Migracija aplikacija iz programskog jezika C#.Net u WebAssembly

Darjan, Baričević

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:848282

Rights / Prava: Attribution-NonCommercial-NoDerivs 3.0 Unported/Imenovanje-Nekomercijalno-Bez prerada 3.0

Download date / Datum preuzimanja: **2024-04-25**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Darjan Baričević

MIGRACIJA APLIKACIJA IZ PROGRAMSKOG JEZIKA C#.NET U WEBASSEMBLY

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Darjan Baričević

Matični broj: 44025/15–R

Studij: Informacijsko i programsko inženjerstvo

MIGRACIJA APLIKACIJA IZ PROGRAMSKOG JEZIKA C#.NET U WEBASSEMBLY

DIPLOMSKI RAD

Mentor :

Doc. dr. sc. Zlatko Stapić

Varaždin, rujan 2020.

Darjan Baričević

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U radu je opisan značaj tehnologije WebAssembly u razvoju modernih web aplikacija i izgradnjih većih ekosustava. Uvodni dio rada sadržava općenitosti tehnologije WebAssembly te ciljevi koji se žele postići u aspektu Web-a primjenom iste. Potom je opisana problematika u postizanju tih ciljeva uz navođenje nekih od prethodnih pokušaja implementacije tehnologija sličnim WebAssemblyu, te je opisan proces razvoja tehnologije. Nakon toga slijedi opis svojstava formata WebAssembly kao što su virtualni stroj, instrukcijski set i slično kako bi se pojasnio princip rada tehnologije i način integracije u Web platformu. Kako bi se dobio dojam o mogućnostima tehnologije, na kraju poglavlja su opisane neke od poznatijih aplikacija iz realne domene. Drugo poglavlje teorijskog dijela rada opisuje Blazor programski okvir i njegove koncepte koji će biti korišteni u praktičnom dijelu rada. Ovdje je veći naglasak stavljen na Blazor WebAssembly inačicu koja koristi spomenutu tehnologiju WebAssembly. Praktični dio rada kroz nekoliko potpoglavlja opisuje migraciju temeljnih slojeva postojeće stolne C# aplikacije u Blazor WebAssembly web aplikaciju. Unutar tih potpoglavlja stavljen je naglasak na sličnosti i razlike u radu stolnih i web .NET aplikacija te na komplementarnost WebAssembly-a sa Javascript tehnologijom u razvoju na korisničkoj strani.

Ključne riječi: razvoj, razvoj programskih proizvoda, migracija, C#, WebAssembly

Sadržaj

1. Uvod 1

2. Metode i tehnike rada 2

3. WebAssembly 3

3.1. Općenito 3

3.2. Zašto WebAssembly? 3

3.2.1. Ciljevi Wasm-a 6

3.2.2. Slučajevi korištenja 7

3.3. Razvoj Wasm-a 8

3.3.1. Prethodni pokušaji 8

3.3.2. Od asm.js-a do minimalno održivog proizvoda 10

3.4. Karakteristike 12

3.4.1. Virtualni stroj 12

3.4.2. Formati 14

3.4.2.1. Binarni format 14

3.4.2.2. Tekstualni format 14

3.4.3. Semantika 16

3.4.3.1. Moduli 16

3.4.3.2. Semantičke faze 17

3.4.3.3. Tipovi podataka 18

3.4.3.4. Memorija 18

3.4.3.5. Instrukcije 19

3.4.4. Nedeterminizam 21

3.4.5. Otkrivanje grešaka 21

3.5. Ugradnja 24

3.5.1. Ugradnja u Web platformu 24

3.5.2. Ugradnja izvan Web platforme 25

3.5.3. Podržani jezici 25

3.6. Aplikacije u realnoj domeni 26

3.7. Budućnost Wasm-a 29

4. Blazor 32

4.1. Općenito o programskom okviru 32

4.2. Inačice 33

4.3. Komponente 34

4.3.1. Komponentni model 35

4.3.2. Životni ciklus . . . 36
4.3.3. Dependency injection . . . 38
4.4. Blazor WebAssembly . . . 40
4.4.1. Kako funkcionira Blazor WebAssembly? . . . 40
4.4.2. Prednosti i nedostatci . . . 42

5. Migracija .NET WinForms aplikacije u Blazor WebAssembly . . . 44
5.1. Analiza prenosivosti i početne postavke . . . 45
5.2. Migracija podatkovnog sloja . . . 46
5.2.1. Razlike u toku podataka . . . 47
5.2.2. Migracija poslovnih modela i konteksta baze podataka . . . 48
5.2.3. Migracija sloja za pristup podacima . . . 49
5.3. Migracija korisničkih kontrola . . . 59
5.3.1. Korištenje biblioteka komponenti treće strane . . . 60
5.3.2. Razvoj vlastitih komponenti . . . 60
5.4. Migracija aplikacijskih resursa . . . 63
5.4.1. Migracija statičkih resursa . . . 63
5.4.2. Migracija dinamičkih resursa . . . 64
5.5. Migracija aplikacijskog stanja . . . 66
5.6. Migracija logike za validaciju korisničkog unosa . . . 72
5.7. Migracija poslovne logike . . . 74
5.8. Migracija sigurnosnog sloja . . . 74
5.8.1. Migracija autentifikacijskog sloja . . . 75
5.8.2. Migracija autorizacijskog stanja . . . 82

6. Zaključak . . . 86

Popis literature . . . 89

Popis slika . . . 91

Popis tablica . . . 92

1. Uvod

Unutar posljednjih 30 godina, odnosno od nastanka Web platforme davne 1991. godine pa sve do danas, zahtjevi okruženja i potrebe korisnika Web-a značajno su se promjenile. Ono što je započelo kao usluga Interneta koja omogućuje pregled i razmjenu hipertekstualnih dokumenata [1], kroz dugi period razvilo se u cijeli ekosustav jednostranih (engl. *SPA, single-page application*) i progresivnih web aplikacija (engl. *PWA, progressive web application*) koje su smještene u oblaku (engl. *cloud-based*), neprestano se testiraju i ažuriraju te imaju bogata i dinamična korisnička sučelja koja se prilagođavaju na svim uređajima. Inicijalno se svaka Web stranica sastojala samo od oznaka koje su bile dio HTML (engl. *hypertext markup language*) specifikacije, zatim se 1994. godine pojavio CSS (engl. *cascading style sheets*) koji je sa svojim stilskim uputama omogućio odvajanje prezentacije i sadržaja, a tek se 1995. godine pojavio prvi skriptni programski jezik LiveScript (odnosno danas poznatiji kao Javascript) koji je dodao dinamiku web stranicama i omogućio odvajanje programske logike od sadržaja. Od tada su HTML, CSS i Javascript deklarirani kao web standardi od strane W3 konzorcija i tako su postali podržani u svakom web pregledniku.

Kroz godine dodavane su nove oznake u HTML specifikaciju koje su omogućile pisanje semantičkog HTML-a, nova svojstva stilskih uputa koja su omogućila razvoj bogatijih i složenijih rasporeda te animacija, a što je sa Javascriptom? Inicijalna zadaća Javascript tehnologije u Webu bilo je upravljanje objektnim modelom dokumenta (engl. *DOM, document object model*), i to je zadaća koju Javascript i dalje vrlo dobro izvršava. Međutim, kao jedini programski jezik koji se izvodi u web pregledniku, očekivanja prema Javascriptu su neprestano rasla. S ciljem proširenja njegovih mogučnosti, pojavljivale su se implementacije raznih dodataka u obliku Javascript biblioteka koje su imale znatno lošije performanse od istih implementacija u drugim jezicima više razine. Postajalo je sve jasnije da Javascript ne može ispunjavati sve teže zahtjeve platforme, ali nitko nije imao ideje kako rješiti taj problem sve do 2014. godine kada je Mozilla zajednica dobila sjajnu ideju razviti podskup Javascript instrukcija niske razine koje će funkcionirati kao rezultat prevođenja (engl. *compilation target*) za programske jezike više razine. Tako je iste godine nastao **asm.js**, a daljnom kolaboracijom sa pružateljima glavnih web preglednika na tržištu postaje podržan u istima i dobiva ime WebAssembly (detaljnije o ovome u poglavlju 3.3.). U prosincu 2019. godine W3 konzorcij službeno proglašava WebAssembly web standardom i tako postaje 4. jezik koji se izvodi u pregledniku, time započinjući revoluciju u Web razvoju i otvarajući bezbrojne mogućnosti. [2].

2. Metode i tehnike rada

U razradi teme rada je, s obzirom na novinu odabranih tehnologija, najviše korištena službena dokumentacija WebAssembly binarnog formata i Blazor programskog okvira kao i javno dostupni video zapisi nedavnih stručnih konferencija iz cijeloga svijeta u kojima se detaljnije pričalo o potencijalima navedenih tehnologija. Također su korišteni i radovi napisani u sklopu tehničkih konferencija i par postojećih knjiga napisanih u vrijeme dok je WebAssembly format još bio u eksperimentalnoj fazi.

Ideja praktičnog dijela rada je da se postojeća stolna .NET aplikacija (Windows Forms biblioteka) migrira u Blazor WebAssembly aplikaciju posluženu na .NET Core poslužitelju. S obzirom da najnovije verzije dviju navedenih .NET izvršnih okruženja imaju dobar dio preklapajućih biblioteka, cilj je da se pokuša što veći dio poslužiteljske strane ponovno iskoristiti dok se klijentski dio (Windows Form kontrole) iznova razvija. Također, postojeća stolna aplikacija i buduća web aplikacija moraju koristiti istu postojeću bazu podataka kako bi poslovna logika ostala nepromijenjena. S time na umu, uvjeti prilikom odabira aplikacije za migraciju su sljedeći:

- aplikacija mora biti javno dostupna na GitHub repozitoriju.
- repozitorij aplikacije mora sadržavati i izvezenu datoteku baze podataka.
- aplikacija mora ostvarivati rezultat od minimalno 90% u alatu *.NET Portability Analyzer* alatu za analizu prenosivosti kako bi se osigurala mogućnost potpune migracije i preostalih 10% sa alternativnim bibliotekama te kako bi što manje poslužiteljskog kôda bilo izmijenjeno.
- aplikacija mora biti dovoljno kompleksna kako bi se što više koncepata migracije moglo opisati (mora imati podatkovni sloj, sigurnosni sloj, servisni sloj itd.).

Nadalje, kako bi provedba praktičnog dijela rada bila moguća potrebno je imati instalirane sljedeće alate i tehnologije:

- Visual Studio Community 2019 - besplatno integrirano razvojno okruženje za razvoj .NET tehnologija.
- .NET Core 3 SDK - razvojni paket koji sadržava Blazor WebAssembly programski okvir.
- SQL Server 2019 - sustav za upravljanje bazama podataka na kojem će biti pohranjena baza podataka stolne i web aplikacije.

Sav programski kôd nalaziti će se na javnom GitHub repozitoriju:

https://github.com/doksara/BlazorShop

3. WebAssembly

U ovom poglavlju biti će opisane karakteristike tehnologije WebAssembly te razlog njene pojave odnosno ciljevi koji se žele riješiti uporabom tehnologije. S obzirom na ciljeve, biti će pobrojani i specifični slučajevi u kojima je smisleno koristiti format WebAssembly. Nakon toga slijedi kratka povijest spomenute tehnologije te slični implementacijski pokušaji koji su prethodili pojavi WebAssemblya. Potom slijedi potpoglavlje u kojem je opisano kako WebAssembly funkcionira na nešto nižoj razini. Potpoglavlje također uključuje sve karakteristike i svojstva formata, a samim time i objašnjenja na koji način su kroz takav dizajn tehnologije ispravljeni problemi prethodnika WebAssemblya. Kako bi se približile trenutne i buduće mogućnosti spomenute tehnologije, za kraj su opisane neke aplikacije iz realne domene te su navedena svojstva koja će biti dodana u budućnosti, a bitna su za proširenje mogućnosti tehnologije.

3.1. Općenito

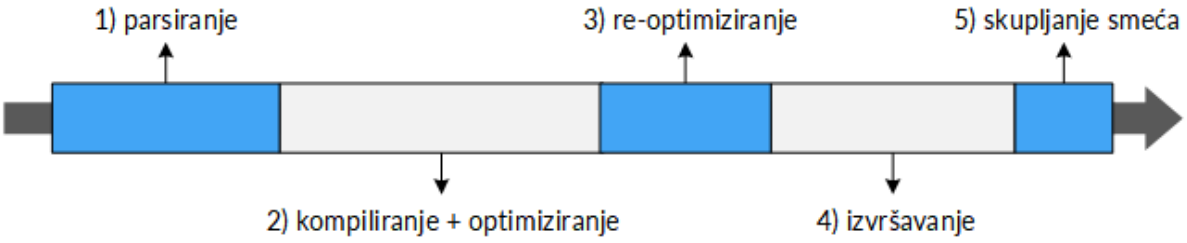
WebAssembly (skraćeno Wasm) je binarni instrukcijski format za virtualne strojeve koji se temelje na stog memoriji (engl. *stack-based virtual machine*) [3]. Dizajniran je na način da bude prenosiv (engl. *portable*) rezultat prevođenja programskih jezika visoke razine poput C, C++ ili Rust i na taj način omogućiti postavljanje (engl. *deployment*) postojećih klijentskih i poslužiteljskih aplikacija na Internet. Unatoč tome što mnogi tvrde da se Wasm pojavio kao zamjena za Javascript i uz to nagoviještavaju "smrt" Javascript-a na klijentskoj strani, istina je zapravo potpuno suprotna [4]. Zapravo je Wasm dizajniran da bude komplement Javascript tehnologiji u razvoju na korisničkoj strani i popunjava njegove nedostatke [5] (više o ovome u narednim poglavljima). Uz pomoć WebAssembly Javascript API-a, moguće je učitati Wasm module u Javascript aplikaciju i na taj način dijeliti funkcionalnosti između njih. To razvojnim programerima omogućava da iskoriste snagu i brzinu Wasm-a i fleksibilnost Javascript-a u jednoj aplikaciji.

3.2. Zašto WebAssembly?

Kroz evoluciju i razvoj Web platforme, Javascript više nije mogao ispunjavati zahtjeve iste u smislu performansi i njegovih mogućnosti, a postoji nekoliko razloga za to. Prije svega, Javascript jezik nije zamišljen da bude brz jer postoje brojne osobine jezika koje uvelike otežavaju mogućnost da se brzo izvršava. Na primjer, poznato je da Javascript ima dinamičke tipove podataka za razliku od npr C/C++ ili drugih jezika visoke razine koji imaju statičke tipove podataka. To znači da se tip neke varijable (odnosno podatka spremljenog unutar varijable) određuje tijekom izvršavanja, a ne tijekom pisanja programa. Nove varijable, funkcije ili bilo koji drugi objekti mogu se kreirati za vrijeme rada programa, a postojeće mogu mijenjati svoj tip podataka. Nije potrebno dodatno naglašavati koliko ova osobina dodaje na fleksibilnosti jezika, pa su stoga razvojni programeri kroz vrijeme jednostavno prihvatili ovaj "blagoslov" u zamjenu za slabije performanse [6]. Taj nedostatak djelomično je riješen 2008. godine kada su se zbog

velike konkurencije glavnih Web preglednika na tržištu pojavili JIT (engl. *just in time*) prevoditelji koji su ubrzali izvođenje Javascript aplikacija do čak 10 puta [7].

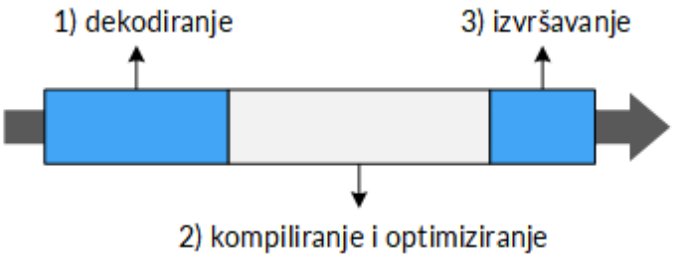
Najjednostavnije rečeno, JIT prevoditelji funkcioniraju na način da konstantno nadziru Javascript kôd i prate koje varijable, funkcije i objekti se često koriste (i kojeg su tipa) te ih označavaju kao "tople" (engl. *warm*) i šalju na kompilaciju kako bi odmah bili spremni za izvršavanje. Na primjer, ukoliko se vrti petlja unutar programa tada će prevoditelj samo provjeriti da li je pojedini segment kôda istog tipa kao i u prošloj iteraciji i, ako je, samo izvući kompiliranu verziju tog segmenta iz privremene memorije umjesto da ga ponovno kompilira. Tada će taj segment postati još "topliji" i biti poslan do optimizirajućeg prevoditelja (engl. *optimizing compiler*) koji će u memoriju spremiti još bržu verziju te funkcije. Slikoviti prikaz cijelog postupka prije izvršavanja Javascript kôda možemo vidjeti na slici ispod:



Slika 1: Proces kompilacije Javascript kôda

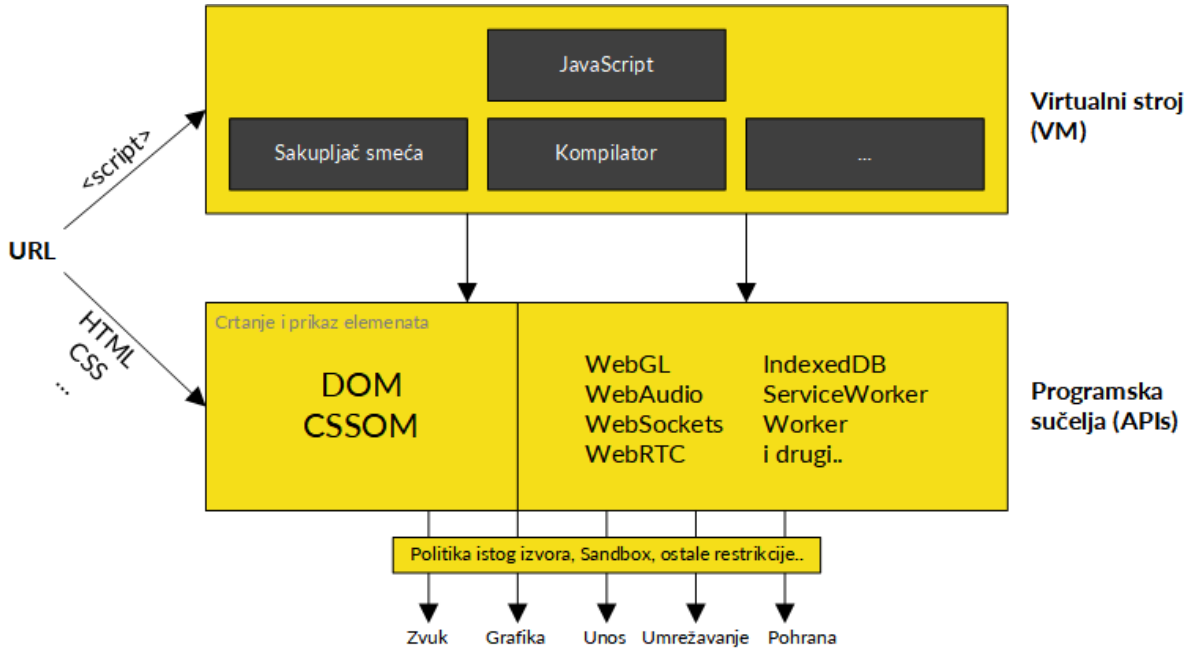
Očigledno je da se ovaj cijeli postupak može skratiti zahvaljujući optimizirajućem prevoditelju ali ga on isto tako može i produljiti. Razlog tomu je taj što optimizirajući prevoditelj, kako bi kreirao bržu verziju kôda, mora raditi pretpostavke. Drugi scenarij iz prethodnog primjera je taj da će optimizirajući prevoditelj pretpostaviti da je pojedini segment kôda iz trenutne iteracije istoga tipa kao i onaj iz prethodne, a zapravo neće biti odnosno napraviti će krivu pretpostavku. Zatim optimizirajući prevoditelj odbacuje sav optimizirani kôd i vraća se na početni korak, a to se naziva deoptimizacija (engl. *bailing out*). Dovoljno je da optimizirajući prevoditelj za vrijeme petlje u prvih 99 iteracija napravi ispravnu pretpostavku i znatno ubrza izvođenje programa, a na 100. iteraciji napravi krivu pretpostavku i može doći do neočekivano smanjenih performansi. Ukoliko tijekom izvođenja programa nekoliko puta dođe do optimizacije popraćene deoptimizacijom kôda, može čak i uzrokovati da krajnje vrijeme izvršavanja bude duže nego da se samo izvršila početna verzija kôda. Takvih problema kod Wasm-a ima znatno manje iz razloga što se u Wasm prevode programski jezici sa jakim tipovima podataka (engl. *strong types*) pa je proces optimizacije kôda uvelike jednostavniji, a isto tako nema koraka skupljanja smeća i koraka raščlanjivanja (parsiranja) jer je Wasm modul već u **bytecode** obliku kao i Javascript kada se prevede (vidi sliku ispod).

Nastavno na mogućnosti Javascript-a, valja napomenuti da razvoj i implementaciju novih osobina otežava činjenica da svaki Web preglednik ima svoj karakteristični **Javascript stroj** (engl. *Javascript engine*) ili **virtualni stroj** (engl. *virtual machine*) koji izvodi gornje prikazani postupak, pa se tako Javascript stroj u Google Chrome pregledniku naziva V8, u Mozilli postoji



Slika 2: Izvršavanje Wasm modula

SpiderMonkey itd. S time na umu, svaki Web preglednik na različiti način izvršava Javascript kôd sa različitim performansama izvođenja, a valja i uzeti u obzir da neki od njih ne podržavaju najnovije osobine ECMAScript speficikacije ili jednostavno ne rade kako bi trebale. Najjednostavnije rečeno, ECMAScript je specifikacija skriptnih jezika razvijena s ciljem standardizacije Javascripta. Isto tako, najnovije verzije glavnih Web preglednika na tržištu najčešće ne podržavaju sve osobine posljednje izdane ECMASCript specifikacije već samo jedan dio njih (popis podržanih osobina prema Web pregledniku mogu se vidjeti na servisu https://caniuse.com/#cats=JS). To je dovelo do pojave posebnih Javascript prevoditelja (kao što je npr. **Babel**) koji omogućuju prevođenje osobina iz novijih izdanja specifikacije u one starije kako bi se mogle izvršavati u svakom pregledniku.

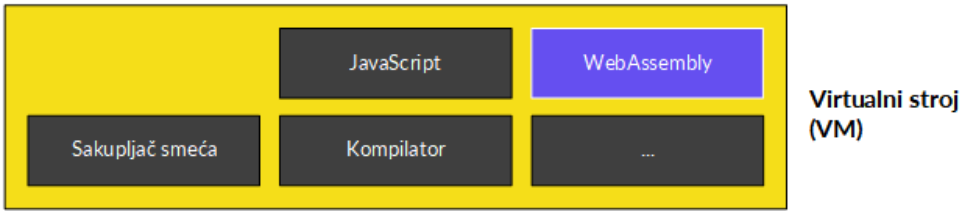


Slika 3: Struktura Web platforme

Osim virtualnog stroja, unutar web preglednika postoji i druga komponenta bez koje Javascript aplikacije ne mogu živjeti, a to su programska sučelja (engl. *API, application programming interface*)(vidi Sliku 3). Virtualni stroj, kako je već navedeno, izvršava Javascript kôd a za to su mu potrebni prevoditelj, interpeter, profiler, sakupljač smeća (engl. *garbage collector*) i drugih alati. Međutim, da bi Javascript kôd izvodio ikakve značajnije stavke poput prihvata unosa, prikaza podataka, reproduciranja zvuka ili video sadržaja, slanje web zahtjeva i ostalih

potrebna su mu programska sučelja koja nudi Web platforma.

Problem u interakciji između ove dvije komponente leži u tome što Javascript kôd prvo mora proći cjelokupan proces prevođenja, kompilacije i optimizacije kroz gore navedene alate da bi se naposlijetku dobio strojni kôd (engl. *machine code*) i pozvalo određeno programsko sučelje. To je primarni razlog slabijih performansi web aplikacija, i u rješenju ovog problema opet uskače Wasm (vidi Sliku 4). Najveća prednost Wasm-a je u tome što je namijenjen da bude kompilacijski cilj, pa su stoga moduli već prevedeni (parsirani) u bytecode oblik prije uključivanja u Javascript aplikaciju. Sukladno tome, preskače se prvi korak parsiranja i skupljanja smeća, a optimizacija je znatno brža pa su Wasm moduli nakon uključivanja gotovo odmah spremni za korištenje.



Slika 4: WebAssembly unutar virtualnog stroja

Posljednje, kao programski jezik koji se interpretira i izvršava u Web pregledniku unutar kontroliranog okruženja (engl. *sandbox*), Javascript ima brojna ograničenja [8]. Zbog sigurnosnih razloga, skripte mogu izvoditi samo zadatke koje se prvenstveno odnose na Web pa stoga ne mogu čitati ili kreirati datoteke u datotečnom sustavu. Nadalje, svaka skripta ima implementiranu politiku istog izvora (engl. *same-origin policy*), pa stoga skripta sa jedne Web stranice nema pristup informacijama druge Web stranice. Zbog istog razloga, skripte ne mogu direktno pristupiti udaljenoj bazi podataka nego se mora slati Web zahtjev (i čekati odgovor) na skriptu koja se nalazi na poslužiteljskoj strani da to učini. Valja napomenuti i da Javascript ne podržava višedretvenost (engl. *multithreading*) nego se sve odvija na glavnoj dretvi što zapravo predstavlja najveću prepreku za izvođenje bilo kakve zahtjevnije stavke.

Na prethodnoj slici vidi se da se Wasm nalazi unutar istog virtualnog stroja kao i Javascript aplikacija, i to znači da za njega vrijede iste restrikcije i sigurnosne politike kao i za Javascript. Prema tome, ukoliko razvojni programer u Wasm-u želi razvijati 2D/3D grafiku onda će biti primoran koristiti WebGL programsko sučelje. Što se tiče sigurnosti i sigurnosnih politika, to znači da je svaki sigurnosni problem vezan uz Javascript (i njegov virtualni stroj) također prisutan i u Wasm-u. Ovo je istovremeno i dobra i loša stvar iz razloga što rješenje jednog sigurnosnog problema na razini Javascript stroja rješava istovremeno i problem na razini Wasm-a, a isto se tako pojava sigurnosnog problema na razini Javascript stroja propagira sve do Wasm-a.

3.2.1. Ciljevi Wasm-a

S obzirom na sve navedene probleme vezane uz Javascript kao tehnologiju koji su bili prisutni prije nastanka Wasm-a (a i danas), pripadnici Mozilla zajednice kao inicijatori ideje morali su puno faktora uzeti u obzir kako Wasm ne bi postao promašaj poput njemu sličnih prethodnih pokušaja (više u poglavlju 3.3.1. Prethodni pokušaji). Stoga, prvenstveni ciljevi

prilikom dizajniranja (engl. *design goals*) formata bili su sljedeći [9]:

a) Biti brz, to jest izvršavati se brzinom približno onoj strojnog kôda (engl. *native code performance*), a pri tome iskorištavajući osnovne mogućnosti svih suvremenih računala.

b) Biti siguran na način da se kôd validira i izvršava u okruženju sigurnom za memoriju, pa s time sprječava neispravnost podataka (engl. *data corruption*) i povrede sigurnosti (engl. *security breach*).

c) Biti neovisan o platformi i sklopovlju na kojem se izvodi, te neovisan o programskom jeziku iz kojeg se prevodi.

d) Biti efikasan tako da se modul može dekodirati, validirati i kompilirati u jednom brzom prolazu kroz njegov sadržaj te da se svaki od tih koraka može izvoditi kao paralelni zadaci, bilo da se radi o JIT ili AOT (engl. *ahead of time*) kompilaciji.

e) Biti modularan na način da se programi mogu podijeliti u manje dijelove koji se mogu odvojeno prenositi, koristiti te spremati u privremenu memoriju (engl. *cache*).

f) Biti interoperabilan tako da programi mogu jednostavno komunicirati sa njihovom okolinom i obrnuto.

Svaki od navedenih ciljeva kao i način njegove realizacije biti će detaljnije objašnjen kroz karakteristike formata u poglavlju 3.4. Karakteristike.

3.2.2. Slučajevi korištenja

Već je nekoliko puta naglašeno koliko Wasm proširuje mogućnosti Javascipt-a (i Web platforme općenito) uzimajući u obzir njegove performanse i interoperabilnost, ali nije puno riječi potrošeno specifično na slučajeve korištenja ove nove tehnologije. Valja naglasiti da ih je stvarno puno te da će cijela lista biti vjerojatno još i duža u vrijeme čitanja ovoga rada. S obzirom da se Wasm može izvršavati unutar te izvan Web preglednika, u nastavku su navedeni neki slučajevi korištenja podijeljeni u te dvije kategorije [3]:

A. **Unutar preglednika**

- Računalne igre (engl. *Gaming*) - klasične Web igre koje se trebaju brzo pokrenuti, koje zahtjevaju veću procesorsku snagu ili obični portali sa Web igrama.
- Računalne biblioteke - prijenos biblioteka implementiranih u Javascriptu na WebAssembly implementaciju radi boljih performansi, općeniti pristup C/C++ stolnim bibliotekama na Web-u, isto vrijedi i za biblioteke koje se izvršavaju u različitim okruženjima i arhitekturama (LibSass) radi univerzalnosti.
- Multimedija - obrada slika i video zapisa, prepoznavanje slika iz izvora, aplikacije za *streaming* glazbe.

- Znanost i tehnologija - bilo kakav oblik znanstvenih vizualizacija i simulacija, virtualna stvarnost (engl. *virtual reality*) i prividna stvarnost (engl. *augmented reality*), CAD alati (engl. *computed-aided design*) za projektiranje tehničkih predmeta.
- Umrežavanje i sigurnost - udaljeno povezivanje računala (engl. *remote desktop connection*), virtualne privatne mreže (engl. *VPN, virtual private network*), enkripcija, postavljanje lokalnog web poslužitelja.

B. **Izvan preglednika**

- Servis za distribuciju računalnih igara (engl. *game distribution service*) radi bolje prenosivosti i sigurnosti.
- Prevođenje nesigurnog kôda na poslužiteljskoj strani i općenito razvoj aplikacija na poslužiteljskoj strani.
- Razvoj hibridnih mobilnih aplikacija (engl. *hybrid native app*) radi boljih performansi zbog izvođenja unutar *webview* komponente.

Osim gore navedenih specifičnih slučajeva korištenja, vrijedi i generalna primjena Wasm-a na način da jednostavno cijeli izvorni kôd (engl. *code base*) postojeće aplikacije prenese u Wasm. To trenutno nije moguće izvesti sa svim postojećim aplikacijama s obzirom na podržane jezike, ali u skorijoj budućnosti će biti moguće (više o ovome u poglavlju 3.5.3. Podržani jezici). Druga generalna primjena je vrlo slična, a to je da se jednostavno samo glavnu funkciju (procesorski najzahtjevniju) aplikacije implementira u Wasm-u dok se korisničko sučelje i dalje implementira u Javascriptu i HTML-u. Prognozira se da će se većina digitalnih agencija u budućnosti okrenuti razvoju ovakvih arhitektura web aplikacija.

3.3. Razvoj Wasm-a

Bilo bi gotovo teško za povjerovati da je Wasm bio prvi pokušaj proširenja Web platforme, jer zasigurno nije. Web platforma ima vrlo zanimljivu povijest, a od njenog postojanja bilo je nekoliko pokušaja proširenja kako bi podržavala različite programske jezike. Nezgrapna rješenja poput priključaka (engl. *plugins*) bila su iza svojeg vremena, a druga rješenja koja su ograničavala korisnika na jedan preglednik bila su pravi recept za katastrofu [7]. U ovom poglavlju biti će spomenuti neki od najpoznatijih prethodnih pokušaja uz njihove nedostatke, te na kraju ukratko opisan razvoj Wasm-a i način na koji on popunjuje te nedostatke.

3.3.1. Prethodni pokušaji

U ovom potpoglavlju biti će opisani neki od najpoznatijih prethodnih pokušaja proširenja Web platforme, a izdvojene su tehnologije i alati koji su svojevremeno bili najinovativniji i samim time imali najveći potencijal postati Web standard. Uz to, navedeni su i glavni problemi svakog od njih koji su ih spriječavali u ostvarivanju spomenutog cilja.

Prvi poznatiji pokušaj datira još iz 1996. godine kada je Microsoft razvio tehnologiju pod nazivom **ActiveX** za potpisivanje (engl. *code signing*) binarnih datoteka za x86 arhitekture kako bi

se kasnije pokretale na Web pregledniku [10]. Zasnivala se u potpunosti na potpisivanju kôda i zbog toga se njena sigurnost nije ostvarivala kroz tehničku specifikaciju tehnologije kao takve, već kroz princip povjerenja što je bio i glavni nedostatak tehnologije jer su binarne datoteke imale nesmetani pristup svim programskim sučeljima Windows platforme [11]. Možda je pregrubo reći da je ActiveX bio promašaj jer je zasigurno predstavljao inovaciju u svoje vrijeme, ali s obzirom na nedostatak sigurnosti i činjenicu da je ograničen samo na Windows platformu i x86 arhitekture kao takav nije bio dovoljno zreo da zaživi kao Web standard.

