

HTML (engl. *Hyper Text Markup Language*) je jezik koji je inicijalno zamišljen za semantičko opisivanje znanstvenih dokumenata, no kroz vrijeme počeo se koristiti i za druge tipove dokumentacije i za razvoj aplikacija. U početku ga je razvio Tim Berners-Lee 1990. godine dok je radio u CERN-u. No kako se počeo koristiti u preglednicima Netscapea i Microsofta, svaki od timova implementirao je nova, vlastita svojstva sa željom da unaprijede svoj proizvod. Kako je svaki preglednik razvio svoju verziju jezika, koje su se ponekad uvelike razlikovale između verzija preglednika istog proizvođača, 1994. godine Tim Berners-Lee osniva W3C koji između ostalog želi napraviti standarde za HTML.[5]

Kroz godine W3C je mijenjao verzije, no 2000-ih počinje zagovarati XML-ovu inačicu HTML-a. Zbog navedenog te kasnijeg zagovaranja kretanja weba u drugom smjeru, velika poduzeća, proizvođači vlastitih preglednika osnivaju grupacija WHATWG. Organizacije rade paralelno ali i surađujući, na budućoj i trenutno aktivnoj verziji HTML5, sve dok 2019. godine nisu i službeno potpisali suradnju na jedinstvenoj verziji otvorenoj za daljnje nadogradnje. Među najvažnijim novim mogućnostima verzije su novi elementi za multimedijski sadržaj koji služe za nativan prikaz audio i video materijala, podrška vektorske grafike, canvas element za grafiku programiranu JavaScriptom te raznolike nove mogućnost skladištenja podataka kod klijenta.[6]

Pisanjem u HTML-u definiramo elemente koji se realiziraju na različite načine, ali najčešći oblik se sastoji od početne oznake, sadržaja i završne oznake. Sama početna oznaka, `<p>`, definira se znakom manje, iza kojeg slijedi naziv elementa, u ovom slučaju p, kao odlomak (engl. *paragraph)* te znak veće. Nakon njega stavljamo sadržaj, u ovom slučaju tekst odlomka te označujemo kraj elementa završnom oznakom koji se od početne razlikuje kosom crtom, `</p>`. Sam sadržaj, osim danog primjera, tekst, može biti i drugi ugniježđeni element, uz određena ograničenja jer ne može svaki element biti dijete drugog.

Unutar početne oznake navodimo i atribute elementa, parove ključa i vrijednosti za konfiguraciju samog elementa. Ti atributi mogu biti globalni, dostupni na svim elementima kao na primjer `id` kojim definiramo identifikacijsku oznaku elementa. Identifikacijskim oznakama dohvaćamo elemente iz CSS i JavaScript koda da bi stranicu učinili dinamičkom. Osim

globalnih postoje i specifični atributi kao `src`, koji se nalazi na `img`, `audio`, `script` i još nekim elementima. Njihova vrijednost pokazuje na URL sadržaj tog elementa. HTML datoteka ima datotečnu ekstenziju .html.

Elemente HTML datoteke preglednici pretvaraju u objektni model dokumenta (engl. *Document Object Model* (DOM)). To je zapravo prikaz HTML dokumenta u računalnoj memoriji u obliku stabla elemenata preko kojeg ostvarujemo manipulaciju HTML dokumenta iz JavaScripta.

HTML5 uvodi nove elemente s naglaskom na semantiku dokumenta kako bi programeru, ali i računalu (rangiranje web stranica unutar tražilice, čitaču ekrana za osobe s poteškoćama u vidu) omogućili lakše raspoznavanje elemenata i njihove primarne upotrebe. Prije HTML-a 5 programeri su koristili `div` kao univerzalni element kod kojeg je semantičko značenje bilo upisano u vrijednost globalnog atributa `id` i/ili `class`. Namjera je da se ta praksa zamijeni novim elementima koji bi intuitivno označavali semantičko. Na primjer element `<div id=navbar"></div>` zamijenimo elementom `<nav></nav>`.

U nastavku slijedi primjer jednostavne HTML datoteke.

```
<!DOCTYPE html>
<html>
  <head>
     <title>Primjer</title>
  </head>
  <body>
     <article>
        <h1>Kako napisati članak</h1>
        <p id="opis">Sadržajni dio HTML dokumenta</p>
        <img src="../slike/primjer.jpg">
     </article>
  </body>
</html>
```

Na početku izraz `<!DOCTYPE html>` definira tip datoteke, koji omogućuje preglednicima ispravan prikaz sadržaja. Nakon njega slijedi `html` element koji sadrži sve ostale elemente. Prvi sljedeći je `head`, klijentu nevidljiv sadržaj, unutar kojeg stavljamo naslov, metapodatke (autor, skup znakova, opis stranice), uključujemo stilove i druge datoteke (JavaScript datoteke). Na kraju imamo `body` koji predstavlja sadržaj same stranice koji će klijentu biti prikazan.[5]

2.3. CSS

CSS (engl. *Cascading Style Sheets*) je jezik koji se koristi za prezentaciju, to jest izgled dokumenta pisanog u HTML-u ili XML-u dokument. HTML-om određujemo semantiku određenog elementa, njegov sadržaj te donekle prezentaciju upotrebom dodatnih elemenata. No da bi u potpunosti mogli urediti izgled sadržaja, od veličine, granice, boje do pozicije u odnosu na druge elemente, koristimo CSS.

Kao i HTML, standardi se razvijaju pod okriljem W3C organizacije, koja je nakon podužeg razvoja verzije CSS2, a tijekom razvoja izrazito velike verzije CSS3, odlučila jezik podijeliti u module upravo s ciljem što ažurnijeg objavljivanja novih preporuka.[7]

CSS uvodimo u HTML datoteku dodavanjem `link` elementa unutar `head` elementa kao što je prikazano u nastavku.

```
<link rel="stylesheet" href="stilovi.css">
```

To je prazan element koji sadrži samo atribute. Njegov `href` atribut predstavlja URL do datoteke unutar koje se nalaze stilske upute za povezani HTML dokument. To je najčešći oblik korištenja, no postoje još dva načina. Jedan je upotreba `style` atributa na svakom pojedinačnom HTML elementu, format koji se najčešće izbjegava zbog svoje slabe čitljivosti, jer CSS dio zna biti veliki komad teksta koji onda smanjuje preglednost HTML dokumenta. Još jedna mana ovog pristupa je što je specifičan za element, tj. ne možemo napisati stilske upute za drugi element kao vrijednost atributa jednog HTML elementa. Zadnja opcija je upotreba `style` elementa unutar kojeg kao i kod zasebne datoteke možemo upisati sve povezane stilske upute.[5]

CSS je jezik koji je baziran na pravilima, a ta pravila definiramo za specifični element ili za određenu grupu elemenata. Jedan izraz sastoji se od selektora, a potom od vitičastih zagrada unutar kojih se nalazi sama uputa. Selektor, kojim odabiremo na što točno primjenjujemo stilsku uputu, u CSS-u može biti naziv elementa, njegova klasa, id elementa ili kombinacija naziva elementa i njegove klase/atributa/specifičnog stanja. Stilska uputa je par ključa i vrijednosti, koji su definirane ključne riječi, odijeljenih dvotočkom, a iza koje slijedi točka sa zarezom koja označuje završetak jedne upute. Unutar vitičastih zagrada mogu se nalaziti višestruke upute.

```
h1 {
     color: yellow;
     font-family: "Times New Roman";
}
```

U primjeru je upit koji obuhvaća sve `h1` elemente unutar HTML dokumenta te će prilikom renderiranja u pregledniku tekst unutar njih biti prikazan žutom bojom te fontom Times New Roman. Osim prikazanih osnovnih oblika uređivanja teksta, CSS nam omogućuje npr. stvaranje sjene na tekstu, organizaciju sadržaja u tablice nepravilnog formata, animacije, media queries (prilagođavanje dizajna veličini klijentovog ekrana), različite mjerne jedinice (također s ciljem što veće responzivnosti aplikacije) itd.. CSS datoteka ima datotečnu ekstenziju .css.

Da bi se olakšala izrada web stranica za CSS su razvijeni određeni okviri i alati za procesiranje. CSS okvir (engl. *framework*) je skup stilskih uputa koji se vrlo često ponavljaju među aplikacijama, a najčešće su manifestirane u obliku klasa koje pridijelimo svome HTML elementu da dobijemo određeni segment dizajna. Za primjer, jedan od najpopularnijih je Bootstrap koji definira različite klase za forme, gumbe, navigaciju i druge komponente. CSS alati za procesiranje su programi koji omogućuju stvaranje CSS-a iz vlastite sintakse, kako bi proširili osnovne mogućnosti CSS-a. Kao primjer navodim varijable, ugnježđivanje stilova, funkcije, matematičke operacije. Primjeri alata za procesiranje su LESS, Sass i Stylus.[7]

2.4. JavaScript

JavaScript je skriptni programski jezik koji nam uz dinamičko ponašanje unutar web aplikacija omogućuje interaktivnost, promjenu sadržaja, definiranja koda koji će odgovoriti na neki događaj, prikaz animacija, kontroliranje multimedije, ukratko sve što nije statička prezentacija informacija. On je uz HTML i CSS treći osnovni dio standardnih web tehnologija potrebnih za razvoj klijentskog dijela aplikacije za web. JavaScript je osim toga i jezik koji se interpretira, tj. kompajlira u trenutku korištenja (engl. *just-in-time compiled*). Iako je najpoznatija njegova upotreba za klijentski dio aplikacija za web, pomoću NodeJS-a, izvršne okoline, JavaScript se može izvršavati i na poslužiteljskoj strani web aplikacija.[8]

Sredinom 1990-ih nastala je želja za dinamikom unutar tadašnji statičkih web stranica. Tako je Brendan Eich 1995. godine unutar Netscape Communications poduzeća osmislio jezik koji je implementiran u tada popularnom pregledniku Netscape Communicator. Unutar par mjeseci postojanja jezik je promijenio nekoliko naziva, Mocha, LiveScript, a kako je u to vrijeme dogovorena primjena Java programskog jezika unutar njihovog preglednika, naposljetku je nazvan JavaScript. Naime, JavaScript je bio zamišljen kao skriptni pomagatelj velikoj i kompliciranoj Javi, orijentiranoj na veće zadatke. Također pretpostavka je bila da ga neće koristiti inženjeri, nego samo amateri i dizajneri za male zadatke. S vremenom u utrku ulazi i Microsoft s vlastitom implementacijom JScript koji je imao svoje razlike u odnosu na Netscape-ovu verziju. Kao što smo vidjeli i kod HTML i CSS, da bi se ostvarila šira upotreba potrebna je

standardizacija. 1996. godine Europska zajednica računalnih proizvođača (engl. *European Computer Manufacturers Association* (ECMA)) izdaje standard ECMA-262, kojim se opisuje ECMAScript jezik (naziv za JavaScript koji ne sadrži zaštitni znak). S godinama se jezik nadograđivao i izlazile su verzije, prva veća među njima je bila ECMAScript 3 nakon koje je nastala pauza od deset godina zbog nemogućnosti postizanja konsenzusa. Nakon nje slijedi ECMAScript 5 s manjim izmjenama te zatim ECMAScript 6 (ECMAScript 2015). Ona donosi mnoge promjene, neke od njih su određeni proizvođači preglednika uveli tek nakon nekoliko godina. Zbog toga nastaju transpajleri, kao Babel, koji kompajliraju JavaScript kod na niže verzije koje podržavaju stariji preglednici. Nakon ECMAScript 6 je odlučeno da će se nova verzija standarda izdavati svake godine sa zasad manjim izmjenama unutar jezika.[9]

JavaScript je jezik sličan u sintaksi Javi odnosno C programskom jeziku. Neki slični elementi su osnovne matematičke operacije, blokovi, grananja i petlje. Ima sedam primitivnih tipova podataka, a to su undefined, null, boolean, string, symbol, number, bigint i posebni strukturni tip object. Kako je JavaScript labavo tipiziran (engl. *loosely typed*), odnosno nije potrebno definirati tip varijable pri deklaraciji, deklaraciju varijabli izvršavamo korištenjem ključnih riječi. Imamo ključnu riječ `var` koja ima domet (engl. *scope*) funkcije i ključnu riječ `let` koji ima opseg bloka te za konstante ključnu riječ `const`. Tip specifične varijable je automatski dodijeljen s obzirom na vrijednost koja je u njoj sadržana i dinamički se mijenja u ovisnosti o promjeni vrijednosti. Blokovi koda unutar JavaScripta definiraju se vitičastim zagradama. Oznaka za završetak linije koda je simbol točka sa zarezom. Grananje se ostvaruje `if`, `else if`, `else` naredbama ili pomoću `switch case` naredbi. Operatori za aritmetičke operacije su standardni kao i u svim višim jezicima. Dostupne su petlje `for`, `while`, `do while` i `for of` petlje implementirane pomoću iteratora, koja je zapravo pregledniji oblik klasične `for` petlje. Komentare ostvarujemo pomoću dvije kose crte za jedan red i kosa crta sa zvjezdicom za više redova.

Slijedi primjer navedenog.

```
//deklariranje varijabli
var osoba = 'Stefan Tropčić';
console.log(`Pozdrav ${osoba}!`);
osoba = 20;
/*primjer
grananja*/
if (osoba > 18 && osoba < 25){
 console.log('Osoba je student.');
}
else {
console.log('Osoba vjerojatno nije student.');
```

```
}
//aritmeticke operacije
var godinaNaFoi = osoba-18;
console.log(`Osoba je vjerojatno student ${godinaNaFoi} godine.`);
//petlja po polju objekata
var studenti = [{'ime' : 'Stefan', 'godina':3},
            {'ime' : 'Mate', 'godina':2},
            {'ime' : 'Matea', 'godina':1},
            {'ime' : 'Ivana', 'godina':4}];
for (let student of studenti){
     console.log(`${student.ime} je student ${student.godina} godine.`);
}
```

JavaScript kod uključujemo u HTML dokument pomoću `script` elementa na dva načina. Prvi je sličan CSS-u i `style` elementu, tako da sadržaj unutar `script` elemenata čini JavaScript kod. Drugi način nam omogućuje da pomoću `script` elementa i njegovog `src` atributa povežemo JavaScript datoteku, tekstualnu datoteku s datotečnom ekstenzijom .js, kao u sljedećem primjeru.[8]

```
<script src="mojJavaScript.js"></script>
```

2.5. JavaScript okviri klijentske strane

Prelaskom zajednice na razvoj jednostraničnih aplikacija počinju se razvijati mnogi okviri i biblioteke kako bi se olakšao razvoj i smanjilo vrijeme potrebno za izradu sve većih, dinamičnijih i interaktivnijih aplikacija za web.

Biblioteke u JavaScriptu su postojale i ranije, prva globalno popularna jQuery nastala je 2006. godine. Imala je za cilj smanjiti količinu JavaScript koda koji programer mora napisati da bi ostvario neki zadatak, kao što i govori sam logo na kojem stoji „piši manje, učini više“ (engl. *write less, do more*).[10] jQuery je olakšao programiranje na klijentu i u smislu što je objedinio različite implementacije JavaScripta u preglednicima u jedinstveni API. Ta univerzalnost omogućila je inovativni razvoj novih aplikacija što je na posljetku dovelo do velikih aplikacija (s obzirom na broj linija koda) često neurednog i neorganiziranog koda. Tada nastaju okviri koji programerima pružaju arhitekturalnu organizaciju. Nastaje Backbone koji pruža tanki sloj nad postojećim jQuery-em, slijedi ga Knockout koji prvi uvodi data-binding. AngularJS je prvi objedinio ključna svojstva koja su programerima i danas bitna vezanje podataka (engl. *data-binding*), usmjeravanje (engl. *routing*) i korištenje predložaka (engl. *templating*). Među većim okvirima bio je i Ember, po svojstvima sličan AngularJS ali s nadograđenim usmjeravanjem.[11]

Kako je rasla popularnost aplikacija za web s godinama su se počeli pojavljivati sve robusniji okviri, s više svojstava i u optimiziranijem obliku. Trenutačno na tržištu ne postoji jedan okvir koji se smatra najboljim, već nekoliko njih, a svaki od njih pruža različito razmišljanje odnosno pristup rješavanju problema izgradnje aplikacije. Upravo je taj pristup, stavka koja olakšava pisanje koda velikih aplikacija, ali i pruža jednostavno održavanje kroz duži vremenski period. Druga vrlo bitna stavka svakog okvira je da pojednostavljuje sinkronizaciju između korisničkog sučelja (engl. *user interface* (UI)) aplikacije i stanja (engl. *state*) unutar te aplikacije. Osim već navedenog, okviri nam direktno ili indirektno pružaju: alate za rad u njima, jednostavniji rad zbog podjele aplikacije na komponente, skalabilnost, usmjeravanje, itd.[12] Danas su, prema nekoliko parametara, od samog broja preuzimanja okvira, preko Stack Overflow i Reddit tema do broja Github repozitorija najpopularniji React, Angular i Vue.js.[13]

React je JavaScript biblioteka za razvoj klijentskih sučelja, nastala unutar Facebooka za njihove potrebe razvoja Facebook aplikacije, a objavljena je kao program otvorenog koda 2013. godine. On sam po sebi nema obilježja okvira jer ne pruža sve značajke koje omogućavaju zaokruženi razvoj aplikacija za web. Međutim, koristi se s raznim drugim paketima za npr. usmjeravanje (React Router) i održavanje stanja aplikacije (Redux) te se kolokvijalno naziva okvirom. React uvodi Virtual DOM, kopiju klijentskog DOM-a sačuvanu u računalnoj memoriji. Pomoću njega dolazi do razlika između dva stanja s ciljem ostvarivanja efikasnije izmjene na sučelju aplikacije.

Angular je okvir koji je također temeljen na komponentama. Razvijen je od strane Angular tima unutar Googlea, nakon što su odlučili napustiti razvoj AngularJS (okvir iz 2010. godine). Angular je nanovo napisan okvir te objavljen 2016. godine kao program otvorenog koda. Koristi TypeScript, superset JavaScripta koji se zajedno s deklarativnim HTML predlošcima u trenutku izgradnje (engl. *build time*) transpajlira u optimalni JavaScript. Programeri koji rade na Angularu odlučili su se za TypeScript, koji je kako mu samo ime kaže orijentiran na tip, odnosno dodaje u JavaScript statičko definiranje tipova. Uz te vrlo bitne stavke kod razvoja velikih projekata, koja uvelike olakšava samo programiranje, TypeScript pruža i potporu za moderna svojstva JavaScripta uvedena novim verzijama ECMAScripta. Na posljetku TypeScript programeru pruža i potporu u okviru izvrsne razvojne okoline koja uvelike olakšava proces razvoja.

Vue je izašao 2014. godine i smatra se najmlađim od navedene trojke, a razvio ga je programer koji je u Googleu radio na AngularJS. Za razliku od prethodno navedenih okvira iza Vuea ne stoji neko veliko poduzeće, nego je razvijen od strane zajednice, što je možda i razlog za sve većim rastom u popularnosti. Po načinu pisanja sličan je Reactu u tome što se i UI dio i interaktivnost pišu unutar komponenti. U Reactu se radi unutar JSX, a unutra Vue unutar

HTML predložaka ili JSX. Glavna osobina Vuea je jednostavnost integracije u postojeće projekte te naglasak na razvoj manjih web aplikacija, za razliku od Reacta, iako je uz dodatne pakete moguće ostvariti i razvoj velikih kompleksnih aplikacija.[12]

2.6. Problem

Kako veličina i kompleksnost web aplikacija iz godine u godinu rastu tako raste i količina koda koju je potrebnu napisati da bi takva aplikacija funkcionirala. Opisani su neki JavaScript okviri koji programerima uvelike olakšavaju razvoj i organizaciju tako puno linija koda. Svaki ima svoje prednosti ali i mane u odnosu na neki drugi no svim okvirima je zajedničko da od full-stack programera zahtijevaju znanje minimalno još jednog programskog jezika kojim se programer koristi na poslužiteljskoj strani web aplikacije i vrlo često znanje SQL-a za rad s podacima u bazi podataka. Neki od trenutno popularnih jezika na poslužitelju su Java, PHP, C#, Ruby i Python. Oni, zajedno s cijelom radnom okolinom oko njih, predstavljaju veliki skup informacija i znanja koji se neprestano mijenja i s kojima programer mora naučiti baratati da bi uspješno razvijao nove aplikacije.

Da bi programer bio u trendu mora konstantno proučavati promjene koje se događaju unutar dva programska jezika, odabranog jezika koji koristi na poslužitelju kojim najčešće stvara aplikaciju za pružanje podataka klijentu i samog jezika JavaScripta koji se u zadnje vrijeme aktivno mijenja, dodavanjem novih značajki unutar jezika. Na to dodajmo česte promjene unutar samih okvira i potrebnih među-alata da bi to sve radilo. Tako se još 2016. godine [14] na internetu pojavio pojam „zamora JavaScripta“(eng. JavaScript fatigue) kojim se prikazuje stanje u klijentskom web razvoju gdje se potrebno znanje(znanje o jeziku, okvirima, transpajlerima, upravljačima modula, bundlerima, mehanizmima za predloške...) za prikaz podataka s poslužitelja mijenja i gomila iz dana u dan.

Uzmemo li u obzir drugi pristup, da cijelu aplikaciju za web pišemo u jednom jeziku, potreban nam je način kako JavaScript primijeniti i na poslužitelju te zamijeniti ustaljene jezike. Dolazimo do već spomenutog NodeJS-a, izvršne okoline temeljene na Chromovu V8 JavaScript stroju koji se koristi u samom pregledniku za izvršavanje klijentskog koda, koja nam u ovoj implementaciji omogućuje sve funkcionalnosti klasičnog poslužitelja. Mana tog pristupa je što se velikom broju programera JavaScript ne sviđa kao jezik najčešće zbog nekih neintuitivnih aspekata jezika, iako je dio odbojnosti vjerojatno nastao i zbog nepotpune edukacije o samom jeziku). Neke od stvari koje mu se zamjeraju su: labava tipizacija(donekle zaobiđeno TypeScriptom), što pretvaranje tipa podataka unutar izraza ponekad ne radi očekivano, nego „nagađa“ što želimo(za razliku od nekih drugih jezika koji se koriste u industriji koji jednostavno izbace grešku), čudno ponašanje osnovnih funkcija(kao String, Number,

Array), stanje između ulaska u domet i deklaracije u kojem varijabla nije dostupna (engl. *temporal dead zone*), što se koristi izvan domene onoga za što je inicijalno zamišljen, različito ponašanje u različitim preglednicima te mnogi drugi. [15]

Uz predstavljene probleme s JavaScriptom, poteškoće stvaraju i učestale promjene u klijentskom okruženju koje developer mora pratiti zajedno s promjenama unutar poslužiteljskog jezika. Osim toga, postoji problem nemogućnosti dijeljenja koda poslovne logike i modela između klijenta i poslužitelja. Dodatni problem može predstavljati i potrebno znanje rada u još jednom radnom okruženju te nemogućnost rada s poznatim bibliotekama koje programer koristi na poslužitelju. Kao rješenje za sve navedeno vidim Blazor, novu tehnologiju koja nam omogućuje razvoj klijentske strane aplikacije, unutar industrijski popularnog poslužiteljskog jezika C#, koji će biti predstavljen u nastavku.

3. Blazor okvir za razvoj klijentske strane web aplikacije

Blazor je moderni okvir, otvorenog koda i višeplatformski (eng. cross platform), za izradu jednostraničnih web aplikacija. Ime mu dolazi od riječi preglednik i Razor(eng. Blazor=Browser+Razor). Prednost Blazora, barem za veliku zajednicu postojećih .NET programera, je mogućnost kreiranja cijele web aplikacije bez korištenja JavaScript programskog jezika, već koristeći jezike C# ili F#.

3.1. Potporne tehnologije

Na početku ovog poglavlja ukratko ćemo proći kroz tehnologije bez kojih razvoj aplikacija u Blazoru ne bi bio moguć. Prvi na redu je WebAssembly standard koji omogućuje izvršavanje „ne JavaScript“ koda unutar preglednika, kojeg su unutar Microsofta iskoristili da dovedu Mono u preglednik. Mono je platforma otvorenog koda koja pruža izvršavanje .NET komponenata. Njegova originalna namjena bila je pružiti mogućnost izvršavanja .NET aplikacija na Linuxu, ali s vremenom uz pomoć Xamarin tima iskoristio se i za izvršavanje mobilnih aplikacija na Windows, Android i iOS mobilnim operativnim sustavima. Mono je napisan u C++ što je bitna činjenica kad spoznamo da je C++ na maloj listi jezika za koje postoje kompajleri u WebAssemblyu.

Slijedi ga SignalR biblioteka za komunikaciju u realnom vremenu koja je kod Blazora iskorištena u poslužiteljskom obliku izvođenja aplikacije. Svaka interakcija korisnika s preglednikom, umjesto da se JavaScriptom obradi u pregledniku, SignalR-om se proslijedi na poslužitelj koji onda uz pomoć Blazor stroja izgradi stablo renderiranja (engl. *render tree*). Drvo se zatim serijalizira i opet SignalR-om prosljeđuje u preglednik, gdje ga generički JavaScript kod koristi za promjenu DOM-a.

Na posljetku ukratko prolazimo Razor stroj i Razor sintaksu, koji se već udomaćio među .NET programerima. Razor je bitan za Blazor što vidimo i po samom imenu, a razlog su Razor komponente. Kao i drugi današnji SPA okviri, i Blazor se bazira na komponentama. Komponente su segmenti korisničkog sučelja koje predstavljaju određenu logičku cjelinu, a moguće ih je opetovano koristiti kroz web aplikaciju. Razor komponente su komponente bazirane na postojećem Razor stroju koji postoji unutar ASP.NET MVC, a s dodanim novim značajkama posebno razvijenim prema potrebama za Blazor. Moguće ih je gnijezditi, ponovno koristiti i dijeliti među projektima. Razor komponente bit će pojašnjene u nastavku kao zasebna cjelina.[16]

3.1.1.WebAssembly

WebAssembly je otvoreni standard od strane W3C koji definira binarni format za instrukcije (engl. *binary instruction format*) koji se izvršava na virtualnom stroju (engl. *virtual machine* (VM)) temeljenom na stogu. WebAssembly (Wasm) definira i programski jezik sličan asembleru, ali to mu nije primarni oblik korištenja. Primarni oblik je kao cilj kompilacije za programske jezike C, C++, Rust, što omogućava pisanje programskog koda za preglednike u drugim jezicima osim JavaScriptu. Pored toga omogućuje i izvršavanje s performansama sličnim kao u aplikacijama koje su pisane u jeziku prirodnom za specifični uređaj (engl. *native performance*). Kako W3C navodi, prilikom razvoja želja im je bila da WebAssembly bude brz, efikasan, portabilan, čitljiv ljudima i ima mogućnost debugiranja, uz to brinuli su se o kompatibilnosti s postojećim tehnologijama. Datoteke koje sadržavaju binarni format imaju ekstenziju .wasm, dok datoteke s kodom čitljivim za ljude .wat.

Problem kod JavaScripta su njegove performanse kod naprednijih primjera korištenja kao 3D igre, VR, AR, uređivanje slika. Tako Wasmu nije cilj zamijeniti JavaScript već raditi paralelno s njim. Naime, VM u pregledniku će odsad osim JavaScripta moći koristiti i Wasm. Wasm moduli su napravljeni da se pozivaju iz JavaScript koda (moguć je i obrnuti oblik interakcije da Wasm poziva određene JavaScript funkcije) upravo za dijelove aplikacija koji su resursno intenzivni. Moduli postižu brže izvršavanje na dva načina.

(1) Wasm datoteke se unutar preglednika trebaju samo dekodirati i spremne su za izvršavanje. Dok je kod Javascripta potrebno više vremena da se aplikacija pripremi za pokretanje, jer se JavaScript mora parsirati u apstraktno sintaksno stablo (engl. abstract syntax tree (AST)) i zatim kompajlirati.

(2) Wasm se optimizira unaprijed prilikom kompajliranja, dok se JavaScript prvo mora kompajlirati u strojni kod te optimizirati u trenutku izvršavanja. Osim toga, različiti preglednici i njihovi interni strojevi različito vrše optimizaciju što nam ne garantira jednaku efikasnost našeg koda unutar različitih preglednika.

Uz navedeno, prednost WebAssemblya je i u potrebnom vremenu preuzimanja resursa na klijentskom računalu, jer su .wasm datoteke same po sebi znatno manje nego JavaScript tekstualne datoteke.

Po osnovnoj definiciji osim u pregledniku Wasm je moguće izvršavati u bilo kojem virtualnom stroju, ali kako je zajednica trenutno fokusirana samo na JavaScript, primjere korištenja za sada vidimo još i kod NodeJS-a.[17]

3.1.2. SignalR

SignalR je biblioteka otvorenog koda koja se može koristiti s bilo kojom ASP.NET aplikacijom za web, a olakšava implementaciju funkcionalnosti komunikacije između klijenta i poslužitelja u realnom vremenu. U prijevodu, uzmimo za primjer editor teksta koji koristi više korisnika istovremeno. Kada ga jedan korisnik koristi i unosi tekst, istovremeno drugi korisnik, bez osvježavanja web stranice, na drugom računalu vidi te promjene sinkrono. Najčešće se takav oblik komunikacije odvija u aplikacijama za razgovor, video igrama, in-time aplikacijama, aplikacijama za pisanje po ploči i za slanje notifikacija.[18]

U jednostavnoj HTTP komunikaciji klijent započinje komunikaciju zahtjevom prema poslužitelju na koji dobiva određeni odgovor. Da bi se postigla komunikacija poslužitelja prema pregledniku, do nastanka i globalne implementacije WebSocket protokola u preglednike, koristili su se principi kao što je tzv. polling (u kojem klijent kroz određeni intervali opetovano poziva poslužitelja kako bi dobio novu informaciju) ili long polling (klijent pošalje zahtjev a poslužitelj ako nema što za odgovoriti u tom trenutku ne šalje prazan odgovor već zahtjev drži otvorenim dok se ne dogodi odgovarajuća promjena). Treća su opcija bili događaji poslani od strane poslužitelja (engl. *Server Sent Events* (SSE)) (jednosmjerna komunikacija s poslužitelja prema klijentu)), no zbog efikasnosti i nepotpune implementacije u sve preglednike ti principi nisu savršeni. WebSocket je protokol koji omogućuje full duplex komunikaciju na jednoj TCP vezi te je standardiziran za preglednike po W3C.[19]

SignalR pruža API za stvaranje komunikacije između poslužitelja i klijenta. Biblioteka sadrži kod za poslužitelja i za klijenta, napisan u čistom JavaScriptu, koji pružaju određenu apstrakciju nad implementacijom. Za komunikaciju koristi hubove, koji omogućavaju da klijenti i poslužitelj pozivaju metode jedan na drugome. Dva ugrađena hub protokola su tekstualni protokol koji omogućuje komunikaciju na temelju JSON formata i binarni protokol temeljen na MessagePacku, čije su poruke značajno manje u odnosu na prvo navedeni. SingalR podržava funkciju automatskog uspostavljanja veze između dvije strane u slučaju bilo kakvog nepoželjnog prekida.