Slijedeći pokušaj vrijedan spomena bio je Google-ov **Native Client** objavljen 2011. godine, ali je prvo stabilno izdanje izašlo tek 2015. godine. Native Client (NaCl) je bio prvi sustav koji je uveo *sandboxing* tehniku za strojni kôd koji se izvodi na Web-u brzinom približnom onoj izvornog strojnog kôda [10]. Oslanja se na statičku validaciju strojnog kôda, pa stoga radi na principu da njegov generator kôda prati određene uzorke i zbog toga podržava samo podskupove x86, ARM i MIPS skupova instrukcija u pregledniku. Jednostavnije rečeno, NaCl je bio u mogućnosti ubaciti modul sa strojnim kôdom specifičnim platformi u Web preglednik kako bi se izvršavao u ograničenom okruženju. S obzirom da je strojni kôd bio specifičan platformi, on inicijalno nije bio prenosiv pa je nedugo zatim razvijen **Portable Native Client** (PNaCl) koji je bio neovisan o arhitekturi na kojoj se izvršava [12]. Princip rada PNaCl tehnologije nešto je sličan Wasm-u jer su aplikacije prevodile prije korištenja (engl. *AOT, ahead-of-time*), a sastojao se od dva elementa [7]:

- ulančanih alata (engl. *toolchains*) koji transformiraju C/C++ kôd u NaCl module.
- izvršne komponente uključene u preglednik koje omogućavaju izvršavanje NaCl modula.

Potrebno je naglasiti da su se NaCl moduli mogli koristiti samo unutar aplikacija i ekstenzija koje su instalirane iz web trgovine na Google Chrome pregledniku, dok su se PNaCl moduli mogli koristiti u svakom pregledniku zahvaljujući Pepperu. Pepper je bilo aplikacijsko programsko sučelje (API) otvorenog kôda koje je omogućavalo komunikaciju između NaCl modula i web preglednika uz mogućnost pristupa sistemskim funkcijama niske razine na siguran način [12]. Zapravo je glavni nedostatak bio taj što je jedino Google Chrome web preglednik imao već uključenu podršku za PNaCl i Pepper, dok ostali glavni web preglednici nisu i zahtjevali su ručnu instalaciju i podešavanje istih. Mozilla je, naprimjer, svoj fokus usmjerila u razvoj *asm.js* tehnologije (više o ovome u sljedećem potpoglavlju) te su odbijali pružiti podršku za Pepper zbog nepotpune specifikacije i dokumentacije. Naposlijetku je Google odlučio odustati od razvoja i održavanja tehnologije što zbog navedenih nedostataka a što zbog pojave WebAssemblya, te je tako Native Client u svibnju 2017. godine proglašen zastarjelim (engl. *deprecated*) [12].

Valja spomenuti i **Emscripten** iako se ne može svrstati u samo "propale pokušaje" kao i navedeni prethodnici jer nije imao isti cilj kao oni, a bio je vrlo bitan za razvoj Wasm-a. Radi se o programskom okviru za prevođenje C/C++ aplikacija u Javascript uz povezivanje istih sa izvršnim okruženjem (engl. *execution environment*) implementiranim u Javascriptu [10]. Kôd se prevodi u posebni podskup Javascripta koji se kasnije razvio u asm.js (detaljnije o ovome u sljedećem potpoglavlju). Njegov temeljni cilj nije bio da proširi web platformu, već da omogući prenošenje postojećih C/C++ aplikacija (ili neke dijelove) u klijentsku stranu web aplikacije. Ko-

```c
// C kôd
int f(int i) {
  return i + 1;
}
```

```c
// Asm.js kôd
function f(i) {
  i = i | 0;
  return (i + 1) | 0;
}
```

Isječak kôda 3.1: C funkcija i njen asm.js ekvivalent

liko je zapravo moćan govori i činjenica da je bio najzaslužniji za prenošenje tehnologija poput Unreal Engine 3, SQLite, MeshLab, AutoCAD, Figma i drugih na web, a koristi se i dan danas [13].

Kroz prethodne primjere bilo je moguće vidjeti da su zapravo svi pokušaji imali brojne prednosti i samo par nedostataka. Međutim, riješavanje tih par nedostataka uvijek je bilo ključno za ostvarivanje cilja tehnologije, a to je postati web standard. Pokušaji ispunjavanja tih bitnih nedostataka uvijek su razvojne programere dovodili do drugih problema ili izbora da se jedna prednost "žrtvuje" s ciljem ispunjena tog nedostatka. Na primjer, Native Client inicijalno nije zahtjevao dodatna sučelja za rad ali nije bio prenosiv. Njegov nasljednik Portable Native Client je bio prenosiv ali je ovisio o Pepper sučelju i samim time zahtjevao mnogo podešavanja. Moguće je zaključiti da je zapravo glavni problem bio taj što se prilikom dizajniranja tehnologije uvijek razmišljalo u smjeru razvoja nekakvog alata ili dodatka koji bi predstavljao sučelje za izvršavanja strojnog kôda stolnih aplikacija u pregledniku, umjesto da se razmišljalo o iskorištavanju postojećih sučelja (Javascript API) i razvoju univerzalnog formata - baš kao što je WebAssembly.

3.3.2. Od asm.js-a do minimalno održivog proizvoda

Već je nekoliko puta do sada spomenut asm.js i njegov utjecaj u vezi sa razvojem Wasm-a, ali nigdje nije detaljnije opisano o čemu se zapravo ovdje radi.

Asm.js je strogi podskup Javascript-a koji može biti korišten kao jezik pretvorbe (engl. *target language*) niske razine za prevoditelj [14]. Razvila ga je kompanija Mozilla još 2013. godine s ciljem prevođenja izvornog C/C++ kôda u Javascript, ali sa zadržavanjem razine performansi znatno višim nego one razine Javascript-a. Izvorni jezik mora imati statičke tipove podataka i ručno upravljanje memorijom (engl. *manual memory management*), a prevodi se najčešće uz pomoć prevoditelja kao Emscripten koji omogućuje kompilaciju jednog izvornog kôda u drugi (engl. *source-to-source*). Navedeni uvjeti omogućavaju inicijalno poboljšanje performansi, a dodatno podizanje performansi omogućava ograničavanje osobina jezika na isključivo one koje su pogodne za AOT optimizaciju. Stoga, podskup dozvoljava samo brojeve, *if* i *while* te ostale jednostavne konstrukte. Nadalje, ne dozvoljava objekte, *string*ove, *closure* ili bilo kakve konstrukte koji zahtjevaju alokaciju memorije hrpe (engl. *heap allocation*). Kako izgleda ekvivalent jednostavne C funkcije nakon kompilacije u asm.js prikazano je u nastavku:

Primjetite dodavanje *bitwise* operatora (|0) na kraju instrukcija koji pretvara svoje operande u 32-bitne pozitivne cijele brojeve (engl. *signed integers*) i vraća rezultat u obliku cijelog broja. Uz

pomoć ovog naizgled beskorisnog operatora moguće je pretvoriti vrijednost u cijeli broj. Ukoliko se primjeni na svaki parametar, osigurava da svaka vrijednost bude pretvorena u odgovarajući tip kada funkcija bude pozvana i na taj način riješi problem slabih tipova podataka u Javascriptu [15]. Uz pomoć tih pretvorbi optimizirajući prevoditelj onda može generirati mnogo učinkovitiji nativni kôd. Za vrijeme koraka optimizicije, kada jedan dio asm.js kôda zove drugi dio asm.js kôda tada neće biti potrebe za pretvorbom tipova podataka jer će svi podaci sigurno biti odgovarajućeg tipa i s time će znatno ubrzati optimizaciju. Isto tako, ovakav oblik kôda daje alatima poput Emscripten-a jasne upute kakav kôd točno treba generirati.

U ožujku 2013. godine, na *Game Developers* konferenciji, Mozilla je objavila da su uspješno prenijeli Unreal Engine okruženje za izgradnju računalnih igara na Web platformu tako što su preveli više od milijun linija C++ izvornog kôda uz pomoć Emscripten alata, a ono što je još bitnije - okruženje je na preglednicima radilo glatko i učinkovito bez potrebe za dodatnim priključcima [16]. Kako bi demonstrirali to, u suradnji sa Epic Games kompanijom razvili su *Epic Citadel* demo igru koja je koristila navedeno okruženje te ga potom predstavili široj publici na konferenciji. Uz to, prikazali su i isječke iz *Sanctuary* demo igre koja je čak i s nekoliko desetaka igrača upravljanih umjetnom inteligencijom na Mozilla Firefox pregledniku radila besprijekorno, što je publiku ostavilo bez daha.

Slika 5: Unreal Engine unutar Epic Citadel demo igre [17]

Unatoč trijumfu na konferenciji i sve većoj popularnosti na tržištu, Mozilla je bila svjesna da projekt ovdje ne završava i da postoji još brdo stvari na kojima će morati poraditi ukoliko žele ostvariti svoju misiju. To se najviše odnosi na nedostatke asm.js jezika koji nikako nisu mogli biti zanemareni [18]:

- svi naputci o tipu podataka (kao bitwise operator iznad) mogu učiniti veličinu datoteke znatno većom
- dobivena asm.js datoteka je Javascript formata, pa i dalje treba biti učitana i parsirana

od strane Javascript stroja pa stoga može znatno usporiti rad na mobilnom uređaju i prouzrokovati veću potrošnju baterije

- dodavanje novih osobina nije moguće bez modificiranja Javascript jezika kao takvog
- Javascript je programski jezik i nije bio zamišljen da postane cilj prevođenja (kao što je asm.js)

S time na umu, odlučili su proglasiti minimalni održivi proizvod (engl. *MVP, minimum viable product)* pod nazivom WebAssembly koji je trebao iskoristiti sve pozitivne aspekte asm.js jezika pri tome uzimajući u obzir i njegove nedostatke. U razvoju softvera, minimalni održivi proizvod predstavlja inkrement proizvoda sa minimalnim osobinama dovoljnim da zadovolji rane korisnike i pruži povratnu informaciju za daljni razvoj. I tako je u travnju 2015. godine W3 konzorcij formirao skupinu pod nazivom *WebAssembly Working Group* koja je bila zadužena za standardiziranje formata i nadgledanje specifikacije te cjelokupnog procesa prijedloga [7]. Inicijalna implementacija podrške za WebAssembly formata u web preglednicima temeljila se na asm.js jeziku, a u narednom periodu je izdana temeljna specifikacija te dokumentacija za Javascript API i Web API. Zahvaljujući dobroj suradnji sa pružateljima glavnih web preglednika, Mozilla je u sljedeće 4 godine uspjela popraviti sve prvobitne nedostatke što je rezultiralo konačnom standardizacijom formata u prosincu 2019. godine. Na koji način se pristupalo u riješavanju svakog pojedinog nedostatka biti će detaljnije objašnjeno u sljedećem potpoglavljima 3.4.1. Virtualni stroj i 3.4.3. Formati.

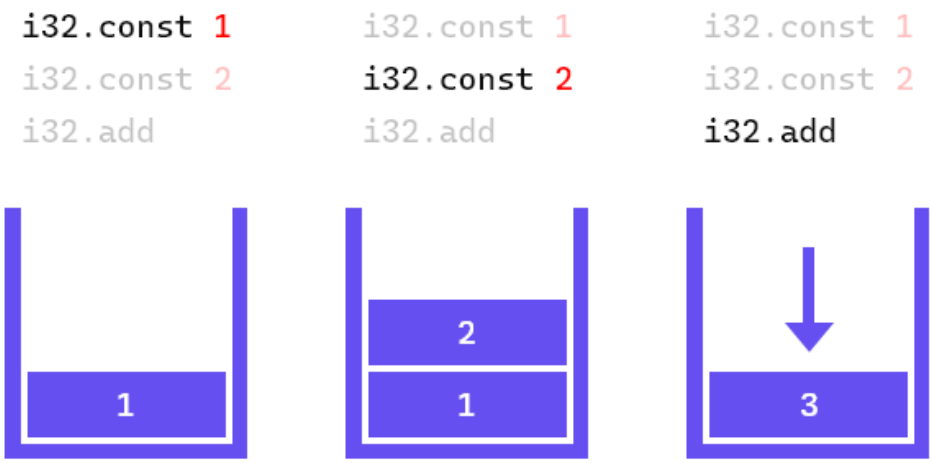
3.4. Karakteristike

Do sada se ponajviše govorilo o povijesti Wasm-a, prethodnim pokušajima i primjerenim slučajevima korištenja dok su karakteristike formata bile pojašnjene samo površno. Ukratko, WebAssembly je binarni instrukcijski format koji se izvodi u virtualnom stroju temeljenom na stogu, a sadrži samo 4 tipa podatka i 67 različitih instrukcija. Dizajniran je na način da podržava kompilaciju u toku (engl. *streaming compilation*), a to znači da Wasm kôd struji putem HTTP-a i samim time je moguće kompilirati Wasm modul kroz više dretvi (samo kompilirati, ne i izvršavati). Nadalje, koristi vrlo jednostavna validacijska pravila za razliku od Javascripta, a podržava uvoz i izvoz funkcija kako bi se mogle dijeliti funkcionalnosti sa istim. To je moguće zahvaljujući činjenici da ima linearnu memoriju koju također dijeli sa Javascriptom. Detaljnije o ovome svemu slijedi u slijedećim potpoglavljima.

3.4.1. Virtualni stroj

U nekim literaturama se spominje kako je Wasm virtualni stroj temeljen na stogu, dok u nekima piše da je on samo instrukcijski format koji se izvodi na virtualnom stroju. Zapravo su obje definicije točne jer je on i jedno i drugo, odnosno instrukcijski format koji se izvodi na prenosivom virtualnom stroju i samim time ne ide jedno bez drugoga [19].

Općenito u računarstvu, virtualni stroj oponaša stvarni računalni sustav u smislu njegove arhitekture i funkcionalnosti ali ne i u fizičkom smislu [20]. U kontekstu Wasm-a, odnosi se na procesni virtualni stroj koji je dizajniran da izvršava programe neovisno o platformi na kojoj se izvodi. Najbolji primjer takvoga stroja je Java virtualni stroj (engl. *Java Virtual Machine*) zahvaljujući kojem se Java kao programski jezik može izvoditi neovisno o platformi. Nadalje, **virtualni stroj temeljen na stogu** sastoji se od dva elementa: instrukcija i stoga, a razlikuje se od tipičnog virtualnog stroja po tome što njegove instrukcije operiraju nad stogom (strukturom), a ne nad brojevima koji se nalaze u registrima unutar procesora [21]. Kako na primjer izgleda zbrajanje dvaju brojeva unutar ovog virtualnog stroja prikazano je na slijedećoj slici:



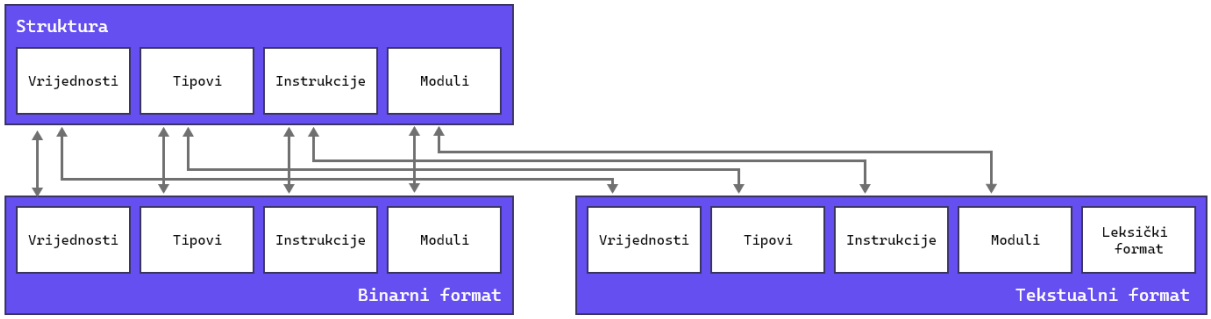
Slika 6: Zbrajanje dvaju brojeva na stogu

S obzirom da su operandi korišteni unutar instrukcije uvijek na poznatom mjestu (na vrhu stoga), instrukcije kao takve ne zahtjevaju memorijske adrese operanada za izvršavanje. Zbog takve arhitekture skupova instrukcija (engl. *instruction set architecture*) se Wasm kao format naziva i **format s nultom adresom** (engl. *zero address format*). S obzirom da se stog kao struktura podataka uvelike koristi u programskim jezicima visoke razine, ovakav tip stroja može oponašati rad programa koji se izvodi na njima što dovodi do znatno boljih performansi.

Ono što je najbitnije naglasiti kod Wasm virtualnog stroja (ili skraćeno WAVM-a) je to da on koristi **LLVM** (engl. *low-level virtual machine* infrastrukturu za kompilaciju WebAssembly kôda u strojni kôd visokih performansi. Ukratko, LLVM infrastruktura predstavlja skup modularnih i ponovno iskoristivih prevoditelja, lanaca alata i drugih tehnologija. Nastavno za performanse strojnog kôda, on može čak i nadjačati performanse izvornih računala u nekim slučajevima zahvaljujući sposobnosti da generira strojni kôd specifično prilagođen za procesor koji izvodi kôd [22]. Isto tako, WAVM spriječava da Wasm kôd pristupi stanju izvan virtualnog stroja i njegove memorije, a isto tako ne dozvoljava poziv funkcije nekog izvornog kôda koji se nije eksplicitno vezao sa Wasm modulom i na taj način ostvaruje sigurnost. Osim za sigurnosti i visoke performanse, WAVM je zaslužan za prenosivost cijelog formata iz razloga što je velik dio virtualnog stroja izvorno napisan u prenosivom C/C++ programskom jeziku.

3.4.2. Formati

WebAssembly kao programski jezik ima dvije konkretne reprezentacije: binarni format i tekstualni format. Oba formata dijele zajedničku strukturu koja je prikazana na Slici 7. Unutar specifikacije Wasm-a se zbog konzistentnosti i jednostavnosti ova struktura modelira kao **abstraktna sintaksta** (engl. *abstract syntax*), dok stvarna reprezentacija može biti kompilirani strojni kôd [19]. Bitno je spomenuti da unutar Wasm-a postoji mehanizam koji se zove atributna gramatika (engl. *attribute grammar*) unutar kojeg su definirana pravila za generiranje svakog formata.



Slika 7: Mapiranje formata

3.4.2.1. Binarni format

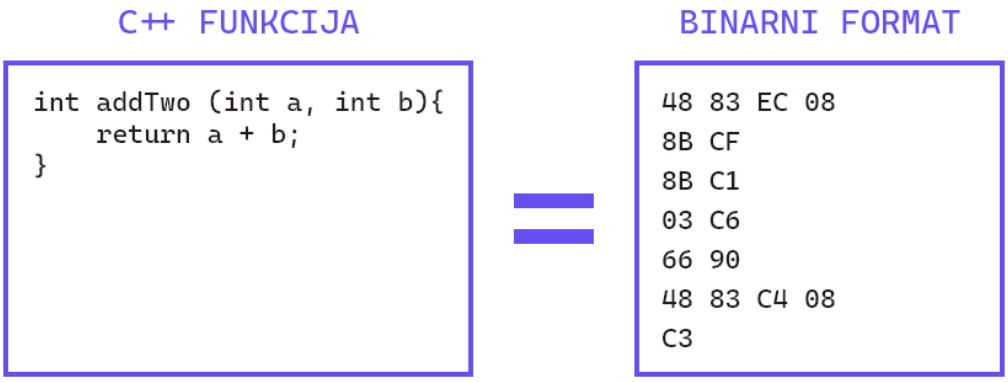
Binarni format je definiran kao rezultat "čvrstog" (engl. *dense*) enkodiranja abstraktne sintakse WebAssembly modula [19]. Jednostavnije rečeno, to znači da je učinkovitiji oblik binarnog zapisa koji omogućuje brže dekodiranje, manje korištenje memorije i u pravilu je znatno manji od tekstualnog formata. Format je definiran isključivo od nizova bajtova, a oni su unutar abstraktne sintakse prikazani kao heksadecimalne vrijednosti (vidi Sliku 8.).

Format je podijeljen u sekcije ovisno o entitetima deklariranim unutar njega (detaljnije o ovome u poglavlju 3.4.3.1. Moduli). Općenito vrijedi da su tipovi funkcija odvojeni u zasebnu sekciju kako bi dozvoljavali dijeljenje memorije. Međutim, tijela funkcija su također u zasebno sekciji nakon deklaracija kako bi se smanjilo vrijeme čekanja da se učita cijela stranica tako što započne kompilacija u toku čim stroj naiđe na tijelo funkcije.

Napomena koju valja imati na umu prilikom uključivanja Wasm modula u Javascript aplikaciju je da je preporučena ekstenzija za datoteke koje sadrže WebAssembly module u binarnom formatu **".wasm"**, a preporučeni MediaType **"application/wasm"** [19].

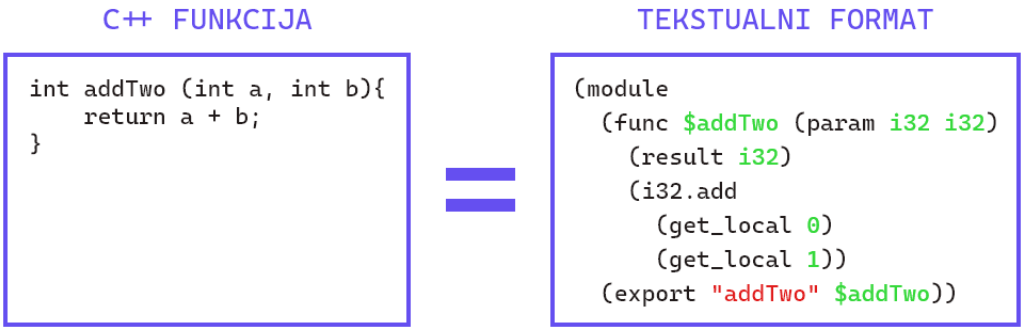
3.4.2.2. Tekstualni format

Tekstualni format je rezultat prevođenja abstraktne sintakse WebAssembly modula u S-izraze (engl. *S-expressions*) [19]. Simbolički izrazi, ili skraćeno S-izrazi, su oblik notacije za ugniježđene liste (strukturirane u obliku stabla). Ukratko, potječu iz programskog jezika *Lisp* i pružaju način za jednostavni prikaz strukture podataka temeljene na listi u tekstualnom obliku



Slika 8: C++ funkcija i njen binarni ekvivalent

(vidi Sliku 9.). Nadalje, tekstualni zapis unutar datoteke je pravilno oblikovan opis WebAssembly modula ako i samo ako je generiran od strane atributne gramatike.



Slika 9: C++ funkcija i njen tekstualni ekvivalent

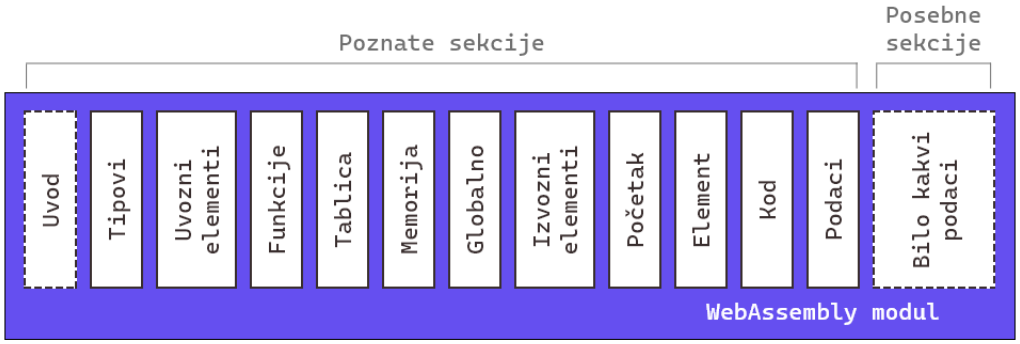
Na samim početcima WebAssemblya kao projekta, mnogi su se pitali koja je korist od ove reprezentacija jezika ukoliko se ne može direktno uključivati u Web platformu. Zašto razvijati još jednu atributnu gramatiku koja će generirati ovaj format i uz to dodatno zakrčiti dokumentaciju formata? Razlog je zapravo vrlo jednostavan, a to je omogućivanje **pronalaženje grešaka** (engl. *debugging*). Zbog implementacije formata kao takve, nije moguće direktno pronalaziti greške u izvornom kôdu. S druge strane, moguće je pronalaziti greške unutar binarnog formata ali takav alat ne bi imao previše smisla jer razvojnom programeru ništa ne znači jedna memorijska lokacija kao npr. '8B C1' u moru heksadekadskih vrijednosti. Tekstualni format omogućuje uvid u izvorni kôd modula koji je uključen u preglednik i na taj način se eliminira problem crne kutije koji je postojao u navedenom NaCl alatu [7]. Isto tako, omogućuje prezentaciju u web preglednicima koji nemaju standardno podržano mapiranje izvornog kôda (engl. *source maps*). Više o mapiranju izvornog kôda i pronalaženju grešaka u poglavlju 3.4.6. Validacija.

Preporučena ekstenzija za datoteke koje sadržavaju WebAssembly modul u tekstualnom formatu je **".wat"**. Isto valja napomenuti da se datoteke sa ovom ekstenzijom uvijek enkodiraju u UTF-8 formatu.

3.4.3. Semantika

3.4.3.1. Moduli

Modul se definira kao prenosiva i izvršna jedinica kôda koju je moguće učitati [7]. Pod modulom se najčešće podrazumijeva .wasm datoteka u binarnom formatu. Kao takav predstavlja statičku reprezentaciju programa, ali ga je moguće instancirati (više u idućem potpoglavlju) i dobiti instancu modula sa promjenjivom memorijom i stogom izvršavanja (engl. *execution stack*). Operaciju instantacije izvršava ugrađivač (engl. *embedder*) kao što je naprimjer Javascript virtualni stroj ili operacijski sustav [10]. Modul je podijeljen u sekcije koje su prikazane na sljedećoj slici:



Slika 10: Struktura WebAssembly modula

Svaka sekcija sastoji se od identifikatora (1 bajt), veličine u bajtovima (4 bajta) i sadržaja koji ovisi od sekcije do sekcije. Važno je napomenuti da uvodna sekcija na početku sadrži jednu posebnu vrijednost (`0x00 0x61 0x73 06D`) koja ugrađivaču daje do znanja da se radi o Wasm modulu i prema kojoj ga je moguće razlikovati od ES6 modula. Nakon toga slijedi vrijednost `0x01 0x00 0x00 0x00` koja označava verziju WebAssembly formata (u ovom slučaju, verziju 1.0). Ono što je također zanimljivo je to da je svaka sekcija opcionalna, pa je tehnički moguće imati modul bez sekcija.

Poznate sekcije se uključuju samo jednom i moraju se pojavljivati u određenom redoslijedu (kao na Slici 10.). Svaka poznata sekcija ima svrhu, a najbitnije su **Uvozni elementi** (engl. *Imports*), **Funkcije** (engl. *Functions*) i **Izvozni elementi** (engl. *Exports*). Uvozni elementi zapravo predstavljaju elemente iz drugih modula kojima se može pristupiti i mogu biti jedno od sljedećeg: funkcije, globalne vrijednosti, linearna memorija i tablice. Isto tako, sekcija Izvozni elementi sadrži elemente koje je moguće izvesti na korištenje drugim modulima ili Javascript API-u. Već je rečeno da se deklaracije funkcija nalaze na početku modula i zapravo samo one se nalaze u sekciji Funkcije, dok se tijela funkcija nalaze u sekciji Kôd (engl. *Code*). Posebne sekcije pružaju način za uključivanje podataka unutar modula koji se ne koriste u poznatim sekcijama [18]. Zbog toga, one se mogu pojavljivati bilo gdje unutar modula odnosno prije, nakon ili između bilo koje dvije sekcije. Za razliku od poznatih sekcija, ukoliko posebna sekcija nije dobro formatirana ona neće vratiti validacijsku grešku.

S obzirom da su moduli kao takvi statički i ne mogu samostalno uvoziti/izvoziti konstrukte,

WebAssembly omogućuje **dinamičko povezivanje** (engl. *dynamic linking*) modula za vrijeme učitavanja i za vrijeme izvođenja kako bi instancirani moduli mogli dijeliti funkcije, linearnu memoriju, tablice i konstante [3]. Nadalje, s obzirom da je način na koji se moduli učitavaju i instanciraju definiran unutar Javascript API-a (ili drugog okruženja na kojem se nalazi), ono kao takvo mora i pružati način za dinamičko instanciranje modula tako što se povežu svi uvozi modula A sa izvozima modula B i obrnuto.

3.4.3.2. Semantičke faze

U prethodnom potpoglavlju spomenuto je da je Wasm modul moguće instancirati i dobiti njegovu dinamičku reprezentaciju, ali potrebno je napomenuti da je to samo jedna od semantičkih faza u malo složenijem životnom ciklusu modula. Naime, postoje tri semantičke faze:

A. **Dekodiranje** (engl. *decoding*) - S obzirom da su Wasm moduli u virtualnom stroju distribuiraju u binarnom obliku, ova faza je zadužena za pretvorbu tog formata u njegovu internu reprezentaciju. Dobivena reprezentacija je modelirana od strane abstraktne sintakse (kako je objašnjeno u poglavlju 3.4.2. Formati).

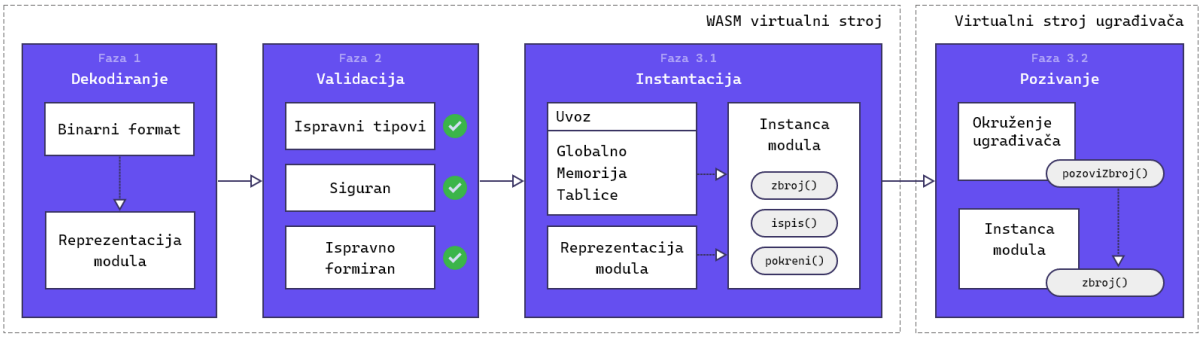
B. **Validacija** (engl. *validation*) - Nakon dekodiranja, virtualni stroj mora provjeriti da li je modul ispravan. To podrazumijeva niz validacijskih provjera kao što su provjere tipova podataka (engl. *type checking*), provjera da li je modul ispravno formiran (da li su sekcije u dobrom redoslijedu) i drugih kako bi se utvrdilo da je modul siguran.

C. **Izvršavanje** (engl. *execution*) - Naposlijetku, ispravan modul može napokon biti izvršen. Ova faza je opet podijeljena u dvije podfaze:

   (a) **Instanciranje** (engl. *instantiation*) - Modul je instanciran i u dinamičkom obliku, što znači da ima svoje stanje i stog izvršavanja [19]. Unutar virtualnog stroja, rezultat ove podfaze biti će skup svih izvezenih funkcija iz modula koji postaju dostupni na korištenje Javascript API-u kao i njegova memorija koju je moguće dijeliti.

   (b) **Pozivanje** (engl. *invocation*) - U ovoj fazi moguće je pozivanje izvezenih funkcija nad instancom modula. Ukoliko su funkciji dani ispravni argumenti, stroj će ju izvršiti i vratiti odgovarajući rezultat.

Potrebno je napomenuti da se posljednja faza izvodi u okruženju koje ugrađuje Wasm (npr. Javascript API), a ne unutar Wasm virtualnog stroja kao prve dvije spomenute faze. U nastavku slijedi jednostavniji dijagramski prikaz semantičkih faza modula:



Slika 11: Semantičke faze Wasm modula

3.4.3.3. Tipovi podataka

WebAssembly ima samo 4 ugrađena tipa podataka, i to su zapravo oni tipovi podataka koji su dostupni u svakom osnovnom ili složenom računalu:

A. `i32` - cjelobrojni tip podatka širine 32 bita, koristi se za adrese u linearnoj memoriji i indeksima u tablicama funkcija

B. `i64` - cjelobrojni tip podatka širine 64 bita

C. `f32` - tip podatka za reprezentaciju brojeva sa pomičnim zarezom širine 32 bita, odnosno jednostruke preciznosti (engl. *single precision*) prema IEEE 754-2019 standardu

D. `f64` - tip podatka za reprezentaciju brojeva sa pomičnim zarezom širine 64 bita, odnosno dvostruke preciznosti (engl. *double precision*) prema IEEE 754-2019 standardu

Valja napomenuti da Wasm, kao i ostali strojevi, ne može razlikovati brojeve sa negativnim predznakom i one bez predznaka (odnosno pozitivne brojeve). Umjesto toga, njihova distinkcija se radi uz pomoć sufiksa `_u` (koji stoji za *unsigned*) za vrijednost bez predznaka ili `_s` (koji stoji za *signed*) za vrijednost sa predznakom. Nadalje, `boolean` vrijednosti reprezentirane su koristeći `i32` tip podatka gdje vrijednost 0 predstavlja lažnu tvrdnju (engl. *false*), a druga svaka vrijednost različita od nule predstavlja istinitu tvrdnju (engl. *true*). Međutim, postavlja se pitanje kako onda Wasm prikazuje druge tipove podataka poput nizova, polja ili objekata? Za sada, potrebno je znati da takva akcija zahtjeva alociranje memorije i stoga će detaljno objašnjenje biti dano u sljedećem potpoglavlju 3.4.3.4 Memorija.