Biblioteka sama procjenjuje koji tip komunikacije je optimalan za danu vezu, na temelju podržane tehnologije u pregledniku i poslužitelju. Kao prvi izbor dolazi WebSocket, a ako klijent ili poslužitelj ne može otvoriti takav tip, biblioteka pokušava sa SSE tehnologijom. Zadnja opcija je long polling.[20]

3.1.3. Razor

Razor je razvijen kao stroj za generiranje predložaka (engl. *view engine*) unutar programskog okvira za web aplikacije ASP.NET MVC 3. Nastao je tijekom 2010. godine od

strane Microsofta kao pristupačnija zamjena za dotadašnje .aspx predloške koji čine V (pogled (engl. *view*) dio Model-View-Controller arhitekture. Donosi pojednostavljenu sintaksu za razvoj pogleda, smanjujući količinu koda koju je potrebno napisati.

Razor sintaksa nije novi programski jezik već je kombinacija HTML-a, C# programskog jezika i Razor ključnih riječi, što ju zbog poznate sintakse među .NET developerima čini veoma popularnom. Microsoftovo integrirano razvojno okruženje (engl. *IDE*) Visual Studio (VS) sadrži alat IntelliSense za popunjavanje koda koji u potpunosti podržava Razor sintaksu te još dodatno olakšava programiranje. Za dizajn i interaktivnost u Razor datotekama, kao i u velikoj većini web aplikacija koristimo CSS i JavaScript respektivno.[21]

Prelazak između HTML dijela u C# vrši se korištenjem `@` simbola nakon kojeg slijedi: sam C# izraz unutar oblih zagrada, izraz sa vitičastim zagradama kojima definiramo blok koda (iz kojega se renderira samo HTML dio), kontrole toka(`if`, `else if`, `else`, `switch`) ili petlje(`for`, `foreach`, `while`, `do while`). Osim navedenog, `@` simbol koristimo i za vezanje podataka između modela i markupa.

```
@*C# izraz*@
<p>3+1=@(3+1)</p>
@*kontrola toka*@
@if(a < b){
     <p>a je manje od b</p>
}
else{
     <p>a je veće ili jednako od b</p>
}
@*petlja*@
@for (var i = 0; i < studenti.Length; i++)
{
     var student = studenti[7];
     <p>Name: @student.JMBAG</p>
     <p>Age: @student.Ime</p>
}
```

Kako je Razor kombinacija HTML i C#, unutar tih datoteka možemo koristiti sve blagodati .NET okruženja, pod čime mislim na bogati skup paketa i biblioteka koje programeru olakšavaju svakodnevni rad. Nužno je najčešće na početak datoteke napisati za C# prepoznatljivu direktivu „`using ...`“ s prefiksom `@` da bi simbol stroju označio prelazak u C# dio datoteke.

Unutar Razora možemo kreirati C# metode za lokalnu ili globalnu upotrebu, diljem aplikacije, koju postižemo pisanjem metoda unutar App_code mape. Metode mogu biti u obliku HTML pomagača, Razor funkcija i Razor delegata.[22]

Svi izrazi unutar Razor sintakse su HTML enkodirani, čime se sprječava mogućnost neovlaštenog korištenja (hakiranja) stranice umetanjem skripti (engl. *Cross Site Scripting* (XSS)).

Sa ASP.NET Core 2.0, Microsoft uvodi još jednu primjenu Razora, a to su Razor Pages. Razor Pages više prati Model-View-ViewModel (MVVM) uzorak dizajna za razliku od Model-View-Controller (MVC) te podsjeća na ASP.NET WebForms pristup razvoju. U odnosu na MVC, Razor Pages je više organizacijski grupiran jer se sav kod za određenu stranicu nalazi u odgovarajućim .cshtml i .cshtml.cs datotekama unutar Pages mape. Upravo ekstenzija .cshtml ukazuje da je to datoteka s Razor sintaksom.

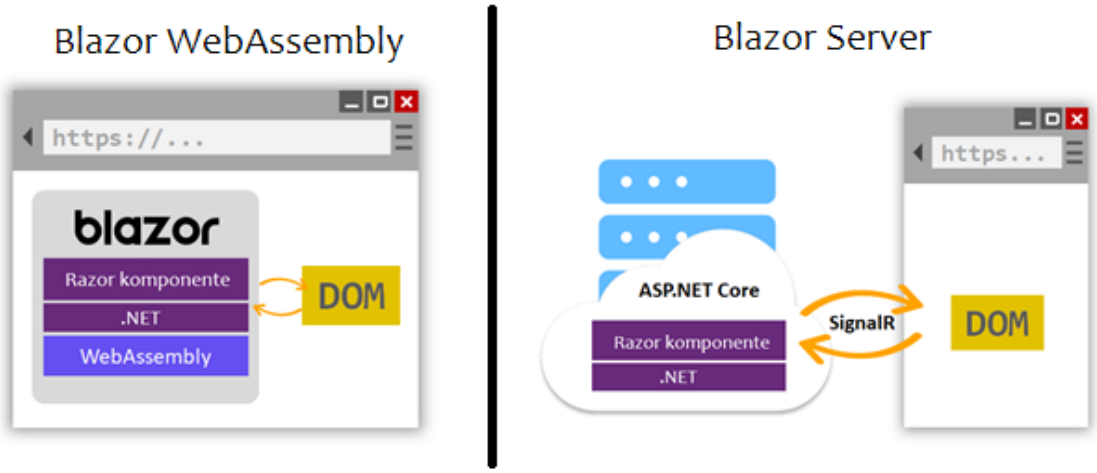
Kao takav, Razor Pages je dobar za jednostavne stranice koje samo prikazuju tekst ili odrađuju jednostavni unos podataka. Uz navedeno, koristi se i za odjeljivanje odgovornosti između dijelova aplikacije. Naime, upravitelji (engl. *controllers*) unutar MVC-a vrlo često znaju narasti u velike datoteke, s akcijama koje poslužuju stranice ali i pomoćnim funkcijama (engl. *helper functions*) i kodom posvećenom poslovnoj logici. Taj okvir je sastavni dio ASP.NET Core MVC aplikacija što znači da zapravo nije zaseban programski okvir, već se može koristiti i u kombinaciji s MVC aplikacijom.

Kao osnovni primjer korištenja Razor Pages imamo defaultni predložak za autentifikaciju kod oblika ASP.NET Core Identity, čak i unutar MVC projekta. Taj oblik autentifikacije možemo proizvoljno uključiti pri kreiranju web aplikacije ili naknadno, da bi uveli već implementiranu autentifikaciju unutar naše web aplikacije.

Unutar .cshtml datoteke vrijede ista pravila Razor sintakse kao i u MVC okviru. Razlika je što na vrh datoteke stavljamo `@page` direktivu koja upućuje da je datoteka zapravo Razor Page te interno čini samu datoteku MVC akcijom, odnosno pri interakciji se ne prolazi kroz upravitelj.[18]

3.2. Oblici izvođenja

Unutar Blazor okvira postoje dva oblika izvođenja (engl. *hosting model*): glavni Blazor WebAssembly (klijentski) te Blazor Server (poslužiteljski). Razlika među njima je u načinu, odnosno lokaciji izvršavanja koda. Kao što možemo vidjeti na slici 2 jedan se izvodi u



Slika 2: Oblici posluživanja (prilagođeno prema [18])

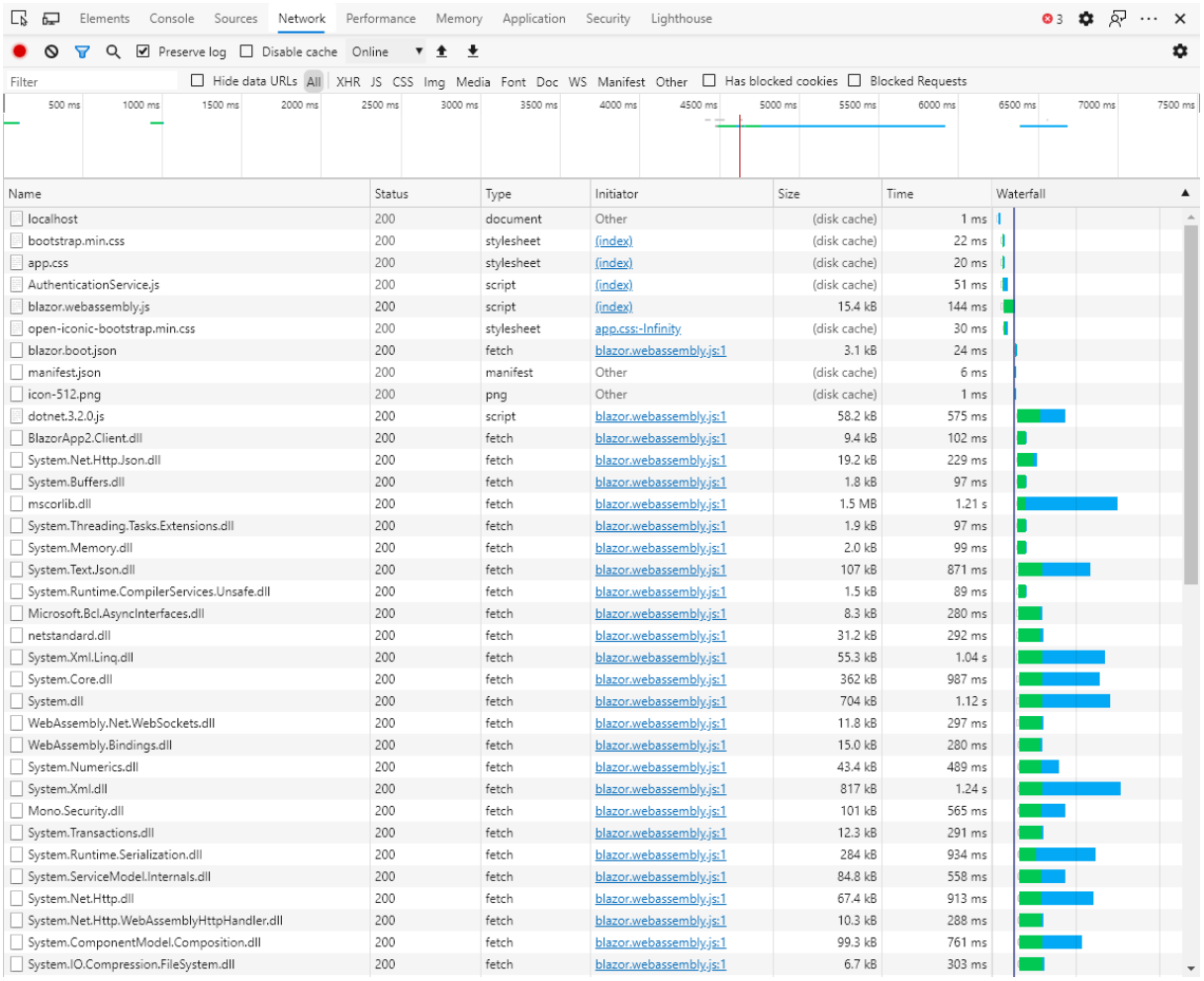
klijentskom pregledniku (Blazor WebAssembly), a drugi (Blazor Server) na samom poslužitelju te se promjene šalju u preglednik pomoću SignalR-a.

3.2.1. Blazor WebAssembly

Kod WebAssembly oblika kod se izvršava unutar preglednika (unutar JavaScript *sandboxa* sa ograničenjima, odnosno zaštitom od zlonamjernog koda na klijentskom uređaju) uz pomoć Mono platforme na UI dretvi web preglednika. Sve C# i Razor datoteke koje čine samu aplikaciju se pri izgradnji (engl *build*) kompajliraju u .NET dinamičke biblioteke sa ekstenzijom .dll. Pri otvaranju aplikacije korisnik unutar svoga preglednika dobiva upravo te datoteke samog projekta i sve preostale datoteke dinamičkih biblioteka koje se koriste kao paketi unutar koda.

Glavni orkestrator u samom pokretanju je blazor.webassembly.js datoteka koja je odgovorna za dohvaćanje, inicijalizaciju aplikacije i pokretanje potrebnih resursa. Ta JavaScript skripta dolazi među prvima u klijentski preglednik. Na Slici 3 u Initiator stupcu vidljivo je da ona poziva ostale datoteke, kao npr. BlazorApp2.Client.dll što je dinamička biblioteka koja se dobije kompajliranjem klijentskog koda istoimenog projekta.[23]

Slijede ga preostale .dll datoteke koje su referencirane unutar projekta. Nakon što klijent dobije sve potrebne resurse, Blazor WebAssembly izvršna okolina pošalje drvo renderiranja JavaScriptu. Uz njegovu pomoć JavaScript interop radi promjene u DOM-u i poziva preglednikove API-je kako bi prikazao stranicu web aplikacije. Svaka sljedeća interakcija klijenta i preglednika rezultira sličnom akcijom. Javascript interop okida određene događaje kao npr. onclick koji se pošalju u C# kod. Blazor zatim izvrši odgovarajuće akcije. Te akcije rezultiraju novom kopijom DOM-a koji se određenim algoritmima uspoređuje sa kopijom DOM-a u memoriji (drvo renderiranja), rezultat je razlika između dvije verzije. Blazor računa razliku s ciljem da se višestruke promjene DOM-a izvrše u jedinstvenoj akciji, koja pruža



Slika 3: Blazor Wasm inicijalno pokretanje (vlastita izrada, dev tool Microsoft Edge)

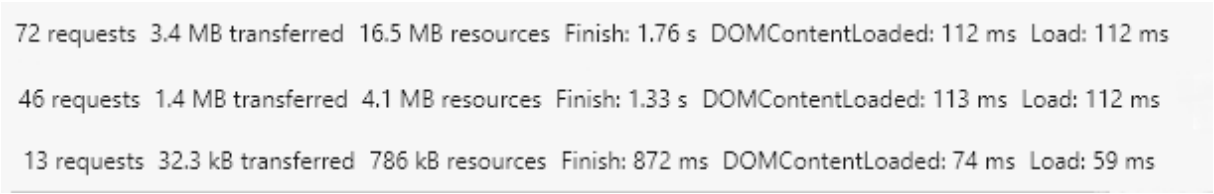
najefikasniji način izmjene. Taj oblik je najefikasniji jer originalni DOM preglednika ne mora znati za međukorake, nego radi izmjene prema finalnoj razlici. Drvo renderiranja je ništa drugo neko skup komponenti (obične C# klase koje nasljeđuju `ComponentBase` klasu). Unutar bazne klase najbitnija je `BuildRenderTree` metoda, koja kao argument dobiva `RenderTreeBuilder` objekt na koji dodaje markup kreiran iz komponente u obliku `RenderTree` objekta.[16]

Aplikacija kao takva je neovisna o nekim dodatnim .NET komponentama ili pluginovima (Silverlight, Adobe Flash). Također, funkcionalnosti koje nisu ovisne o pozivima prema poslužitelju moguće je izvršavati i u offline modu, kao bilo koju drugu JavaScript aplikaciju. Kao i trenutne SPA aplikacije, za dio funkcionalnosti su ipak potrebni novi podaci, odnosno mogućnost interakcije s pozadinskim poslužiteljem (engl. *backend*). Blazor Wasm aplikacija može komunicirati s poslužiteljem preko web API poziva i/ili SignalR. Samim time nije ni limitiran programski jezik poslužitelja, koji ne mora nužno biti pisan unutar C#. Tako i .NET Core CLI (engl. *command line interface*) za stvaranje predložaka sadrži zastavicu hosted koji definira hoće li aplikacija koristiti .NET Core Web API projekt ili ćemo kroz Blazor konzumirati neki drugi API.

```
dotnet new blazorwasm --hosted
```

Taj flag osim tipa API-a definira i oblik posluživanja Wasm (klijentskog) dijela aplikacije. Ako je postavljena zastavica hosted klijentski dio je poslužen od strane ASP.NET Core aplikacije (API). U slučaju da ne koristimo hosted opciju Wasm dio se može poslužiti kroz statički web poslužitelj ili servis.

Kako je već nekoliko puta naglašeno, brzina same aplikacije je među bitnijim faktorima na webu. Pogledom na Sliku 2 možemo vidjeti da je za inicijalno pokretanje potrebno stvarno puno datoteka što se manifestira lošim korisničkim iskustvom kroz dug period čekanja da bi se

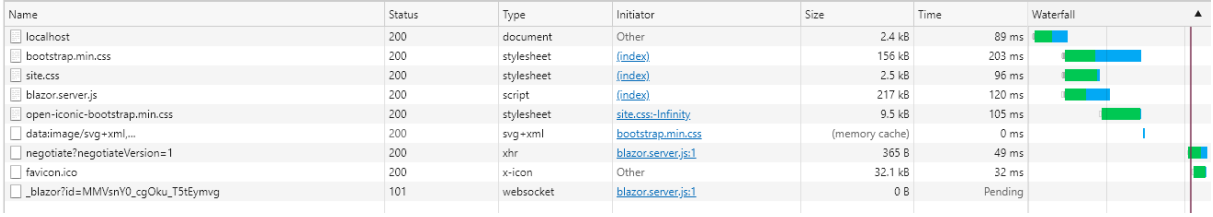


Slika 4: Blazor Wasm veličina aplikacije (vlastita izrada, dev tool Microsoft Edge)

uopće prikazala početna stranica aplikacije. Blazor Wasm optimizira veličinu početnog preuzimanja u tri koraka. Prvo se uz pomoć *Intermediate Language linkera* odstrani sav kod, odnosno reference koje se ne koriste. Zatim se pri komunikaciji s preglednikom datoteke mogu kompresirati, a na kraju se generičke datoteke dinamičnih biblioteka i platforma za izvršavanje .NET keširaju u preglednik.[18] Pregled veličina aplikacije nakon kompajliranja u Debug modu, Release modu(sa IL linkerom) te nakon ponovnog osvježavanja (upotreba keša) može se vidjeti na Slici 3.

3.2.2.Blazor Server

Kod Blazor Server oblika izvođenja aplikacija se izvršava na poslužitelju unutar „klasične“ ASP.NET Core aplikacije. Za razliku od Wasm verzije kod inicijalnog pokretanja aplikacije nema toliko datoteka koje je potrebno preuzeti na klijentsko računalo. Tu su neke uobičajene datoteke za svaku aplikaciju na webu, tipa HTML i CSS datoteke. Ono što Blazor Server aplikaciju razlikuje od ostalih aplikacija za web je JavaScript skripta blazor.server.js. Ta JavaScript datoteka je zadužena za uspostavljanje veze prema poslužitelju pomoću SignalR-a.[18] Na Slici 4 je prikazana Network kartica kod pokretanja aplikacije u kojem, osim datoteka, na posljednjem mjestu vidimo otvorenu Websocket vezu.



Slika 5: Blazor Server inicijalno pokretanje (vlastita izrada, dev tool Microsoft Edge)

Kod ovog oblika drvo renderiranja kreira se na poslužitelju te se zatim serijalizirano uz pomoć SignalR-a šalje klijentskom pregledniku. Slično kao i kod Wasm oblika JavaScriptom deserijalizirano drvo se koristi za promjenu DOM-a u pregledniku. U slučaju interakcije klijenta s aplikacijom događaji se umjesto unutar preglednika obrađuju na poslužitelju. Serijaliziraju se i uz pomoć otvorene veze šalju na poslužitelja koji ih prvo pripremi za obradu. Tamo se na isti efikasan način akcije obrađuju i daju rezultat u obliku razlika stanja DOM-a, prije i poslije. Te se razlike zatim serijaliziraju i šalju klijentu, odnosno u preglednik. Kao i kod inicijalnog pokretanja naposljetku se preko JavaScripta događaju promjene u DOM-u.[16]

Postojeće ASP.NET Core aplikacije koje također koriste Razor sintaksu se na prvu ne razlikuju toliko od Blazor Server aplikacija. No, pogledamo li malo preciznije, vidimo veliku razliku u načinu renderiranja markupa. Kod Razor Pages ili predložaka svaka linija rezultira tekstualnim HTML-om spremnim za prikaz, a poslužitelj odbacuje bilo kakve tragove i/ili stanje o riješenom zahtjevu. Kod Blazor Servera, kao i kod Wasm oblika aplikacija, trenutno aktivna komponenta ima svoju kopiju DOM-a. Ta kopija je u binarnom obliku (objekti u memoriji) i sadržava stanje sa svojstvima i varijablama. Klijent u svom pregledniku može dobiti tekstualni HTML ili binarnu datoteku. Tekstualni HTML se šalje kada programeri aplikacije na samoj aplikaciji implementiraju mogućnost prerenderiranja. Ta opcija se koristi kad želimo ubrzati prikaz nekog statičnog dijela aplikacije. Binarna datoteka se koristi kada poslužitelj klijentu šalje najmanji skup DOM akcija koje su potrebne da bi se promijenio UI na klijentu.[23]

Važno je primijetiti da svaki klijent komunicira s poslužiteljem preko minimalno jedne veze ili kruga (engl. *circuit*), a uz to poslužitelj mora upravljati i stanjem komponente (već navedena svojstva, varijable, servisi, resursi). Također, svaki novi prozor preglednika dovodi do nove veze i novog stanja koje poslužitelj mora pamtiti. Ta veza i podaci o stanju čuvaju se na poslužitelju sve dok korisnik ne ugasi prozor ili se usmjeri na vanjski URL. Osim željenog gašenja aplikacije može doći i do neželjenog, npr. prekidanje internet veze korisnika. U tom slučaju, programeri kroz konfiguraciju mogu postaviti željeno vrijeme čuvanja podataka s ciljem ostvarivanja iskustva neprekidnog korištenja do ponovnog ostvarivanja veze. To sve rezultira brigom o poslužiteljskim resursima koja se kod ovog oblika izvođenja mora uzeti u obzir pri planiranju i samoj izgradnji (pisanju koda).

Kada je riječ o brzini ovdje nemamo problem s inicijalnim pokretanjem kao kod Wasm oblika, no postoji problem kod UI kašnjenja. Naime, kako se svaki događaj šalje na poslužitelja, tamo obrađuje i povratno šalje u korisnikov preglednik, može doći do problema u situacijama kada su korisnik i poslužitelj udaljeni geografski.

Aplikacije koje koriste ovaj oblik izvođenja su najčešće u potpunosti napisane u C# programskom jeziku, odnosno konzumiraju web API koji je unutar istog poslužitelja. No, kao

što je slučaj i s Wasm oblikom, postoji mogućnost korištenja bilo kojeg API-a uz pomoć `IHttpClientFactory`. Osnovni predložak dobijemo sljedećom instrukcijom.[18]

```
dotnet new blazorserver
```

Kao i kod svake aplikacije važno je voditi se konceptima čistog koda i poznatih uzorka dizajna. No kod Blazora, ako ikada želimo ponovno koristiti određene komponente ili čak prebaciti aplikaciju iz jednog oblika u drugi, jako je bitan Separation of Concerns (SoC) princip dizajna. Specifično primjena tog uzorka za dohvaćanje podataka, gdje je na poslužiteljskom obliku Blazora bitno ne upasti u zamku i direktno pozivati bazu podataka, već je potrebno napraviti apstrakcijski sloj servisa koji nam dohvaćaju podatke.

3.2.3.Usporedba oblika izvođenja

Kod izrade aplikacije jedan od bitnijih koraka je odabir tehnologija u kojima će aplikacija biti izrađena. Te tehnologije ovise o mnogim čimbenicima koje je moguće prilagoditi potrebama, ali i nekim čimbenicima koje nije moguće prilagoditi. Da bismo znali koju tehnologiju odabrati, najčešće uspoređujemo prednosti i mane užeg izbora. U nastavku slijedi tablica s usporedbom prednosti i mana određenog oblika izvođenja složena uz pomoć Microsoftove dokumentacije [18], jedne od opširnijih trenutno dostupnih knjiga o Blazoru [16], video konferencije o Blazoru [24] i mog osobnog iskustva s razvojem u Blazoru.

Tablica 1: Usporedba Blazor oblika izvođenja

	Blazor WebAssembly (BWA)	Blazor Server (BS)
Zavisnost	aplikacija za sebe(nema ovisnost o poslužitelju), radi i offline	ovisi o poslužitelju toliko da ne može raditi bez njega
Incijalno pokretanje	velika količina podataka	mali broj datoteka, potrebno samo uspostaviti vezu
Resursi poslužitelja	posao većinom obavlja preglednik	poslužitelj čuva puno toga u memoriji
Serverless	aplikaciju je moguće poslužiti na CDN	nije moguće poslužiti na CDN
Preglednik	ne podržavaju svi preglednici WebAssembly	nema potrebe za WebAssembly-jem, neki stariji preglednici ne podržavaju promise u JS (moguće riješiti uz dodatne pakete)

Debugiranje	novija tehnologija, postoje problemi	skoro pa klasična ASP.NET aplikacija
UX	brza, jer interakcija ne mora nužno komunicirati sa poslužiteljem	spora, jer svaka interakcija ide preko mreže
Skalabilnost	kao i kod klasičnih ASP.NET aplikacija	slaba, jer mora čuvati stanje svakog korisnika
PWA	podržava mogućnost pretvaranja aplikacije u PWA	ne podržava

3.3. Komponente

Komponente su klase koje definiraju markup za prikaz i obrađuju korisnikove događaje. One čine jednu logičku cjelinu, a definiraju dijelove UI-a kao što su stranica, dialog ili forma.

Uobičajeni način implementacije je datoteka sa .razor ekstenzijom, koja u sebi sadrži kod (kombinacija HTML i C#) napisan Razor sintaksom. Drugi mogući oblik je pomoću djelomičnih datoteka u kojem se unutar .razor datoteke nalazi samo markup, a C# kod iz `@code` taga izvučemo u zasebnu istoimenu datoteku ekstenzije .razor.cs. Treći oblik je pomoću baznih klasa gdje .razor datoteka pomoću `@inherits` taga nasljeđuje baznu klasu u kojoj je implementiran kod. Komponente koje čine web stranicu, s `@page` direktivom, najčešće se nalaze unutar Pages mape, dok one koje specificiraju manji segment stavljamo ili u Shared mapu, ili neku zasebnu, ili dolaze iz vanjskih paketa.[18]

3.3.1. Vezanje podataka

U slučaju kad u aplikaciji imamo neku formu u koju klijenti unose podatke da bi došli do svih unesenih podataka, jedan od pristupa je iteracija kroz članove objekta modela i dohvaćanje vrijednosti iz odgovarajućih elemenata na formi. Taj pristup je mukotrpan i podložan greškama, no okviri kao Angular i React rješavaju taj problem uz mogućnost vezanja podataka (engl. *data binding*).

```
<h1 style="color:@headingColor">@headingText </h1>
@code {
     private string headingColor = "red";
     private string headingText = "Ovo je element h1";
}
```

Unutar markupa pomoću Razora uključujemo mogućnost dinamičkog renderiranja. U gornjem primjeru C# varijable `headingColor` i `headingText` su članovi klase komponente definirane u `@code` bloku. One se pri kompajliranju zajedno s markupom pretvaraju u klasu istog imena kao i datoteka u kojoj se nalaze. Prilikom renderiranja ovaj `h1` element će sadržavati tekst iz varijable `headingText` koji će biti boje definirane unutar `headingColor` varijable. Ovo je primjer jednosmjernog vezanja od komponente prema DOM-u.

```
<input @bind="inputText" />
<p> Trenutno: @inputText </p>
@code {
     private string inputText = "Probni tekst";
}
```

U ovom primjeru je pomoću HTML atributa `@bind` specifičnog za Razor komponentu ostvareno dvosmjerno vezivanje. Naime, pri početnom renderiranju se u `input` elementu prikaže vrijednost „Probni tekst“, a promjenom teksta unutar tog elementa će se promijeniti vrijednost `inputText` varijable unutar objekta komponente. No to se ovdje neće dogoditi automatski kako upisujemo vrijednost jer se komponente rerenderiraju tek na određeni događaj. U ovom slučaju događaj koji se čeka je `onchange` koji se okida kada element izgubi fokus. Također, u svakom trenutku programer može komponentu prisiliti na rerenderiranje uz poziv metode `StateHasChanged` iz `ComponentBase` klase. Osim samog vezivanja, ključna riječ `bind` nam pruža i osnovni oblik validacije, što znači da u slučaju vezivanja varijable tipa `int`, element ne bi dozvolio unos drugačijeg tipa, nego bi vratio vrijednost na zadnju koja je prošla validaciju.

Kako nam ponekad želja nije da se vezivanje događa na `onchange`, moguće je prilagoditi događaj na koji će se okidati. Na primjer, responzivnija promjena teksta može se postići na sljedeći način.

```
<input @bind="inputText" @bind:event="oninput" />
<p> Trenutno: @inputText </p>
@code {
     private string inputText = "Probni tekst";
}
```

U ovom primjeru će na svaki pritisak tipkovnice doći do vezivanja i posljedično rerenderiranja same komponente te ćemo promjene u `p` elementu vidjeti nakon svakog znaka.

Zadani HTML atribut za koji se radi vezivanje je `value`, no postoje slučajevi kada to ne želimo. U slučaju kada želimo npr. vezivati stil elementa koristimo `@bind-style`. Općenito koristimo sljedeću sintaksu `@bind-{atribut}` i `@bind-{atribut}:event`.