3.4.3.4. Memorija

Glavno mjesto pohrane svakog WebAssembly programa je zapravo jedno veliko polje (engl. *array*) bajtova koje se najčešće naziva **linearna memorija** (engl. *linear memory*) ili samo memorija [10]. Kao što je već rečeno, svaki modul ima samo jednu linearnu memoriju ali ona može biti dijeljena s drugim modulima uz pomoć dinamičkog povezivanja. Potrebno je istaknuti da naziv 'linearna' u ovom kontekstu znači da se memorija inicijalizira i proširuje

fiksnom veličinom. Jedinica memorije je stranica (engl. *page*) i ona je uvijek veličine 64 KiB. Ovakva struktura memorije je ključna u sigurnosnom aspektu Wasm-a iz razloga što moduli kao takvi nemaju direktan pristup memoriji sustava na kojem se izvode. Umjesto toga, modul se inicijalizira sa inicijalnom veličinom memorije i po potrebi proširuje. Nadalje, linearna memorija funkcionira isto kao struktura hrpe u C/C++ jeziku izuzev toga što program uvijek prije pristupa memoriji provjerava je li tražena vrijednost izvan njenih granica (engl. *out of bounds*). Isto tako, veličina stranice nije specifična za svaki operacijski sustav jer bi dinamička veličina mogla uzrokovati probleme sa prenosivosti formata.

Zbog ovakve strukture memorije su zapravo najpogodniji jezici za rad sa Wasm-om bili C/C++ jer oni imaju ugrađeno ručno upravljanje memorijom (kao i Wasm). Shodno tome, C/C++ kao ni Wasm nemaju automatsko upravljanje memorijom odnosno nešto što se naziva **skupljanje smeća** (engl. *garbage collection*). Skupljanje smeća je oblik automatiziranog upravljanja memorijom koji omogućuje da se memorija objekata koji se više ne koriste u programu automatski dodijeli dalje po potrebi. Zbog toga Wasm (kao i C/C++) za reprezentaciju složenijih tipova podataka zahtjeva alokaciju memorije. Naprimjer, za prikaz niza znakova (engl. *string*) Wasm će svaki pojedini znak spremiti kao 1 bajt i u konačnici će rezultirati pohranom niza bajtova u memoriji. Prilikom učitavanja modula u Javascript, Wasm će sučelju prenijeti indeks lokacije te funkcije u linearnoj memoriji kako bi Javascript stroj mogao znati gdje se u dijeljenoj memoriji nalazi tražena funkcija te međuspremnik (engl. *buffer*) iz nje [4]. Međuspremnik se unutar Javascript API-a implementira uz pomoć `ArrayBuffer` konstrukta koji također sadrži sučelja za dohvaćanje i proširivanje memorije. Posljednje, potrebno je unutar programa iterirati kroz međuspremnik, bajt po bajt, te dohvaćati tekstualnu vrijednost iz Unicode kôda uz pomoć statičke metode `String.fromCharCode()` i nakraju sve znakove ulančati (engl. *concatenate*) u jedan niz znakova. Razvojni inženjeri formata Wasm svjesni su složenosti ovog procesa i zbog toga planiraju u skorijoj budućnosti implementirati referentne tipove podataka (više o planu načina implementacije u poglavlju 3.6. Budućnost Wasm-a).

3.4.3.5. Instrukcije

Kao što je već spomenuto, format WebAssembly definira 67 različitih instrukcija. One operiraju unutar stroja koji se temelji na stogovima na način da upravljaju vrijednostima koje se nalaze na stogu operanada (engl. *operand stack*), pri tome uzimajući (engl. *pop*) vrijednosti ma vrhu stoga i vraćajući (engl. *push*) rezultat na stog (vidi Sliku 6). One zapravo predstavljaju akcije koje određuju što treba raditi sa stogom, a ako bi to točno odredile svaka instrukcija mora imati definiran tip funkcije. Tip funkcije opisuje potrebne tipove podataka potrebnih argumenata (vrijednosti koje se uzimaju sa stoga) i tip podatka rezultata koji se vraća (vrijednosti koju se stavlja na stog). Naprimjer, instrukcija zbrajanja `i32.add` ima tip `[i32 i32]` $\longrightarrow$ `[i32]` što znači da sa stoga uzima dvije vrijednosti tipa `i32` i vraća jednu.

Nadalje, prema specifikaciji instrukcije su podijeljene u 6 kategorija [19]:

1) Numeričke instrukcije (engl. *numeric instructions*) - pružaju osnovne operacije nad numeričkim vrijednostima određenog tipa. Za svaki tip opet postoje nekoliko podkategorija:

(a) Konstante - vraćaju statičku konstantu

(b) Unarne operacije - uzimaju jednu vrijednost sa stoga i na stog stavljaju jednu vrijednost odgovarajućeg tipa

(c) Binarne operacije - uzimaju dvije vrijednosti sa stoga i na stog stavljaju jednu vrijednost odgovarajućeg tipa

(d) Testovi - uzimaju jednu vrijednost odgovarajućeg tipa i vraćaju Boolean cjelobrojnu vrijednost (0 ili 1)

(e) Usporedbe - uzimaju dvije vrijednosti odgovarajućeg tipa i vraćaju Boolean cjelobrojnu vrijednost (0 ili 1)

(f) Pretvorbe - uzimaju sa stoga vrijednost jednog tipa i na stog vraćaju vrijednost drugoga tipa

Valja i spomenuti da prve tri spomenute podkategorije dolaze u 2 oblika koji se razlikuju po anotaciji `sx` koja određuje trebaju li se operandi interpretirati kao pozitivni ili negativni cjelobrojni tipovi.

2) Parametarske instrukcije - operiraju nad parametrima (odbacivanje i odabiranje operanada)

3) Instrukcije varijabli - služe za postavljanje i dohvaćanje lokalnih i globalnih varijabli

4) Memorijske instrukcije - služe za dohvaćanje i pohranjivanje u memoriju te proširivanje iste

5) Kontrolne instrukcije - određuju tijek izvođenja programa. Ovdje valja izvdojiti `if`, `block` i `loop` strukturne instrukcije čija je zadaća jednaka istoimenim konstruktima u svijetu računarstva

6) Izrazi - ovdje spadaju tijela funkcija i inicijalizacijske vrijednosti za globalne varijable koje se zapravo interpretiraju kao niz instrukcija završen sa ključnom riječi `end`

Kategorija	**Popis instrukcija**
Numeričke instrukcije	clz, ctz, popcnt, add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr, abs, neg, ceil, floor, trunc, nearest, sqrt, min, max, copysign, eqz, eq, ne, lt, gt, ge, le, wrap, convert, promote, demote, reinterpret, extend
Parametarske instrukcije	drop, select
Instrukcije varijabli	local.get, local.set, local.tee, global.get, global.set
Memorijske instrukcije	load, load_8, load_16, load_32, store, store_8, store_16, store_32, memory.size, memory.grow
Kontrolne instrukcije	block, loop, if, unreachable, nop, br, br_if, br_table, return, call, call_indirect
Izrazi	bilo koji niz instrukcija

Tablica 1: Popis instrukcija prema kategorijama

U tablici iznad moguće je vidjeti popis svih instrukcija podijeljenih prema već opisanim kategorijama. Zbog njihovog broja one nisu u istoj tablici i opisane, ali je iz njihovog naziva moguće zaključiti o kojoj se odgovarajućoj instrukciji sklopovlja radi. Na primjer, operacija `add` odgovara operaciji zbrajanja, operacija `sub` odgovara operaciji oduzimanja itd.

3.4.4. Nedeterminizam

U računarstvu, programski jezik se kategorizira kao nedeterministički ako u svom radu može definirati tzv. "točke izbora" (engl. *choice points*), odnosno alternative izvođenja toka programa. Za razliku od `if-else` naredbe, odabir alternative nije specificiran od strane programera već od strane programa koji u vrijeme izvođenja mora odabrati alternativu uz pomoć neke od generalnih metoda [23]. Programer kroz programski kôd direktno ili indirektno definira alternative, a program mora kasnije izabrati jednu od njih. Bitno je naglasiti da je dizajn programskog jezika zapravo ono što ga čini nedeterminističkim. Zahvaljujući dizajnu jezika, točnije načinu izvođenja nije uvijek moguće točno odrediti koji je sljedeći korak u programu pa se stoga javljaju već spomenute točke izbora. Najčešća metoda izbora je zapravo implementirana kroz sustav traganja unatrag (engl. *backtracking*) unutar kojeg ako neka alternativa ne uspije, program se jednostavno vraća korak unatrag i isprobava drugu alternativu sve dok ne pronađe onu ispravnu. Upravo zbog ovakvih višestrukih izbora, nedeterministički programi nemaju polinomno vrijeme izvođenja već eksponencijalno. Isto tako, karakterizira ih da za jedan ulaz mogu davati više različitih izlaza.

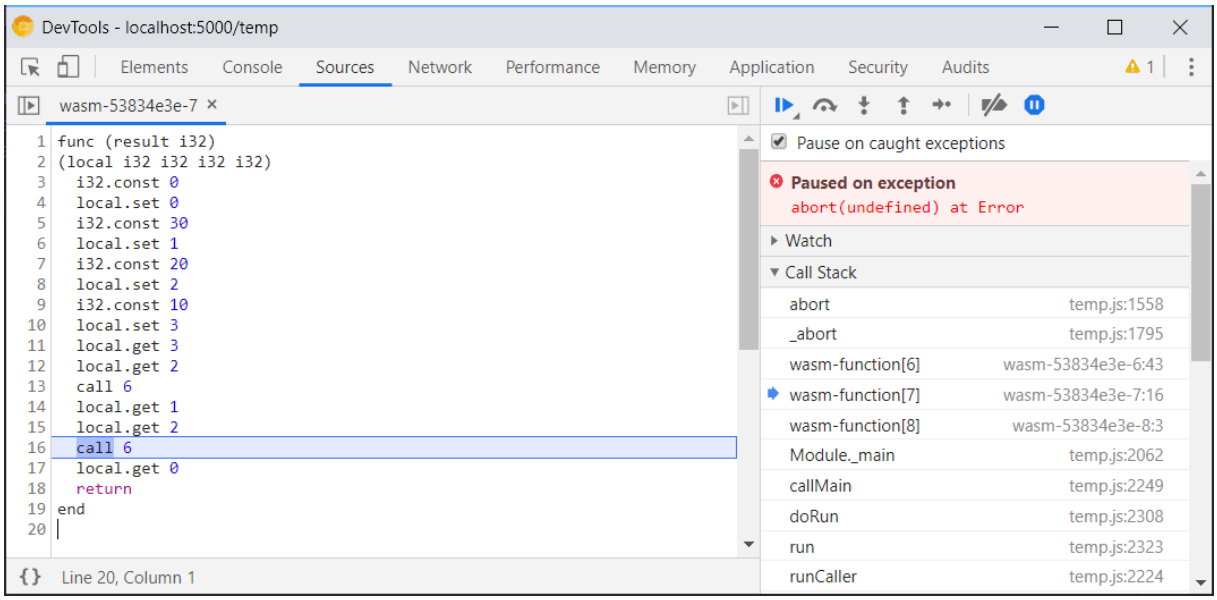
WebAssembly kao programski jezik ostvaruje samo ograničeni nedeterminizam i to na lokalnoj razini. U ovom konteksu, 'ograničeni' znači da samo u nekoliko specifičnih slučajeva dolazi do nedeterminističkog izvršavanja odnosno odabira alternativa u toku rada programa. To su prvenstveno slučajevi koji uključuju nove i buduće osobine formata koje su vrlo nepredvidljive, na primjer: API pozivi dretvi, određivanje predznaka rezultata koji nije broj (engl. *NaN, not a number*) i drugi slučajevi definirani unutar specifikacije formata. Nadalje, 'lokalni' znači da kada dođe do nedeterminističkog izvršavanja, taj će se odraziti samo na lokalno stanje odnosno neće uzrokovati posljedice u nekoj drugoj komponenti.

3.4.5. Otkrivanje grešaka

Sastavni dio razvoja svakog jednostavnog sustava pa sve do onih složenijih je otkrivanje grešaka (engl. *debugging*). Većina modernih razvojnih okruženja poput Visual Studio-a, Android Studio-a, NetBeans-a i drugih u sebi imaju ugrađene alate za otkrivanje grešaka. Takvi alati omogućuju postavljanje točki prekida (engl. *breakpoints*) na kojima se zaustavlja izvođenje programa te je tada moguće provjeriti vrijednosti pojedinih varijabli, mijenjati vrijednost ili ih dodati na listu za pregled kako bi se moglo vidjeti njihovo stanje u svakom pojedinom trenutku. Također je moguće ispitati stog poziva funkcija, ispitati iznimke, kreirati snimak memorije u nekom trenutku pa kasnije usporediti performanse i slično. Sve ove navedene mogućnosti alata za otkrivanje grešaka postoje već dugi niz godina i nisu teške za implementirati iz tehničke perspektive jer se otkrivanje grešaka radi na izvornom kôdu poznatog programskog jezika i

dostupnog unutar izvršnog okruženja. Međutim, Wasm je binarni format i kao takav može se dobiti kompiliranjem gotovo bilo kojeg programskog jezika pa ponekad taj podatak ne može biti poznat ugrađivaču. Isto tako, zamišljen je da ga se ugrađuje u druga okruženja i tamo izvršava i stoga je takvo "udaljeno" pronalaženje grešaka vrlo nezgodno.

S obzirom da je Emscripten bio prvobitni lanac alata za kompilaciju Wasm-a, on pruža neke mogućnosti za otkrivanje grešaka unutar modula. Za otkrivanje grešaka prevoditelja, moguće uključiti način rada za otkrivanje grešaka (engl. *debug mode*) tako da se postavi `EMCC_DEBUG` varijabla okruženja na 1 ili jednostavno pridoda `-v` zastavica prije kompilacije u naredbenom retku. U računarstvu, zastavica označava vrijednost koja služi kao signal za pojedini proces i tako odredi tok programa, a najčešće je binarnog tipa [24]. Nadalje, Emscripten nudi nekoliko zastavica (-g0, -g1, -g2, -g3, -g4) koje progresivno uključuju više i više informacija uz produkcijski kôd [18]. Pod informacijama ovdje se misli na DWARF informacije koje su dio DWARF standarda - podatkovnog formata za otkrivanje grešaka [25]. DWARF format je inicijalno razvijen za otkrivanje grešaka u Linux ELF (engl. *executable and linkable format*) binarnim datotekama, a s vremenom je pronašao svoje mjesto u prevoditeljima poput Emscripten-a i drugih [26]. U principu, Wasm stroj nema nikakve koristi od ovih informacija ali okruženja poput Mozilla Firefox i Google Chrome web preglednika dakako imaju (više o ovome u nastavku). Nastavno na zastavice, svaka opcija uz binarni format generira i tekstualni format koji sadržava navedene DWARF informacije. Ovdje valja napomenuti da mnogi web preglednici nude mogućnost direktnog pregleda tekstualnog formata Wasm modula i postavljanje prekidnih točaka unutar istoga i tako omogućavaju osnovno otkrivanje grešaka (vidi Sliku 12).

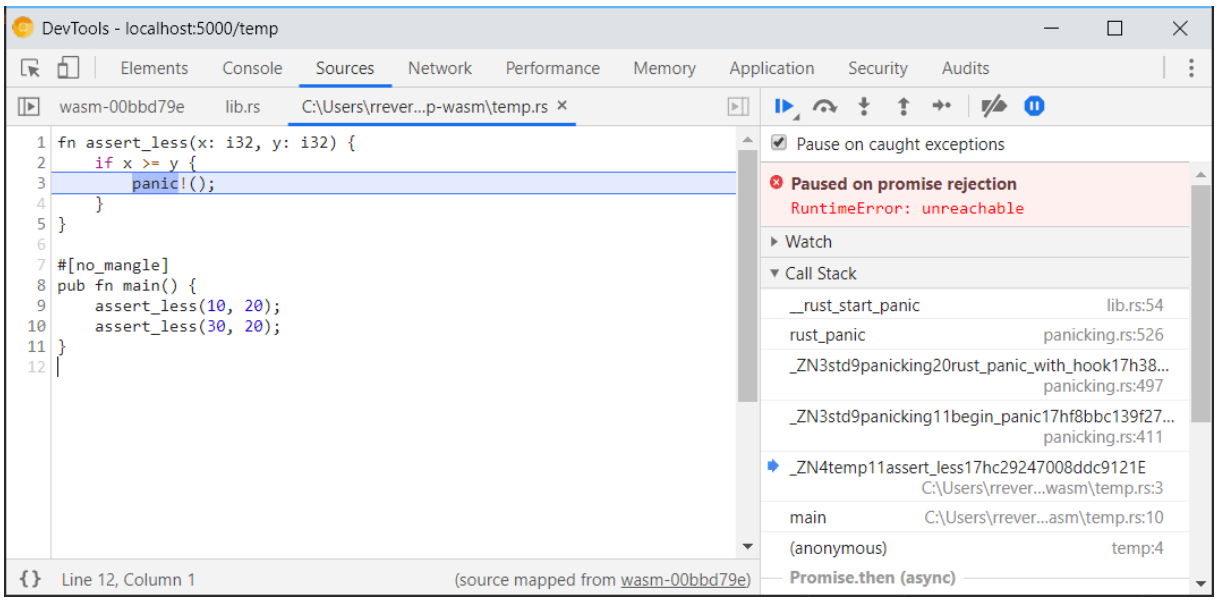


Slika 12: Otkrivanje grešaka na tekstualnom formatu unutar Chrome DevTools [27]

Iako ovaj pristup radi sa bilo kojim Wasm modulom i pomaže otrkiti greške u manjim, izoliranim funkcijama, nije nikako praktično za veće aplikacije gdje je znatno teže mapirati između tekstualnog formata i izvornog kôda. Kako bi se ovaj problem rješio, Emscripten i Chrome DevTools su zajedničkim snagama prilagodili mapiranje izvornog kôda za WebAssembly [27]. Mape izvornog kôda (engl. *source maps*) definiraju format za mapiranje između originalnog

izvornog kôda i produkcijskog (kompiliranog) kôda tako da web preglednik može imati uvid u izvorni kôd i tako otkrivati greške [28]. Mape izvornog kôda su prvobitno razvijene kako bi pomogle u otkrivanju grešaka unutar transpiliranog Javascript kôda [26]. Specifičnije, u slučajevima kada je Javascript kôd dobiven iz drugih jezika poput CoffeScript-a, TypeScript-a i/ili je jednostavno minimiziran pa je direktno otkrivanje grešaka u rezultirajućem kôdu otežano. Unutar Emscripten-a, mape izvornog kôda moguće je kreirati tako da se unutar naredbenog retka prilikom kompilacije doda zastavica `-g4` i shodno tome ona pruža najviše informacija. Ova osobina radi na principu da bilježi pomake u memorijskim lokacijama kôd kompilinarnog modula i izvornog kôda i tako je moguće otkriti gdje se nalazi greška u izvornom kôdu. Međutim, kao što je već navedeno, mape izvornog kôda su razvijene za tekstualne formate za jasnim mapiranjima u Javascript konstrukte i vrijednosti, a ne za binarne formate poput Wasm-a koji imaju proizvoljne tipove podataka i linearnu memoriju. S obzirom na to, ova integracija je dosta ograničena i zahtjeva puno "varanja" i "zaobilaženja", te nije široko podržana izvan Emscriptena.

Sada konačno dolazi već spomenuti DWARF do izražaja, jer upravo ovaj format rješava problem navedene integracije. Već je rečeno da izvršna okruženja nemaju koristi od ovih informacija ali Web preglednik, točnije DevTools ima puno koristi. Uz pomoć ovih informacija, DevTools može ispravno napraviti mapiranje izvornog kôda i samim time moguće je potpuno otkrivanje grešaka unutar Wasm modula bez zadiranja u tekstualni format, već direktno u izvornom kôdu (vidi Sliku 13). Valja spomenuti da i dalje postoje neke osobine specifične za WebAssembly jezik koje i dalje nisu implementirane, a u budućnosti bi se trebale implementirati za potpunu kompatibilnost.



Slika 13: Otkrivanje grešaka u izvornom kôdu unutar Chrome DevTools [27]

3.5. Ugradnja

WebAssembly format je karakterističan po tome što on ne definira na koji način se njegovi moduli učitavaju u stroj koji ga izvršava, niti kako se izvode ulazno-izlazne operacije, šalju web zahtjevi i slično [10]. Umjesto toga, razvojni programeri i dizajneri ovog formata prebacili su taj implementacijski detalj na izvršno okruženje (engl. *runtime*) u koje se ugrađuje Wasm. To podrazumijeva da okruženje ima već ugrađenu implementaciju Wasm formata, što je sasvim logičan uvjet. Nadalje, takav dizajn čini ovaj format vrlo laganim (engl. *lightweight*) u smislu njegove veličine i lakše prenosivosti na druge platforme. Tako su se, osim navedenog WAVM izvršnog okruženja u pregledniku, pojavila i mnoga druga izvršna okruženja kao što je *Wasmer* napisan u Rust-u, *Life* napisan u Go-u, *wasm3* u C-u i dr.

Već je u prethodnim poglavljima spomenuto da je prilikom ugradnje moguće dinamički povezati module na način da se uvozni elementi jednog modula povežu sa izvoznim elementima drugog modula. No, jedna bitna stvar koja ovdje nije spomenuta, a uvelike dodaje na interoperabilnosti formata je da moduli ne moraju biti od istoga proizvođača (engl. *producer*). To znači da je u jednog web aplikaciji moguće uključiti (i povezati) module koji su nastali kompilacijom različitih programskih jezika i korištenjem različitih lanaca alata. Zbog toga WebAssembly kao format ne daje prednost ni jednom specifičnom programskom jeziku ili alatu za kompilaciju, što nije bio slučaj sa prethodnim pokušajima. Zainteresirani proizvođači mogu kreirati zajednička aplikacijska binarna sučelja (engl. *ABI, application binary interface*) nad Wasm-om tako da moduli mogu interoperirati međusobno u heterogenim sustavima. Za razliku od homogenih sustava koji koriste jedan tip procesora ili jezgri, heterogeni sustavi koriste više različitih tipova procesora ili jezgri i znatno su kompleksniji. Nastavno na binarna sučelja, takav pristup razdvajanja briga (engl. *separation of concerns*) ključan je za univerzalnost Wasm-a kao formata [10].

3.5.1. Ugradnja u Web platformu

Jedna od primarnih svrha Wasm-a je da se izvodi na Web platformi, primjerice da se ugradi u Web preglednik. Dizajneri Wasm-a su prije izdavanja minimalnog održivog proizvoda definirali glavne ciljeve (engl. *high-level goals*) Wasm-a kao formata, a jedan od njih je bio slijedeći [3]:

"Dizajnirati format da način da se izvršava unutar i dobro integrira sa postojećom Web platformom; omogućiti da bude kompatibilan sa prijašnjim Web standardima, poštivati iste sigurnosne politike platforme i pristupati funkcionalnostima Web preglednika kroz Web API-e.."

Valja napomenuti i da je jedan od ciljeva bio da Wasm podržava i ugradnju izvan Web platforme, ali ne toliko strogo definirano i kao unutar nje.

Javascript API trenutno je jedino sučelje unutar Web platforme koje pruža vezu sa Wasm izvršnim okruženjem i tako omogućava kompilaciju modula. Osim toga, pruža i mogućnost ograničene refleksije kompiliranih modula, pohranu i dohvaćanje modula iz lokalne pohrane u izvanmrežnom radu, poziv izvezenih funkcija modula, referenciranje memorije modula i drugo.

U računarstvu, refleksija označava svojstvo programskog jezika da preispituje samog sebe. Osim instantnog (sinkroniziranog) kompiliranja Wasm modula, sučelje pruža i mogućnost asinhrone kompilacije modula koristeći `Promise` JavaScript konstrukta. Taj pristup opet ima svoje prednosti i nedostatke koje treba uzeti u obzir, ali one su izvan konteksta ovog rada.

Ono što valja naglasiti kod ugradnje unutar Web-a je Javascript-ov sistem modula (engl. *module system*) koji omogućava integraciju Wasm modula u platformu na intuitivan i prirodan način. Na isti način kojim se uključuju Javascript moduli u Web aplikaciju te izvoze i uvoze funkcije iz Javascript modula moguće je i uključiti WebAssembly modul u Web aplikaciju te izvesti i uvesti funkcije između njegovih modula. Jedini preduvjet u ovom pristupu je pridržavanje konvencija oko ekstenzija datoteke i formata enkodiranja kako je navedeno u potpoglavlju 3.4.2.1. Binarni format. U budućnosti, kada Wasm bude podržavao sakupljanje smeća, moduli će moći pristupiti Javascript DOM API-u i tako direktno upravljati korisničkim sučeljem i tako dodatno proširiti mogućnosti jezika.

3.5.2. Ugradnja izvan Web platforme

Iako je WebAssembly dizajniran da se izvodi na Web-u, poželjno je i da bude moguće njegovo izvršavanje u drugim okruženjima od minimalnih ljuski (engl. *shell*) operacijskog sustava pa sve do većih sustava poput poslužitelja u podatkovnim centrima, IoT uređajima, mobilnim i stolnim aplikacijama i drugim [3].

Od 2009. godine, Javascript se počinje izvoditi i na poslužiteljskoj strani u obliku *Node.js* izvršnog okruženja koje koristi V8 Javascript stroj (inače je integriran u Google Chrome preglednik). Nadalje, 2014. godine pojavio se *Electron* programski okvir za razvoj grafičkih stolnih aplikacija koji koristi Chromium-ov stroj za obradu i *Node.js* izvršno okruženje te tako omogućava izvođenje Javascript-a izvan Web platforme. S pojavom Wasm-a, zainteresirani razvojni programeri počeli su razvijati sučelja koja omogućavaju integraciju navedenih tehnologija sa Wasm-om na isti način kako to radi Javascript API unutar Web platforme. Tako je izdana specifikacija za WebAssembly sistemsko sučelje (engl. *WASI, WebAssembly System Interface*) te je prema njoj razvijeno WASI API za *Node.js*, sučelje koje omogućuje komunikaciju WebAssembly-a sa operacijskim sustavom kroz funkcije nalik POSIX funkcijama [29]. Specifikaciju za navedeno sučelje definirala je Mozilla grupa, a definirana je tako da bude prenosiva na bilo koju platformu. S obzirom da postoje neka preklapanja u radu Wasm-a unutar Web platforme i izvan nje, pa je u tim segmentima Mozilla grupa mogla pružiti osnovne gradivne elemente i podršku za razvoj vlastitog sučelja.

Općenito, WebAssembly izvan Web platforme ne mora (i ne može) koristiti primarna sučeljima koje ono pruža kao što su već spomenuti WebGL, WebAudio, WebSocket itd. S time na umu, moguće ga je pokretati na gotovo svakoj platformi što je ključno za portabilnost formata.

3.5.3. Podržani jezici

Nakon izdanja minimalnog održivog proizvoda u kojem je fokus bio na programskim jezicima C/C++, u međuvremenu su tri jezika razvili podršku za WebAssembly a to su Rust,

TypeScript i C#. Trenutno postoje i mnogi aktivni projekti koji bi donijeli podršku za WebAssembly i drugim programskim jezicima, ali oni još nisu završeni ili nisu spremni za produkcijsku okolinu. U nastavku je prikazan popis tih projekata podijeljen u dvije skupine:

a) Završeni ili spremni za produkcijsku okolinu

   1) **C i C++** - postojana podrška od početka formata kroz *Emscripten* lanac alata
   2) **Rust** - podrška u samoj specifikaciji Rust jezika kroz *Rustup* lanac alata
   3) **C#** - programski okvir *Blazor* koji omogućuje kompilaciju C# jezika u Wasm (puno više o ovome u poglavlju 4. Blazor)
   4) **TypeScript** - projekt *AssemblyScript*, prevoditelj koji omogućuje kompilaciju TypeScript skupa u Wasm uz pomoć Binaryen lanca alata
   5) **Go** - standardni prevoditelj jezika pruža mogućnost kompilacije Go formata u WebAssembly

b) Nezavršeni ili nespremni za produkcijsku okolinu

   1) **Java** - projekt *TeaVM*, prevoditeljski prevoditelj koji je inicijalno razvijen da kompilira Javu u Javascript, sada pruža i podršku za WebAssembly
   2) **Python** - projekt *Pyodide*, distribucije Python okruženja prenesenog u WebAssembly koje sadržava ključne pakete za znanstveni rad (Numpy, Pandas i matplotlib)
   3) **Kotlin** - projekt *Kotlin/Native* osmišljen sa idejom kompilacije Kotlin-a u okruženja gdje virtualni strojevi nisu postojani, a sadrži i LLVM prevoditelj koji omogućuje kompilaciju jezika u WebAssembly format

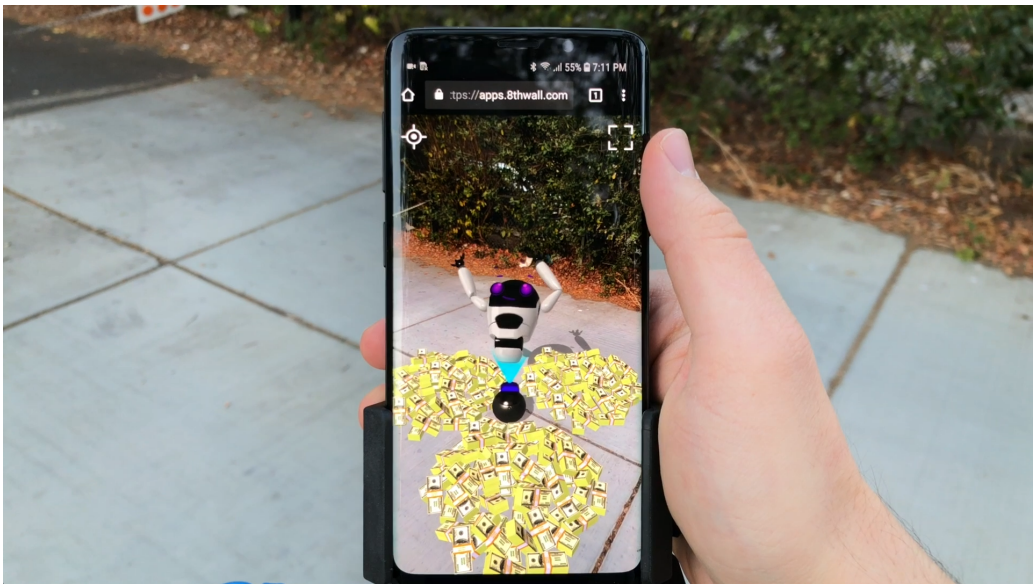
Valja napomenuti da je ovo vrlo skraćeni popis projekata koji uključuje samo najkorištenije programske jezike u razvoju softvera. Puni popis sadrži više od 40 projekata, a moguće ga je vidjeti na *awesome-wasm-langs* GitHub repozitoriju (https://github.com/appcypher/awesome-wasm-langs).

3.6. Aplikacije u realnoj domeni

S obzirom da je WebAssembly još poprilično novi element u području Web razvoja, on u mnogima i dalje izaziva skepticizam oko njegove primjene u bilo kakvim ozbiljnijim aplikacijama. Međutim, WebAssembly se pokazao kao spreman "igrač" već u nekoliko aplikacija koje su predviđene za korištenje velikih skupina ljudi. Pod ovime se nipošto ne misli na demo aplikacije poput već spomenutih *Epic Citadel*, *Epic Zen Garden*, *Tanks* i ostalih aplikacija razvijenih u svrhu prezentacije. U nastavku su opisane nekoliko aplikacija iz različitih domena koje najbolje demonstriraju moć i široku primjenu Wasm-a:

1) **8th Wall** - web aplikacija koja omogućuje razvojnim programerima da na Webu kreiraju i objavljuju objekte i iskustva proširene stvarnosti u realnom vremenu. Kreatori ove aplikacije iskoristili su mogućnost *Unity* okruženja da izlazni rezultat kompilacije (engl. *output*)

bude u Wasm formatu i tako donijeli iskustvo proširene stvarnosti na Web. Svakako je potrebno naglasiti da Unity razvojno okruženje, osim proširene stvarnosti, nudi i mogućnosti kreiranja 3D grafike i 2D virtualne stvarnosti pa se stoga može očekivati još mnogo ovakvih aplikacija u budućnosti.

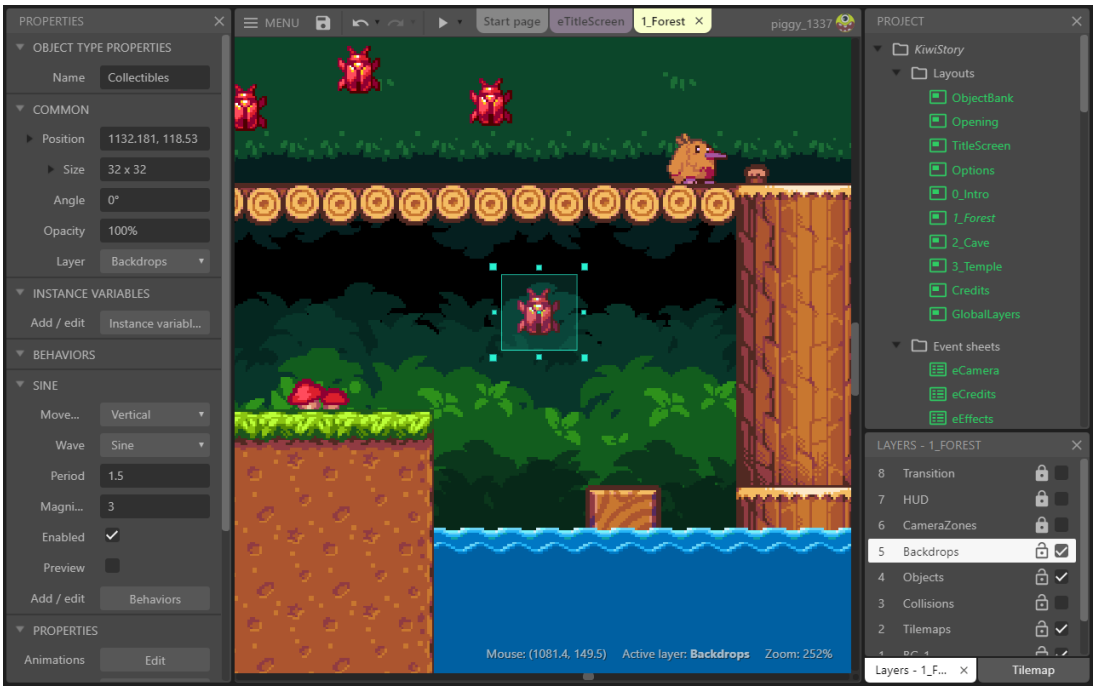


Slika 14: 8th Wall web aplikacija [30]

2) **Amped Studio** - moderni web studio i produkcijsko okruženje za obradu glazbe. S obzirom da svi programi za obradu digitalnog zvuka rade mnogo procesorski-zahtjevnih poslova kroz razne algoritme kako bi se dodali razni efekti i dobio konačan zvuk, Wasm je svakako najbolji izbor. Do sada, ovakvi tipovi aplikacija isključivo su postojali u obliku stolnih rješenja, dok su web rješenja kao *Bandlab* ili *Soundation* imale znatno lošije performanse i ograničene mogućnosti.