Za neke varijable osim samog vezivanja bitan nam je njihov format ispisa, ali i upisa. Najčešći primjer takvog slučaja je datum. Pri korištenju `input` elementa za datum, možemo mu pridjeliti `@bind:format` atribut kojem bismo proslijedili oblik koji želimo, na klasičan .NET način.[16]

3.3.2. Obrada događaja

Razor nam unutar svojih komponenti omogućava i obradu događaja (engl. *event handling*). Način kojim to radimo je vrlo intuitivan: unutar HTML elementa na koji želimo vezati događaj definiramo atribut imena `@on{događaj}` čija vrijednost je ime metode. Slijedi primjer iz originalnog predloška koji dobijemo pri kreiranju Blazor projekta unutar Visual Studia.

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
      currentCount++;
    }
}
```

Naime, klikom miša okida se metoda `IncrementCount`, koja je zapravo metoda koja obrađuje događaj (engl. *event handler*). U Blazoru event handleri ne prate u potpunosti klasični uzorak događaja iz .NET-a te im se po želji mogu deklarirati parametri koji nasljeđuju `EventArgs`, odnosno u ovom slučaju `MouseEventArgs`.[18]

Za jednostavnije metode kao prethodno navedena, Razor podržava i lambda sintaksu u kojoj ne moramo definirati samu metodu. Slijedi kraći primjer.

```
<button @onclick="@(()=>currentCount++)">Click me</button>
```

U slučaju da ne želimo uobičajeno ponašanje na određeni događaj, postoji mogućnost njegovog isključivanja, i naknadno pisanja vlastite implementacije, pomoću atributa `@on{događaj}:preventDefault="true"`. Još jedna opcija koju je moguće isključiti je propagiranje događaja - kada imamo događaj koji se dogodi unutar komponente on se propagira prema vanjskoj komponenti. Na primjer, na micanje mišem unutar komponente djeteta okinut će se `onmousemove` i na vanjskoj komponenti, a gasimo ga pomoću atributa `@on{događaj}:stopPropagation`. [16]

3.3.3. Gniježđenje

Gniježđenje je uzorak u kojem jedna komponenta unutar svog markupa sadržava drugu komponentnu kao bilo koji drugi HTML element. Sintaksa je `<{imeKomponente} />`. Komponentu koja sadržava drugu komponentu nazivamo roditeljskom, a sadržanu zovemo komponenta dijete. Kada dođemo do takve strukture komponenti vrlo brzo nastane želja za komunikacijom između roditelja i djeteta u smislu prosljeđivanja vrijednosti varijabli ili okidanja određenih metoda. Ta komunikacija vrlo često mora biti obostrana, od roditelja prema djetetu, ali i obrnuto.

Komponente mogu imati parametre koji su definirani kao `public` svojstva na samoj komponenti i označeni `[Parameter]` atributom. Na ovaj način možemo proslijediti vrijednost od roditelja prema djetetu.

```
@*KomponentaDijete.razor*@
<h1>@Naslov</h1>
@code {
    [Parameter]
    public string Naslov { get; set; }
}
@*KomponentaRoditelj.razor*@
<KomponentaDijete Naslov="Naslov iz roditelja" />
```

Komponenti djetetu možemo proslijediti i sadržaj koji predstavlja HTML, a renderirati će se na zadanom mjestu unutar komponente.

```
@*KomponentaDijete.razor*@
<div>@Sadrzaj</div>
@code {
    [Parameter]
    public RenderFragment Sadrzaj { get; set; }
}
@*KomponentaRoditelj.razor*@
<KomponentaDijete>
    <p>Ovdje je upisan sadržaj</p>
</KomponentaDijete>
```

U slučaju kada komponenta dijete renderira markup čiji je HTML element opisan atributa nije efikasno popisati sve moguće atribute, tada koristimo spljoštavanje atributa (engl. *splatting*) (pojava kada koristimo rječnik da proslijedimo sve željene parametre). Postavljanje `CaptureUnmatchedValues=true` nam omogućuje da se bilo koji parametar koji proslijedimo djetetu, a imenom nije uparen za postojeće svojstvo, veže na ovaj rječnik.[18]

```
@*KomponentaDijete.razor*@
<input @attributes="AtributiZaInput" size="50"/>
@code {
   [Parameter(CaptureUnmatchedValues = true)]
   public Dictionary<string, object> AtributiZaInput { get; set; }
}
@*KomponentaRoditelj.razor*@
<KomponentaDijete placeholder="Input tekst" required="required"
size="20" />
```

Ponekad postoji želja da se neka vrijednost proslijedi kroz više slojeva komponenti, a nije najbolja praksa prosljeđivati kroz slojeve uzastopno. U tom slučaju koristimo `CascadingValue` koji nam omogućuje da unutar nasljednika koristimo tu vrijednost koja se veže preko njezinog tipa. U prijevodu, možemo proslijediti samo jedan parametar tipa `int` unutar jednog stabla nasljedstva, osim ako uz vrijednost ne proslijedimo i ime koje zadamo unutar `Name` atributa.

```
@*KomponentaRoditelj.razor*@
<CascadingValue Value="broj">
@*komponente nasljednici*@
</CascadingValue>
@code {
   private int broj = 9;
}
@*neki od nasljednika*@
@code {
   [CascadingParameter]
   public int broj {get; set;}
}
```

U slučaju obrnute komunikacije koristimo funkcije povratnog poziva (engl. *callback*) koje moraju biti istog imena kao i ime parametra komponente sa sufiksom `Changed`. Definirani callback pozivamo s metodom Invoke te mu proslijedimo parametre, ako smo ih zadali.

```
@*KomponentaDijete.razor*@
<button @onclick="PovecajVrijednost"/>
@code {
   [Parameter]
   public int Vrijednost { get; set; }
   [Parameter]
   public EventCallback<int> VrijednostChanged { get; set; }
   private Task PovecajVrijednost(){
     Vrijednost++;
```

```
        return VrijednostChanged.InvokeAsync(Vrijednost);
      }
   }
   @*KomponentaRoditelj.razor*@
   <KomponentaDijete @bind-Vrijednost="vrijednost" />
   <p>Vrijednost: @vrijednost</p>
   @code {
      private int vrijednost = 0;
   }
```

Taj callback je zapravo događaj sam po sebi te bi istu stvar ostvarili da smo koristili sljedeću sintaksu.

```
<KomponentaDijete @bind-Vrijednost = "vrijednost" @bind-Vrijednost:event = "VrijednostChanged">
```

Osim vezanja svojstva, možemo im pridružiti neki svoj event handler i odraditi neku drugu operaciju, jer je vezanje uz `bind` upravo to - obrada događaja preko implementiranog event handlera.[16]

3.3.4. Ugrađene komponente

Neki trenutno dostupni klijentski okviri incijalno dolaze s malo ugrađenih značajki. Do tih značajki potrebnih u svakoj aplikaciji, kao što su forme za unos podataka ili usmjeravanje u SPA aplikacijama, dolazimo dodavanjem biblioteka u ovom ili onom oblik. Blazor sam po sebi dolazi s puno ugrađenih funkcionalnosti te rijetko treba posezati za drugim bibliotekama.

Osnovna komponenta u kojoj živi cijela aplikacija zove se `App` te predstavlja `app` DOM element. Ona u sebi sadrži `Router` komponentu koja pruža usmjeravanje, odnosno renderira određenu straničnu komponentu u ovisnosti o putanji u pregledniku. Također nam omogućava da u straničnim komponentama definiramo parametre kroz putanju unutar `@page` direktive. `Router` unutar sebe sadrži `Found` komponentu koja uz pomoć `RouteView` komponente inicijalizira traženu stranicu, a definira i zadani predložak cijele aplikacije. Dodatno drugo izravno dijete je `NotFound` komponenta u kojoj možemo definirati stranicu koja će se korisniku prikazati ako putanja ne vodi ni do jedne komponente. Za navigiranje između komponenti koristimo `NavLink` koji ima ugrađenu mogućnost vizualne aktivacije pomoću CSS-a kada smo na odgovarajućoj putanji.

Za forme postoji `EditForm` komponenta unutar koje onda koristimo dorađene komponente. Te dorađene komponente su u osnovi HTML elementi s dodatnim značajkama kao npr. obrada vrijednosti koje nije moguće pretvoriti u odgovarajući tip – slova unutar `InputNumber`. Komponenta za forme nam omogućuje i validaciju u ovisnosti o

`DataAnnotationsima` na modelu za koji je komponenta vezana. Aktiviranje validacije je relativno jednostavno uključivanjem `DataAnnotationsValidator` komponente unutar `EditForm` komponente. Postoji i komponenta `ValidationSummary` koja nam na jednom mjestu slijedno prikaže sve postojeće greške unutar forme. `EditForma` nam u ovisnosti o validaciji pruža i razne događaje na koje možemo reagirati, kao što su `OnSubmit`, `OnValidSubmit` i `OnInvalidSubmit`.[18]

Za sigurnost aplikacije Blazor nam omogućuje nekoliko komponenti, a one se sve temelje na `AuthenticationStateProvider` servisu. Servis definira kada je korisnik autentificiran te omogućuje pristup korisnikovim podacima (uloge, claimovi, itd.), koji se onda koriste se za autorizaciju. `AuthorizeView` je vanjska komponenta koju stavljamo u bilo koju aplikativnu komponentu, a ima dvije komponente koje su joj djeca `Authorized` i `NotAuthorized`. Te dvije komponente omogućuju odnosno onemogućuju UI pristup određenom sadržaju tj. kodu. Osnovni način ponašanje te komponente temelji se na činjenici da ako je korisnik autentificiran, on je i autoriziran, a ako želimo ciljanu autorizaciju koristimo atribute `Roles` i `Policy` na komponenti. `AuthorizeView` radi autorizaciju samo na sadržaju, ne i na usmjeravanju, za autorizaciju na usmjeravanju na straničnim komponentama koristimo `Authorize` atribut (`@atribute [Authorize]`) koji također ima atribute `Roles` i `Policy`. Ako želimo složiti osnovno ponašanje unutar neautoriziranih situacija na razini cijele aplikacije, na `Router` komponenti umjesto `RouteView` koristimo `AuthorizeRouteView` zajedno s `Authorized`, `NotAuthorized` i `Authorizing` komponentama djecom.[23]

3.3.5. Metode životnog ciklusa

Kao i komponente u nekim drugim klijentskim okvirima tako, i Razor komponente unutar Blazora imaju metode kroz njihov životni vijek unutar kojih se programeri mogu uključiti i odraditi dodatne operacije. Postoje četiri grupe metoda u ovisnosti o koraku u životnom ciklusu, a dolaze od `ComponentBase` klase.

U prvoj grupi nalazi se metoda `SetParametersAsync` koja postavlja članove komponente u ovisnosti o argumentima koje proslijedi roditeljska komponenta. U pitanju su članovi sa `[Parameter]` i `[CascadingParameter]` atributima.

Zatim slijede metode koje se okidaju pri inicijalizaciji komponente `OnInitialized` za sinkronu i `OnInitializedAsync` za asinkronu operaciju. U asinkronu metodu najčešće stavljamo pozive prema izvoru podataka za početno postavljanje vrijednosti te je bitno uzeti u obzir mogućnost da izvor ne odgovori na vrijeme. U skladu s tim, potrebno je napisati logiku koja će pokazivati komponentu u ovisnosti o tome ima li sve podatke ili ne (tipa tekst „Učitava se...“ ili neki *spinner*). U slučaju da je odabran Blazor Server oblik izvođenja s mogućnosti

prerenderiranja, ove metode okidaju se dvaput inicijalno i drugi put kad preglednik uspostavi vezu.

Slijedi ih grupa metoda sa `OnParametersSet` i `OnParametersSetAsync` koje se okidaju nakon inicijalizacije ili nakon što se roditeljska komponenta rerenderira i proslijedi nove vrijednosti parametara. Na posljetku imamo grupu s `OnAfterRender` i `OnAfterRenderAsync` metodama. Najbitniji primjer korištenja tih metoda je kad na Blazor Serveru želimo koristiti JavaScript direktno, jer su to jedine metode koje nam garantiraju da imamo pristup korisnikovom pregledniku.[18]

3.4. Interoperabilnost s JavaScriptom

Blazor je nova tehnologija i Microsoft nije implementirao sve API-je preglednika u njemu. Uzmemo li u obzir da su kroz dugo vrijeme postojanja JavaScripta implementirani brojni efektivni paketi, Microsoft je odlučio pružiti mogućnost programeru da iz .NET dijela aplikacije pozove JavaScript metode, ali i obrnuto. Ta mogućnost zove se interoperabilnost s JavaScriptom (engl. *JavaScript interoperability* (JS interop)).

Da bi omogućili pozivanje JavaScript metode potrebno je u komponentu iz dependency injection (DI) spremnika unijeti servis koji implementira `IJSRuntime`. On ima definirane metode `InvokeVoidAsync` i `InvokeAsync<TValue>` koje kao parametre imaju naziv JavaScript metode i listu parametara same JavaScript metode. Također moramo definirati povratni tip, koji se kao i parametri, mora moći serijalizirati JSON-om. Sljedeći primjer će pritiskom na gumb iz Blazora pozvati JS metodu koja će ispisati poruku u konzolu na pregledniku.

```
//unutar primjerJSInterop.js
window.ispisiPoruku=(poruka) => {
   console.log(poruka)
}


@*unutar Pages/_Host.cshtml za poslužiteljski oblik ili
wwwroot/index.html za Wasm oblik*@
<script src="primjerJSInterop.js"></script>


@*unutar komponente koja poziva JS*@
@inject IJSRuntime JSRuntime;
<button @onclick="PozoviJS">Ispisi poruku</button>
@code {
   private async Task PozoviJS(){
      await JSRuntime.InvokeVoidAsync("ispisiPoruku", "Poruka iz .net-a
   okruzenja");
```