3) **Construct3** - web uređivač računalnih igara i također najbitnija stavka na ovom popisu. Razvojni programeri puno su vremena uložili u sam dizajn i arhitekturu sustava i tako su napravili uređivač koji spaja najbolje od Web-a i najbolje od izvornoga i na taj način iskoristili puni potencijal oba svijeta. Izvorno stolne biblioteke napisane u C/C++ jezicima poput *FFmpeg* za video dekodiranje, *Box2D* stroj za fiziku i druge jednostavno su kompilirali u WebAssembly i uključili u Web platformu [31]. S druge strane, na Web platformi koristili su WebGL za prikaz grafike, WebAudio za reprodukciju zvuka pa čak i WebRTC za komunikaciju u realnom vremenu. Jedna zanimljivost oko WebRTC sučelja je ta da pruža mogućnost prijenosa proizvoljnih binarnih podataka putem podatkovnog kanala (engl. *data channel*). Imajući na umu to i veličinu izvornog kôda projekta od svega nekoliko megabajta, moguće je u nezamislivo kratkom vremenu drugoj osobi poslati igricu na testiranje. Na ovom primjeru moguće je vidjeti kako je uz korištenje postojećih biblioteka i sučelja koja nam već nudi Web platforma moguće razviti sustav bogatih osobina i izvrsnih performansi.

4) **Google Earth** - web aplikacija koja omogućuje virtualni 3D prikaz Zemljine površine. Ova stavka je na neki način izuzetak na ovoj listi jer inicijalno nije razvijena uz pomoć Wasm-



Slika 15: Construct 3 web uređivač igara [32]

a, već uz pomoć spomenutog Native Client-a (istoga je Google Chrome odbacio zbog Wasm-a). Isto tako, inicijalna aplikacija bila je dostupna isključivo na Google Chrome pregledniku upravo zbog Native Client-a koji je radio samo na navedenom pregledniku. S pojavom Wasm-a, razvojni programeri Google Earth aplikacije iskoristili su Emscripten da pojedine dijelove starije izvorne aplikacije napisane u C++ kompiliraju u WebAssembly. Na taj način uspjeli su donijeti Google Earth aplikaciju na sve preglednike sa još boljim performansama.

5) **Tensorflow.js** - popularna Javascript biblioteka otvorenog tipa za pripremu i postavljanje modela strojnog učenja (engl. *machine learning*). Poznato je da strojno učenje zahtjeva veliku procesorsku moć kako bi se izvelo mnoštvo matematičkih operacija nad velikom količinom podataka, pripremio model za taj skup podataka i naposlijetku izradilo predviđanje. Matematičke operacije se konstantno ponavljaju i zato ovaj slučaj korištenja zahtjeva strojni kôd kojega je moguće ubrzati kada je god to moguće, što nije slučaj sa Javascript-om. Prethodna izdanja biblioteka su na poslužiteljskoj strani koristile WebGL, a od ožujka 2020. godine postoji i mogućnost korištenja Wasm-a. S takvom tranzicijom poboljšali su performanse na većem skupu uređaja, prvenstveno mobilnim uređajima koji nemaju podršku za WebGL ili jednostavno nemaju snažan grafički procesor [33].

Na prethodnih nekoliko primjera bilo je moguće vidjeti kako WebAssembly zamjenjuje Javascript i popunjava njegove nedostatke u gotovo svim domenama Web razvoja. Potpuni popis svih projekata koji su koristili WebAssembly u svom razvoju i koji će ga koristiti moguće je vidjeti na *madewithwebassembly* (https://madewithwebassembly.com) web stranici.

3.7. Budućnost Wasm-a

U prethodnom potpoglavlju moglo se vidjeti da se Wasm-om čak i kao minimalni održivi proizvod već koristi u implementaciji složenih sustava. Taj popis će se definitivno znatno proširiti u budućnosti kada budu implementirane sve najavljene osobine formata. Međutim, dodavanje novih osobina izvodi se kroz proces standardizacije koji uključuje nekoliko izvršnih okruženja i preglednika i samim time traje dugo. Isto tako, dizajneri formata ne ovise isključivo o sebi već i o pružateljima drugih web preglednika čija je zadaća implementirati standardizirane osobine. Svi aktivni prijedlozi osobina dostupni su na njihovoj službenoj GitHub dokumentaciji, a podijeljeni su u prema fazama u kojem se nalaze gdje Faza 0 (engl. *Phase 1*) označava ne započeti proces, a Faza 4 gotovo završeni proces standardizacije. U nastavku su opisane nekoliko najiščekivanijih osobina iz različitih faza procesa koje će u najvećoj razini proširiti mogućnosti Wasm-a i otkloniti postojeće probleme:

1) **Višedretveni rad** (engl. *concurrency*) - poznato je da većina programskih jezika danas ima podršku za višedretvenost i stoga bi ista bila poželjna u Wasm-u. Problem kod implementacije ove osobine je taj što Javascript kao takav ne podržava višedretveni rad nego se sve izvodi na glavnoj (UI) dretvi. Ovaj prijedlog donio bi osnovne module za rad sa dretvama nalik POSIX dretvama (pthread) na način da bi iskoristili već postojeće Javascript sučelje za Web radnike [31]. Web radnici (engl. *web workers*) su dretve koje mogu raditi u pozadini aplikacije bez da ometaju glavnu dretvu. Uz pomoć web radnika i dijeljenog spremnika (SharedArrayBuffer) ostvario bi se višedretveni rad, a navedeni prijedlog donio bi i atomarne operacije (engl. *atomic operations*) koje bi omogućile da dretve učitavaju ili zapisivaju podatke u jednom koraku. Trenutno već postoje mnoge stolne biblioteke koje iskorištavaju dretve u svome radu i nisu još pogodne za prijenos na Web, a u budućnosti bi ova osobina omogućila razvoj kompleksnijih Web sustava koji u pozadini moraju izvoditi nekoliko desetaka procesa.

2) **Sakupljanje smeća** - također jedna od osobina mnogih programskih jezika danas koja razvojnog programera oslobađa od ručnog upravljanja memorijom. WebAssembly trenutno nema sakupljača smeća nego jednostavno ugrađivaču pruža nakupinu memorije koju je po potrebi moguće proširiti. S time na umu, okruženje koje ugrađuje Wasm i dalje upravlja memorijom modula pa nema prevelikog smisla da Wasm kao takav imeplementira vlastiti sakupljač smeća. Umjesto toga, integrirati će se uz sakupljač smeća koji postoji u okruženju ugrađivača i tako olakšati dijeljenje stanja memorije te poboljšati interoperabilnost s istim [34]. Trenutno, programski jezici koji se kompiliraju u Wasm, a zahtjevaju sakupljanje smeća nemaju drugog izbora nego taj modul isto kompilirati u Wasm zajedno sa cijelom implementacijom. Naprimjer, AssemblyScript ima uključen modul pod nazivom *makeshiftGC* koji služi za sakupljanje smeća, a sami modul znatno dodaje na veličini cijele implementacije. Potrebno je naglasiti da je ovaj prijedlog trenutno u Fazi 1 zbog cjelokupne kompleksnosti te da je ovisnost o sakupljanju smeća razlog zašto jezici poput Scala i Elm i dalje nisu dobili svoje Wasm implementacije.

3) **Povezivanje sa okruženjem ugrađivača** (engl. *host bindings*) - zanimljiva osobina

```
(func $swap (param i32 i32) (result i32 i32)
        (get_local 1) (get_local 0)
)
```

Isječak kôda 3.2: Primjer Wasm tekstualnog koda za vraćanje dviju vrijednosti u funkciji

koja će omogućiti WebAssembly modulima da direktno upravljaju Javascript objektima te objektima modela dokumenta (DOM objektima). Trenutno sučelje između Wasm-a i Javascript stroja vrlo je ograničeno, prvenstveno zbog složenijih tipova podataka. Zbog toga, ukoliko bi se htjelo iz Wasm modula upravljati objektnim modelom dokumenta, taj pothvat bi zahtjevao mnogo dodatnog kôda na razini sučelja za samo jednu osnovnu operaciju. Plan implementacije ove osobine uključuje dodavanje još jedne poznate sekcije modula koja bi sadržavala anotacije koje bi davale upute koji mehanizam povezivanja ili sučelje treba biti korišteno. Zanimljivo je da Rust već ima sličan alat *wasm-bindgen* koji omogućuje da se složeniji objekti poput nizova znakova sa lakoćom prenose do Javascript stroja s tim da alat ne mjenja modul dodavajući mu anotacije, već metapodatke.

4) **Referentni tipovi podataka** (engl. *reference types*) - već spomenuta osobina koja bi proširili Wasm-ov trenutni skup od samo 4 tipa podataka i dodala `anyref` tip koji omogućuje modulima da referenciraju tipove podataka iz okruženja ugrađivača. Trenutno je za dijeljenje referentnih tipova podataka potrebno iste serijalizirati u linearnu memoriju i referencu na lokaciju pohrane poslati ugrađivaču, što svakako nije prikladno. Važno je naglasiti da je prijedlog ove osobine već u Fazi 4 (gotovo implementiran) jer je njegova implementacija glavna stepenica prije ostvarivanja drugih prijedloga.

5) **Višestruke povratne vrijednosti funkcije**) (engl. *multi-value returns*) - osobina koja bi omogućila da Wasm virtualni stroj umjesto maksimalno jedne vrijednosti, vraća višestruke vrijednosti sa stoga. U poglavlju 3.4.1. Virtualni stroj objašnjeno je da isti operira nad stogom, a takva struktura podataka dozvoljava istodobno uzimanje samo jedne vrijednosti i to one koja se nalazi na vrhu stoga. S time na umu, ovaj prijedlog na prvu se može činiti kao teško izvediv ali uz malo razmišljanja može se zaključiti da se ovdje radi o samo prividnom ograničenju [34]. Na primjer, moguće je definirati funkciju poput ove

   koja jednostavno slijedno vraća vrijednost na vrhu stoga, a zatim vrijednost nakon nje. Iako ova osobina ne donosi gotovo ništa posebno razvojnim programerima koji bi samo ugrađivali WebAssembly, nužna je za implementaciju drugih osobina poput rukovanja iznimkama (slijedi u nastavku).

6) **Rukovanje iznimkama** (engl. *exception handling*) - bez previše objašnjavanja, ovaj prijedlog donio bi mogućnost rukovanja iznimkama kao u većini današnjih programskih jezika. S obzirom na poprilično oskudne kontrolne tokove, WebAssembly trenutno ne nudi ništa slično rukovanju iznimkama. Umjesto toga, alati poput Emscripten-a moraju simulirati ovu funkcionalnost i takva opcija standardno nije uključena već ju je potrebno samostalno uključiti i podesiti. Zanimljivo je da timu zaduženom za standardizaciju ovo već drugi pokušaj za implementaciju osobine, jer prvi pokušaj nije bio dovoljno interoperabilan sa

okruženjem ugrađivača. To dovoljno govori o kompleksnosti ovog problema, a bitno je i naglasiti da implementacija ovog prijedloga strogo ovisi o prijedlogu za referentne tipove podataka te o prijedlogu za višestruke povratne vrijednosti funkcije.

Kroz sve ove opisane buduće osobine formata WebAssembly moguće je dobiti dojam o ozbiljnosti razvoja cjelokupne tehnologije. Iako su ispunjeni inicijalni ciljevi tehnologije WebAssembly u smislu njegove univerzalnosti, prenosivosti i integracije sa drugim sustavima, razvojni tim tu ne planira stati. Umjesto toga, najavljene su brojne nove osobine koje uvelike proširuju mogućnosti jezika i olakšavaju njegovu integraciju s drugim sustavima. Valja istaknuti i da se realizacija pojedinih osobina na prvu činila gotovo neizvediva što je predstavilo izazov za razvojni tim koji je naposljetku uvijek predložio elegantno rješenje. Najbolji primjer je implementacija višedretvenog rada uz pomoć sučelja za Web radnike unutar Javascripta koji standardno ne podržava višedretveni rad.

Zaključno, jedan od ciljeva cijelog prethodnog poglavlja bio je prikazati prošlost tehnologije WebAssembly odnosno proces njegova nastanka, njegove sadašnje mogućnosti u aspeku podržanih programskih jezika i aplikacija u realnoj domeni, te konačno njegovu budućnost u smislu mogućnosti i novih osobina. Primarni cilj poglavlja bio je da se navedu i opišu glavni koncepti koji karakteriziraju ovaj format i tako demonstrira princip njegova rad na najnižoj razini. Uz ostvareni dojam o njegovom radu na nižoj razini i načinu integracije sa Web platformom mnogo je lakše shvatiti i princip rada programskog okvira *Blazor WebAssembly* koji ovu tehnologiju koristi na višoj razini. O programskom okviru *Blazor* i njegovoj inačici *Blazor WebAssembly* biti će više riječi u sljedećem poglavlju.

4. Blazor

U ovom poglavlju biti će ukratko opisani glavni koncepti programskog okvira *Blazor*. Biti će pobrojane sve trenutne i najavljene inačice programskog okvira sa najvećim naglaskom na inačicu *Blazor WebAssembly* koja će biti korištena u praktičnom dijelu rada za migraciju .NET aplikacije. Za taj dio bitno je znati kako se *Blazor WebAssembly* izvodi u web pregledniku pa je izdvojeno posebno potpoglavlje *4.4.1. Kako funkcionira Blazor WebAssembly?* koje to detaljnije opisuje. Također, na kraju su definirani prednosti i nedostatci ove inačice kako bi se dobio dojam o ograničenjima okvira prije nego što uopće započne migracija aplikacije.

4.1. Općenito o programskom okviru

Blazor je novi programski okvir otvorenog kôda (engl. *open source*) za razvoj modernih jednostranih web aplikacija koje se izvode na više platformi (engl. *cross-platform*). Razvio ga je Microsoft još 2018. godine kao proširenje ASP.NET programskog okvira (zato se i u nekim literaturama naziva samo osobinom ASP.NET okvira), a naziv je dobio od riječi Browser + Razor. Ovaj programski okvir nudi sve mogućnosti modernih Javascript okvira s time da u ovom slučaju razvojni programeri mogu na klijentskoj strani pisati C#, a ne Javascript kao inače. To omogućava razvojnim programerima slično iskustvo kao i kod razvoja standardnih .NET aplikacija u kojem je moguće koristiti samo jedan programski jezik za kompletni razvoj. Za usporedbu, kod tipičnog razvoja jednostranih web aplikacija sa Javascript programskim okvirima poput React, Angular ili Vue potrebno je koristiti NPM za instalaciju brojnih paketa bez kojih okviri ne mogu funkcionirati, potrebno je podesiti razvojno okruženje za otkrivanje grešaka, instalirati biblioteke za testiranje i dr. Isto tako, postavljanje aplikacije na poslužitelj zahtjeva da ona bude upakirana (engl. *bundled*) za produkciju sa alatima kao što je *Webpack* kojeg je također potrebno instalirati i postaviti što nije jednostavno ukoliko se radi o složenijoj aplikaciji. Nije potrebno ni naglašavati da sve pakete treba uskladiti da rade zajedno, održavati ih i provjeravati imaju li sigurnosne propuste i slično. S druge strane, Visual Studio kao integrirano razvojno okruženje nudi sve prethodno navedene mogućnosti u gotovom, bez ikakvih naprednih postavki pa tako su korisniku dostupne gotovo sve .NET standardne biblioteke (umjesto NPM paketa) i MS Build alat za automatizaciju izgradnje aplikacija (umjesto Webpacka).

Već je rečeno da Blazor ima sve mogućnosti modernih okvira za razvoj jednostranih web aplikacija, a one su sljedeće [35]:

1) **Arhitektura temeljena na komponentama** (engl. *component-based architecture*) - komponente su primarni element svake Blazor aplikacije i mogu se ponovno koristiti, a implementiraju se kao Razor komponente (mnogo više u sljedećem poglavlju).

2) **Usmjeravanje** (engl. *routing*) - klijent može preusmjeriti na drugi pogled ili drugu komponentu koristeći usmjeravanje. Ovo je najbitnija mogućnost svakog okvira za razvoj jednostranih web aplikacija jer ona daje taj osjećaj jednostranosti, odnosno korisnik zbog izmjene URL-a dobije dojam da je otvorio novu stranicu dok se u pozadini samo putem

AJAX poziva izmjenio HTML sadržaj bez osvježavanja stranice i narušavanja korisničkog iskustva.

3) **Osnovni raspored elemenata** (engl. *layout*) - moguće je kreirati osnovni raspored za elemente korisničkog sučelja koji su isti na svakoj stranici (zaglavlje, podnožje, navigacija i slično).

4) **Povezivanje događaja** / **podataka** (engl. *data and event biding*) - jedno od najmoćnijih osobina u web razvoju jer je moguće podatke na pogledu povezati sa onima unutar poslovne logike te kada dođe do promjene u jednom, automatski se promjene izvršavaju i na drugom (dvostrano povezivanje) što uvelike poboljšava korisničko iskustvo.

5) **Forme i validacija** - Blazor standardno pruža kreiranje interaktivnih formi i njihovu validaciju na korisničkoj strani i/ili na poslužiteljskoj strani kroz životni ciklus komponenti, dok s druge strane okviri poput React-a ili Angular-a zahtjevaju instalaciju dodatnih biblioteka za validaciju forme.

6) **Autentikacija i autorizacija** - dostupne su mnoge biblioteke i ugrađeni atributi za ostvarivanje autentikacije i autorizacije, a moguće je i postići autentikaciju sa raznim web servisima poput Facebook-a, Google-a i slično.

7) **Ubrizgavanje ovisnosti** - jedna od najljepših osobina ne samo Blazora, već i cjelokupnog .NET Core programskog okvira jer omogućuje korištenje raznih servisa tako da ih se "ubrizga" u komponentu (nije potrebno instanciranje).

8) **Interoperabilnost sa Javascript-om** - Blazor pristupa sučeljima web preglednika kroz Javascript interoperabilnost (poznatije kao Javascript interop) te je stoga moguće pozivanje Javascript funkcija unutar C# kôda i obrnuto.

Od svih spomenutih, najbitnija je osobina ubrizgavanja ovisnosti koja omogućuje slabo povezivanje komponenata i samim time razvoj čišče arhitekture aplikacije. Valja naglasiti da izuzev Angulara ni jedan programski okvir za razvoj jednostranih web aplikacija ne posjeduje ovu osobinu. Također, svi programski okviri izuzev Angulara zahtjevaju instalaciju dodatnih biblioteka za usmjeravanje komponenti i validaciju formi koje Blazor nudi u gotovome što mu svakako daje veliku prednost.

4.2. Inačice

Kako bi se dodatno prilagodio potrebama razvojnih programera, prilikom kreiranja projekta *Blazor* nudi mogućnost odabira jedne od Blazor inačica (engl. *edition*). Trenutno postoje dvije inačice Blazor programskog okvira, svaka sa svojim posebnostima:

1) **Blazor Server** - prva inačica Blazor programskog okvira na kojoj se aplikacije vrte na ASP.NET Core poslužitelju u Razor formatu. U ovom modelu, udaljeni klijenti odnosno preglednici se ponašaju kao tanki klijenti s obzirom na veličinu datoteka koje se moraju

prenijeti. Preglednik mora samo preuzeti stranicu male veličine, a osvježavanje korisničkog sučelja obavlja se putem SignalR veze. SignalR je Microsoftova biblioteka za asinhrono slanje poruka do web preglednika u realnom vremenu. Nadalje, šalju se SignalR poruke/obavijesti koje su zapravo binarne poruke male veličine na osnovu kojih poslužitelj napravi određenu akciju i po potrebi osvježi sučelje.

2) **Blazor WebAssembly** - inačica Blazora koja se u potpunosti izvodu u korisnikovom pregledniku (bez potrebe za poslužiteljem) zahvaljujući WebAssemblyu, a prva stabilna verzija objavljena je tek u svibnju 2020. godine.

Odabir između navedenih inačica kod razvoja programskog proizvoda ovisi u potpunosti o zahtjevima sustava. Naprimjer, ukoliko sustav zahtjeva mogućnost obrade nekoliko desetaka tisuća zahtjeva tada je bolji izbor Blazor WebAssembly jer kod druge opcije web preglednik kreira novu SignalR vezu za svakog novog korisnika i s time nije skalabilna. S druge strane, ukoliko zahtjeva biblioteke koje nisu dostupne u web pregledniku onda je Blazor Server definitivni izbor. Osim navedenih, Microsoft je najavio još 3 inačice Blazora koje su trenutno u razvoju:

1) **Blazor PWA** - inačica prilagođena za razvoj progresivnih web aplikacija.

2) **Blazor Hybrid** - programski okvir izvorne platforme koji prikazuje korisničko sučelje korištenjem web tehnologija (HTML/CSS).

3) **Blazor Native** - programski okvir koji prikazuje korisničko sučelje specifično izvornoj platformi (ne koristi HTML/CSS).

Migracija .NET stolne aplikacije u *Blazor* web aplikaciju gotovo sigurno bi bila lakša sa odabirom inačice *Blazor Server* s obzirom da se sva poslovna logika nalazi na poslužitelju koji uz to pruža mogućnosti za rad sa bazom podataka i druge servise. Međutim, ovaj se rad bavi prvenstveno tehnologijom WebAssembly pa će stoga u praktičnom dijelu rada biti korištena inačica *Blazor WebAssembly* uz neke dodatne postavke koje će omogučiti migraciju svih funkcionalnosti stolne aplikacije (više u poglavlju 5.1. Analiza prenosivosti i početne postavke).

4.3. Komponente

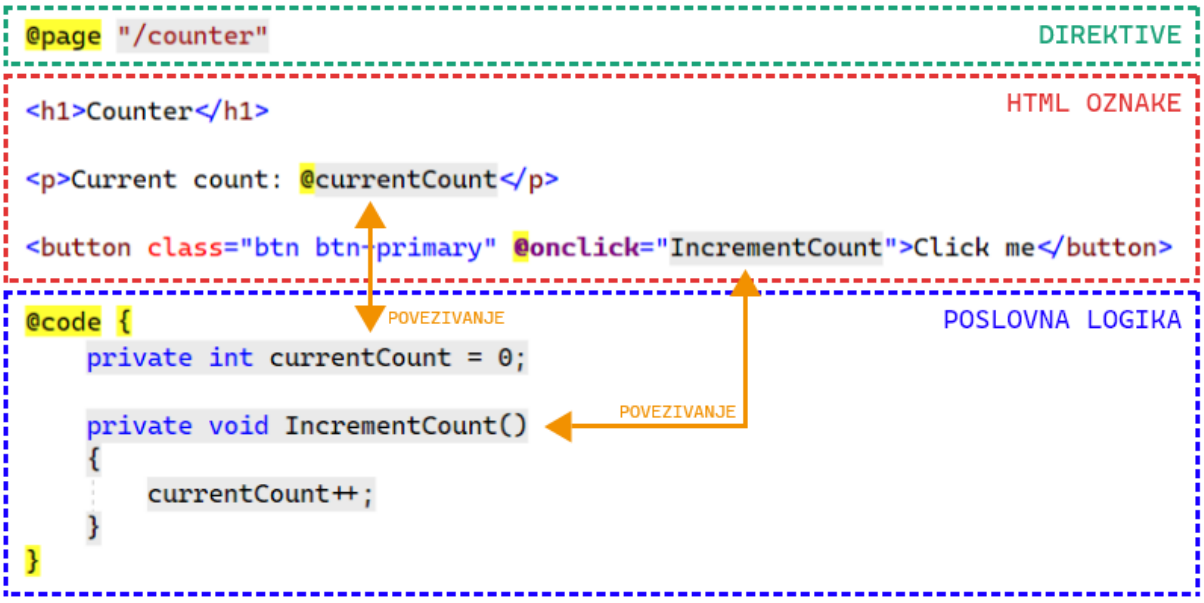
Blazor aplikacije sastoje se od ponovno iskoristivih komponenata implementiranih kao Razor komponente. Razor je programska sintaksa ASP.NET programskog okvira, a koristi se u razvoju dinamičkih web stranica omogućujući miješanje HTML sintakse sa C# ili VB.NET sintaksom. S time na umu, Razor komponente mogu sadržavati C# kôd, HTML oznake i CSS stilske upute. Unutar samih, C# im omogućava procesuiranje programske logike za dohvaćanje podataka i mapiranje istih na potrebna mjesta unutar komponente, rukovanje događajima, usmjeravanje i slično. Nadalje, Razor komponenta nije ništa drugo nego jedan komad korisničkog sučelja kao naprimjer forma, modal, dijaloški okvir i slično. Komponenta može čak biti i jedna cijela stranica, a tada se deklarira kao Razor stranica (engl. *Razor page*). U inicijalnom Blazor projektu, Razor komponente se nalaze unutar `Shared` direktorija, dok se Razor stranice nalaze

unutar `Pages` direktorija. Inače, uvijek ih se može razlikovati prema `@page` direktivi koju svaka Razor stranica mora sadržavati kako bi se moglo usmjeriti na nju (detaljnije u sljedećem potpoglavlju). Općenito, komponente su fleksibilne i lagane pa ih je moguće ponovno iskorištavati, ugniježđivati jednu unutar druge pa čak i dijeliti između nekoliko projekata.

Valja napomenuti da Razor komponente moraju imati ekstenziju *.razor* ili čak *.cshtml* zbog kompatibilnosti sa starijim verzijama VS razvojnog okruženja. Također, generalno pravilo kod imenovanja komponenti je da uvijek počinju sa velikim slovom (kao i klase).

4.3.1. Komponentni model

Na slici ispod prikazana je komponenta `Counter` koja predstavlja jednostavan brojač, a sastoji se od 3 sekcije: direktive, HTML oznake (engl. *markup*) ili pogled te sekcija sa poslovnom logikom (engl. *business logic*).



Slika 16: Komponentni model Blazor okvira

U sekciji direktiva komponenti je moguće dodati određene sposobnosti poput usmjeravanja ili ubrizgavanja raznih servisa. Na gornjem primjeru, definiran je put (engl. *route*) `"/counter"` za komponentu pa će korisniku biti prikazana ova komponenta kada posjeti navedenu rutu. Sljedeća sekcija sadrži standardne HTML oznake koje je, zahvaljujući Razor-u, moguće miješati sa C# kôdom. Komponenta prikazuje trenutno stanje brojača tako da jednostavno referencira `currentCount` varijablu unutar kôda i tako se zapravo izvodi dvostrano povezivanje podataka. Također je izvršeno povezivanje događaja uz pomoć posebnog Razor-ovog atributa `@onclick` koji će na klik gumba pozvati proslijeđenu funkciju `IncrementCount` unutar kôda. Funkcija će inkrementirati vrijednost varijable stanja brojača te će se zahvaljujući dvostranom povezivanju automatski ažurirati i vrijednost na korisničkom sučelju.

Iako se ovaj komponentni model na prvu čini pomalo nezgrapan jer se poslovna logika i pogled nalaze u jednoj datoteci, zapravo je vrlo jednostavan i skalabilan. Kada komponenta postane

```
public override async Task SetParametersAsync(ParameterView
 ↪  parameters)
{
    // API poziv
    await ...
    await base.SetParametersAsync(parameters);
}
```

Isječak kôda 4.1: SetParametersAsync metoda životnog ciklusa komponente

teža za održavanje, uvijek je moguće premjestiti poslovnu logiku u izdvojenu klasu pod uvjetom da klasa ima naziv `nazivKomponente.razor.cs` te da se istoj klasi postavi modifikator `partial`. U objektno-orijentiranom programiranju, modifikator `partial` označava da drugi dijelovi klase, strukture ili sučelja mogu biti definirani unutar istog polja imena (engl. *namespace*).

4.3.2. Životni ciklus

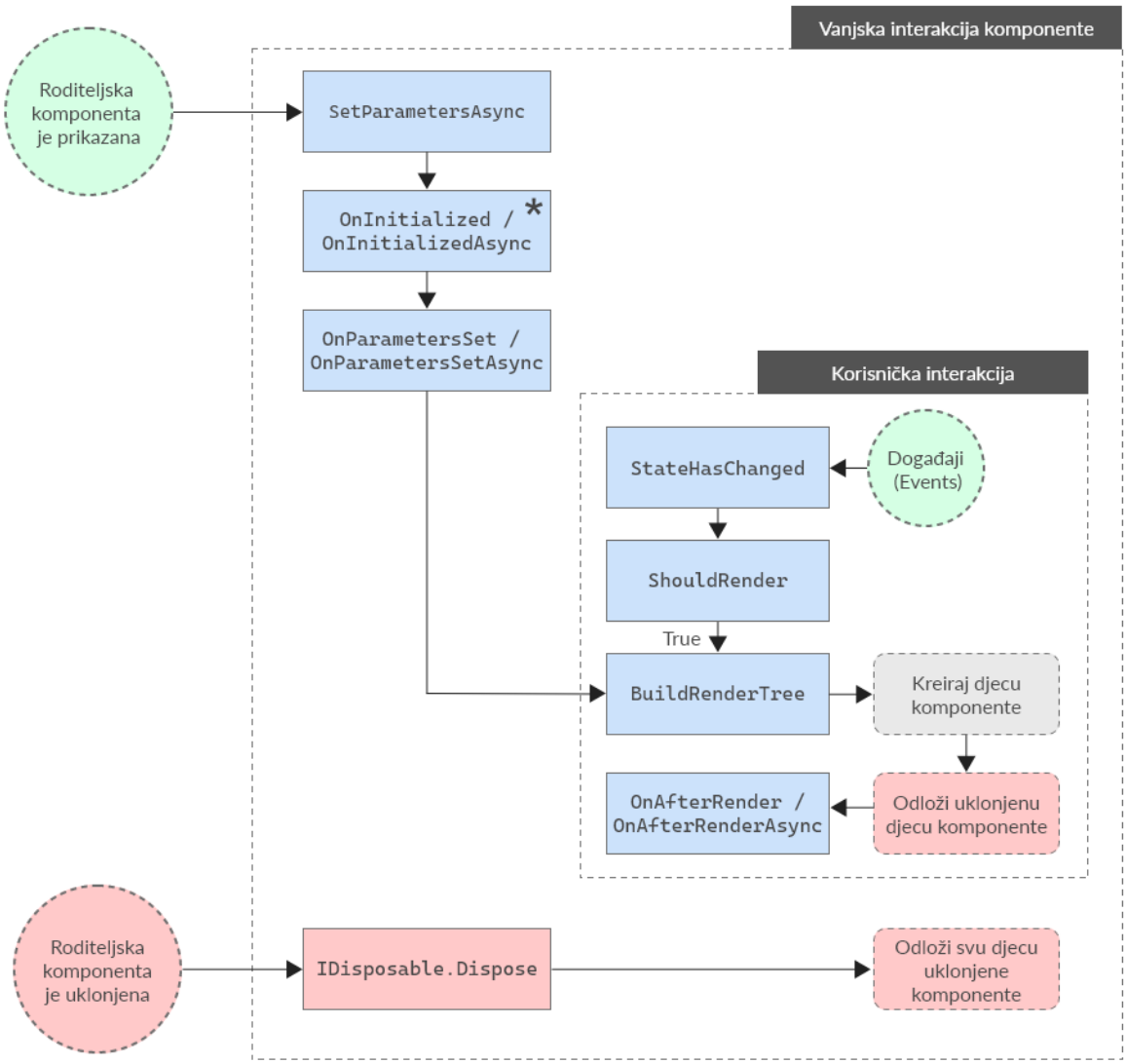
Blazor komponente imaju nekoliko virtualnih metoda koje je moguće nadjačati i tako promjeniti ponašanje aplikacije. Svaka od metoda se izvršava u različitim vremenima tijekom životnog ciklusa komponente, a neke od metoda se čak mogu izvršavati više puta tijekom navedenog ciklusa.

Životni ciklus započinje sa metodom `SetParametersAsync` koja se izvršava svaki put kada se roditeljska komponenta (unutar stabla komponenti) prikaže ili osvježi. Ta roditeljska komponenta može proslijediti parametre koje se nalaze unutar `ParameterView` instance. Dobra je praksa pozivati vanjske API servise unutar ove faze jer je moguće asinhrone pozive raditi ovisno o parametrima koje su proslijeđeni komponenti. Uz to, komponenta u ovoj fazi još nije prikazana pa je moguće sve podatke pripremiti na sučelje i tako poboljšati korisničko iskustvo.

Potrebno je napomenuti da je ovo jedina faza životnog ciklusa koja se uvijek izvodi asinhrono (zato i ima sufiks `Async`) dok ostale faze mogo biti u sinhronoj i asinhronoj varijaciji. Zbog toga uvijek mora vraćati tip `Task` i imati modifikator `async` kako bi označio da se radi o asinhronoj radnji.

Nakon što su parametri proslijeđeni komponenti, izvršava se metoda `OnInitialized` (ili `OnInitializedAsync`) koja funkcionira vrlo slično kao i `SetParametersAsync` s time da ova koristi interno stanje komponente a ne parametre. Zapravo ju je moguće predočiti kao konstruktor komponente u kojem se postavlja inicijalno stanje. Upravo zbog toga, ova faza se izvršava samo jednom tijekom ciklusa i to prvi puta kada je komponenta kreirana. Ukoliko se promjeni stanje roditeljske komponente, ova metoda se neće ponovno izvršiti već će se umjesto nje izvršiti `OnParametersSet` (ili `OnParametersSetAsync`). Logično, ako se radi o novokreiranoj komponenti onda se prethodno navedena metoda neće izvršiti.

Na dijagramu iznad moguće je vidjeti da prethodno navedene faze životnog ciklusa spadaju u vanjsku interakciju komponente, odnosno pozivaju se indirektno. S druge strane, postoje i još



Slika 17: Životni ciklus Blazor komponente

neke metode koje omogućuju bolje upravljanje stanjem komponenti i one se izvode na korisničku interakciju. Metoda `ShouldRender` poziva se svaki put prije prikazivanja komponente i standardno uvijek vraća `true`, odnosno govori programskom okviru da komponentu uvijek treba ponovno prikazati kada se njeno stanje ili stanje roditelja promjeni. Ovu metodu moguće je nadjačati i zatim provjeravati što se zapravo u stanju komponente promjenilo te ovisno o tome utjecati na ponovni prikaz komponente. Ovakve uštede možda nisu značajne kada se na pregledniku prikazuje pet komponenti, ali kada se nalazi njih 500 dok su se podaci promjenili samo u njih 20 tada nije potrebno ostalih 480 ponovno prikazivati i s time se značajno poboljšavaju performanse. Nadalje, ponovni prikaz komponente moguće je prisiliti na programerski način s pozivom funkcije `StateHasChanged` i na taj način obavijestiti programski okvir da se stanje komponente promjenilo. Ova funkcija korisna je unutar asinhronih metodi koje se ne izvršavaju u trenutku već tek kada se završi asinhrona radnja. Metoda koja je zadužena za izgradnju stabla komponenti je `BuildRenderTree` i ona se uvijek izvodi nakon `OnParametersSet` ili ukoliko `ShouldRender` vrati `true` vrijednost (vidi dijagram iznad). Ova metoda zapravo po-

hranjuje sadržaj komponente u memoriju, pa ukoliko se na sučelju prikazuje lista tada će se ta lista spremiti u memoriju. Sve promjene na listi automatski će se propagirati i na sučelje, odnosno ukoliko se stavka doda u listu tada će se i novo dijete kreirati, a ukoliko se stavka ukloni iz liste tada se dijete odlaže. Ukoliko komponenta implementira sučelje `IDisposable`, nakon uklanjanja iz stabla pozvati će se metoda `Dispose` unutar koje se izvode razni poslovi čišćenja.

Posljednja metoda koja se izvršava u ciklusu je `OnAfterRender` (ili `OnAfterRenderAsync`) i, shodno nazivu, izvršava se tek onda kada se komponenta prikaže na sučelju odnosno kada postoji unutar objektnog modela dokumenta. Nakon izvršavanja ove metode će se postaviti zastavica `firstRender` na vrijednost `true` i uz pomoć koje će okvir znati treba li i idući put pozivati metodu `OnInitialized` odnosno radi li se o prvom prikazu komponente ili ne. Valja napomenuti da nije sigurno referencirati komponente uz pomoć `@ref` direktive sve dok ova metoda upravo zbog činjenice da komponenta još nije u potpunosti prikazana na sučelju.

4.3.3. Dependency injection

Ubrizgavanje ovisnosti (engl.*dependency injection*) je tehnika najbolje prakse razvoja softvera koja omogućuje slabo povezivanje (engl. *loose coupling*) klasa i olakšava provođenje jediničnih testova. Tehnika nije vezana ni uz jedan specifični programski jezik niti programski okvir, a koristi se u razvoju na poslužiteljskoj strani i korisničkoj strani. Ideja tehnike je ta da aplikacije mogu registrirati razne servise u centralizirano servisno spremište (engl. *service container*) kako bi ti servisi bili dostupni kroz cijelu aplikaciju. Naprimjer, ukoliko u aplikaciji postoji servis (klasa) za slanje e-mail poruka kojeg koriste nekoliko drugih klasa tada sve one ovise o konkretnoj implementaciji servisa i s time su usko vezane na tu implementaciju. Kroz ubrizgavanje ovisnosti, klase su vezane uz sučelje kojega implementira taj servis i s time je omogućena slaba povezanost i olakšano testiranje. Kada se provode jedinični testovi, potrebno je unutar kontejnera samo zamijeniti pravi servis sa klasom koja ga oponaša (engl. *mock class*) umjesto da se to radi na svim klasama koje koriste servis.

Specifično, ova tehnika Blazor WebAssembly aplikacijama omogućuje:

- dijeljenje jedinstvene instancu servisne klase kroz više komponenti koristeći *singleton* servis.
- odvajanje komponenti od konkretnih implementacija servisnih klasa kroz referencu apstrakcije (sučelja).

U Blazoru (i .NET Core-u općenito) prilikom registriranja servisa moguće je odrediti životni vijek servisa tako da ga konfiguriramo sa jednom od ponuđenih nekoliko opcija:

1) **Scoped** - servis sa ovom postavkom traje sve dok je postojana HTTP veza sa poslužiteljem, odnosno kroz jednu korisničku sesiju. S obzirom da u Blazor WebAssembly aplikacijama nema stalne HTTP veze sa poslužiteljem, ovaj servis se ponaša isto kao Singleton servis.

2) **Singleton** - programski okvir kreira samo jednu instancu servisne klase, a sve komponente koje zahtjevaju servis dobivaju istu instancu.

3) **Transient** - svaki puta kada komponenta zatraži servis iz servisnog spremišta, dobiti će novu instancu servisne klase.

U Blazor WebAssembly inačici, servisi se registriraju unutar `Program.cs` klase koja je zapravo ishodišna klasa aplikacije (kao npr. `Main.cs` u konzolnim aplikacijama). Naprimjer, ukoliko se želi registrirati servisna klasa za slanje e-mail poruka `EmailService` koja implementira `IEmailService` sučelje i konfigurirati ju kao `Singleton` servis, to se može napraviti na sljedeći način:

```
builder.Services.AddSingleton<IEmailService, EmailService>();
```

Isječak kôda 4.2: Registracija servisne klase unutar DI spremišta

Valja napomenuti da je `builder` instanca klase `WebAssemblyHostBuilder` već dostupna unutar `Main` metode i ne treba ju posebno kreirati ni podešavati. Nadalje, kada se prethodno registrirani servis želi koristiti unutar komponente potrebno ga je samo "ubrizgati" sa `@Inject` direktivom u zaglavlju komponente i dodati mu željeni naziv:

```
@Inject IEmailService emailService
```

Isječak kôda 4.3: Ubrizgavanje servisne klase u komponentu

Blazor WebAssembly nudi i tri već prekonfigurirana servisa: `HttpClient`, `IJSRuntime` i `NavigationManager` i oni su ključni u razvoju bilo kakve složenije web aplikacije. Njih programski okvir već nudi u gotovom, to jest nije ih potrebno registrirati u servisni kontejner niti dodatno konfigurirati već samo "ubrizgati" u komponentu kao u prethodnom primjeru. U nastavku slijedi opis navedenih servisa:

Servis	**Konfiguracija**	**Opis**
HttpClient	Scoped	Pruža metode za slanje HTTP zahtjeva i primanje HTTP odgovora putem resursa definiranim URI-em
IJSRuntime	Singleton (WebAssembly) Scoped (Server)	Predstavlja instancu Javascript izvršnog okruženja i omogućuje pozivanje Javascript metoda (onih koje pruža Web API ili onih definiranih od strane korisnika)
NavigationManager	Singleton (WebAssembly) Scoped (Server)	Sadrži pomoćne metode za rad sa URL-om i upravljanjem stanjem navigacije

Tablica 2: Servisi zadani od Blazor programskog okvira

4.4. Blazor WebAssembly

Blazor WebAssembly je inačica Blazor programskog okvira koja odvaja ovisnost aplikacije od poslužitelja i izvodi se u potpunosti na korisnikovom web pregledniku pa se zbog toga nekada i naziva klijentski (engl. *client-side*) Blazor [36]. Shodno tome, nije potrebna stalna HTTP veza kao što je to slučaj sa Blazor Server inačicom. Takav arhitekturalni model je moguć zahvaljujući primarno web standardima, a najviše novom WebAssemblyu. Prva pomisao mnogih bila je da se radi o još jednom dodatku kao što je bio propali Microsoft Silverlight koji je zahtjevao instalaciju dodatka u web preglednik kako bi pokretao WPF aplikacije sa vrlo ograničenim mogućnostima, ali Blazor WebAssembly to definitivno nije. Blazor WebAssembly nije dodatak, ni transpilirani JavaScript a ni skup gotovih web kontrola kao što je to nudio ASP.NET WebForms. Radi se o programskom okviru koji kompilira C# kôd u WebAssembly i izvodi ga u pregledniku uz korištenje standardnih web tehnologija i sučelja te standardnih .NET biblioteka (detaljnije u sljedećem potpoglavlju). Ovdje opet valja istaknuti da nisu sve biblioteke .NET standardna dostupne u pregledniku, već je njihov skup ograničen. S time na umu, ova inačica je pogodna za statičke web aplikacije (kao što su i React/Angular) samo dohvaćaju podatke putem AJAX poziva i ne zahtjevaju biblioteke koje se mogu izvoditi samo na poslužiteljskoj strani.

4.4.1. Kako funkcionira Blazor WebAssembly?

Već je nekoliko puta rečeno da se Blazor WebAssembly aplikacije izvode u potpunosti u korisnikovom pregledniku jer se cijeli C# kôd kompilira u WebAssembly. Međutim, postavlja se pitanje: kako je to moguće ako ne postoji izvršno okruženje da pokreće kôd? Stvar je u tome da se zajedno sa komponentama, stranicama i ostalim projektnim datotekama u WebAssembly kompilira i prilagođeno .NET okruženje koje se naziva Mono.

Mono je kompletna platforma otvorenog kôda koja se temelji na .NET programskom okviru i omogućava razvojnim programerima izgradnju aplikacija koje se izvode na različitim platformama [37]. To je moguće zbog dizajna platforme kao takve te Mono izvršnog okruženja koje implementira ECMA opću infrastrukturu jezika (engl. *CLI, common language infrastructure*) i ne sadrži biblioteke specifične platformi na kojoj se izvodi. Shodno tome, aplikacije se mogu izvoditi na Linux-u, macOS-u, Windows-u te bilo kakvim računalima sa x86, x86-64 ili ARM arhitekturom procesora. Zato je bilo bitno da se prilikom dizajna ovog okvira nije implementiralo izvršno okruženje kao što je standardni .NET koji bi radio samo na Windows platformi, već okruženje kao što je Mono koje se može izvoditi na svim platformama - baš kao i WebAssembly.

Kada se Blazor WebAssembly aplikacija izgradi i pokrene u web pregledniku (putem komandne linije ili putem integriranog razvojnog okruženja), prvo se sav kôd iz jezika više razine (C#, F# itd.) i Razor datoteke kompiliraju u .NET binarne datoteke ili DLL-ove [35]. Zatim, cijelo Mono izvršno okruženje zajedno s tim datotekama prenosi se u web preglednik. Konačno, Blazor na klijentskoj strani (`blazor.webassembly.js`) koji služi kao glavni okvir aplikacije radi pokretanje .NET okruženja i učitavanje binarnih datoteka aplikacije. On također pruža Javascript interoperabilnost te s time omogućuje klijentskoj strani upravljanje objektnim modelom

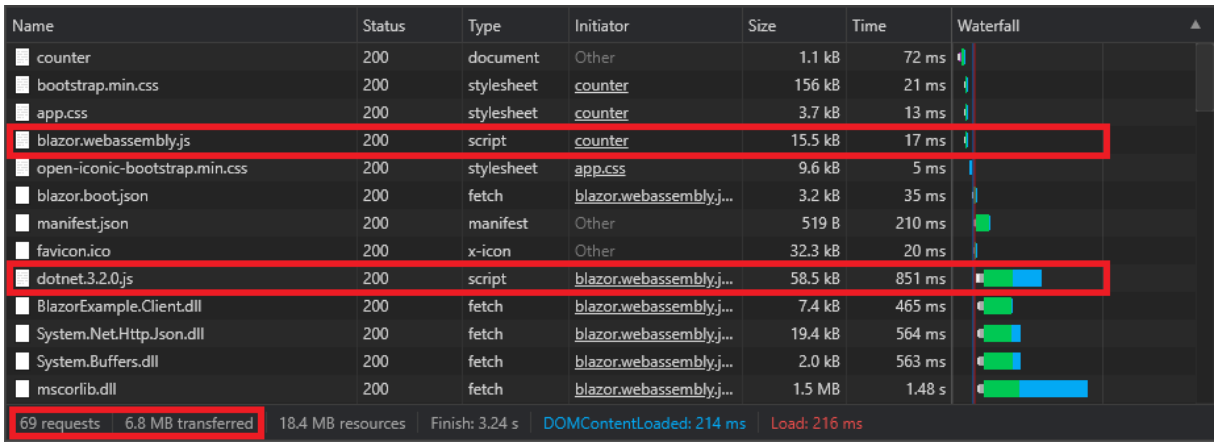
dokumenta (engl. *DOM manipulation*).

Radi demonstracije, pokrenuta je Blazor WebAssembly ogledna aplikacija u pregledniku te je uzet isječak iz kartice Mreža kako bi se vidjele sve datoteke koje su preuzete u preglednik (vidi Sliku 17). U preglednik se, osim statičkih datoteka (ikone, postavke, stilske upute), preuzelo još i mnogo drugih datoteka od kojih su najbitnije sljedeće:

1) `blazor.webassembly.js` - Javascript biblioteka koja služi kao glavni okvir aplikacije i omogućuje Javascript interoperabilnost, a samo nju je potrebno uključiti u HTML stranicu (vidi sliku ispod).

2) `dotnet.3.2.0.wasm` - Mono izvršno okruženje kompilirano u Wasm modul (nije prikazano na slici).

3) `dotnet.3.2.0.js` - Javascript biblioteka koja služi kao omotač za prethodno navedeno izvršno okruženje kojega ono uljučuje i izvodi sve operacije specifične okruženju (vidi sliku ispod).



Slika 18: Blazor WebAssembly aplikacija u pregledniku

Također valja primjetiti da je web preglednik napravio 69 FTP zahtjeva, odnosno po jedan za svaku datoteku koju je bilo potrebno preuzeti i sveukupno preuzeo čak 6.8 megabajta podataka. Usporedbe radi, veličina prosječne zapakirane React ili Angular aplikacije kada se preuzme u pregledniku je manja od jedan megabajt. Iako ove 3 prethodno navedene biblioteke obavljaju sve operacije, na slici je moguće primjetiti da one zapravo zauzimaju najmanje mjesta već su najveći "teret" DLL (engl. *dynamic link library*) datoteke koje čine srž Mono izvršnog okruženja. Tako naprimjer *mscorlib.dll* biblioteka koja je jezgra cijelog okruženja sama zauzima 1.5 megabajta. Zbog tolikog tereta, Blazor WebAssembly imaju znatno sporije učitavanje u usporedbi sa Blazor Server aplikacijama, ali samo prvi puta kada se pokreću. Razlog tomu je taj što se većina ovih datoteka sprema u privremenu memoriju web preglednika, pa ih on već nakon sljedećeg pokretanja može samo učitati iz privremene memorije pa se vrijeme učitavanja smanjuje nekoliko desetaka puta. Također, već je spomenuto da Javascript ne podržava višedretvenost pa proces učitavanja dodatno usporava činjenica da se sve odvija na glavnoj (UI) dretvi.

4.4.2. Prednosti i nedostatci

Izvođenje .NET kôda u pregledniku sa modernim i bogatim korisničkim sučeljem uvijek zvuči prekrasno, ali dolazi sa svojim nedostatcima. Neki od njih su već navedeni u prethodnom potpoglavlju, a u ovom će ukratko biti rezimirani i ostali nedostatci te usporedba pojedinih segmenata sa drugom inačicom Blazor Server.

Prednosti Blazor WebAssembly inačice su sljedeće [38]:

- ✓ **Niski dodatni troškovi zadaća** - s obzirom da se sve izvodi u web pregledniku, postoje vrlo mali dodatni troškovi (engl. *overhead*) kod izvođenja osnovnih zadaća jer se ne mora čekati na poslužiteljski dio da obavi svoj posao ili na razmjenu SignalR poruka kao što je to slučaj kod Blazor Server inačice.
- ✓ **RESTful tip aplikacije** - postoji samo jedno izdvojeno (odspojeno okruženje), a za interakciju sa drugim sustavima (dohvaćanje i slanje podataka) koristi se Blazor-ov HTTP klijent i ostali Web API koje pruža preglednik.
- ✓ **Mogućnost izvanmrežnog rada** - s obzirom da poslužiteljski dio ne postoji, aplikacija koje ne ovise o drugim sustavima već postoje samo statičke komponente mogu raditi i u izvanmrežnom (engl. *offline*) načinu.
- ✓ **Progresivna web aplikacija** - postoji mogućnost da se kreirana web aplikacija jednostavno prilagodi u progresivnu web aplikaciju i tako pruži korisniku doživljaj nativne stolne odnosno mobilne aplikacije.
- ✓ **Jednostavno postavljanje** - kod statične Blazor WebAssembly aplikacije, vrlo je jednostavno postavljanje na poslužitelj jer isti ne mora imati instalirane nikakve određene tehnologije potrebne za pokretanje (kao Apache za PHP, Tomcat za Javu itd) jer je sve zapravo Javascript i Wasm kod kojeg web preglednik može pokretati samostalno. Tako je naprimjer moguće i aplikaciju postaviti na GitHub Pages servis u samo nekoliko klikova.

Nedostatci Blazor WebAssembly inačice su sljedeći [38]:

- ✗ **Sporo prvo pokretanje** - kao što je već objašnjeno, zbog veličine izvršnog okruženja i učitavanje na samo jednoj dretvi, prvo pokretanje aplikacije u pregledniku može biti znatno sporije u usporedbi sa Blazor Server aplikacijama.
- ✗ **Odspojeno okruženje** - kako se radi o odspojenom okruženju (engl. *disconnected environment*) bez poslužitelja, ne mogu se koristiti brojne biblioteke i klase koje .NET nudi (za rad sa bazom podataka, ulazno-izlazne operacije i slično) već je potrebno slati Web zahtjeve prema vanjskim servisima koji će obaviti traženu zadaću, dok Blazor Server aplikacije imaju pristup svim sučeljima koje jednostavno pozivaju slanjem SignalR poruke.
- ✗ **Sigurnost** - s obzirom da se cijelo okruženje zajedno sa kompiliranim kôdom i DLL-ovima preuzima u korisnikov web preglednik, svaki klijent ima uvid u programski kôd aplikacije

i to predstavlja problem iz sigurnosnog aspekta. Međutim, to nije problem svojstven Blazoru ili WebAssemblyu već klijentske strane općenito jer se i kod bilo koje druge Javascript aplikacije isto preuzima u web preglednik i također ga je moguće pregledati. S time na umu, atribute poput API ključeva ili korisničkih podataka nikada ne treba stavljati kao dio kôda, već učitavati iz varijabli okruženja ili slično.

Potrebno je naglasiti da se prethodnom listom prednosti i nedostataka te usporedbom sa Blazor Server inačicom nije ni na koji način htjelo pokazati da je jedna od njih bolja odnosno gora. Svaka od njih ima svoje prednosti i nedostatke, te svaka od njih ima svoju primjenu ovisno o sustavu koji se razvija i njegovim slučajevima korištenja.

5. Migracija .NET WinForms aplikacije u Blazor WebAssembly

U ovom poglavlju opisan je postupak migracije .NET Windows Forms u Blazor WebAssembly (posluženom na .NET Core 3.1 poslužitelju), odnosno migracije stolne aplikacije u web aplikaciju. Poseban naglasak stavljen je na sličnosti i razlike u radu .NET stolnih i web aplikacija, to jest dijelove aplikacije koji se lako migriraju i one dijelove čija migracija predstavlja problem. Bitno je i naglasiti da će buduća web aplikacija preslikavati odabranu postojeću stolnu aplikaciju u smislu njenih funkcionalnosti i dizajna, no ne i u načinu implementacije tih funkcionalnosti već će pratiti najbolje tehnike i prakse koje se koriste u razvoju *Blazor* aplikacije i web razvoju općenito.

Za migraciju odabrana je ProphetShop aplikacija (https://github.com/isaiasvallejos/shop) koja zadovoljava sve uvjete definirane u poglavlju 2. Metode i tehnike rada. Radi se o jednostavnoj aplikaciji za trgovinu odjeće i obuće koja ima dvije vrste korisnika: klijent (engl. *customer*) i voditelj (engl. *manager*). Klijenti se moraju registrirati i zatim mogu kreirati narudžbe, te imaju pregled svoje košarice i aktivne narudžbe. Također, mogu ažurirati svoj profil sa novim korisničkim podacima. Voditelji kreiraju nove proizvode te mogu izmijenjivati i brisati one stare, te obrađuju narudžbe proizvoda kreiranih od strane klijenata. Prilikom obrade, svakoj narudžbi pridružuje se (i kod promjene ažurira) status narudžbe (u tijeku, otkazano, odobreno, isporučeno itd.). Struktura projekta je sljedeća:

Slika 19: Struktura ProphetShop projekta

U direktoriju `App` nalaze se klase vezane uz stanje aplikacije: klasu `Session` koja predstavlja aktivnog korisnika te klasu `ShopContext` koja predstavlja kontekst baze podataka. Unutar direktorija `Business` nalaze se klase koje sadržavaju poslovnu logiku pripadajućeg modela, odnosno jednostavne algoritme za izračunavanje popusta, dostave, ukupne vrijednosti proizvoda i sl. Zatim, u mapi `Components` nalaze se korisničke kontrole koje zajedno čine korisničko sučelje aplikacije. Unutar mape `DAO` (engl. *data access object*) nalazi se sloj za pristup podacima koji zapravo sadrži kontekst baze podataka i koristeći njega omogućuje dohvaćanje i pohranu podataka. Za svaki model postoji pripadajuća klasa koja služi za upravljanje njegovim podacima. Mapa `Migrations` sadrži promjene iz poslovnih modela njihove pripadajuće entitete u bazi podataka, a zapravo je dio *Entity Framework* okvira za objektno-relacijsko mapiranje kojega koristi stolna aplikacija i nema nikakve veze sa migracijama koje se opisuju u

ovom poglavlju. Posljednje, mapa `Models` sadrži poslovne modele s kojima aplikacija radi, dok mapa `Util` sadrži uslužne klase korištene u ostatku aplikacije.

5.1. Analiza prenosivosti i početne postavke

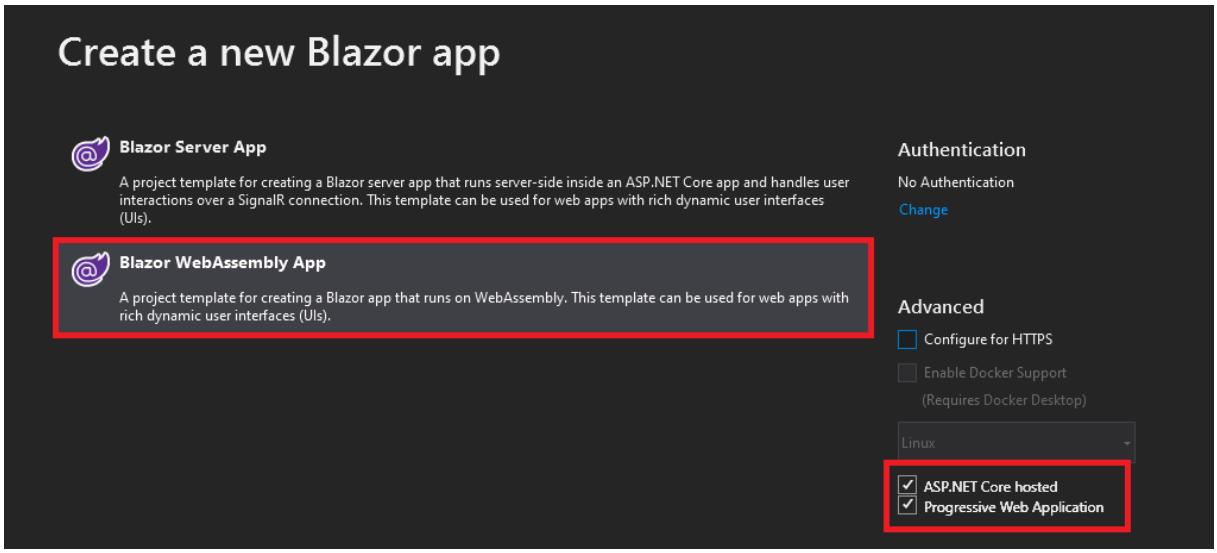
Prije početka postupka migracije, potrebno je provesti analizu prenosivosti .NET aplikacije uz pomoć besplatnog alata *.NET Portability Analyzer* dostupnog na *Visual Studio Marketplace* trgovini. Nakon što je alat dodan, unutar Visual Studio razvojnog okruženja potrebno je ići na **Analyze > Portability Analyzer Settings** na alatnoj traci i unutar opcija označiti **.NET Core 3.1** kao ciljnu platformu i označiti .HTML format izvještaja. Nakon spremanja postavki, desnim klikom na projekt nudi se opcija *Analyze Project Portability* čijim odabirom se otvara izvještaj u web pregledniku:

Slika 20: .NET Portability Report

Iako je 100% najpoželjniji mogući rezultat, ovdje je potrebno naglasiti da ovo ne znači da cijeli projekt može samo "kopirati i zalijepiti" ili jednim klikom gumba prenijeti u web aplikaciju. Ovo se odnosi samo na poslužiteljski dio aplikacije koji se pokreće na *.NET Core 3* poslužitelju, dok je klijentski dio aplikacije koji sadrži korisničko sučelje svakako potrebno iznova konstruirati. Isto tako, alat radi na principu analiziranja biblioteka i provjeravanja postoji li njihov ekvivalent u ciljnoj platformi, pa ako je naprimjer u *.NET Framework* aplikaciji korištena biblioteka *EntityFramework* (kao što je to ovdje slučaj), alat će prepoznati da ju je moguće migrirati u *.NET Core 3* okruženje sa ekvivalentnom bibliotekom *Microsoft.EntityFramework.Core* i tu neće javljati odstupanje. Nadalje, buduća web aplikacija se više neće pokretati unutar jednog okruženja kao što je to slučaj sa stolnom aplikacijom već će postojati klijentski dio koji se izvršava u pregledniku te poslužiteljski dio koji obrađuje zahtjeve pa sama implementacija te veze između strana predstavlja novi skup problema. Upravo taj skup problema i njihovo rješavanje je opisano u sljedećim potpoglavljima.

Za početak potrebno je unutar razvojnog okruženja kreirati novi projekt klikom na **File > New > Project** te odabrati *Blazor App* i zatim odrediti naziv projekta, te nakon toga u dijaloškom okviru odabrati postavke prema Slici 21. Ovdje je najvažnije označiti opciju *ASP.NET Core hosted* koja

u principu omogućuje da se obje strane web aplikacije poslužuju na istoj mrežnoj utičnici (engl. *port*). Zahvaljujući tome, obje strane imaju isto ishodište (engl. *origin*) i mogu jednostavno komunicirati, dok bi u drugom slučaju bilo potrebno postavljati složene CORS opcije kako bi se omogućila njihova komunikacija putem HTTP zahtjeva.



Slika 21: Kreiranje novog Blazor WebAssembly projekta

Nakon pritiska na gumb *Create*, okruženje će kreirati rješenje koje se sastoji od 3 projekta:

1) `BlazorShop.Client` - predstavlja klijentski dio aplikacije koji sadržava korisničke kontrole zajedno sa njihovim rukovateljima događaja, validaciju formi, aplikacijsko stanje te dio sloja sigurnosti (autentifikacija i autorizacija)
2) `BlazorShop.Server` - predstavlja poslužiteljski dio aplikacije koji sadržava sloj za pristup podacima u obliku repozitorija, poslovnu logiku te sloj za pristup aplikacijskim resursima
3) `BlazorShop.Shared` - predstavlja dijeljeni dio aplikacije kojega koriste oba sloja, stoga je najbolja praksa ovdje držati poslovne modele i objekte za prijenos podataka

5.2. Migracija podatkovnog sloja

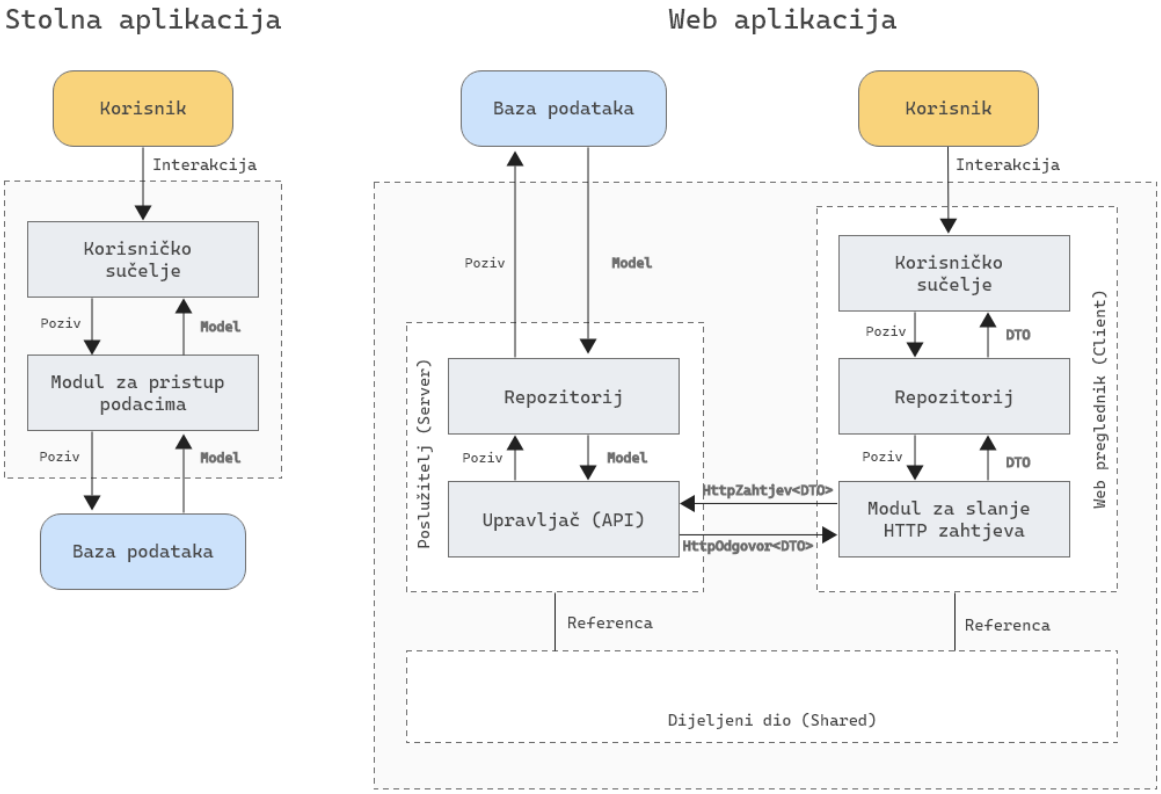
S obzirom da je podatkovni sloj jedan od ključnih slojeva bez kojeg aplikacija ne može raditi, taj sloj je potrebno prvo migrirati. U stolnoj aplikaciji je za pristup bazi podataka korišten *Entity Framework* programski okvir za objektno-relacijsko mapiranje, a kako bi aplikacije radile sa jednakim modelima taj isti okvir će biti korišten i u web aplikaciji. Shodno tome, potrebno je unutar projekta `BlazorShop.Server` otvoriti `Package Manager` konzolu i unijeti sljedeće naredbe kako bi se instalirali potrebni paketi:

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Prvi navedeni paket sadrži srž programskog okvira dok drugi sadrži spojnik (engl. *connector*) za SQL Server bazu podataka. Posljednji navedeni paket pruža dodatne alate za rad s bazom podataka unutar komandnog sučelja i sasvim je opcionalan.

5.2.1. Razlike u toku podataka

Prije nego što se započne se migracijom podatkovnog sloja, potrebno je prikazati razlike u toku podataka između .NET stolne i web aplikacije kako bi bilo jasno zašto se pojedini dio sloja migrirao na određeni način.



Slika 22: Razlike u tokovima podataka između stolne i web .NET aplikacije

Naprimjer, unutar stolne aplikacije korisnik će svojom interakcijom otvoriti formu za prikaz liste proizvoda, i zatim će forma (Korisničko sučelje) pozvati određenu metodu modula za pristup podacima koji će pozvati određenu metodu iz konteksta baze podataka i dobiti listu modela proizvoda kao odgovor. Ta lista modela se proslijeđuje nazad do korisničkog sučelja te se iz modela iščitavaju podaci i prikazuju na sučelju.