```
        }
    }
```

Obrnuti slučaj je također moguć. Unutar JavaScripta pozovemo `DotNet.invokeMethodAsync` s parametrima, redom: ime dinamičke biblioteke, ime metode koja je unutar komponente označena s `[JSInvokable]` te parametrima.[18]

3.5. Dodatne biblioteke

Iako je Blazor tek nedavno službeno pušten u upotrebu oko njega se već skupila poveća zajednica programera koji aktivno programiraju biblioteke za što lakši i ugodniji razvoj Blazor aplikacija. Tako postoji velika količina paketa raznih funkcionalnosti. Te funkcionalnosti uključuju npr. autorizaciju i autentifikaciju, čiji paketi implementiraju različite oblike `AuthenticationStateProvidera` ovisno o potrebi. Servisi s kojima su paketi napravili integraciju su IdentityServer, Azure Active Directory i Auth0. Sljedeća bitna funkcionalnost pokrivena paketima je za korištenje localStoragea i sessionStoragea. Kao i kod SPA aplikacija pisanih u drugim okvirima javlja se potreba za održavanjem stanja kroz cijelu aplikaciju. Reactove biblioteke za održavanja stanja, Redux i Flux, već su prepisane i primjenjive na Blazor aplikacijama. Postoje i omotači (engl. *wrapperi*) koji olakšavaju pristup nekim API-jima preglednika kao za geolokaciju ili rad s datotekama.

Osim paketa otvorenog koda razvijenih od strane zajednice, na tržište ulaze i velike organizacije koje su i do sada bile Microsoftovi suradnici u pojedinim okvirima. Telerik koji ima svoje biblioteke za UI komponente za WinForm, WPF, UWP, MVC, ASP.NET Core, ali i za Angular, React i Vue, napravio je biblioteku i za Blazor. Slično je i s DevExpressom koji je okrenut samo na Microsoft tržište, a također proizvodi biblioteke s UI komponentama.

Unit testove kao i u drugim .NET Core aplikacijama možemo provesti uz pomoć xUnita ili NUnit ili sličnih paketa. No razvijeni su paketi specifično za Razor komponente kao Razor.Components.Testing.Library i bUnit koji nam omogućuju testiranje uz pomoć uspoređivanja dobivenog HTML-a, testiraju event handlere, kaskadne parametre i sve glavne funkcionalnosti samog Blazora.[24]

4. Aplikacija

U ovom poglavlju bit će opisana aplikacija koja je napravljena u Blazoru s ciljem prezentiranja koncepata razvoja navedenih u prijašnjim poglavljima. Aplikacija služi kao društvena aplikacija za web na kojoj korisnici imaju priliku za upoznavanje novih ljudi službenog ime UpoznajMe (engl. *MeetMe*). Kao ideja za aplikaciju te njezin kostur uzet je praktični vodič (engl. *tutorial*) „Razvoj aplikacije sa ASPNET Core i Angularom iz početka“ (engl. *Build an app with ASPNET Core and Angular from scratch*) sa stranice *Udemy*.[25]

4.1. Opis aplikacije

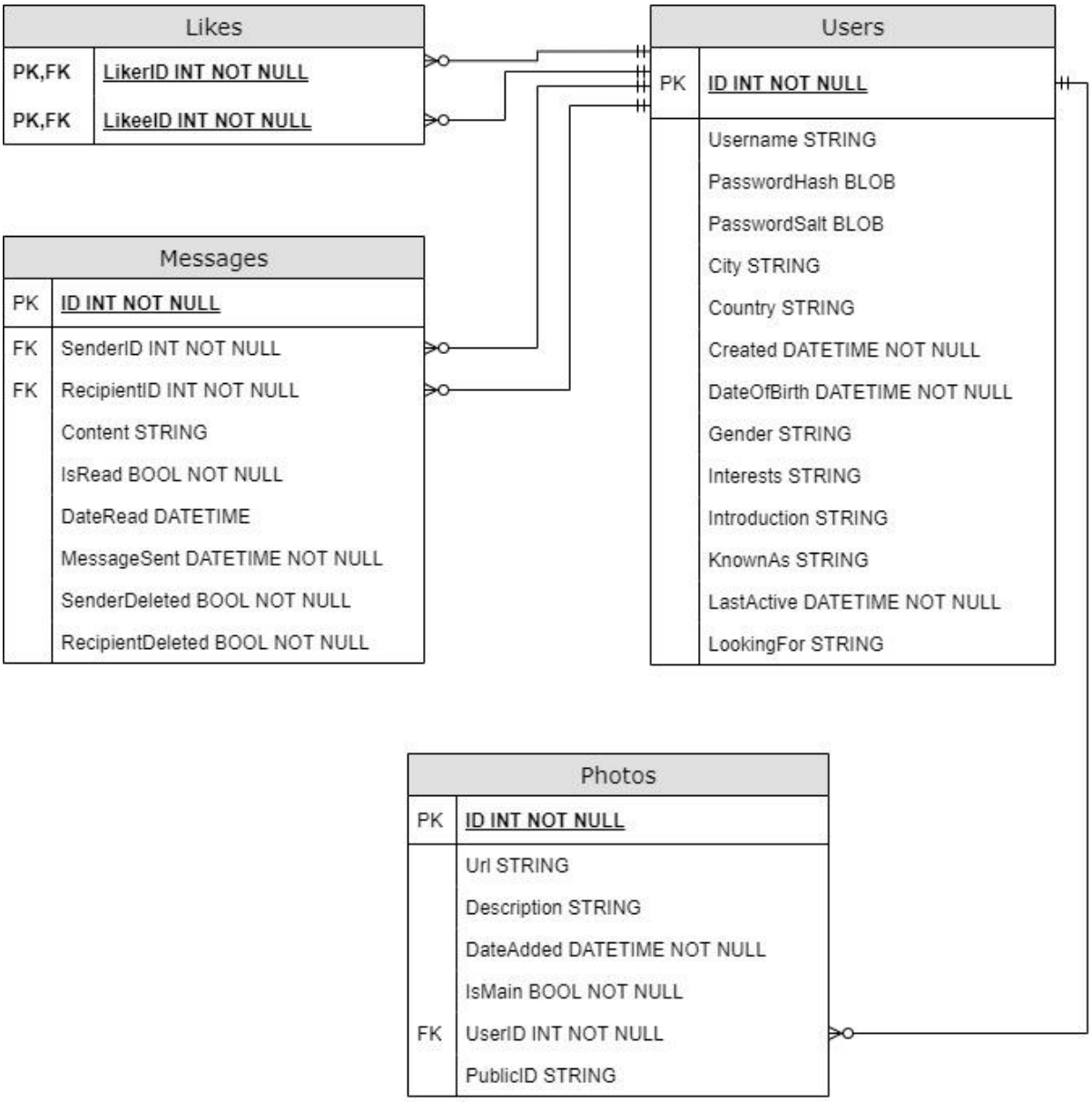
Princip aplikacije je jednostavan, na osnovu određenog filtera kao što su raspon godina, spol i „zadnje aktivan“ ili „najnovije registrirani član“, svaki korisnik dobije popis registriranih korisnika. Preko tog popisa korisnici dolaze do osoba koje im se čine zanimljive za druženje, a ako su im i privlačne postoji mogućnost „sviđanja“. Svaki korisnik kroz popis ima pristup podacima koje drugi korisnici ispunjavaju kako bi dali neke osnovne informacije o sebi. Neke od informacija su kratki opis u kojem korisnici ukratko opišu svoju osobnost, što točno traže na stranici, koji su im interesi (kako bi zainteresirani korisnici našli zajedničke točke) te lokaciju na kojoj se nalaze. Osim tekstualnih podataka na ovakvom tipu aplikacije bitan je i vizualni prikaz, tako da svaki korisnik ima svoju galeriju sa svojim slikama. Sa samog popisa korisnici također mogu pristupiti slanju poruke nekoj od osoba bez potrebnog prvotnog označivanja osobe kao da im se sviđa.

Osim liste sa korisnicima po filteru, postoji i pregled korisnika koje smo mi označili da nam se sviđaju i korisnika koji su nas označili zbog lakše navigacije do korisnika s kojima je komunikacija vjerojatnija. Same poruke podijeljene su u tri grupacije, slično kao i kod e-mail servisa. Nepročitane poruke, zaprimljene ali pročitane poruke, i odaslane poruke. U konačnici tu je mogućnost uređivanja svoga profila s već navedenim stavkama te galerija u koju dodajemo slike. Unutar galerije slika mora postojati barem jedna slika koja je odabrana kao glavna, odnosno profilna.

Kako je ovo aplikacija za web, novi korisnici ulaze u sustav procesom registracije. U samom tom procesu već daju osnovne osobne podatke (korisničko ime, datum rođenja, lokacija, lozinka) te su pri prvom loginu u mogućnosti upoznati novu osobu.

4.2. Baza podataka

Aplikacija koristi relativno jednostavan model podataka. Sama je aplikacija pozadinskog sustava razvijena je u .NET Core Web API projektu uz pomoć Entity Framework Core ORM-a (engl. *object relational mapping*). Korišten je *code first approach*. U prijevodu to



Slika 6: ERA dijagram aplikacije (vlastita izrada)

znači da je objektni model aplikacije složen prvotno u C# kodu preko klasa uz definiranje dodatnih pravila mapiranja kao stranih ključeva i n-tost veza unutar relacijske baze podataka. Svaka klasa sadrži i veze prema vezanim klasama tj. moguće je kroz kod pristupiti povezanim entitetim kroz navigacijska svojstva. Taj model je uz pomoć EF Corea direktno pretvoren u

tablice baze podataka. Osim objektnog modela, EF za potrebe verzioniranja migracija radi posebnu tablicu __EFMigrationsHistory koja ni na koji način nije vezana s tablicama modela.

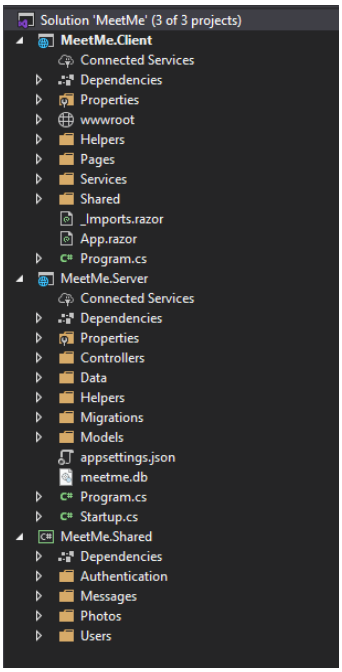
Sam sustav za upravljanje relacijskim bazama podataka (engl. *Relational database management system* (RDBMS)) u ovoj aplikaciji je SQLite najviše zbog jednostavne inicijalizacije na računalu programera. U slučaju isporuke aplikacije uz minimalne prepravke unutar Web API projekta moguće je uključiti novi pružatelj baze podataka (engl. *database provider*) te prebaciti aplikaciju na drugi tip RDBMS-a npr. SQL Server.

Kako se cijeli sustav temelji na korisnicima, tako je i klasa odnosno tablica `Users` najbitniji dio. Tablica sadrži polja kojima će se korisnik predstaviti drugim korisnicima na stranici, npr. od kuda dolaze, koliko su stari ili koji su njihovi interesi. Osim podataka za profil, tablica sadrži i polja koja služe sustavu za prepoznavanje korisnika: `Username`, `PasswordHash` i `PasswordSalt` te datumsko polje koje bilježi zadnju interakciju sa sustavom. Tablica `Photos` sadrži podatke o slikama pojedinog korisnika te je stranim ključem `UserID` vezana na tablicu korisnika. Ostale vrijednosti u tablici su `PublicID` koji predstavlja identifikaciju u vanjskom sustavu Cloudinary te `Url` putanja do same slike. Tu su još tekstualno polje u kojem se nalazi opis slike, datum kada je slika dodana te bool polje koje govori jel li slika glavna odnosno profilna. Unutar tablice `Likes` posprema se zapis kada jedan korisnik okine akciju sviđanja nad profilom druge osobe. `LikerID` predstavlja `ID` korisnika kojemu se sviđa neki drugi korisnik odnosno `LikeeID`. Oba polja čine primarni ključ za tablicu `Likes`, a ujedno su i strani ključ prema `Users` tablici. Na posljetku imamo tablicu `Messages` u koju spremamo poruke koje korisnici izmjenjuju. Ta tablica, uz samu poruku, sadrži i osnovne metapodatke komunikacije kao što su: pošiljatelj, primatelj, vrijeme slanja, vrijeme čitanja te zastavice je li poruka pročitana, je li obrisana od strane pošiljatelja ili od strane primatelja.

4.3. Rješenje (engl. *solution*) aplikacije

Samo rješenje aplikacije početno je strukturirano uz .NET Core CLI naredbu `dotnet new blazorwasm --hosted`. Ta naredba kreira rješenje koje se sastoji od tri projekta: projekta za klijentsku stranu, projekta za poslužiteljsku stranu aplikacije i projekta unutar kojeg se nalazi kod kojeg zajedno koriste, odnosno dijele, klijentski i poslužiteljski projekt.

Na slici 6 vidimo strukturu UpoznajMe aplikacije iz Solution Explorera unutar Visual Studio IDE. Projekt pod nazivom MeetMe.Client sadrži kod pisan u Blazor okviru i predstavlja klijentski dio aplikacije. Projekt za poslužiteljsku stranu aplikacije ima sufiks Server te ima strukturu Web API projekta. Na posljetku imamo projekt sa sufiksom dijeljeni (engl. *Shared*) unutar kojeg se nalaze modeli, klase koje koristi i poslužiteljski i klijentski projekt.



Slika 7: Struktura aplikacije (vlastita izrada)

4.3.1. Poslužiteljska strana aplikacije

Pozadinski sustav je klasičan .NET Core Web API projekt. Sastoji se od četiri kontrolera: `Auth`, `Users`, `Photos` i `Messages`. `Auth` kontroler je zadužen za registraciju novih korisnika te login i logout funkcionalnosti. Autentifikacija i autorizacija je riješena uz jednostavnu implementaciju JSON Web tokena (JWT). Token se kreira pri loginu u sustav te se šalje klijentu. Klijent, odnosno klijentska aplikacija pri svakom sljedećem pozivu prema pozadinskom sustavu koristi taj token za autorizaciju i dohvaćanje resursa potrebnih za rad.