S druge strane, unutar web aplikacije korisnik će svojom interakcijom otvoriti stranicu za prikaz liste proizvoda, i zatim će ta komponenta (Korisničko sučelje) pozvati određenu metodu repozitorija za proizvode. U ovom kontekstu, repozitorij je samo uzorak dizajna (Repository Pattern) koji se ponaša kao sloj apstrakcije nad kontekstom baze podataka i zapravo je ekvivalentan modulu za pristup podacima u stolnoj aplikaciji. Razlog postojanja takvog sloja na klijentskoj strani je taj što komponente ne bi smjele direktno zvati modul za slanje HTTP zahtjeva jer bi to rezultiralo neodrživim komponentama koje sadrže previše logike. Umjesto toga, klijentski repo-

zitorij sakriva svu tu logiku od komponente i omogućava joj da samo pozove određenu metodu koja će dohvatiti tražene podatke. Isto tako, poslužiteljski repozitorij služi da upravljač (API) ne bi imao previše poslovne logike pa se unutar njega najčešće implementiraju složeni upiti nad bazom podataka i slično. Nastavno na tok podataka, repozitorij poziva određenu funkciju modula za slanje HTTP zahtjeva (`HTTPClient` servis kojega nudi Blazor) i on šalje HTTP zahtjev prema određenoj krajnjoj točki na kojoj se nalazi upravljač (engl. *Controller*) koji je zadužen za obradu HTTP zahtjeva. On poziva određenu metodu repozitorija koja opet iz baze dohvaća modele baze podataka sve do upravljača. Međutim, kada se podaci vraćaju prema klijentu tada upravljač radi transformacije modela u objekte za prijenos podataka (engl. *data transfer object - DTO*). Takvi objekti služe za enkapsulaciju minimalnih potrebnih podataka za izvršavanje funkcija iz jednog podsustava u drugi. Razlog njihova korištenja je taj što uvelike smanjuju količine podataka koje se prijenose iz jedne strane u drugu, a to je količina tereta koji sadržava HTTP zahtjev je bitan faktor u radu web aplikacije. Naprimjer, ukoliko je za prikaz komponente proizvoda na sučelju potrebno samo naziv proizvoda i cijena, nije potrebno u odgovoru slati još i kategoriju, sliku, popust, zalihu i ostalo. Ovo možda neće imati neki preveliki utjecaj ako se radi o 5 proizvoda, ali ukoliko se na sučelju prikazuju nekoliko stotina proizvoda onda definitivno hoće. Ovakvi troškovi kod stolne aplikacije ne postoje jer se radi o "debelom klijentu" koji može direktno pristupiti bazi podataka bez slanja HTTP zahtjeva prema vanjskom servisu, dok je kod web aplikacije zbog ograničenja preglednika uvijek potrebno slati zahtjev prema vanjskom servisu za izvođenje bilo kakve složenije radnje. Sličan primjer je i sa kreiranjem proizvoda putem forme na korisničkom sučelju u kojem slučaju se šalje objekt za prijenos podataka koji sadrži minimalne podatke kako bi se mogao kreirati model (bez ID-a, datuma kreiranja i ostaloga). Shodno tome, repozitorij klijentske strane uvijek radi sa DTO-ovima jer komponente rade samo sa DTO-ovima dok repozitorij poslužiteljske strane uvijek radi sa modelima kako bi ih mogao dohvaćati i spremati u bazu podataka jer baza podataka radi isključivo sa modelima.

5.2.2. Migracija poslovnih modela i konteksta baze podataka

Migracija poslovnih modela vrlo je jednostavna jer su oni ni više ni manje nego POCO (engl. *plain old CLR object*) klase koje sadrže samo primitivna svojstva i stoga ih je moguće samo kopirati i zalijepiti u željeni projekt. Međutim, kontekst baze podataka nije moguće samo kopirati jer sadrži reference na biblioteke koje ne postoje u *.NET Core 3* okruženju. Stoga, potrebno je unutar `Package Manager` konzole izvršiti naredbu koja će automatski kreirati modele iz baze podataka i njen kontekst:

```
Scaffold-DbContext "Server=LAPTOP-5ALT62PH;Database=BlazorShop;
Trusted_Connection=True;MultipleActiveResultSets=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Data
```

Nakon pokretanja naredbe će se unutar mape `Data` na poslužiteljskom dijelu kreirati svi poslovni modeli (mapirani iz entiteta baze podataka) i kontekst baze podataka. Kreirani entiteti trebaju biti doslovno jednaki onima koji već postoje na stolnoj aplikaciji. S obzirom da se oni koriste u obje strane aplikacije, potrebno ih je premjestiti u `BlazorShop.Shared` projekt unutar novokreirane mape `Models` i podesiti polje imena. Zatim, kako bi se kontekst baze podataka

(`DbContext`) mogao koristiti kao servisna klasa, potrebno ju je registrirati unutar DI spremišta sa postavkama koje sadrži vezu baze podataka:

```
services.AddDbContext<BlazorShopContext>(option =>
 ↪  option.UseSqlServer(Configuration.GetConnectionString
("DefaultConnection")));
```

Isječak kôda 5.1: Dodavanje konteksta baze podataka u DI spremište

S obzirom da unutar postavki još nisu definirani podaci za spajanje na SQL Server bazu podataka, potrebno je na `BlazorShop.Server` projektu ažurirati `appsettings.json` datoteku i unutar postojeće sekcije `ConnectionStrings` kreirati stavku:

```
"DefaultConnection": "Server=LAPTOP-5ALT62PH;Database=BlazorShop;
Trusted_Connection=True;MultipleActiveResultSets=True;"
```

Isječak kôda 5.2: Dodavanje podataka o vezi na bazu podataka unutar postavki

S time je omogućeno spajanje na bazu podataka i rad s podacima baze podataka, pa je migracija poslovnih modela i konteksta baze podataka završena.

5.2.3. Migracija sloja za pristup podacima

Sloj za pristup podacima je jedan od najkompleksnijih slojeva u web aplikaciji i postoji na klijentskoj strani kao i na poslužiteljskoj strani. Preporuča se da migracija krene od poslužiteljske strane jer ju je moguće testirati putem stolnih API klijenata kao što je npr. *Postman*. Shodno modelu prikazanom u prethodnom poglavlju, prvo se kreira poslužiteljski repozitorij. Dobra je praksa kod .NET Core aplikacija prvobitno kreirati generički repozitorij (Generic Repository Pattern) - način implementacije repozitorija koji radi na još jednom višem nivou apstrakcije, a sadrži osnovne metode za rad sa podacima svakoga modela. Prvo se kreira sučelje:

```
public interface IRepository<TEntity> where TEntity : class
{
    TEntity Get(object id);
    ValueTask<TEntity> GetAsync(object id);
    IEnumerable<TEntity> GetAll();
    void Add(TEntity entity);
    void AddRange(IEnumerable<TEntity> entities);
    void Remove(TEntity entity);
    TEntity Single(Expression<Func<TEntity, bool>> predicate);
    Task<TEntity> SingleAsync(Expression<Func<TEntity, bool>>
     ↪  predicate);
    void RemoveRange(IEnumerable<TEntity> entities);
}
```

Isječak kôda 5.3: Kreiranje sučelja generičkog repozitorija

Moguće je vidjeti da se ovdje nalaze sve osnovne metode za rad sa entitetima iz baze podataka: dohvaćanje jednog entiteta, dohvaćanje liste entiteta, brisanje jednog entiteta, brisanje liste entiteta itd. Nakon kreiranja sučelja, potrebno je isto implementirati:

```
public class Repository<TEntity> : IRepository<TEntity> where
 ↪  TEntity : class
{
    protected readonly DbContext _context;
    protected readonly DbSet<TEntity> _entities;

    public Repository(DbContext context)
    {
        _context = context;
        _entities = _context.Set<TEntity>();
    }

    public virtual void Add(TEntity entity)
    {
        _entities.Add(entity);
    }

    public virtual void AddRange(IEnumerable<TEntity> entities)
    {
        _entities.AddRange(entities);
    }

    public virtual TEntity Get(object id)
    {
        return _entities.Find(id);
    }

    public virtual IEnumerable<TEntity> GetAll()
    {
        return _entities.AsEnumerable();
    }

    public virtual ValueTask<TEntity> GetAsync(object id)
    {
        return _entities.FindAsync(id);
    }

    public void Remove(TEntity entity)
    {
        _entities.Remove(entity);
    }

    public virtual void RemoveRange(IEnumerable<TEntity>
     ↪  entities)
    {
        _entities.RemoveRange(entities);
    }
```

```
        public TEntity Single(Expression<Func<TEntity, bool>>
        ↪  predicate)
        {
            return _entities.SingleOrDefault(predicate);
        }

        public Task<TEntity> SingleAsync(Expression<Func<TEntity,
        ↪  bool>> predicate)
        {
            return _entities.SingleOrDefaultAsync(predicate);
        }
    }
```

Isječak kôda 5.4: Implementacija sučelja generičkog repozitorija

Moguće je primjetiti da repozitorij koristi kontekst baze podataka koji je kreiran u prethodnom koraku. Samim time, kod rada sa podacima upravljačka klasa neće pozivati metode konteksta direktno nego će pozivati repozitorij. Zatim je potrebno kreirati repozitorije za svaki model odnosno entitet baze podataka. Postavlja se pitanje: zašto je potrebno kreirati repozitorij za svaki pojedinačni entitet ukoliko već postoji generički repozitorij? Razlog tomu je taj što će ti repozitoriji samo nasljeđivati ovaj generički i dodavati napredne metode po potrebi (složenije upite od ovih postojećih osnovnih). Nadalje, ti repozitoriji također moraju implementirati vlastito sučelje kako bi ih se moglo registrirati kao servisne klase unutar DI spremišta. S primjenom ovakvih repozitorija poboljšava se fleksibilnost sustava i iskorištava najmoćnije svojstvo *.NET Core 3* i *Blazor WebAssembly* okvira, a to je ubrizgavanje ovisnosti. Radi usporedbe, modul za pristup podacima (DAO) na stolnoj aplikaciji implementiran je na način da klase za pristup podacima sadrže samo statičke metode unutar kojih se kreira kontekst baze podataka i na kraju korištenja odlaže uz pomoć `using` klauzule. Ovo je ujedno i najčešći način implementacije sloja za pristup podacima na stolnim .NET aplikacijama s obzirom da okvir kao takav ne nudi svojstvo ubrizgavanja ovisnosti.

Sljedeće je potrebno kreirati sučelje repozitorija za pristup podacima entiteta proizvoda:

```
public interface IProductRepository : IRepository<Product>
{
    Task<IEnumerable<Product>> GetProductsByCategory(int id);
}
```

Isječak kôda 5.5: Kreiranje sučelja repozitorija za upravljanje proizvodima

Sučelje za početak ima samo jednu metodu za dohvaćanje svih proizvoda ovisno o njihovoj kategoriji, a kasnije će se po potrebi dodavati i druge metode. Treba primjetiti da sučelje nasljeđuje sučelje generičkog repozitorija sa definiranim entitetom `Product`. Ono što je također na prvi pogled vidljivo kod `IProductRepository` sučelja njegova jednostavnost. Kada ne bi

postojao generički repozitorij kao nadklasa, tada bi u kôdu postojalo šest repozitorija koji bi sadržavali devet gotovo jednakih metoda koje bi se razlikovale jedino po potpisu metode. Ovakav pristup omogućava da podklase repozitorija sadržavaju vrlo malo kôda, a ukoliko za neki entitet nisu potrebne nikakve složenije metode tada takav repozitorij doslovno može implementirati prazno sučelje. Nakon toga, potrebno je implementirati sučelje:

```
public class ProductRepository : Repository<Product>,
 ↪  IProductRepository
{
    protected BlazorShopContext BlazorShopContext => _context as
     ↪  BlazorShopContext;

    public ProductRepository(BlazorShopContext context) :
     ↪  base(context) { }

    public async Task<IEnumerable<Product>>
     ↪  GetProductsByCategory(int id)
    {
        return await _entities.Where(p => p.CategoryId ==
         ↪  id).ToListAsync();
    }
}
```

Isječak kôda 5.6: Implementacija sučelja repozitorija za upravljanje proizvodima

Sa završenim početnim repozitorijem moguće je krenuti na implementaciju upravljačkih klasa koji će služiti kao servis klijentskoj aplikaciji (API) tako što će obrađivati njegove HTTP zahtjeve i pozivati metode prethodno kreiranih repozitorija ili druge servisne klase. Unutar *.NET Core* okruženja, upravljačke klase moguće je kreirati desnim klikom na već postojeću mapu `Controllers` i zatim **Add > Controller > API Controller - Empty**. Važno je da upravljačka klasa ima sufiks *Controller* jer se putanja do krajnje točke određuje putem prefiksa naziva klase, odnosno cijelog naziva što ide prije navedenog sufiksa. Naprimjer, putanja do ove upravljačke klase biti će `api/product`, ali to je uvijek moguće izmijeniti putem dekoratora iznad definicije klase. Za početak, klasa će sadržavati samo dvije jednostavne metode: za dohvaćanje svih proizvoda, i novokreiranu metodu za dohvaćanje proizvoda putem ID-a kategorije.

```
[Route("api/[controller]")]
[ApiController]
public class ProductController : ControllerBase
{
    private readonly IProductRepository _productRepository;

    public ProductController(IProductRepository
     ↪  productRepository)
    {
        _productRepository = productRepository;
    }
```

```
        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            return Ok(await _productRepository.GetAll());
        }

        [HttpGet("bycategory/{id}")]
        public async Task<IActionResult> GetAllByCategory([FromRoute]
        ↪  int id)
        {
            return Ok(await
            ↪  _productRepository.GetProductsByCategory(id));
        }
    }
```

Isječak kôda 5.7: Upravljačka klasa za upravljanje proizvodima

Kada bi se sada pokušao testirati ovaj API putem nekoga klijenta, korisnik bi dobio poruku o internoj pogrešci a to je zato što repozitorij koji je ubrizgan u konstruktoru nije registriran unutar DI spremišta. Stoga, potrebno je unutar `Startup.cs` klase u metodi `ConfigureServices` dodati sljedeću liniju kôda:

```
services.AddScoped<IProductRepository, ProductRepository>();
```

Isječak kôda 5.8: Registracija repozitorija za upravljanje proizvodima u DI spremište

Nakon ovoga moguće je testirati API putem nekog stolnoj API klijenta i dobiti sve proizvode u odgovoru. Sukladno dijagramu toka podataka prikazanom u prethodnom poglavlju, sljedeći element u sustavu koji je potrebno realizirati je onaj koji će omogućiti slanje i primanje HTTP zahtjeva na korisničkoj strani. On kao takav već postoji na klijentskoj strani, a radi se o `HTTPClient` servisu kojeg Blazor WebAssembly već nudi u gotovome. Međutim, on kao takav može raditi sa osnovnim tipovima podataka i u tijelu poruke može biti jedino niz znakova. Samim time, sve DTO-ove koji se šalju kao zahtjev prema poslužiteljskoj strani prvo je potrebno serijalizirati u niz znakova. Upravo zbog toga je najbolja praksa napraviti napraviti servisnu klasu koja će omotavati `HTTPClient` i dodati još sloj za serijalizaciju i deserijalizaciju podataka. Unutar `BlazorShop.Client` projekta kreira se sučelje sa osnovnim HTTP metodama:

```
public interface IHttpService
{
    Task<HttpResponse<T>> Get<T>(string url);
    Task<HttpResponse<object>> Post<T>(string url, T data);
    Task<HttpResponse<object>> Delete(string url);
    Task<HttpResponse<object>> Put<T>(string url, T data);
}
```

Isječak kôda 5.9: Kreiranje sučelja za slanje HTTP zahtjeva putem HTTP klijenta

Važno je da odgovori svake metode budu omotani u `HttpResponse` omotač koji će odgovoru osim tijela poruke pridodati i još neke bitne informacije koje se generiraju na poslužitelju, a bitne su za otkrivanje grešaka. Zatim je potrebno implementirati sučelje:

```
public class HttpService : IHttpService
{
    private readonly HttpClient _httpClient;
    private JsonSerializerOptions defaultJsonSerializerOptions =>
        new JsonSerializerOptions() { PropertyNameCaseInsensitive =
        ↪ true };

    public HttpService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<HttpResponse<T>> Get<T>(string url)
    {
        var responseHttp = await _httpClient.GetAsync(url);

        if (responseHttp.IsSuccessStatusCode)
        {
            var response = await Deserialize<T>(responseHttp,
            ↪ defaultJsonSerializerOptions);

            return new HttpResponse<T>(response, true, responseHttp);
        }
        else
        {
            return new HttpResponse<T>(default, false, responseHttp);
        }
    }

    public async Task<HttpResponse<object>> Post<T>(string url, T
    ↪ data)
    {
        var dataJson = JsonSerializer.Serialize(data);
        var stringContent = new StringContent(dataJson,
        ↪ Encoding.UTF8, "application/json");
        var response = await _httpClient.PostAsync(url,
        ↪ stringContent);

        return new HttpResponse<object>(null,
        ↪ response.IsSuccessStatusCode, response);
    }

    public async Task<HttpResponse<object>> Put<T>(string url, T
    ↪ data)
```

```
    {
        var dataJson = JsonSerializer.Serialize(data);
        var stringContent = new StringContent(dataJson,
        ↪ Encoding.UTF8, "application/json");
        var response = await _httpClient.PutAsync(url,
        ↪ stringContent);

        return new HttpResponse<object>(null,
        ↪ response.IsSuccessStatusCode, response);
    }

    public async Task<HttpResponse<object>> Delete(string url)
    {
        var responseHTTP = await _httpClient.DeleteAsync(url);

        return new HttpResponse<object>(null,
        ↪ responseHTTP.IsSuccessStatusCode, responseHTTP);
    }

    private async Task<T> Deserialize<T>(HttpResponseMessage
    ↪ httpResponse, JsonSerializerOptions options)
    {
        var responseString = await
        ↪ httpResponse.Content.ReadAsStringAsync();
        return JsonSerializer.Deserialize<T>(responseString,
        ↪ options);
    }
}
```

Isječak kôda 5.10: Implementacija generičkog HTTP klijenta

Nakon kreiranja modula za slanje HTTP zahtjeva, potrebno je kreirati repozitorij za rad sa podacima proizvoda. Metode sučelja repozitorija na klijentskoj strani uvijek moraju preslikavati metode pripadajućeg repozitorija na poslužiteljskoj strani:

```
public interface IProductRepository
{
    Task<List<Product>> GetAll();
    Task<List<Product>> GetAllByCategory(int id);
}
```

Isječak kôda 5.11: Kreiranje sučelja za upravljanje proizvodima na klijentskoj strani

Važno je povratni tip metoda uvijek definirati kao `Task` koji predstavlja asinhronu radnju kako bi aplikacija u web pregledniku mogla nesmetano nastaviti sa radom dok se u pozadini šalju ili dohvaćaju podaci. Isto tako, metode koje implementiraju sučelje moraju imati modifikator `async` kako bi se odgovor mogao iščekivati sa `await` klauzulom koja dolazi u paru sa navedenim modifikatorom:

```
class ProductRepository : IProductRepository
{
    private readonly IHttpService _httpService;
    private readonly string baseURL = "api/product";

    public ProductRepository(IHttpService httpService)
    {
        this._httpService = httpService;
    }

    public async Task<List<Product>> GetAll()
    {
        var response = await
        ↪  _httpService.Get<List<Product>>(baseURL);

        if (!response.Success)
        {
            throw new ApplicationException(await response.GetBody());
        }

        return response.Data;
    }

    public async Task<List<Product>> GetAllByCategory(int id)
    {
        var response = await
        ↪  _httpService.Get<List<Product>>($"{baseURL}/bycategory/{id}");

        if (!response.Success)
        {
            throw new ApplicationException(await response.GetBody());
        }

        return response.Data;
    }
}
```

Isječak kôda 5.12: Implementacija sučelja za upravljanje proizvodima na klijentskoj strani

Nakon kreiranja repozitorija i modula za slanje HTTP zahtjeva potrebno je iste registrirati kao servisne klase unutar DI spremišta na isti način kako je to objašnjeno u poglavlju 4.3.3. Dependency injection. Tada je napokon moguće unutar komponenti korisničkog sučelja pozivati repozitorij i dohvaćati podatke. Međutim, trenutno još nisu migrirane komponente korisničkog sučelja pa samim time te podatke još nije moguće prikazati na sučelju. Umjesto toga, kako bi se testirala funkcionalnost jednostavno će ih se ispisati u konzoli preglednika. U svrhu testiranja, kreira se prazna komponenta desnim klikom na mapu `Pages` i **Add > Razor component** i odabire se naziv `ProductList.razor`. Unutar komponente se ubrizgava servisna klasa prethodno kreiranog repozitorija i nadjačava metoda životnog ciklusa `OnInitializedAsync`

u kojem se poziva repozitorij, iščekiva odgovor i ispisuje u konzolu preglednika:

```
@page "/proizvodi"
@inject IProductRepository productRepository

<h3>Proizvodi</h3>

@code {
    protected override async Task OnInitializedAsync()
    {
        var response = await productRepository.GetAll();

        foreach (var p in response.Data)
        {
            Console.WriteLine(p.Name);
        }
    }
}
```

Isječak kôda 5.13: Testiranje repozitorija za upravljanje proizvodima

S obzirom da se podaci unutar odgovora odmah serijaliziraju u povratni tip podataka, nije ih moguće samo ispisati u konzolu, već je potrebno iterirati po listi i ispisivati podatke. U prethodnom isječku kôda postaje jasno zašto je dobro imati repozitorij i generički HTTP servis jer je za dohvaćanje potrebnih podataka u komponentu potrebno pozvati doslovno jednu liniju kôda. Kada prethodni elementi ne bi postojali na klijentskoj strani, komponenta bi morala koristiti obični `HTTPClient` i sama podešavati odredišnu putanju do krajnje točke, serijalizirati i deserijalizirati podatke, ispitivati da li postoji tijelo odgovora, ispitivati greške i ostale zadaće koje komponenta prema definiciji ne bi trebala raditi. Samim time, jedna kôda pretvorila bi se u njih barem 30 te bi komponente ubrzo postale neodržive. Nastavno na testiranje, za primjer su ispisani nazivi prethodno kreirana 3 proizvoda u bazi podataka. Ispis u konzoli bi trebao izgledati ovako:

Slika 23: Ispis konzole u web pregledniku

Sada je moguće vidjeti da tok podataka od poslužiteljske strane pa sve do korisničke strane potpuno ispravno radi. Međutim, ukoliko se pogleda dijagram toka podataka prikazan u prošlom potpoglavlju, moguće je primjetiti da i dalje ne postoji transformacija modela u DTO objekte. Samim time se do korisničkog sučelja šalju modeli baze podataka, što nije ispravno iz već spomenutih razloga. Stoga, potrebno je u upravljačkoj klasi implementirati sloj koji će obavljati transformacije modela u DTO objekte. Valja napomenuti i da se u nekim literaturama koristi i termin pogledni model (engl. *ViewModel*) jer se model transformira u model prikladan za prikaz na pogledu odnosno korisničkom sučelju. Nadalje, transformacije je moguće obavljati

ručno tako da jednostavno svaki puta kreiramo novu instancu DTO-a i popunimo ju samo sa potrebnim podacima iz modela. Taj pristup je prikladan kada se unutar sustava ne radi previše transformacija ili kada modeli nemaju previše atributa, ali ako to nije slučaj onda ovakav pristup može biti vrlo nezgodan. Umjesto toga, koriste se biblioteke treće strane koje te transformacije rade automatski uz pomoć prethodno definiranih postavki. Najpoznatija biblioteka koja obavlja takvu zadaću je *AutoMapper*, pa je nju potrebno instalirati sa sljedećom naredbom unutar `Package Manager` konzole na poslužiteljskoj strani:

```
Install-Package AutoMapper -Version 10.0.0
```

Nakon instalacije paketa, potrebno je kreirati profil *AutoMapper* biblioteke unutar kojega su definirane sve postavke prema kojima će se moći automatski napraviti transformacija podataka. Unutar mape `Profiles` na poslužiteljskoj strani kreira se klasa koja nasljeđuje `Profile` klasu:

```
public class AutoMapperProfile : Profile
{
    public AutoMapperProfile()
    {
        CreateMap<Product, ProductViewModel>();
    }
}
```

Isječak kôda 5.14: Mapiranje modela proizvoda u pogledni model

S obzirom da se u ovoj transformaciji samo sa modela "skidaju" pojedini nepotrebni podaci odnosno svojstva i ne dodaju nova svojstva, `AutoMapper` zna kako mapirati podatke iz jedne klase u drugu tako da ispituje nazive svojstava i samim time nije potrebno nikakve dodatne opcije postavljati. Zatim, kreira se `ProductViewModel` klasa za prikaz proizvoda na sučelju:

```
public class ProductViewModel
{
    public int Id { get; set; }
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public double Value { get; set; }
    public bool Promotion { get; set; }
    public int PromotionPercentage { get; set; }
    public int Quantity { get; set; }
    public string ImageName { get; set; }
}
```

Isječak kôda 5.15: Kreiranje poglednog modela za prikaz proizvoda

Sljedeće je potrebno registrirati servis unutar DI spremišta. Registracija servisa se, u ovom slučaju, radi nešto drugačije nego registracija ostalih servisnih klasa. Unutar postojeće metode

`ConfigureServices` u klasi `Startup` na poslužiteljskoj strani dodaje se sljedeća linija kôda:

```
services.AddAutoMapper(typeof(Startup));
```

Isječak kôda 5.16: Dodavanje AutoMapper servisa u DI spremište

Već je rečeno da je upravljačka klasa zadužena za transformaciju modela (a to je moguće vidjeti i na dijagramu toka podataka), pa je stoga unutar konstruktora klase potrebno ubrizgati servisnu klasu `IMapper`. Zatim, potrebno je prilagoditi metode upravljačke klase za proizvode da koriste ubrizganu instancu koja će obaviti transformaciju iz proslijeđene liste modela:

```
[HttpGet]
public async Task<IActionResult> GetAll()
{
    var products = await _productRepository.GetAll()
    return Ok(_mapper.Map<List<ProductViewModel>>(products));
}

[HttpGet("bycategory/{id}")]
public async Task<IActionResult> GetAllByCategory([FromRoute] int id)
{
    var products = await _productRepository.GetProductsByCategory()
    return Ok(_mapper.Map<List<ProductViewModel>>(products));
}
```

Isječak kôda 5.17: Mapiranje modela unutar upravljačke klase

Posljednje, potrebno je metode unutar sučelja i klasa kod modula za slanje HTTP zahtjeva i repozitorija na korisničkoj strani izmijeniti da rade sa `ProductViewModel` modelom, a ne `Product` modelom sukladno prikazanom dijagramu toka podataka.

Za kraj valja napomenuti da u ovom poglavlju nije migriran kompletni sloj rada s podacima, već su samo implementirani i opisani temeljni konstrukti koji služe kao podloga za daljnju migraciju. Zahvaljujući takvom dizajnu, samo prateći prethodno opisani proces vrlo je lako dodati upravljačke klasu i repozitorije za rad s podacima drugih modela i to bez puno promjena u kôdu.

5.3. Migracija korisničkih kontrola

Nakon što je omogućeno dohvaćanje podataka na korisničkoj strani, logično bi bilo da se zatim migriraju korisničke kontrole kako bi se ti podaci mogli prikazivati na korisničkom sučelju u obliku komponenti. No prije nego što se započne sa migracijom korisničkih kontrola, važno je spomenuti da je razvoj stilskih uputa komponenata (CSS) pomalo otežan, a postoji nekoliko razloga za to. Prvi je bio taj što Blazor WebAssembly inicijalno nije podržavao statičke datoteke poput `.css` datoteka, već su razvojni programeri morali koristiti već uključenu *Boot-*

strap biblioteku, a vlastite stilove definirati unutar HTML-a komponente što nije baš pogodno za kompleksnije dizajnove stranica. Isto tako, Blazor WebAssembly aplikacije se standardno uvijek spremaju u privremenu memoriju zajedno sa stilovima komponenti pa je samim time kod najmanje promjene u `.css` datotekama potrebno ponovno očistiti privremenu memoriju, inače će na zaslonu biti prikazan stari stil. Posljednje, *Visual Studio* razvojno okruženje nema dobru integraciju sa CSS predprocesorima, pa je razvoj složenijih stilova otežan. Upravo zbog toga su se već pojavile brojne biblioteke komponenti koje već imaju predefinirane stilove i HTML stukturu, a na razvojnom programeru preostaje samo da implementira programsku logiku.

5.3.1. Korištenje biblioteka komponenti treće strane

Prilikom odabira biblioteka komponenti valja obratiti pozornost na njihovu dostupnost. Većina pružatelja nude svoje biblioteke na besplatni probni rok koji traje 30 dana, a zatim se moraju plaćati. Stoga je za ovu priliku odabrana besplatna `Razden` biblioteka otvorenog kôda koja nudi mnogo jednostavnih korisničkih kontrola s kojima će svakako razvoj korisničkog sučelja biti znatno ubrzan. Instalirati biblioteku moguće je sa jednostavnim unosom sljedeće komande unutar `Package Manager` konzole u `BlazorShop.Client` projektu:

```
Install-Package Radzen.Blazor -Version 2.11.14
```

5.3.2. Razvoj vlastitih komponenti

Iako prethodno instalirana `Radzen` biblioteka komponenti nudi mnogo generički komponenti kao što su gumbovi, tekstualni dijalozi, tablice sa podacima, padajući meni i druge, ipak je složenije komponente potrebno samostalno implementirati. Te složenije komponente u pravilu u sebi koriste generičke komponente iz biblioteka. U ovom potpoglavlju opisana je migracija jednostavne komponente za prikaz proizvoda.

Dizajn komponente biti će izrađen prema dizajnu iste korisničke kontrole na stolnoj aplikaciji:

Slika 24: Dizajn komponente za prikaz proizvoda na stolnoj aplikaciji

Osim prikazanog dizajna, korisnička kontrola ima i rukovatelj događaja za klik miša na gumb "Kupi". Prema tome, struktura komponente `ProductCard` može izgledati ovako:

```
<article class="product-card">
    <div class="product-card__imagecontainer">
        <img class="product-card__image" src="" alt="@Product.Name"
        ↪ />
    </div>
    <div class="product-card__content">
        <h3 class="product-card__title">@Product.Name</h3>
        <p
        ↪ class="product-card__price">@FormatPrice(Product.Value)</p>
    </div>
    <RadzenButton Text="Dodaj u košaricu"
                  ButtonStyle="ButtonStyle.Success"
                  Icon="add_shopping_cart"
                  Style="width: 100%"
                  Click="@(() =>
                  ↪ AddProductToCart.InvokeAsync(Product))" />
</article>

@code {
    [Parameter] public ProductViewModel Product { get; set; }
    [Parameter] public EventCallback<ProductViewModel>
↪   AddProductToCart { get; set; }

    private string FormatPrice(double price)
    {
        return $"{price}.00 HRK";
    }
}
```

Isječak kôda 5.18: Kreiranje komponente za prikaz proizvoda

Parametre koje se prenose u komponentu moguće je definirati sa `[Parameter]` dekoratorom, a to su `ProductViewModel` koji sadrži podatke potrebne za prikaz komponente na sučelju te `EventCallback<ProductViewModel>` rukovatelj događaja koji će se aktivirati kada korisnik klikne na komponentu. Razlog zašto se rukovatelj događaja treba prenositi kao parametar je taj što u web aplikacijama komponente nikada ne smiju direktno upravljati aplikacijskim stanjem ili stanjem aktivne stranice, već se svo stanje kao i rukovatelji događaja trebaju nalaziti unutar stranica odnosno u ovom slučaju komponenata sa `@page` direktivom. Sa takvim pristupom lakše se upravlja komponentama jer se sva logika nalazi unutar stranica, a komponente je moguće ponovno iskoristiti na način da im se u nekoj drugoj stranici proslijeđuje opet po potrebi neki drugi rukovatelj događaja. Stoga, sljedeće je potrebno kreirati stranicu `ProductList` koja će imati putanju `/proizvodi` na kojoj će se prikazivati lista proizvoda koje korisnik može pregledavati:

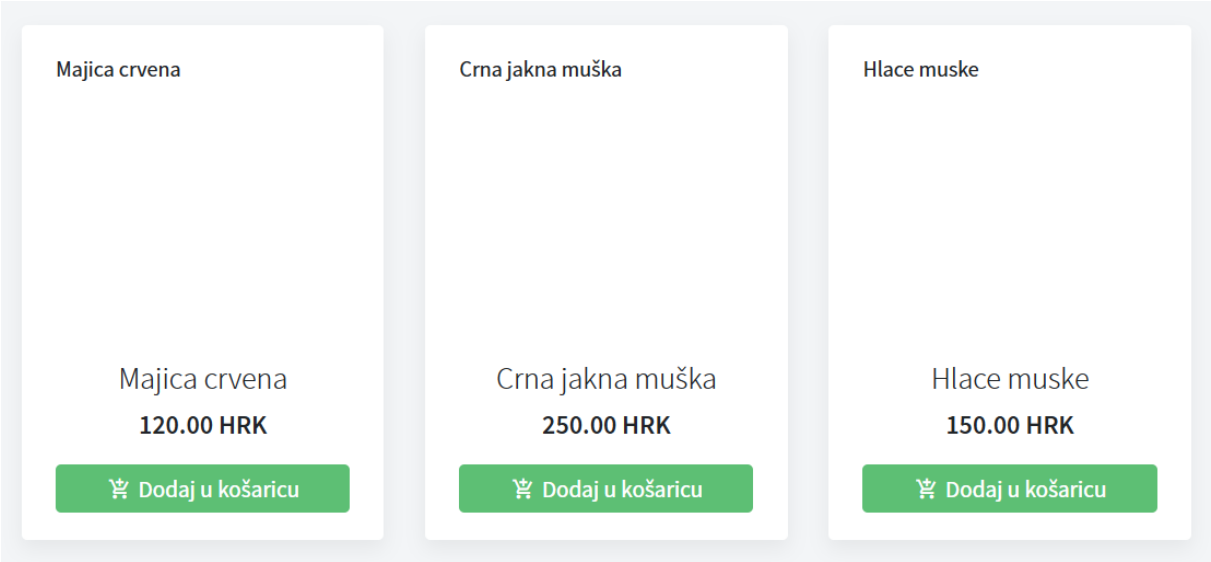
```
@page "/proizvodi"
```

```
@inject IProductRepository productRepository

<div class="row mt-2">
    @if (Products != null)
    {
        @foreach (var p in Products)
        {
            <div class="col-3 mb-2">
                <ProductCard Product="p" AddProductToCart="AddToCart"
                 ↪  />
            </div>
        }
    }
    else
    {
        <p>Učitavam proizvode...</p>
    }
</div>

@code {
    private List<OrderProductSubmitDTO> Order { get; set; }
    public List<ProductViewModel> Products { get; set; }

    protected override async Task OnInitializedAsync()
    {
        Products = await productRepository.GetAll();
    }

    private void AddToCart(ProductViewModel vm)
    {
        Order.Add(new OrderProductSubmitDTO
        {
            ProductId = vm.Id,
            Quantity = 1,
            Value = vm.Value
        });
    }
```

Isječak kôda 5.19: Kreiranje stranice za prikaz liste proizvoda

Odmah je i kreiran `OrderProductSubmitDTO` objekt za prijenos podataka stavke narudžbe korisnika jer je za kreiranje entiteta `OrderProduct` u bazi podataka potrebno samo ID proizvoda, njegovu vrijednost i količinu pa stoga nije potrebno slati i ostale podatke. Unutar HTML-a se iterira po prethodno dohvaćenim proizvodima iz repozitorija te se dodaje prethodno kreirana komponenta `ProductCard` kojoj se proslijeđuju podaci i rukovatelj događaja `AddToCart` definiranim unutar stranice. Rukovatelj jednostavno u listu stavki narudžbe dodaje novu stavku koja joj se prosljeđuje. Ovdje je također moguće vidjeti kako se provodi transformiranje iz jednog DTO-a u drugi DTO, a to je da se jednostavno ručno mapiraju svojstva objekata iz jednoga

u drugi. Ukoliko se pokrene aplikacija, ono što bi se trebalo prikazati na ekranu izgleda ovako:



Slika 25: Prikaz liste proizvoda na webu

Vidljivo je da se na sučelju ne prikazuju slike proizvoda, a to je zbog toga što unutar komponente `ProductCard` element sa slikom proizvoda nema svoju izvornu putanju (`src=""`). Problem leži u tome što se na stolnoj aplikaciji slike serijaliziraju i spremaju u bazu podataka u obliku niza znakova, a kada se proizvodi dohvaćaju iz baze podataka onda se deserijaliziraju nazad u instacu `Image` klase i prikazuju na sučelju. Ovakav pristup na web aplikaciji nikako se ne preporuča jer bi se na klijentskoj strani mora implementirati mehanizam za određivanje ekstenzije slike i deserijalizaciju što bi predstavljalo procesorski zahtjevnu radnju i samim time loše korisničko iskustvo. Najbolja praksa kod web aplikacija je da se dinamički resursi poput slika komponenti pohranjuju na servisima treće strane, a upravo će migracija takvih resursa biti opisana u sljedećem potpoglavlju.

5.4. Migracija aplikacijskih resursa

Resursi web aplikacije dijele se na statičke resurse koji se poslužuju zajedno sa klijentskim dijelom aplikacije u web pregledniku, te na dinamičke resurse koji se nalaze na nekim servisima treće strane i referencirani su na klijentskoj aplikaciji. Statički resursi se u pravilu rijetko mijenjaju i mogu sadržavati pozadinske slike stranica, logotipove, fontove i slične datoteke dok dinamički resursi predstavljaju resurse koje se često mijenjaju.

5.4.1. Migracija statičkih resursa

Statički resursi Blazor WebAssembly aplikacije postavljaju se unutar postojećeg direktorija `wwwroot` unutar kojega se inicijalno nalaze `css` direktorij sa stilskim uputama te izvan njega `index.html` koja predstavlja ishodišnu stranicu te `manifest.json` unutar kojega se nalaze svojstva aplikacije. Ovdje je moguće kopirati logotip *ProphetShop* stolne aplikacije, korištene ikone i slično.

5.4.2. Migracija dinamičkih resursa

Kao što je već rečeno, dinamički resursi se zbog njihovog broja pohranjuju na vanjskim servisima treće strane. Stoga će slike proizvoda web aplikacije biti pohranjene na *Microsoft Azure Storage* servisu koji osim pohrane teksta pruža i mogućnost pohrane multimedijskih datoteka u obliku *Blob* formata. Razlog odabire servisa *Microsoft Azure Storage* je taj što što *.NET Core* programski okvir pruža biblioteke za vrlo jednostavan rad sa servisom. S obzirom da stolna aplikacija deserijalizira i prikazuje slike koje se nalaze u bazi podataka, a takve nisu pogodne za prikaz na klijentskoj strani tada se javlja problem iz uvjeta da aplikacije moraju koristiti istu bazu podataka. Stoga, ideja je da se implementira mehanizam na poslužiteljskoj strani koji prilikom dohvaćanja proizvoda iz baze podataka uvijek prvo provjeri da li se slika trenutnog proizvoda već nalazi prenesena na servisu te ju u suprotnom prenosi do spremišta servisa i do klijenta šalje `URI` putanju. Na taj način će slike koje se nalaze u bazi podataka te one koje se nalaze na spremištu servisa uvijek biti usklađene. Na poslužiteljskoj strani potrebno je instalirati sljedeći paket:

```
Install-Package Microsoft.Azure.Storage.Blob -Version 11.2.2
```

Kako bi programski okvir znao putanju do spremišta servisa i ključ za autentifikaciju računa, potrebno je ažurirati `launchSettings.json` postavke sa novim redom unutar sekcije `ConnectionStrings` i unijeti podatke dostupne na web sučelju servisa:

```
"AzureStorageConnection":
 ↪  "DefaultEndpointsProtocol=https;AccountName=blazorshop;
AccountKey=key;EndpointSuffix=core.windows.net"
```

Isječak kôda 5.20: Unos podataka za povezivanje na AzureStorage servis unutar postavki

Valja napomenuti da je potrebno zamijeniti `key` vrijednost sa ključem prikazanim na sučelju *Azure Storage* web aplikacije koji ovdje nije napisan iz sigurnosnih razloga. Nakon toga kreira se sučelje `IFileStorageService` unutar mape `Services/Storage`:

```
public interface IFileStorageService
{
    Task<string> SaveFile(byte[] content, string fileName, string
     ↪  containerName);
}
```

Isječak kôda 5.21: Kreiranje sučelja za spremanje datoteka na AzureStorage servis

Zatim je potrebno implementirati prethodno kreirano sučelje:

```
public class AzureStorageService : IFileStorageService
{
    private readonly string _connectionString;
```

```
        public AzureStorageService(IConfiguration configuration)
        {
            _connectionString =
            ↪  configuration.GetConnectionString("AzureStorageConnection");
        }

        public async Task<string> SaveFile(byte[] content, string
        ↪  fileName, string containerName)
        {
            var account =
            ↪  CloudStorageAccount.Parse(_connectionString);
            var client = account.CreateCloudBlobClient();
            var container =
            ↪  client.GetContainerReference(containerName);

            await container.CreateIfNotExistsAsync();
            await container.SetPermissionsAsync(new
            ↪  BlobContainerPermissions
            {
                PublicAccess = BlobContainerPublicAccessType.Blob
            });

            var blob = container.GetBlockBlobReference(fileName);

            if (!await blob.ExistsAsync())
            {
                await blob.UploadFromByteArrayAsync(content, 0,
                ↪  content.Length);
                blob.Properties.ContentType = "image/png";

                await blob.SetPropertiesAsync();
            }

            return blob.Uri.ToString();
        }
    }
```

Isječak kôda 5.22: Implementacija sučelja za spremanje datoteka na AzureStorage servis

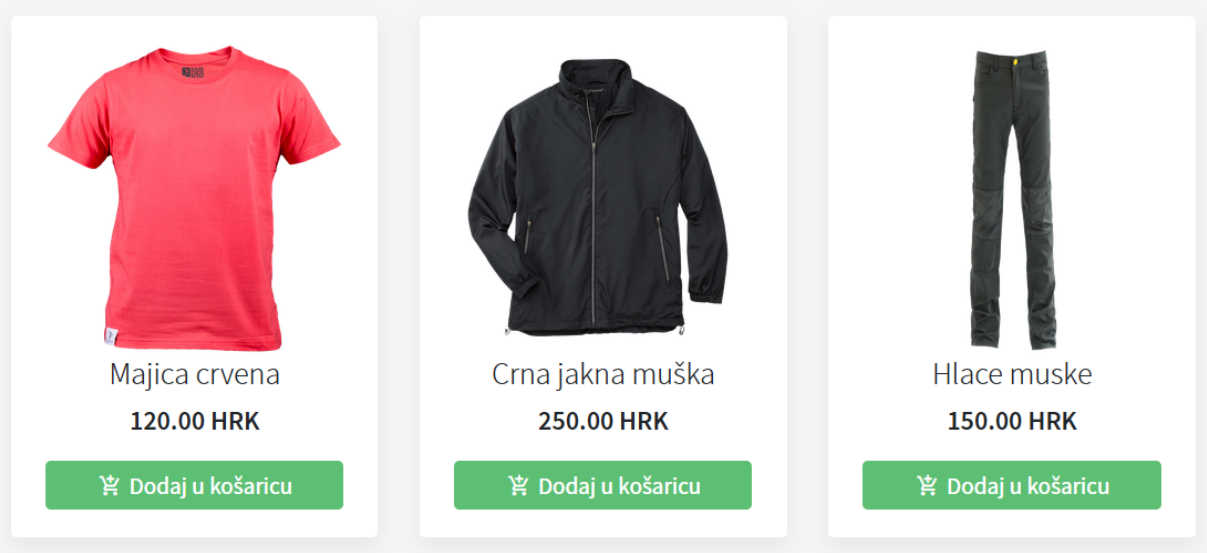
Prethodna metoda spaja se prvo na servis putem `CloudBlobClient` klijenta i onda kreira spremište sa proslijeđenim nazivom ukoliko ne postoji već. Zatim iz naziva datoteke provjerava da li već takva postoji te ju prenosi ukoliko ne postoji. Na kraju vraća `URI` putanju do resursa u obliku niza znakova. S obzirom da se radi o servisnoj klasi, potrebno ju je prvo registrirati unutar DI spremišta. Zatim ju je moguće koristiti u upravljačkoj klasi unutar koje će biti implementiran mehanizam sinkronizacije podataka. Metoda za dohvaćanje liste proizvoda biti će malo izmijenjena:

```
[HttpGet]
public async Task<IActionResult> GetAll()
```

```
{
    var products = await _productRepository.GetAll()

    foreach (var p in products)
    {
        await _azureStorage.SaveFile(p.ImageContent, p.ImageName,
        ↪  "images");
    }

    return Ok(products);
}
```

Isječak kôda 5.23: Implementacija mehanizma za sinkronizaciju dinamičkih resursa

Nakon implementiranog mehanizma moguće je `URI` putanje slika slati nazad do klijenta i referencirati ih u `ProductCard` komponenti. Samim time bi sučelje `ProductList` stranice sada trebalo izgledati ovako:



Slika 26: Prikaz liste proizvoda sa slikama na webu

5.5. Migracija aplikacijskog stanja

Upravljanje stanjem jedno je od kompleksnijih koncepata kod razvoja klijentskog dijela web aplikacije zbog čega se uz standardne Javascript programske okvire poput *Angular* ili *React* uvijek koriste i biblioteke za upravljanje stanjem poput *Redux*, *Mobx*, *Recoil* i druge. Kod komponentno-baziranih programskih okvira svaka komponenta kao dio stranice može imati svoje lokalno stanje (da li je navigacijska traka proširena ili ne), svaka stranica kao skup komponenti ima svoje stanje (podaci koji se dohvaćaju i prosljeđuju komponentama, popis stavki u košarici) te postoji i aplikacijsko stanje (da li je korisnik autentificiran ili ne) pa samim time

upravljati svime istovremeno može biti nezgodno. Isto tako, ukoliko na stranici postoje komponente A, B, C i D koje su međusobno ugniježdene jedna unutar druge tada slanje podataka od komponente A do komponente D znači proslijeđivanje istih podataka kroz komponente B i C što nije prikladno. Iako su se već pojavile brojne biblioteke za upravljanje stanjem *Blazor* aplikacije, one neće biti korištene za migraciju stanja iz razloga što se želi prikazati kako se to radi ručno uz pomoć koncepata koje već nudi sami *Blazor* okvir.

U slučaju BlazorShop web aplikacije, ono što se želi postići je da stanje autentifikacije korisnika bude dostupno uvijek kroz cijelu aplikaciju (opisano u poglavlju 5.8 Migracija sigurnosnog sloja) te da popis stavki u košarici bude perzistentan. Kod stolne Windows Forms aplikacije ta perzistencija ne predstavlja problem jer instanca objekta postoji sve dok aplikacija radi. Samim time, ukoliko se otvori druga forma tada instanca objekta stanja i dalje postoji. Stanje se najčešće implementira kao Singleton uzorak dizajna pa mu je moguće pristupiti u svakoj formi. S druge strane, unutar klijentske web aplikacije instance svih komponenti korisničkog sučelja jedne stranice su perzistentne za vrijeme dok je ta stranica aktivna pa ukoliko korisnik osvježi stranicu tada se stanje vraća na ono početno. Shodno tome, ukoliko bi korisnik dodao nekoliko stavki u košaricu i u međuvremenu izgubio mrežnu vezu i osvježio stranicu, nestale bi mu sve dodane stavke u košarici. Jedno moguće rješenje ovog problema je to stanje pohranjivati unutar baze podataka, ali to bi značilo da svaki unos stavke u košaricu ili uklanjanje stavke iz košarice predstavlja jedan HTTP zahtjev prema poslužiteljskoj strani što nikako nije pogodno za korisničko iskustvo, a može se dogoditi i desinkronizacija stanja. Povoljnije je u ovom slučaju koristiti `localStorage` pohranu web preglednika koju je moguće koristiti kroz Javascript interoperabilnost i `IJSRuntime` servis koji nudi *Blazor* okvir. Podaci unutar `localStorage` pohrane web preglednika su uvijek perzistentni i lako im se pristupa, a brišu se programski na zahtjev korisnika ili kada se obriše memorija preglednika. Prvo je potrebno kreirati klasu (DTO) koja će predstavljati stanje košarice:

```
public class OrderState : INotifyPropertyChanged
{
    private int customerId;
    private string payment;
    private int discountPercentage;
    private ObservableCollection<OrderProductSubmitDTO> orderProduct;

    public OrderState()
    {
        OrderProduct = new
        ↪  ObservableCollection<OrderProductSubmitDTO>();
    }

    public int CustomerId
    {
        get => customerId;
        set
        {
            customerId = value;
            RaisePropertyChanged();
```

```
        }
    }

    public string Payment
    {
        get => payment;
        set
        {
            payment = value;
            RaisePropertyChanged();
        }
    }

    public int DiscountPercentage
    {
        get => discountPercentage;
        set
        {
            discountPercentage = value;
            RaisePropertyChanged();
        }
    }

    public ObservableCollection<OrderProductSubmitDTO> OrderProduct
    {
        get => orderProduct;
        set
        {
            orderProduct = value;
            RaisePropertyChanged();
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void RaisePropertyChanged([CallerMemberName] string
     ↪  propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
         ↪  PropertyChangedEventArgs(propertyName));
    }
}
```

Isječak kôda 5.24: Kreiranje klase aplikacijskog stanja

Ideja je da se promjena stanja promatra (engl. *observable state*) i propagira kroz sve komponente koje koriste to stanje, pa stoga klasa mora implementirati `INotifyPropertyChanged` sučelje koje će omogućavati da se kroz poziv njegove metode `RaisePropertyChanged` obavijesti o promjeni stanja i šalje događaj komponentama koji sadrži novu vrijednost svojstva koje

se promjenilo. Klasa je svojevrstan objekt za prijenos podataka (DTO) jer sadrži sve potrebne podatke za kreiranje entiteta narudžbe (`Order`) i njegovih stavki (`OrderProduct`) pa će ga biti moguće nakon završene kupovine samo poslati do poslužitelja i unijeti narudžbu u bazu podataka. Sljedeće je potrebno kreirati servisnu klasu koja će komunicirati sa Javascript izvršnim okruženjem te pohranjivati i dohvaćati stanje iz lokalne pohrane preglednika:

```
public sealed class OrderStateProvider
{
    private const string KeyName = "ORDER";

    private readonly IJSRuntime _jsRuntime;
    private bool _initialized;
    private OrderState _order;

    public event EventHandler Changed;

    public bool AutoSave { get; set; } = true;

    public OrderStateProvider(IJSRuntime jsRuntime)
    {
        _jsRuntime = jsRuntime;
    }

    public async ValueTask<OrderState> Get()
    {
        if (_order != null)
            return _order;

        if (!_initialized)
        {
            var reference = DotNetObjectReference.Create(this);
            await
             ↪  _jsRuntime.InvokeVoidAsync("BlazorRegisterStorageEvent",
             ↪  reference);
            _initialized = true;
        }

        OrderState result;

        var str = await
         ↪  _jsRuntime.InvokeAsync<string>("BlazorGetLocalStorage",
         ↪  KeyName);
        if (str != null)
        {
            result = JsonSerializer.Deserialize<OrderState>(str) ??
             ↪  new OrderState();
        }
        else
        {
            result = new OrderState();
        }
```

```
            result.PropertyChanged += OnPropertyChanged;
            _order = result;
            return result;
        }

        public async Task Save()
        {
            var json = JsonSerializer.Serialize(_order);
            await _jsRuntime.InvokeVoidAsync("BlazorSetLocalStorage",
             ↪  KeyName, json);
        }

        private async void OnPropertyChanged(object sender,
         ↪  PropertyChangedEventArgs e)
        {
            if (AutoSave)
            {
                await Save();
            }
        }

        [JSInvokable]
        public void OnStorageUpdated(string key)
        {
            if (key == KeyName)
            {
                _order = null;
                Changed?.Invoke(this, EventArgs.Empty);
            }
        }
    }
```

Isječak kôda 5.25: Implementacija mehanizma za perzistenciju aplikacijskog stanja

Servisna klasa će prilikom poziva metode `Get()` za dohvaćanje stanja vratiti stanje ukoliko već postoji, a ukoliko ne postoji onda kreirati novo. Isto tako, proslijediti će Javascript okruženju referencu na vlastito okruženje kako bi se mogla metoda iz C# kôda pozivati unutar Javascript-a i tako omogućiti da se stanje sinkronizira. Ukoliko stanje već postoji, ono se dohvaća u obliku niza znakova i deserijalizira u `OrderState` instancu i vraća korisniku. Također, kada se okine događaj `OnPropertyChanged` automatski se poziva metoda `Save()` unutar koje se opet stanje serijalizira u obliku niza znakova i zapisuje u lokalnu pohranu web preglednika i time je ostvarena sinkronizacija u oba smjera. Javascript funkcije koje se pozivaju unutar ove servisne klase potrebno je kreirati i ubaciti u `index.html` datoteku:

```
function BlazorSetLocalStorage(key, value) {
    localStorage.setItem(key, value);
}
```

```
function BlazorGetLocalStorage(key) {
    return localStorage.getItem(key);
}

function BlazorRegisterStorageEvent(component) {
    window.addEventListener("storage", async e => {
        await component.invokeMethodAsync("OnStorageUpdated", e.key);
    });
}
```

Isječak kôda 5.26: Kreiranje Javascript funkcija za upravljanje aplikacijskim stanjem

Kreirane metode nisu ništa drugo nego apstrakcija postojećih metoda Javascript API-a za rad sa lokalnom pohranom, a važno je primjetiti da se unutar metode `BlazorRegisterStorageEvent` događa pozivanje C# kôda komponente iz Javascript-a. Posljednje, potrebno je kreirati komponentu koja će omotavati cijelu aplikaciju odnosno predstavljati će ishodišnu komponentu koja će proslijeđivati stanje aplikacije uz pomoć `CascadingValue` svojstva koja omogućuje da se kaskadna vrijednost dohvaća u bilo kojoj komponenti dijeteta toga omotača.

```
@inject OrderStateProvider orderStateProvider
@implements IDisposable

@if (state == null)
{
    <p>loading...</p>
}
else
{
    <CascadingValue Value="@state"
     ↪  IsFixed="false">@ChildContent</CascadingValue>
}

@code{
    private OrderState state = null;

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    protected override async Task OnInitializedAsync()
    {
        orderStateProvider.Changed += OrderStateChanged;
        await Refresh();
    }

    public void Dispose()
    {
        orderStateProvider.Changed -= OrderStateChanged;
```

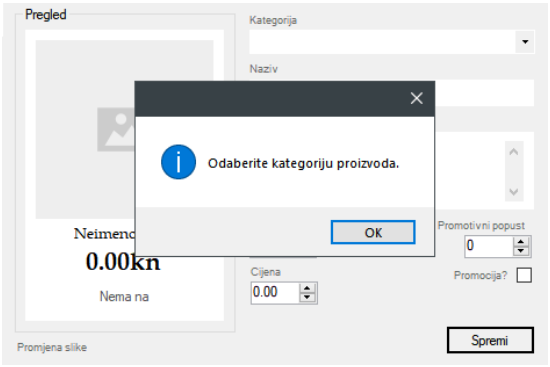
```
    }

    private async void OrderStateChanged(object sender, EventArgs e)
    {
        await InvokeAsync(async () =>
        {
            await Refresh();
            StateHasChanged();
        });
    }

    private async Task Refresh()
    {
        state = await orderStateProvider.Get();
    }
}
```

Isječak kôda 5.27: Kreiranje komponente za dijeljenje aplikacijskog stanja

U komponentu se prvo ubrizgava prethodno kreirana servisna klasa `OrderStateProvider` te se na promjenu njegova stanja registrira događaj komponente `OrderStateChanged` koji automatski dohvaća novo stanje i osvježava ga tako da se promjena propagira do svih komponenti dijece. Komponenta također ima parametar `ChildContent` koji omogućava da komponenta na toj sekciji prikazuje sav onaj sadržaj koji omotava. Posljednje, potrebno je ishodišnu komponentu `MainLayout` omotati sa ovom komponentom i s time je omogućeno perzistencija stanja i njegova propagacija do svih komponenti aplikacije.

5.6. Migracija logike za validaciju korisničkog unosa

Na stolnoj aplikaciji validacija se vrši na način da se nakon pritiska gumba za slanje forme provjeravaju sva polja za unos te ovisno o ispravnosti polja prikazuje poruka greške ili se forma podnosi dalje. Ako se pokušaju unijeti sva prazna polja na formi za kreiranje novog proizvoda, dobila bi se ovakva poruka:



Slika 27: Validacija forme za unos proizvoda na stolnoj aplikaciji

Iako je ovakav pristup svakako ispravan i na klijentskoj strani web aplikacije, nije preporučljiv iz razloga što `Blazor` programski okvir nudi bolje komponente za validaciju korisničkog unosa. U ovom slučaju, biti će korištene komponente za validaciju iz *Radzen* biblioteke koje su samo omotači za standardne komponente za validaciju uz nekoliko dodatnih mogućnosti. Komponente za validaciju moraju uvijek biti unutar `RadzenTemplateForm` komponente, a HTML struktura tekstualnog polja za unos kategorije proizvoda izgleda ovako:

```
<div class="row mb-2">
    <div class="col-md-4 align-items-center d-flex">
        <RadzenLabel Text="Kategorija" />
    </div>
    <div class="col-md-8">
        <RadzenTextBox Style="width: 100%"
                       Name="Category"
                       @bind-Value="@NewProduct.CategoryName" />
        <RadzenRequiredValidator Component="Category"
                                 Text="Unesite kategoriju proizvoda!"
                                 ↪ />
    </div>
</div>
```

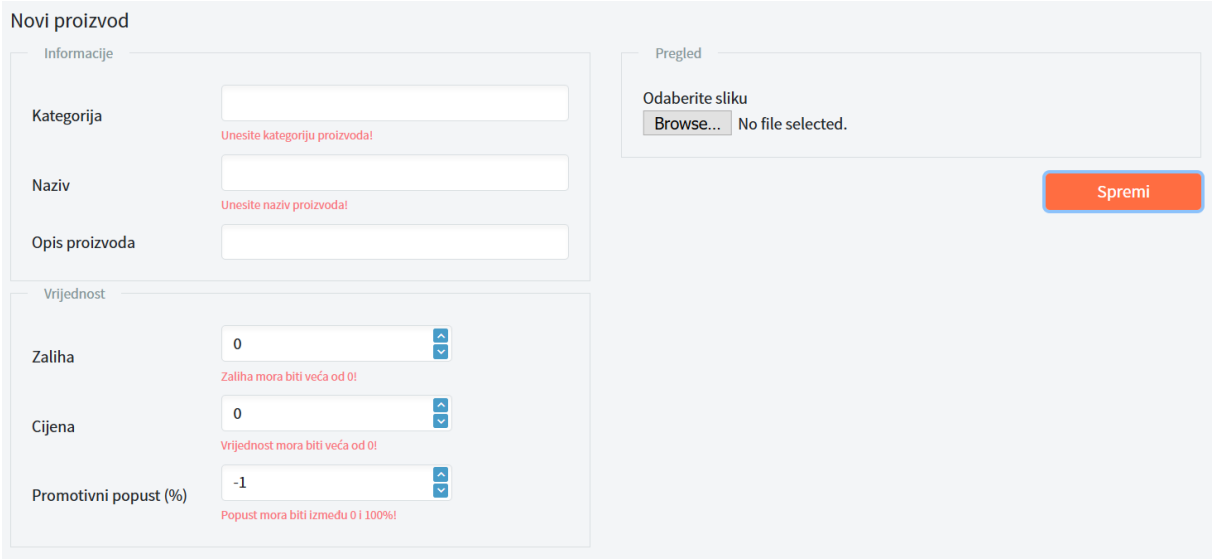
Isječak kôda 5.28: Dodavanje komponente za validaciju korisničkog unosa teksta

U prethodnom isječku koristi se komponenta `RadzenRequiredValidator` koja jednostavno provjerava da li je vrijednost svojstva s kojim je povezano ispunjena ili ne te ovisno o tome ispisuje poruku greške koja je definirana. Ako se, naprimjer, želi ispitati da li je numerički unos unutar nekog raspona tada se može koristiti `RadzenNumericRangeValidator` komponenta:

```
<div class="row mb-2">
    <div class="col-md-4 align-items-center d-flex">
        <RadzenLabel Text="Cijena" />
    </div>
    <div class="col-md-8">
        <RadzenNumeric TValue="double" Name="Value"
        ↪ @bind-Value="@NewProduct.Value" />
        <RadzenNumericRangeValidator Component="Value" Min="1"
        ↪ Text="Vrijednost mora biti veća od 0!" />
    </div>
</div>
```

Isječak kôda 5.29: Dodavanje komponente za validaciju korisničkog unosa vrijednosti

Na sličan način definiraju i validatori za sva ostala polja forme. Ukoliko se na web aplikaciji pokuša poslati prazna forma, dobio bi se ovakav prikaz:



Slika 28: Validacija forme za unos proizvoda na web aplikaciji

5.7. Migracija poslovne logike

Poslovna logika nalazi se unutar `Business` mape stolne aplikacije, a sadrži klase `Orders` i `Products` a sadrže algoritme za izračunavanje vrijednosti popusta proizvoda, iznosa dostave proizvoda, ukupne vrijednosti košarice i ostalo. S obzirom da se radi o logici koja ne koristi nikakve posebne biblioteke, moguće je klase jednostavno prenijeti unutar `BlazorShop.Client` projekta i podesiti polja imena. S time je migracija poslovne logike aplikacije završena.

5.8. Migracija sigurnosnog sloja

Sigurnosni sloj stolne `ProphetShop` aplikacije je, u najmanju ruku, vrlo tanak. To karakteristika vrijedi i općenito za stolne aplikacije jer se, prema definiciji stolne aplikacije, izvode na korisnikovom računalu a ne unutar web preglednika pa su samim time manje podložnije raznim napadima. Kod web aplikacija, HTTP zahtjevi se konstantno šalju prema drugim web servisima i lako se mogu presresti od strane zlonamjernih alata. Uz to, kako je objašnjeno u prethodnom poglavlju, klijentska strana web aplikacija gubi svoje stanje svaki puta kada se stranica ponovno učita. Samim time, ukoliko se autentifikacijsko stanje korisnika negdje ne pohrani onda bi se ono izgubilo svaki puta kada se stranica ponovno učita i korisnik bi se opet morao prijaviti. Zbog toga, autentifikacijsko stanje mora biti pohranjeno unutar web preglednika kroz `localStorage` ili `sessionStorage` sučelja koje Javascript nudi. Način na koje se to stanje pohranjuje i izmijenjuje sa poslužiteljem ovisi o korištenoj autentifikacijskoj shemi koja je implementirana, a u ovom slučaju biti će korištena JSON Web Token (JWT) autentifikacija jer je najviše korištena u web razvoju, a `.NET Core 3` okvir nudi mnoge biblioteke za jednostavnije postavljanje sheme. Valja i napomenuti da se autentifikacija mora implementirati na poslužiteljskoj strani kao i na korisničkoj strani web aplikacije. Poslužiteljska strana obavlja generiranje i provjeru primljenih žetona (engl. *token*), dok se na korisničkoj strani iz žetona iščitavaju podaci

i ovlasti autentificiranog korisnika.