`Users` kontroler služi za dohvaćanje podataka o specifičnom korisniku i/ili više korisnika zajedno. Osim metoda za dohvaćanje sadrži i metodu za promjenu profila ulogiranog korisnika te metodu koja se koristi pri akciji sviđanja drugog korisnika. `Messages` kontroler služi za komunikaciju. Sadrži metode za dohvaćanje, kreiranje i brisanje poruka. Tu je i metoda koja se okida kada korisnik pročita poruku, a sprema podatak o vremenu kada je poruka pročitana. Preostao je Photos kontroler koji služi za rad sa slikama. Sadrži metode za dohvaćanje, dodavanje i brisanje slika te metodu koja neku sliku čini glavnom (profilnom). Svaka metoda unutar navedenih kontrolera sadrži provjere smije li korisnik sa danim JWT-om uopće obaviti tu akciju - ako nema potrebna prava API mu vraća odgovor sa HTTP status kodom 401 koji predstavlja neuspješnu autorizaciju (engl. *Unauthorized*). U tablici 2 nalazi se popis svih krajnjih točaka (engl. *endpoint*) unutar aplikacije.

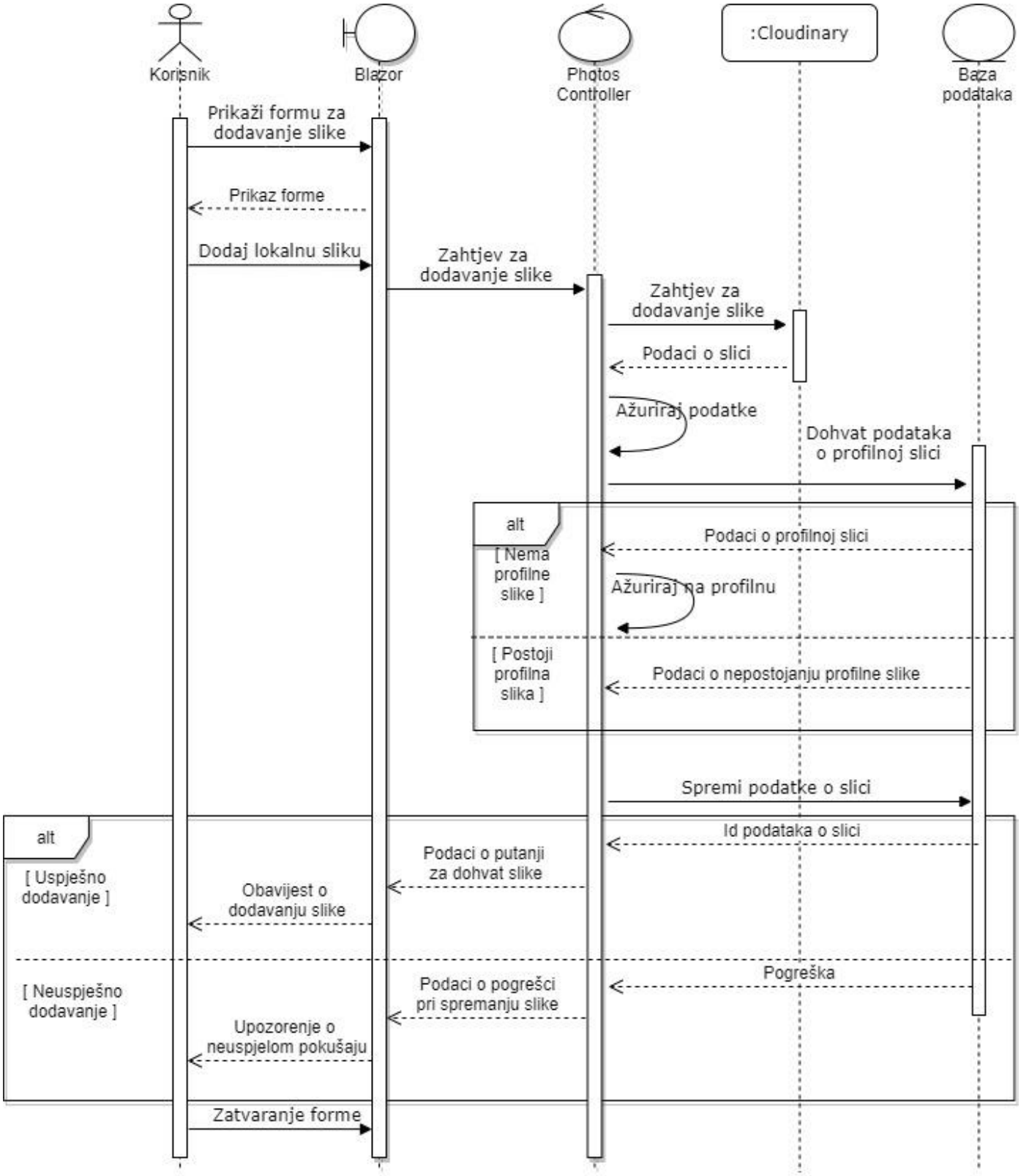
Tablica 2: Popis API krajnjih točaka aplikacije

Putanja	**HTTP metoda**	**Opis funkcije postavljene na putanji**
api/auth/register	POST	Krajnja točka na koju se šalju podaci korisnika za registraciju u aplikaciju
api/auth/login	POST	Putanja na koju se korisnik ulogira na aplikaciju, ako je zadovoljavajući username i password kao odgovor dobiva JWT token
api/users/{UserParams}	GET	Putanja na kojoj se kao odgovor dobiju korisnici unutar nekih zadanih parametara spol, godina,... putanja pruža i paginaciju
api/users/{id}	GET	Putanja za prikaz jedinstvenog korisnika sa specifičnim id-em
api/users/{id}	PUT	Putanja za promjenu podataka na profilu, svaki korisnik može urediti podatke samo za sebe, poslužitelj razaznaje na osnovi JWT tokena
api/{id}/like/{recipientId}	POST	Putanja na kojoj korisnik sa id lajka korisnika sa recipientId
api/users/{userId}/messages/{id}	GET	Dohvaćanje poruke sa jedinstvenim id-em
api/users/{userId}/ messages/{MessageParams}	GET	Dohvaćanje svih poruka nekog usera, sa određenom zastavicom (npr. Unread), paginacija
api/users/{userId}/ messages/thread/{recipientId}	GET	Dohvaćanje svih poruka u razgovoru sa korisnikom recipientId
api/users/{userId}/ messages/{MessageForCreationDto}	POST	Slanje nove poruke
api/users/{userId}/messages/{id}	POST	Brisanje vlastite poruke

api/users/{userId}/messages/{id}/read	POST	Označivanje poruke kao pročitane od strane korisnika userId
api/users/{userId}/photos/{id}	GET	Dohvaćanje slike sa id-em
api/users/{userId}/photos/{id} [FromBody]{ PhotoForCreationDto}	POST	Dodavanje nove slike za korisnika unutar repozitorija
api/users/{userId}/photos/{id}/setMain	POST	Postavljanje specifične slike kao glavne profilne
api/users/{userId}/photos/{id}	DELETE	Brisanje slike iz repozitorija

Moduli autentifikacije i interakcije sa sustavom za upravljanje bazom podataka razvijeni su uz pomoć uzorka repozitorija (engl. *repository pattern*) da bi odvojili logiku unutar kontrolera od logike sloja za rad sa podacima. Za mapiranje klasa, odnosno njihovih svojstava, korištena je biblioteka AutoMapper. Takva biblioteka je korisna kada imamo različite klase u sloju za rad s podacima u odnosu na sloj za prezentaciju.

Za rad sa slikama unutar sustava korišten je Cloudinary. To je aplikacija za upravljanje multimedijskim podacima(engl. media asset management (MAM)). Korišten je kako se ne bismo morali brinuti o spremanju slike i njihovom posluživanju. Uz samo posluživanje MAM nam često nudi i osnovne mogućnosti uređivanja medije. U ovom projektu korištena je mogućnost pospremanja slike u formatu preddefiniranih dimenzija. U nastavku, na slici 7, slijedi dijagram slijeda koji opisuje proces spremanja slike. Korisnik preko klijentske aplikacije, pisane uz pomoć Blazora, poziva `Photos` kontroler. Kontroler odrađuje interakciju sa sustavom Cloudinary, u koji se sprema slika, i bazom podataka, u kojoj se spremaju metapodaci slike. Na kraju korisnik dobije obavijest jesu li podaci uspješno spremljeni ili je došlo do neke pogreške.



Slika 8: Dijagram slijeda dodavanja slike unutar aplikacije (vlastita izrada)

4.3.2. Dijeljeni kod

Unutar rješenja aplikacije projekt pod nazivom MeetMe.Shared sadrži klase koje dijele poslužitelj i klijent. Te klase, odnosno objekti kojima se prenose podaci između dva odvojena procesa čine uzorak dizajna pod nazivom objekti za prijenos podataka (engl. data transfer object (DTO)).

Sam projekt podijeljen je u četiri mape: Authentication, User, Messages i Photos. Mape su napravljene tako da predstavljaju kontrolere unutar kojih se te klase koriste. Neke klase pojavljuju se kao argument HTTP zahtjeva s klijentske strane (Blazor), odnosno kao parametar metode u kontroleru. Druge su povratni tip metode u kontroleru, odnosno povratni tip HTTP odgovora na klijentskoj strani. Tako u Authentication mapi imamo klase `UserForRegisterDto` i `UserForLoginDto` koje se koriste kada s klijentske strane proslijedimo poslužiteljskoj strani podatke o korisniku koji se želi registrirati ili ulogirati. Druge dvije klase `RegisterResult` i `LoginResult` koriste se kao povratni tip metoda `Register` i `Login`, a obavještavaju klijentsku stranu aplikacije o uspješnosti poziva. Sadržaj preostalih mapa je veoma sličan, u prijevodu sadrži klase koje se koriste kao ulaz ili izlaz komunikacije između klijenta i poslužitelja.

4.3.3.Klijentska strana aplikacije

Klijentski projekt sastoji se od nekoliko mapa. Datoteke koje se nalaze izvan mapa su `_Imports.razor`, `App.razor` i `Program.cs`. Unutar `_Imports.razor` datoteke navedene su sve reference na klase koje se koriste u komponentama. U `App.razor` datoteci je definirana korijenska komponenta aplikacije. Ona u sebi ima definirane komponente za usmjeravanje unutar aplikacije (`Found`, `NotFound`, `Router`), komponente koje definiraju stanje autentifikacije (`CascadingAuthenticationState`, `AuthorizeRouteView`) te komponentu (`LayoutView`) koja referencira komponentu za glavni raspored akplikacije. `Program.cs` datoteka je početna točka koja postavlja samu Blazor WebAssembly aplikaciju. U toj se datoteci inicijalizira korijenska komponenta i registriraju svi servisi koji se koriste.

`wwwroot` mapa sadrži statičke datoteke aplikacije koje se poslužuju direktno klijentu. Tu se nalaze HTML, CSS i JavaScript datoteke te slike. Najbitnija među njima je `index.html` datoteka koja predstavlja početnu stranicu aplikacije, a zadužena je za preuzimanje preostalih datoteka unutar te mape. CSS datoteke organizirane su u mapi pod nazivom CSS. Tu se nalazi datoteka `app.css` u kojoj se nalazi CSS aplikacije, te podmape `Bootstrap` i `Open-iconic` koje dolaze sa predloškom aplikacije, a uključuju istoimene okvire. Osim navedenog tu je i mapa pod nazivom `alertify` koja sadrži CSS datoteke malog JavaScript okvira za izradu notifikacija u pregledniku. U `script` mapi nalazimo datoteku okvira Alertify i `application.js` skriptu u kojoj su registrirane Alertify metode, `success` i `error`, a koje se koriste unutar aplikacije.

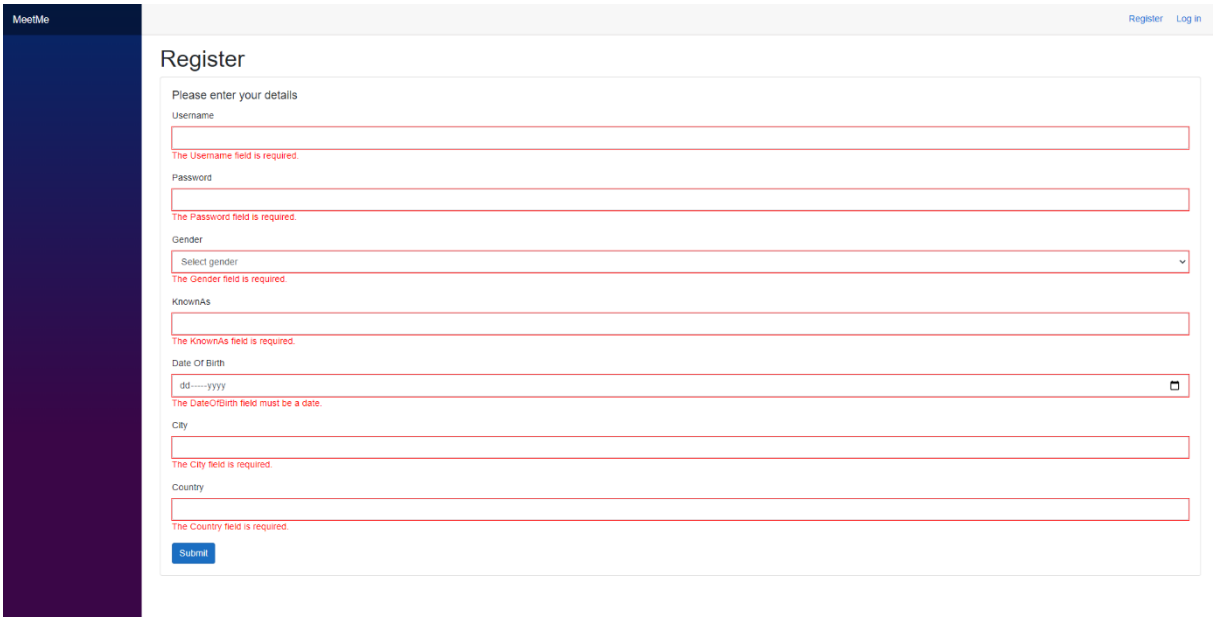
U `Services` mapi nalaze se klase koje se koriste kao servisi unutar aplikacije, a u DI spremnik aplikacije registrirane su pomoću `AddScoped` metode unutar `Program.cs` datoteke. Klase, odnosno servisi, `AuthService` i `MemberService` odrađuju komunikaciju prema poslužitelju. `AuthService` zadužen je za obradu akcija vezanih uz registraciju, te login i logout. Taj servis uz pomoć Nuget paketa `Blazored.LocalStorage` čuva stanje autentifikacije. Naime, `Blazored.LocalStorage` nam omogućuje da pohranjujemo podatke u lokalnu pohranu preglednika u tekstualnom obliku. `AuthService` prilikom `login` akcije u tu pohranu sprema JWT token koji će biti korišten za autentifikaciju pri svakoj budućoj interakciji sa poslužiteljem. `MemberService` je omotač oko preostalih endpointa na poslužiteljskoj strani koji nisu zaduženi za autentifikaciju. On sa određenim ulaznim vrijednostima koje dobije od komponenti preko integriranog HTTP klijenta u Blazor aplikaciji, odrađuje komunikaciju s poslužiteljem. Neke od akcija koje `MemberService` izvršava su dohvaćanje korisnika, promjena podataka unutar profila ili razmjena poruka. Posljednji razvijeni servis naziva `AlertifyService` poziva JavaScript okvir Alertify preko ugrađene funkcionalnosti JS interop.

Unutar `Helpers` mape nalaze se pomoćne metode korištene unutar klijentskog dijela aplikacije. Tu se nalazi i klasa `ApiAuthenticationStateProvider`, koja je prilagođena klasa koja nasljeđuje `AuthenticationStateProvider`. Ta klasa je zapravo servis koji komponentama `AuthorizeView` i `CascadingAuthenticationState` daje informaciju o statusu autentifikacije klijenta. U nastavku je prikazana premošćena metoda `GetAuthenticationStateAsync`. Ona preko servisa za rad s lokalnom pohranom unutar preglednika dohvati autentifikacijski token (JWT) i spremi ga u varijablu `savedToken`. Potom se provjerava tekstualna vrijednost `savedToken` varijable te ako je dohvaćena vrijednost null, prazan string ili razmak metoda vraća prazno stanje autentifikacije. To prazno stanje spomenutim komponentama označava kako klijent nije autentificiran.