5.8.1. Migracija autentifikacijskog sloja

U stolnoj aplikaciji postoji samo autentifikacija korisnika putem forme prijave gdje se, u slučaju da postoji korisnik s tim podacima, uspješno prijavljuje u aplikaciju. U bazu podataku je spremljena *hash* (šifrirana) vrijednost lozinke, a ne stvarna (nešifrirana) vrijednost iz sigurnosnih razloga i takav pristup se uvijek primjenjuje u razvoju softvera. Stoga će taj dio autentifikacije biti preslikan na poslužiteljskoj strani web aplikacije. Prvo je potrebno unutar `Package Manager` konzole unijeti sljedeću naredbu kako bi se instalirala `JwtBearer` biblioteka za rad sa JWT-om:

```
Install-Package Microsoft.AspNetCore.Authentication.JwtBearer -Version
3.1.7
```

Zatim je potrebno unutar `launchSettings.json` postavki poslužitelja definirati javni ključ korišten za autentifikaciju i izdavatelja certifikata. S obzirom da se radi o ne-produkcijskoj web aplikaciji, unijete su proizvoljne vrijednosti:

```
"Jwt": {
    "Key":
    ↪  "3d8cc8e74936bb1d424594988614cfd01ac57ae845ba54d6842c7f266d7d9b52",
    "Issuer": "blazorshop.com"
}
```

Isječak kôda 5.30: Dodavanje Json Web Token javnog ključa

Potom je na poslužiteljskoj strani potrebno kreirati servis koji će obavljati autentifikaciju uz pomoć prethodno instalirane biblioteke. Kreira se `IAuthenticationService` sučelje sa metodom za prijavu korisnika:

```
public interface IAuthenticationService
{
    Task<HttpResponse<UserToken>> Login(string username, string
    ↪  password);
}
```

Isječak kôda 5.31: Kreiranje sučelja autentifikacijskog servisa

Nakon toga, potrebno je implementirati sučelje:

```
public class AuthenticationService : IAuthenticationService
{
    private readonly BlazorShopContext _context;
    private readonly IConfiguration _configuration;
```

```
public AuthenticationService(BlazorShopContext context,
↪  IConfiguration configuration)
{
    _context = context;
    _configuration = configuration;
}

public async Task<HttpResponse<UserToken>> Login(string username,
↪  string password)
{
    HttpResponse<UserToken> response = new
    ↪  HttpResponse<UserToken>();
    User user = await _context.User.FirstOrDefaultAsync(x =>
    ↪  x.Username.ToLower().Equals(username.ToLower()));

    if (user == null)
    {
        response.Success = false;
    }
    else if (!VerifyPasswordHash(password, user.PasswordHash,
    ↪  user.PasswordSalt))
    {
        response.Success = false;
    }
    else
    {
        response.Data = CreateToken(user);
    }

    return response;
}

public bool VerifyPasswordHash(string password, byte[]
↪  passwordHash, byte[] passwordSalt)
{
    using (var hmac = new
    ↪  System.Security.Cryptography.HMACSHA512(passwordSalt))
    {
        var computedHash =
        ↪  hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
        for (int i = 0; i < computedHash.Length; i++)
        {
            if (computedHash[i] != passwordHash[i])
            {
                return false;
            }
        }
        return true;
    }
}
```

```
    private UserToken CreateToken(User user)
    {
        List<Claim> claims = new List<Claim>
        {
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
            new Claim(ClaimTypes.Name, user.Username),
            new Claim(ClaimTypes.Role, user.Discriminator)
        };

        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(
        _configuration.GetSection("Jwt:Key").Value));
        var creds = new SigningCredentials(key,
         ↪  SecurityAlgorithms.HmacSha512Signature);
        var expiration = DateTime.UtcNow.AddDays(1);

        JwtSecurityToken token = new JwtSecurityToken(
            issuer: null,
            audience: null,
            claims: claims,
            expires: expiration,
            signingCredentials: creds);

        return new UserToken()
        {
            Token = new JwtSecurityTokenHandler().WriteToken(token),
            Expiration = expiration
        };
    }
}
```

Isječak kôda 5.32: Implementacija sučelja autentifikacijskog servisa

Iako implementira samo jednu metodu sučelja, ova klasa sadrži i nekoliko pomoćnih metoda koje koristi u svom rada, a najbitnija je `CreateToken`. Njoj se prosljeđuje korisnik iz baze podataka iz kojeg se iščitavaju i potom na žeton zapisuju korisničko ime, naziv i uloga prema kojoj će se kasnije moći obavljati autorizacija korisnika. Klasa `Claim` predstavlja potvrdu s kojom je moguće asocirati korisnika s njegovim podacima na isti način kako osobna iskaznica potvrđuje identitet osobe u realnom svijetu. Lista tih potvrda zapisuje se `JwtSecurityToken` žeton zajedno sa datumom isteka valjanosti žetona te potpisnim vjerodajnicama (engl. *signing credentials*) koji su iščitani iz prethodno kreirane sekcije u konfiguraciji. Metoda vraća instancu klase `UserToken` koja sadrži samo žeton serijaliziran u niz znakova uz pomoć ponuđene metode `JwtSecurityTokenHandler().WriteToken(token)` te datum isteka žetona. Nakon kreiranja servisne klase, potrebno je kreirati upravljačku klasu `AuthController` koja će biti zadužena za obradu autentifikacijskih HTTP zahtjeva te, između ostalog, koristiti kreirani servis:

```
[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
    private readonly IAuthenticationService _authService;

    public AuthController(IAuthenticationService authService)
    {
        _authService = authService;
    }

    [HttpPost("login")]
    public async Task<ActionResult<UserToken>> Login([FromBody]
    ↪  UserLoginDTO user)
    {
        HttpResponse<UserToken> response = await
        ↪  _authService.Login(user.Username, user.Password);

        if (!response.Success)
        {
            return BadRequest(response);
        }

        return Ok(response);
    }
}
```

Isječak kôda 5.33: Kreiranje upravljačke klase za autentifikaciju korisnika

U konstruktor upravljačke klase ubrizgava se prethodno kreirani servis, a prilikom primanja HTTP zahtjeva na krajnju točku `"api/auth/login"` poziva se `Login` metoda servisne klase i proslijeđuje joj se korisničko ime i lozinka korisnika. Ukoliko korisnik ne postoji ili podaci nisu ispravni jednostavno se odgovoru neće priložiti i žeton korisnika čime se korisnik neće moći autentificirati. Prije nego što je moguće testirati ovu krajnju točku, potrebno je registrirati prethodno kreiranu servisnu klasu i `JwtBearer` servis unutar `ConfigureServices` metode `Startup` klase:

```
services.AddScoped<IAuthenticationService, AuthenticationService>();
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new
        ↪  SymmetricSecurityKey(Encoding.ASCII.GetBytes(
        Configuration.GetSection("Jwt:Key").Value)),
        ValidateIssuer = false,
```

```
        ValidateAudience = false,
        ValidateLifetime = true
    };
});
```

Isječak kôda 5.34: Registracija servisne klase za autentifikaciju u DI spremište

Prikazane opcije autentifikacijske sheme preuzete su iz službene *ASP.NET* dokumentacije i nemaju nekakav prevelik utjecaj na izvođenje web aplikacije pa stoga neće biti dodatno objašnjavane. Sa registracijom servisa u DI spremnik završava implementacija autentifikacijskog sloja na poslužiteljskoj strani. Nakon toga, potrebno je implementirati autentifikacijski sloj na klijentskoj strani aplikacije. Tamo će se također autentifikacijske funkcije provoditi pozivom metoda servisa, tako da je prvo potrebno kreirati `ILoginService` sučelje:

```
public interface ILoginService
{
    Task Login(string token);
    Task Logout();
}
```

Isječak kôda 5.35: Kreiranje sučelja za autentifikaciju korisnika na korisničkoj strani

Osim implementiranja kreiranog sučelja, servisna klasa mora nasljeđivati još i postojeću klasu `AuthenticationStateProvider` koju nudi *Blazor* kako bi se mogla nadjačati metoda za dohvaćanje autentifikacijskog stanja.

```
public class JwtAuthenticationStateProvider :
↪  AuthenticationStateProvider, ILoginService
{
    private readonly IJSRuntime _js;
    private readonly HttpClient _httpClient;
    private readonly string TOKENKEY = "TOKENKEY";
    private AuthenticationState Anonymous =>
        new AuthenticationState(new ClaimsPrincipal(new
        ↪  ClaimsIdentity()));

    public JwtAuthenticationStateProvider(IJSRuntime js, HttpClient
    ↪  httpClient)
    {
        _js = js;
        _httpClient = httpClient;
    }

    public async override Task<AuthenticationState>
    ↪  GetAuthenticationStateAsync()
    {
        var token = await _js.GetFromLocalStorage(TOKENKEY);
```

```
    if (string.IsNullOrEmpty(token))
    {
        return Anonymous;
    }

    return BuildAuthenticationState(token);
}

public AuthenticationState BuildAuthenticationState(string token)
{
    _httpClient.DefaultRequestHeaders.Authorization = new
    ↪  AuthenticationHeaderValue("bearer", token);

    return new AuthenticationState(new ClaimsPrincipal(new
    ↪  ClaimsIdentity(ParseClaimsFromJwt(token), "jwt")));
}

public IEnumerable<Claim> ParseClaimsFromJwt(string jwt)
{
    var claims = new List<Claim>();
    var payload = jwt.Split('.')[1];
    var jsonBytes = ParseBase64WithoutPadding(payload);
    var keyValuePairs =
    ↪  JsonSerializer.Deserialize<Dictionary<string,
    ↪  object>>(jsonBytes);

    keyValuePairs.TryGetValue(ClaimTypes.Role, out object roles);

    if (roles != null)
    {
        if (roles.ToString().Trim().StartsWith("["))
        {
            var parsedRoles =
            ↪  JsonSerializer.Deserialize<string[]>(roles.ToString());

            foreach (var parsedRole in parsedRoles)
            {
                claims.Add(new Claim(ClaimTypes.Role,
                ↪  parsedRole));
            }
        }
        else
        {
            claims.Add(new Claim(ClaimTypes.Role,
            ↪  roles.ToString()));
        }

        keyValuePairs.Remove(ClaimTypes.Role);
    }
```

```
            claims.AddRange(keyValuePairs.Select(kvp => new
            ↪  Claim(kvp.Key, kvp.Value.ToString())));

            return claims;
        }

        private byte[] ParseBase64WithoutPadding(string base64)
        {
            switch (base64.Length % 4)
            {
                case 2: base64 += "=="; break;
                case 3: base64 += "="; break;
            }
            return Convert.FromBase64String(base64);
        }

        public async Task Login(string token)
        {
            await _js.SetInLocalStorage(TOKENKEY, token);
            var authState = BuildAuthenticationState(token);
            NotifyAuthenticationStateChanged(Task.FromResult(authState));
        }

        public async Task Logout()
        {
            await _js.RemoveItem(TOKENKEY);
            _httpClient.DefaultRequestHeaders.Authorization = null;
            NotifyAuthenticationStateChanged(Task.FromResult(Anonymous));
        }
    }
```

Isječak kôda 5.36: Implementacija sučelja za autentifikaciju korisnika na korisničkoj strani

U konstruktor servisne klase se ubrizgava se `IJSRuntime` uz pomoć kojeg se pozivaju metode za pohranu, dohvaćanje i brisanje žetona iz lokalne pohrane te `HttpClient` servis nad čijom instancom je moguće postavljanje uobičajenih zaglavlja HTTP zahtjeva. To je vrlo bitna stvar jer je moguće postaviti da se uz slanje HTTP zahtjeva uvijek u zaglavlje priloži i žeton korisnika uz kojega je moguće napraviti autorizaciju korisnika na poslužiteljskoj strani te prihvatiti ili odbiti zahtjev ovisno o statusu autorizacije. Metoda `BuildAuthenticationState` upravo iz tog zaglavlja iščitava vrijednost žetona i kreira autentifikacijsko stanje prema onome što se nalazi u žetonu. Ukoliko se u lokalnoj pohrani ne nalazi žeton, tada se korisnik identificira kao anonimni dok se u suprotnom prema vrijednosti žetona izgrađuje autentifikacijsko stanje. Najbitnija metoda je zapravo `ParseClaimsFromJwt` koja je također preuzeta iz službene dokumentacije, a služi da iz žetona kao niza znakova iščita tvrdnje korisnika koje će biti dostupne u svim komponentama aplikacije. Također, postoje i metode `Login` i `Logout` koje jednostavno u lokalnu pohranu spremaju ili brišu žeton te obaviještavaju o promjeni autentifikacijskog stanja. Sa registracijom servisne klase unutar DI spremišta, moguće ju je ubrizgati

u komponente. Međutim, ukoliko se pažljivo promotri kôd servisne klase moguće je vidjeti da se unutar nje nigdje ne šalje HTTP zahtjev prema poslužiteljskoj strani jer ona kao takva nije zadužena za to već samo za uspostavu autentifikacije. Za slanje i primanje podataka su, kako je već objašnjeno, zaduženi repozitoriji. Prema uputama za migraciju podatkovnog sloja kreiran je `UserRepository` repozitorij za slanje podataka za prijavu prema krajnjoj točki poslužitelja i dohvaćanje žetona. Zatim je prema uputama za migraciju korisničkih kontrola i migraciju validacije kreirana komponenta stranice za prijavu korisnika koja koja poziva repozitorij i servisnu klasu za autentifikaciju. Programska logika komponente izgleda ovako:

```
public UserLoginDTO userLogin = new UserLoginDTO();

public async Task HandleLogin()
{
    var userToken = await userRepository.Login(userLogin);

    await loginService.Login(userToken.Data.Token);
    navigationManager.NavigateTo("/");
}
```

Isječak kôda 5.37: Implementacija funkcije za prijavu korisnika na stranici Prijava

Na klik gumba poziva se metoda koja šalje `UserLoginDTO` objekt koji sadrži korisničko ime i lozinku korisnika prema krajnjoj točki poslužitelja i dobavlja žeton ovisno o rezultatu autentifikacije. Zatim se žeton sprema u lokalnu pohranu web preglednika i vraća korisnika na početnu stranicu nakon uspješne autentifikacije. Posljednje, potrebno je ishodišnu komponentu `App.razor` omotati sa `CascadingAuthenticationState` komponentom koju nudi *Blazor*, a služi da se stanje autentifikacije propagira i bude dostupno kroz cijelu aplikaciju.

5.8.2. Migracija autorizacijskog stanja

Kako je već navedeno u uvodu poglavlja, stolna aplikacija ima dvije vrste korisnika: kupac (Customer) i voditelj prodaje (Manager) te svaki od njih ima različite ovlasti. Međutim, unutar aplikacije ne postoji nikakav mehanizam autorizacije korisnika već jednostavno postoje odvojene forme za prijavu svake vrste korisnika te različita sučelja glavnog izbornika aplikacije i samim time ponuđenih opcija u izborniku. U smislu sakrivanja elemenata korisničkog sučelja od korisnika i nedozvoljavanja pristupa pojedinim dijelovima aplikacije kao mehanizmima autorizacije, ovdje se radi o tipičnom pristupu kod razvoja *Windows Forms* aplikacija. Razlog tomu je taj što *Windows Forms* okvir ne nudi mogućnost kreiranja predložaka formi prema kojem bi se dinamički kreiralo sučelje ovisno o ulozi trenutno aktivnog korisnika, pa je stoga kreiranje jedne forme koja bi se prilagođavala ulogama korisnika teško izvedivo. Drugi razlog je taj što se unutar stolne aplikacije ne može jednostavno posjetiti određena forma aplikacije, odnosno stranica kao što je to slučaj sa web aplikacijama pa se ti dijelovi aplikacije jednostavno ne moraju autorizirati.

S druge strane, *Blazor* kao i mnogi drugi programski okviri za razvoj web aplikacija nude mo-

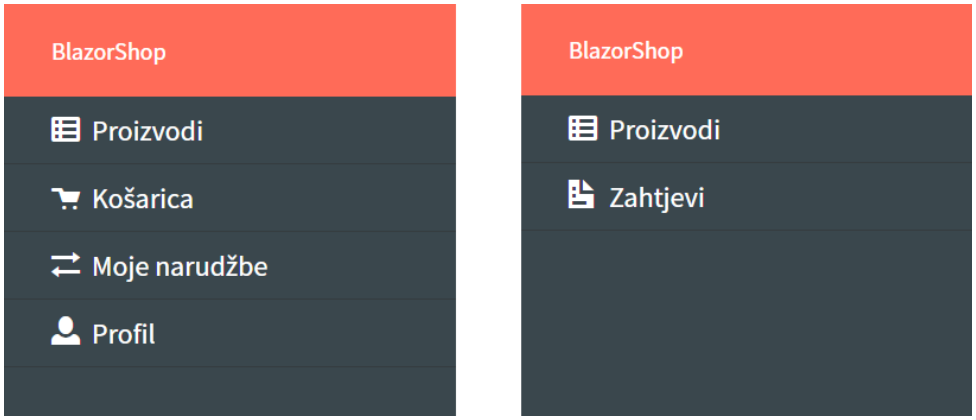
gućnosti za dinamičko generiranje HTML sadržaja ovisno o razini autorizacije korisnika kao i mogućnosti zaštite pojedinih stranica za korisnike koji nemaju dovoljno visoka prava pristupa. Specifičnije, *Blazor* nudi `AuthorizeView` komponentu koja ima parametar `Roles` prema kojoj se provjerava da li korisnik zadovoljava tu ulogu i ovisno o tome se prikazuje sadržaj. Ulogu korisnika moguće je liste tvrdnji (`Claim`) zapisanog na žetonu koji se generira na poslužiteljskoj strani, a taj dio funkcionalnosti je već implementiran u prethodnom potpoglavlju. Stoga, moguće je odmah krenuti na implementaciju autorizacijskog sloja na korisničkoj strani. Ideja je da se kreira komponenta navigacije `NavMenu` koja će dinamički prikazivati poveznice prema ostalim stranicama ovisno o ulozi prijavljenog korisnika. Struktura komponente izgleda ovako:

```
<nav class="navigation">
    <ul class="navigation__list">
        <li class="navigation__item">
            <NavLink class="navigation__link" href="proizvodi"
             ↪  Match="NavLinkMatch.All">
                <span class="oi oi-spreadsheet mr-1"
                 ↪  aria-hidden="true"></span> Proizvodi
            </NavLink>
        </li>

        <AuthorizeView Roles="Manager">
            <Authorized>
                <li class="navigation__item">
                    <NavLink class="navigation__link" href="zahtjevi"
                     ↪  Match="NavLinkMatch.All">
                        <span class="oi oi-document mr-1"
                         ↪  aria-hidden="true"></span> Zahtjevi
                    </NavLink>
                </li>
            </Authorized>
            <NotAuthorized>
                <li class="navigation__item">
                    <NavLink class="navigation__link" href="kosarica"
                     ↪  Match="NavLinkMatch.All">
                        <span class="oi oi-cart mr-1"
                         ↪  aria-hidden="true"></span> Košarica
                    </NavLink>
                </li>
            </NotAuthorized>
        </AuthorizeView>

        <AuthorizeView Roles="Customer">
            <Authorized>
                <li class="navigation__item">
                    <NavLink class="navigation__link" href="narudzbe"
                     ↪  Match="NavLinkMatch.All">
                        <span class="oi oi-transfer mr-1"
                         ↪  aria-hidden="true"></span> Moje narudžbe
                    </NavLink>
                </li>
```

```
                    <li class="navigation__item">
                        <NavLink class="navigation__link" href="profil"
                        ↪  Match="NavLinkMatch.All">
                            <span class="oi oi-person mr-1"
                            ↪  aria-hidden="true"></span> Profil
                        </NavLink>
                    </li>
                </Authorized>
            </AuthorizeView>
        </ul>
</nav>
```

Isječak kôda 5.38: Kreiranje navigacijske trake sa odgovarajućim poveznicama

Na prethodnoj HTML strukturi moguće je vidjeti kako se uz pomoć podkomponenti `Authorized` i `NotAuthorized` dinamički generira sadržaj sučelja ovisno o tome da li korisnik odgovara onoj ulozi definiranoj u parametru nadkomponente `AuthorizeView`. Na primjer, ukoliko se radi o voditelju prodaje, tada se želi prikazati poveznica na stranicu aktivnih zahtjeva odnosno narudžbi korisnika dok se u suprotnom želi prikazati poveznicu na stranicu sa košaricom (jer voditelj nabave ne kupuje proizvode). Na slici ispod s lijeve strane je prikazano generirana navigacijska traka kupca, a s desne voditelja prodaje:



Slika 29: Prikaz navigacijske trake kupca i navigacijske trake voditelja prodaje

Valja naglasiti da samo sakrivanje sadržaja korisničkog sučelja nije dovoljno kako bi se zaštitile pojedine sekcije web aplikacije. Na primjer, samo voditelj prodaje je autoriziran da kreira novi proizvod i upravlja narudžbama korisnika, pa se te sekcije (stranice) moraju zaštititi. To se realizira uz pomoć tako da se doda `@attribute [Authorize]` atribut unutar deklaracijskog dijela komponente, a također je moguće i navesti koje se uloge želi propustiti. Osim komponenti korisničkog sučelja, moguće je i zaštiti komponente poslužiteljske strane odnosno upravljačke klase koje obrađuju HTTP zahtjeve klijentske strane. Zaštita upravljačkih klasa se provodi iz razloga što je, osim putem formi na korisničkom sučelju, slanje zahtjeva prema njima moguće i putem bilo kojeg alata slanje HTTP poziva kao što je *Postman*, *Insomnia* i drugi. Želi se postići da se obrađuju samo zahtjevi autoriziranih korisnika (osim zahtjeva za prijavu), a ne

svi zahtjevi. Zaštita upravljačkih klasa provodi se na način da se iznad definicije klase doda dekorator `[Authorize]` kojem je opet moguće proslijediti uloge koje se želi propustiti. S obzirom da osim upravljačke klase za autentifikaciju postoji samo ona za rad sa proizvodima, nju je potrebno zaštititi na prethodno spomenut način.

Sa implementacijom zaštite upravljačkih klasa završava migracija sigurnosnog sloja, kao i migracija svih temeljnih slojeva koji su postojani unutar *ProphetShop* stolne aplikacije. Bitno je još jednom naglasiti da je cilj ovoga poglavlja bio opis na koji način se migriraju pojedini elementi stolne aplikacije na web aplikaciju, a ne da se opiše kompletni razvoj web aplikacije. Kroz ponavljanje tih koraka migracije moguće je vrlo lako implementirati sve ostale funkcionalnosti aplikacije i, eventualno, migrirati kompletnu stolnu aplikaciju.

6. Zaključak

WebAssembly je novi binarni instrukcijski format za virtualne strojeve temeljene na stogu, te od nedavno i četvrti Web standard za razvoj web stranica uz HTML, CSS i Javascript. Format je dizajniran na način da bude prenosiv rezultat prevođenja viših programskih jezika pa s time omogućuje ponovno iskorištavanje kôda napisanom u jednom programskom jeziku i izvršavanje unutar okruženja ugrađivača koji izvršava WebAssembly. Osim prenosivosti i mogućnosti izvršavanja na svakoj platformi, WebAssembly karakterizira brzina jer se tipovi podataka određuju prije izvršavanja programa, a ne za vrijeme izvršavanja programa kao što je to slučaj s Javascriptom pa ga je samim time moguće jednostavnije ubrzati. Iako ta razlika upućuje na to da se WebAssembly pojavio kao zamjena za Javascript koji sve teže ispunjuje zahtjeve platforme, zapravo se pojavio kao komplement koji ispunjava nedostatke Javascripta. Ovo se najviše odnosi na procesorski zahtjevne zadatke koje Javascript teško izvodi upravo zbog spomenutog sporog ubrzanja. Nastavno na prenosivost, valja istaknuti da je jedna od primarnih ideja praktičnog dijela ovog rada bila da se migrira postojeća stolna .NET aplikacija u Blazor WebAssembly okvir za razvoj web aplikacija koji također koristi C# programski jezik za razvoj aplikacija i tako ponovno iskoristi programski kôd. Iako je migracija uspjela zahvaljujući brojnim mogućnostima Blazor WebAssembly okvira (ponajviše interoperabilnosti sa Javascriptom), nije iskorištena očekivana količina programskog kôda. To se zapravo moglo i pretpostaviti s obzirom da se radilo o stolnoj aplikaciji sa grafičkim sučeljem za koju ne postoje jasna mapiranja u web sučelja ili biblioteke. Smatram da bi se ostvarila veća iskoristivost kôda sa migracijom neke C/C++ biblioteke bez grafičkog sučelja jer bi se osnovna stolna sučelja mogla jednostavno mapirati u web sučelja (socket.h u WebSocket, OpenGL u WebGL itd.).

Sa svime spomenutim, može se bez pretjerivanja reći da je WebAssembly jedna od najbitnijih tehnologija u domeni razvoja softvera prvenstveno iz razloga što pristupa rješavanju problema prenosivosti aplikacija i ponovne iskoristivosti kôda. Za vrijeme pisanja ovog rada već su mnoge kompanije prihvatile WebAssembly i iskoristile njegove mogućnosti te migrirale vlastite biblioteke na Web ili jednostavno zamijenile postojeće Javascript komponente sa njihovim WebAssembly implementacijama umjesto da iste iznova razvijaju. Smatram da će se, uz daljnji razvoj tehnologije i proširenje mogućnosti kroz planirane nove osobine, sve više i više kompanija početi koristiti WebAssembly s ciljem smanjenja troškova razvoja. Iako je WebAssembly poprilično novi element u Web razvoju, ne treba ga odbijati i biti skeptičan prema njegovu korištenju već ga prihvatiti jer definitivno predstavlja budućnost web razvoja kakvog poznajemo danas.

Popis literature

[1] Wikipedija, *World Wide Web — Wikipedija, Slobodna enciklopedija*, //hr.wikipedia.org/w/index.php?title=World_Wide_Web&oldid=5225114, [Online; accessed 1-May-2020], 2019.

[2] A. van der Hiel. (2019). World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation, World Wide Web Consortium, adresa: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.

[3] (). WebAssembly. [Online; accessed 1-May-2020], adresa: https://webassembly.org/.

[4] C. Eberhardt. (). WebAssembly and the Death of JavaScript - JS Monthly - February 2018. Pristupljeno 23.6.2020, adresa: https://www.youtube.com/watch?v=pBYqen3B2gc.

[5] (Lipanj 2020). WebAssembly Concepts. Pristupljeno: 31.8.2020, adresa: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

[6] L. Clark. (). A Cartoon Intro to WebAssembly | JSConf EU. Pristupljeno 6.5.2020, adresa: https://www.youtube.com/watch?v=HktWin_LPf4.

[7] M. Rourke, *Learn WebAssembly, Build web applications with native performance using Wasm and C/C++*. Livery Place 35 Livery Street Birmingham B3 2PB, UK.: Packt Publishing Ltd, 2018, ISBN: 978-1-78899-737-9.

[8] S. Chapman, „What Javascript Cannot Do”, 2019. adresa: https://www.thoughtco.com/what-javascript-cannot-do-2037666.

[9] A. Rossberg. (). WebAssembly Specification. Pristupljeno 10.5.2020, adresa: https://webassembly.github.io/spec/core/index.html.

[10] A. R. Andreas Haas, D. L.Schuff, B. L.Titzer, D. Gohman, L. Wagner, A. Zakai, M. Holman i J. Bastien, „Bringing the web up to speed with WebAssembly”, *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, srpanj 2017, str. 185–200. adresa: https://doi.org/10.1145/3062341.3062363.

[11] E. D. B. Abhinav Jangda Bobby Powers i A. Guha, „Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code”, *Proceedings of the 2019 USENIX Annual Technical Conference*, Renton, WA, USA, srpanj 2019, str. 107–120, ISBN: 978-1-939133-03-8. adresa: https://www.usenix.org/system/files/atc19-jangda.pdf.

[12] Wikipedia contributors, *Google Native Client — Wikipedia, The Free Encyclopedia*, [Online; accessed 1-September-2020], 2020. adresa: https://en.wikipedia.org/w/index.php?title=Google_Native_Client&oldid=973604804.

[13] ——, *Emscripten — Wikipedia, The Free Encyclopedia*, [Online; accessed 1-September-2020], 2020. adresa: https://en.wikipedia.org/w/index.php?title=Emscripten&oldid=939699826.

[14] D. Herman, L. Wagner i A. Zakai. (kolovoz 2014). asm.js Specification. Pristupljeno 19.5.2020, adresa: http://asmjs.org/spec/latest.

[15] Wikipedia contributors, *Asm.js — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Asm.js&oldid=930996350, [Online; accessed 19-May-2020], 2019.

[16] Unknown. (svibanj 2013). 'Epic Citadel' Demo Shows the Power of the Web as a Platform for Gaming, adresa: https://blog.mozilla.org/futurereleases/2013/05/02/epic-citadel-demo-shows-the-power-of-the-web-as-a-platform-for-gaming/.

[17] R. R, „Epic Citadel in Firefox now!", Pristupljeno 1.8.2020, svibanj 2013. adresa: https://www.mono-live.com/2013/05/epic-citadel-in-firefox-now.html.

[18] G. Gallant, *WebAssembly in action, With examples using C++ and Emscripten*. 20 Baldwin Road PO Box 761 Shelter Island, NY 11964: Manning Publications Co., 2019, ISBN: 9781617295744.

[19] A. Rossberg. (prosinac 2019). WebAssembly Core Specification, W3C Recommendation, adresa: https://www.w3.org/TR/wasm-core-1/.

[20] Wikipedia contributors, *Virtual machine — Wikipedia, The Free Encyclopedia*, [Online; accessed 29-May-2020], 2020. adresa: https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=959523551.

[21] ——, *Stack machine — Wikipedia, The Free Encyclopedia*, [Online; accessed 29-May-2020], 2020. adresa: https://en.wikipedia.org/w/index.php?title=Stack_machine&oldid=952444375.

[22] Unknown. (). Github - WAVM/WAVM: WebAssembly Virtual Machine. Pristupljeno 30.5.2020, adresa: https://github.com/WAVM/WAVM.

[23] Wikipedia contributors, *Nondeterministic programming — Wikipedia, The Free Encyclopedia*, [Online; accessed 13-July-2020], 2019. adresa: https://en.wikipedia.org/w/index.php?title=Nondeterministic_programming&oldid=909137143.

[24] P. Christensson. (svibanj 2018). Flag Definition. Retrieved 2020, Sep 1 from https://techterms.com.

[25] Wikipedia contributors, *DWARF — Wikipedia, The Free Encyclopedia*, [Online; accessed 26-July-2020], 2019. adresa: https://en.wikipedia.org/w/index.php?title=DWARF&oldid=925857616.

[26] W. Scott. (lipanj 2019). WebAssembly Debugging, The current state of interactive debugging for WebAssembly and useful tips on how to do better. Pristupljeno: 26.7.2020, adresa: https://medium.com/oasislabs/webassembly-debugging-bec0aa93f8c6.

[27] I. Stepanyan. (lipanj 2020). Improved WebAssembly debugging in Chrome DevTools. Pristupljeno: 26.7.2020, adresa: https://developers.google.com/web/updates/2019/12/webassembly.

[28] M. Contributors. (ožujak 2019). Use a source map. Pristupljeno: 1.9.2020, adresa: https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map.

[29] Unknown. (2020). WebAssembly System Interface (WASI). [Pristupljeno: 30.6.2020], adresa: https://nodejs.org/api/wasi.html.

[30] A. Turner, „Made With WebAssembly - 8th Wall", Pristupljeno 1.8.2020, veljača 2020. adresa: https://madewithwebassembly.com/showcase/8thwall.

[31] A. Danilo i D. Gangluri. (). WebAssembly: Real World Applications. Pristupljeno 4.7.2020, adresa: https://www.youtube.com/watch?v=ysFJHpS-O08.

[32] A. Turner, „Made With WebAssembly - Construct 3", Pristupljeno 1.8.2020, veljača 2020. adresa: https://madewithwebassembly.com/showcase/construct-3.

[33] D. Smilkov, N. Thorat i A. Yuan. (ožujak 2020). Introducing the WebAssembly backend for TensorFlow.js, adresa: https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html.

[34] C. Ebarhardt. (srpanj 2018). The future of WebAssembly - A look at upcoming features and proposals, adresa: Pristupljeno%205.7.2020.

[35] J. Trivedi, *Building a Web App with Blazor and ASP .Net Core, Create a Single Page App with Blazor Server and Entity Framework Core*. BPB Publications, 2020, ISBN: 9389845459.

[36] A. Freeman, *Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*. Apress, 2020, ISBN: 9781484254400. adresa: https://books.google.hr/books?id=o5npDwAAQBAJ.

[37] T. M. Runtime. (prosinac 2018). Alexander Köplinger. Pristupljeno: 30.7.2020, adresa: https://www.mono-project.com/docs/advanced/runtime/.

[38] E. Charbeneau. (studeni 2019). Goodbye Client Side JavaScript, Hello C# Blazor. Pristupljeno: 31.7.2020, adresa: https://www.youtube.com/watch?v=hYaWm2xZv9E.

Popis slika

1. Proces kompilacije Javascript kôda 4
2. Izvršavanje Wasm modula 5
3. Struktura Web platforme 5
4. WebAssembly unutar virtualnog stroja 6
5. Unreal Engine unutar Epic Citadel demo igre [17] 11
6. Zbrajanje dvaju brojeva na stogu 13
7. Mapiranje formata 14
8. C++ funkcija i njen binarni ekvivalent 15
9. C++ funkcija i njen tekstualni ekvivalent 15
10. Struktura WebAssembly modula 16
11. Semantičke faze Wasm modula 18
12. Otkrivanje grešaka na tekstualnom formatu unutar Chrome DevTools [27] 22
13. Otkrivanje grešaka u izvornom kôdu unutar Chrome DevTools [27] 23
14. 8th Wall web aplikacija [30] 27
15. Construct 3 web uređivač igara [32] 28
16. Komponentni model Blazor okvira 35
17. Životni ciklus Blazor komponente 37
18. Blazor WebAssembly aplikacija u pregledniku 41
19. Struktura ProphetShop projekta 44
20. .NET Portability Report 45
21. Kreiranje novog Blazor WebAssembly projekta 46
22. Razlike u tokovima podataka između stolne i web .NET aplikacije 47
23. Ispis konzole u web pregledniku 57

24. Dizajn komponente za prikaz proizvoda na stolnoj aplikaciji 60
25. Prikaz liste proizvoda na webu 63
26. Prikaz liste proizvoda sa slikama na webu 66
27. Validacija forme za unos proizvoda na stolnoj aplikaciji 72
28. Validacija forme za unos proizvoda na web aplikaciji 74
29. Prikaz navigacijske trake kupca i navigacijske trake voditelja prodaje 84

Popis tablica

1. Popis instrukcija prema kategorijama 20

2. Servisi zadani od Blazor programskog okvira 39

Popis isječaka kôda

3.1. C funkcija i njen asm.js ekvivalent . . . 10
3.2. Primjer Wasm tekstualnog koda za vraćanje dviju vrijednosti u funkciji . . . 30
4.1. SetParametersAsync metoda životnog ciklusa komponente . . . 36
4.2. Registracija servisne klase unutar DI spremišta . . . 39
4.3. Ubrizgavanje servisne klase u komponentu . . . 39
5.1. Dodavanje konteksta baze podataka u DI spremište . . . 49
5.2. Dodavanje podataka o vezi na bazu podataka unutar postavki . . . 49
5.3. Kreiranje sučelja generičkog repozitorija . . . 49
5.4. Implementacija sučelja generičkog repozitorija . . . 51
5.5. Kreiranje sučelja repozitorija za upravljanje proizvodima . . . 51
5.6. Implementacija sučelja repozitorija za upravljanje proizvodima . . . 52
5.7. Upravljačka klasa za upravljanje proizvodima . . . 53
5.8. Registracija repozitorija za upravljanje proizvodima u DI spremište . . . 53
5.9. Kreiranje sučelja za slanje HTTP zahtjeva putem HTTP klijenta . . . 54
5.10. Implementacija generičkog HTTP klijenta . . . 55
5.11. Kreiranje sučelja za upravljanje proizvodima na klijentskoj strani . . . 55
5.12. Implementacija sučelja za upravljanje proizvodima na klijentskoj strani . . . 56
5.13. Testiranje repozitorija za upravljanje proizvodima . . . 57
5.14. Mapiranje modela proizvoda u pogledni model . . . 58
5.15. Kreiranje poglednog modela za prikaz proizvoda . . . 58
5.16. Dodavanje AutoMapper servisa u DI spremište . . . 59
5.17. Mapiranje modela unutar upravljačke klase . . . 59
5.18. Kreiranje komponente za prikaz proizvoda . . . 61
5.19. Kreiranje stranice za prikaz liste proizvoda . . . 62
5.20. Unos podataka za povezivanje na AzureStorage servis unutar postavki . . . 64
5.21. Kreiranje sučelja za spremanje datoteka na AzureStorage servis . . . 64
5.22. Implementacija sučelja za spremanje datoteka na AzureStorage servis . . . 65
5.23. Implementacija mehanizma za sinkronizaciju dinamičkih resursa . . . 66
5.24. Kreiranje klase aplikacijskog stanja . . . 68
5.25. Implementacija mehanizma za perzistenciju aplikacijskog stanja . . . 70
5.26. Kreiranje Javascript funkcija za upravljanje aplikacijskim stanjem . . . 71
5.27. Kreiranje komponente za dijeljenje aplikacijskog stanja . . . 72
5.28. Dodavanje komponente za validaciju korisničkog unosa teksta . . . 73
5.29. Dodavanje komponente za validaciju korisničkog unosa vrijednosti . . . 73

5.30. Dodavanje Json Web Token javnog ključa 75
5.31. Kreiranje sučelja autentifikacijskog servisa 75
5.32. Implementacija sučelja autentifikacijskog servisa 77
5.33. Kreiranje upravljačke klase za autentifikaciju korisnika 78
5.34. Registracija servisne klase za autentifikaciju u DI spremište 79
5.35. Kreiranje sučelja za autentifikaciju korisnika na korisničkoj strani 79
5.36. Implementacija sučelja za autentifikaciju korisnika na korisničkoj strani 81
5.37. Implementacija funkcije za prijavu korisnika na stranici Prijava 82
5.38. Kreiranje navigacijske trake sa odgovarajućim poveznicama 84