```
public override async Task<AuthenticationState>
GetAuthenticationStateAsync()
{
     var savedToken = await
            _localStorage.GetItemAsync<string>("authToken");
     if (string.IsNullOrWhiteSpace(savedToken))
     {
            return new AuthenticationState(new ClaimsPrincipal(new
                   ClaimsIdentity()));
     }
     return new AuthenticationState(new ClaimsPrincipal(new
            ClaimsIdentity(ParseClaimsFromJwt(savedToken))));
}
```

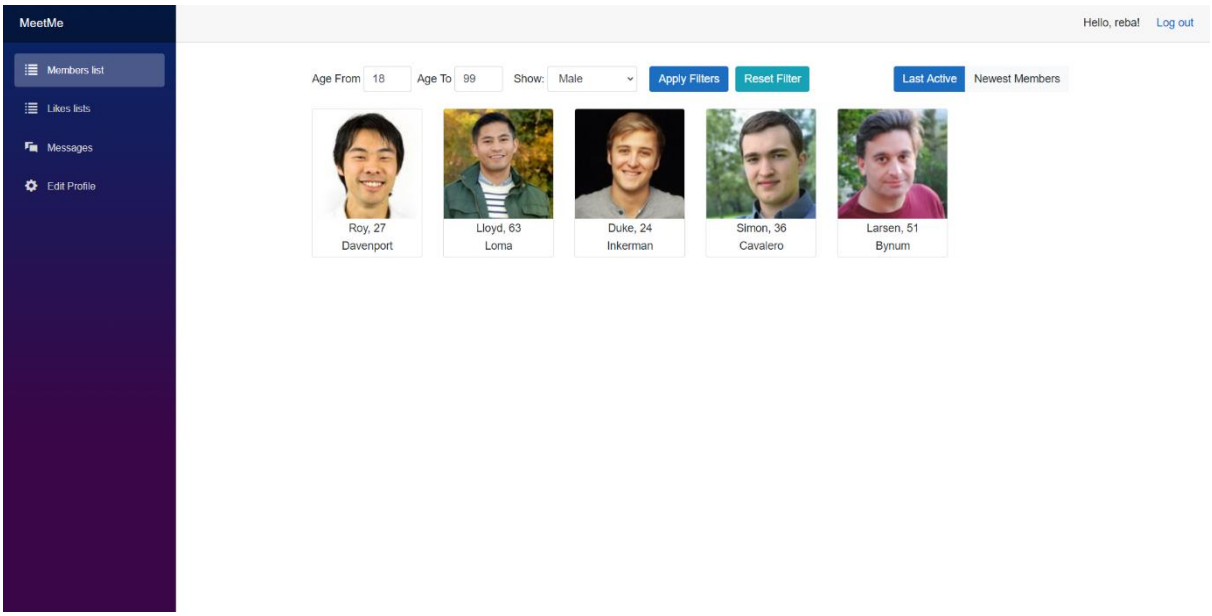
U slučaju da se u lokalnoj pohrani nalazi JWT token, metoda vraća objekt stanja autentifikacije, `AuthenticationState`, u kojem se nalaze sve dozvole (engl. *claimovi*) na dozvoljene akcije proslijeđeni od poslužiteljske strane aplikacije.

Slijedi `Pages` mapa sa stranicnim komponentama. Dolaskom na stranicu aplikacije prvo se prikazuje `Index` komponenta koja sadrži samo pozdravni tekst. `Register` je komponenta koja prikazuje formu u koju korisnik unosi osnovne informacije o sebi. Ta komponenta koristi ugrađenu komponentu `EditForm` za prikaz polja forme. Osim toga, `Register` komponenta koristi i ugrađenu validaciju podataka. Na slici 8 prikazan je primjer neuspješne validacije podataka unutar forme za registraciju.



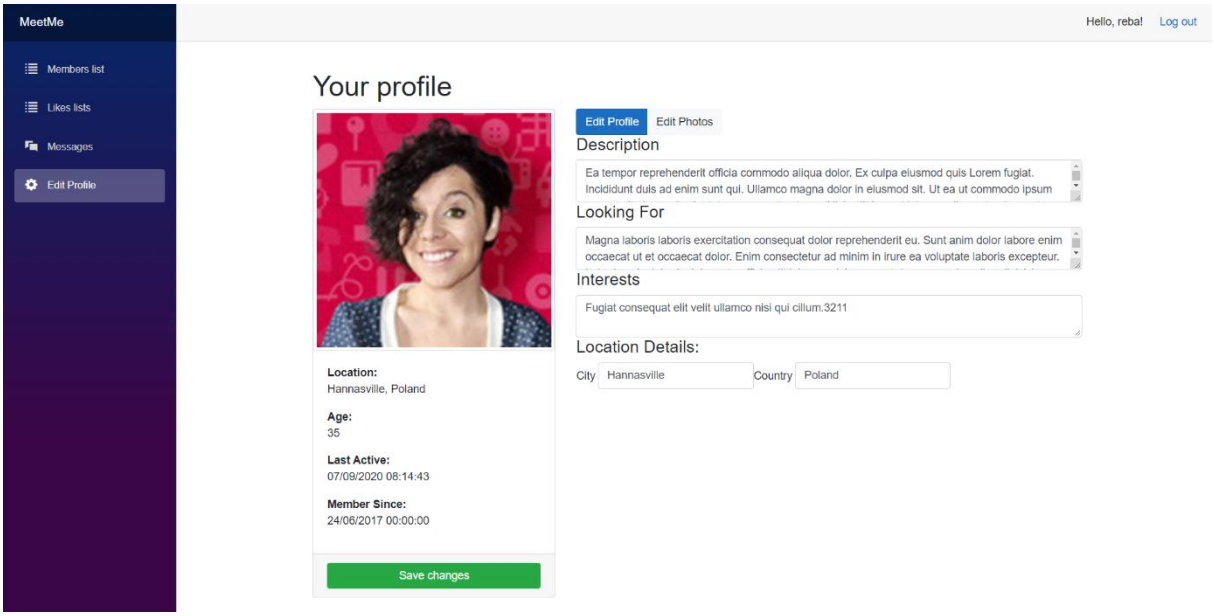
Slika 9: Validacija unutar forme za registraciju (vlastita izrada)

Uz komponentu za registraciju za proces autentifikacije napravljene su i komponente `Login` i `Logout`. `Login` komponenta je mala forma sa poljima za `username` i `password`, dok `Logout` komponenta uz pomoć `AuthService` servisa odjavljuje korisnika te sa `NavigationManager` klasom usmjeri korisnika na početnu stranicu. `MemberList` komponenta prikazuje sve korisnike unutar aplikacije s filterom, radi lakšeg pregleda, može se vidjeti na slici 10. Svaki korisnik je u listi prikazan pomoću `MemberCard` komponente. `MemberCard` komponenta nije stranična komponenta, već pomoćna te se nalazi u Shared mapi. Ponovno se koristi u `LikesList` stranicnoj komponenti. `LikesList` komponenta prikazuje korisnike koje je ulogirani korisnik označio da mu se sviđaju ili korisnike koji su ulogiranog korisnika označili kao da im se sviđa.



Slika 10: MemberList komponenta (vlastita izrada)

Na `Messages` stranični komponenti nalazi se pregled ulaznih i odlaznih poruka. `EditProfile` je komponenta u kojoj ulogirani korisnik uređuje svoj profil, npr. uređuje opis ili interese te galeriju vlastitih slika, prikazana je na slici 11. `Detailed` komponeta je također stranična, njena putanja sastoji se od riječi `detailed` i parametara `id` i `tab`. Parametar `id` predstavlja jedinstvenu identifikaciju korisnika čiji profil je prikazan, dok parametar `tab` određuje na kojoj kartici (engl. *tab*) će se profil prikazati.



Slika 11: EditProfile komponenta (vlastita izrada)

Preostala je `Shared` mapa unutar koje se nalaze pomoćne komponente koje se višestruko koriste kroz cijelu aplikaciju. `MainLayout` komponenta se sastoji od dvije podkomponente, a koristi se na svakoj stranici aplikacije. Naime, `MainLayout` komponenta je glavni raspored aplikacije. U sebi sadrži `NavMenu` komponentu koja prikazuje bočni izbornik sa poveznicama na sve postojeće stranice u aplikaciji. Druga je `LoginDisplay` komponenta, koja se nalazi na vrhu svake stranice te ovisno o stanju autentifikacije prikazuje gumbe za registraciju i login ili pozdravnu poruku ulogiranom korisniku („Hello, {korisnik}!“) te gumb za logout. Osim `MainLayout` komponente u mapi se nalazi i već spomenuta `MemberCard` komponenta. Na kraju imamo dvije komponente `TabControl` i `TabPage` koje omogućuju kartični prikaz podataka u straničnoj komponenti bez promjene putanje.

5. Zaključak

Blazor je novi okvir razvijen od strane Microsofta za razvoj klijentske strane aplikacija za web. On dolazi kao alternativa JavaScriptu koji dvadesetak godina ima monopol nad sferom razvoja klijentske strane aplikacija za web. Do sada je uobičajena opcija bila koristiti JavaScript u kombinaciji s nekim drugim poslužiteljskim jezikom. Činjenica da web programer mora znati dva odvojena programska jezika kako bi razvio neku aplikaciju za web rezultirala je velikom ulaznom barijerom za programera u obliku dugotrajnog obrazovanja. Blazor donosi nešto što je rijetko koja alternativa pridonijela. Naime, Blazor je okvir za C# programski jezik, što znači da nam Blazor pruža mogućnost razvoja klijentske i poslužiteljske strane aplikacija za web unutar jednog programskog jezika.

Istina je da su neke alternative kao Dart i Lua već pružile razvoj u jednom programskom jeziku, ali Blazorova prednost je ogromna zajednica .NET programera, velik broj postojećih alata za programiranje u C# i sama dugovječnost jezika u odnosu na navedene alternative. Također kao što je navedeno u radu i JavaScript je moguće izvršavati na poslužitelju ali mane samog jezika su nešto što je i natjeralo programere da počnu razvijati alternative JavaScriptu.

U okviru ovoga rada ukratko su prikazane trenutno dostupne tehnologije za razvoj web aplikacija. Zatim je u središnjem dijelu objašnjen način funkcioniranja Blazora i pokazane su osnove okvira potrebne za razvoj jedne aplikacije za web. Na kraju je ukratko pokazana aplikacija razvijena u C#-u uz Blazor okvir.

Razvoj same aplikacije bio je jednostavniji i brži u usporedbi s razvojem slične aplikacije unutar Angular okvira. Za početak, sam proces izgradnje klijentskog i poslužiteljskog koda znatno je brži kod aplikacije razvijene pomoću Blazor okvira. Naime, u slučaju Blazora izgradnja cjelokupne aplikacije se pokreće na pritisak jednog gumba u Visual Studiju dok sam kod aplikacije pisane u Angular okviru morao odvojeno izgraditi klijentski dio aplikacije od poslužiteljskog dijela. Druga prednost je bila prilikom samog debugiranja, naime klijentski i poslužiteljski dio Blazor aplikacije sam mogao debugirati unutar VS bez dodatnog korištenja alata ugrađenih u preglednik. Također, svaku izmjenu u nekom od modela odradio sam na jednom mjestu i automatski je bila vidljiva u cijeloj aplikaciji, za razliku od Angular aplikacije kod koje je izmjene bilo potrebno raditi dvaput, na klijentskom i poslužiteljskom dijelu. Na kraju, iako je Blazor okvir tek nedavno pušten u upotrebu, za sve probleme s kojima sam se susreo prilikom razvoja aplikacije unutar Blazor okvira na stranicama namijenjenima za pomoć programerima (kao što je Stack Overflow) već postoje odgovori.

U konačnici, Blazor je nova tehnologija koja se svakodnevno mijenja i razvija te dolazi sa skupom prednosti i nedostataka, no ciljevi koji se nalaze u planu rada (engl. *roadmap*) za

Blazor, kao što je mogućnost odvojenih CSS datoteka za komponente, dohvaćanje koda komponente u trenutku otvaranja a ne unaprijed (engl. *lazy loading*) i direktno kompajliranje u Wasm (engl. *ahead of time compilation*), zasigurno će poboljšati sveukupno iskustvo korištenja okvira. U konačnici, kao i kod drugih tehnologija, globalna prihvaćenost i vrijeme će pokazati hoće li Blazor uspjeti opstati u dinamičkom svijetu razvoja aplikacija za web.

Popis literature

[1] Starlink (bez dat.) Starlink [Na internetu]. Dostupno: https://www.starlink.com/ [pristupano 25.08.2020.]

[2] A. Mesbah, A. van Deursen, "Migrating Multi-page Web Applications to Single-page AJAX Interfaces," 11th European Conference on Software Maintenance and Reengineering (CSMR'07), Amsterdam, 2007, pp. 181-190, doi: 10.1109/CSMR.2007.33. [Na internetu]. Dostupno: IEEE Xplore, https://ieeexplore.ieee.org/ [pristupano 09.08.2020.]

[3] M. Wasson, „ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET“, MSDN Magazine, sve. 28, izd. 11, studeni 2013.

[4] S. Richard, „What are Progressive Web Apps?“, [Blog post]. 24.02.2020. [Na internetu]. Dostupno: https://web.dev/what-are-pwas/ [pristupano 20.07.2020.]

[5] J. C. Jackson, Web technologies: a computer science perspective, Upper Saddle River, NJ, USA, Pearson Education 2007

[6] WHATWG (bez dat.) HTML Living Standard [Na internetu]. Dostupno: https://html.spec.whatwg.org/ [pristupano 09.07.2020.].

[7] Mozilla (bez dat.) CSS [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/CSS [pristupano 09.07.2020.]

[8] Mozilla (bez dat.) JavaScript [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/JavaScript [pristupano 10.07.2020.]

[9] S. Peyrott, „A Brief History of JavaScript“, [Blog post]. 16.01.2017.[Na internetu]. Dostupno: https://auth0.com/blog/a-brief-history-of-javascript/ [pristupano 10.07.2020.].

[10] The Jquery Foundation (bez dat.) jQuery API [Na internetu]. Dostupno: https://api.jquery.com/ [pristupano 11.07.2020.].

[11] A. Lundiak, „History'n'Evolution of JS MV* frameworks“, [Blog post]. 25.09.2014.[Na internetu]. Dostupno: https://worknme.wordpress.com/2014/09/25/history-and-evolution-of-js-mvc-mvv-frameworks/comment-page-1/ [pristupano 11.07.2020.].

[12] Mozilla (bez dat.) Understanding client-side JavaScript frameworks [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks [pristupano 11.07.2020.].

[13] T. Krotoff, „Frontend Frameworks Popularity“, [Blog post]. 12.12.2019. [Na internetu]. Dostupno: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190 [pristupano 11.07.2020.]

[14] J. Aguinaga, „How it feels to learn JavaScript in 2016“, [Blog post]. 03.10.2016.[Na internetu]. Dostupno: https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f [pristupano 08.08.2020.]

[15] Quora, „Why is JavaScript so hated?“(bez dat.) [Na internetu]. Dostupno: https://www.quora.com/Why-is-JavaScript-so-hated [pristupano 08.08.2020.]

[16] P. Himschoot, *Microsoft Blazor*, New York, NY, USA, Apress. 2020.

[17] Mozilla (bez dat.) WebAssembly [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/WebAssembly [pristupano 21.06.2020.].

[18] Microsoft (bez dat.) ASP.NET documentation [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1 [pristupano 19.06.2020.].

[19] T. A. Gamage, „HTTP and Websockets: Understanding the capabilities of today's web communication technologies“, [Blog post]. 19.11.2017. [Na internetu]. Dostupno: https://medium.com/platform-engineer/web-api-design-35df8167460 [pristupano 20.06.2020.]

[20] M. Aponte, *Building Single Page Applications in .NET Core 3*, New York, NY, USA, Apress. 2020.

[21] S. Guthrie, "Introducing “Razor” – a new view engine for ASP.NET", [Blog post]. 03.06. 2010. [Na internetu]. Dostupno: https://weblogs.asp.net/scottgu/introducing-razor [pristupano 19.06.2020.]

[22] A. Troelsen i P. Japikse, *Pro C# 7: With .NET and .NET Core*, New York, NY, USA, Apress. 2017.

[23] Dev Apps, (18.06.2020.) „BlazorDay 2020“, Youtube [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=XoizucRjxgU [pristupano 09.08.2020.]

[24] Progress Telerik, (srpanj 2020.) „Blazing into Summer“, Youtube [Video datoteka], Dostupno: https://www.youtube.com/playlist?list=PLvmaC-XMqeBZO_LOC4oWsMIGgva7_4JbZ [pristupano 01.08.2020.]

[25] N. Cummings, (05.2020) „Build an app with ASPNET Core and Angular from scratch“, Udemy [Video datoteke], Dostupno: https://www.udemy.com/course/build-an-app-with-aspnet-core-and-angular-from-scratch/ [pristupano 10.06.2020.]

Popis slika

Popis tablica