Implementacija arhitekture klijent-poslužitelj korištenjem deklarativnog programskog jezika

Kranjec, Filip

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Filip Kranjec

IMPLEMENTACIJA ARHITEKTURE KLIJENT-POSLUŽITELJ KORIŠTENJEM DEKLARATIVNOG PROGRAMSKOG JEZIKA

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Kranjec

Matični broj: 50563

Studij: Informacijsko i programsko inžinjerstvo

IMPLEMENTACIJA ARHITEKTURE KLIJENT-POSLUŽITELJ KORIŠTENJEM DEKLARATIVNOG PROGRAMSKOG JEZIKA

DIPLOMSKI RAD

Mentor :

Prof. dr. sc. Markus Schatten

Varaždin, rujan 2024.

Filip Kranjec

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

Svrha ovoga rada jest prikazati jedan od načina implementiranja arhitekture klijent-poslužitelj koja u velikoj razini može ubrzati proces implementacije i razdvojiti sustav na više dijelova. Teorijski dio sastojat će se od objašnjena klijent-poslužitelj arhitekture i vrsta takvih arhitektura, usporedbe različitih arhitektura te objašnjenja funkcijskog programiranja koje postaje sve popularnije u današnje vrijeme. Također će biti opisani svi alati koji su korišteni za implementaciju takvog sustava i razlog korištenja tih alata. Za praktični dio rada implementiran je sustav koristeći objašnjene tehnologije i alate. Implementirani sustav služi kao platforma za e-učenja te će biti prikazani svi glavni ekrani i programski kod kojim se ti ekrani služe

Ključne riječi: funkcijsko programiranje, Erlang, WebAssembly, Angular, Rust, klijent-poslužitelj

Sadržaj

1. Uvod

Jedan od većih problema prilikom razvoja sustava u današnje vrijeme jest kreiranje arhitekture sustava koji će biti robustan, skalabilan i prenosiv. Pošto je prilikom izgradnje sustava zapravo najskuplja stavka sama implementacija sustava za koju je potrebno vrijeme nužno je razviti arhitekturu pomoću koje je implementacija razvojem sve jednostavnija za održavanje i nadograđivanje. Iz tog razloga korišten je programski jezik Erlang pomoću kojega je moguće implementirati podatkovni i aplikacijski sloj unutar jednog projekta. Također jako zanimljiva tema jest WebAssembly najviše zbog svojih performansi unutar preglednika, ali i zbog prenosivosti pomoću koje je moguće implementirati algoritme, strukture te zapravo potpune funkcionalnosti koje se koriste na više prezentacijskih slojeva kao na primjer unutar preglednika ili čak unutar mobilne aplikacije. Time se postiže sigurnost da će svaka klijentska aplikacija koristiti jednake algoritme i još važnije imate unificirane strukture.

Početak ovog rada opisuje samu arhitekturu klijent poslužitelj, vrste koje postoje poput jedno razinske, dvo razinske arhitekture te prednosti korištenja navedene arhitekture prilikom izgradnje sustava. Sljedeće na redu jest funkcijska paradigma programiranja pošto Erlang koristi tu paradigmu te ju je moguće koristiti i u Rust-u pomoću kojega je implementiran WebAssembly dio sustava te unutar TypeScript-a i Angular-a koji u zadnjih nekoliko godina i potiču programere da koriste funkcijsko programiranje.

Svaki od korištenih alata unutar praktičnog dijela potkrijepljen je teorijskom podlogom unutar koje se dublje ulazi u samu sintaksu programskih jezika, okvira te biblioteka koje su korištene kako bi se ubrzala i olakšala implementacija.

Za praktični dio rada implementiran je sustav koji se temelji na klijent poslužitelj arhitekturi te koristi četiri zasebna projekta koji su međusobno povezani. Za bazu podataka koristi se Mnesia koja je ugrađena u Erlang sustav te je odvojena u zaseban projekt koji se koristi kao biblioteka unutar projekta gdje je implementiran REST API također koristeći Erlang i biblioteku Cowboy. Prezentacijski sloj aplikacije implementiran je koristeći programski okvir Angular unutar kojega je uveden projekt za dohvaćanje podataka s poslužitelja koji je napisan u Rust-u te kompajliran u WebAssembly kako bi se unificirale strukture koje se koriste te funkcije koje su potrebne da bi aplikacija komunicirala s poslužiteljem.

2. Metode i tehnike rada

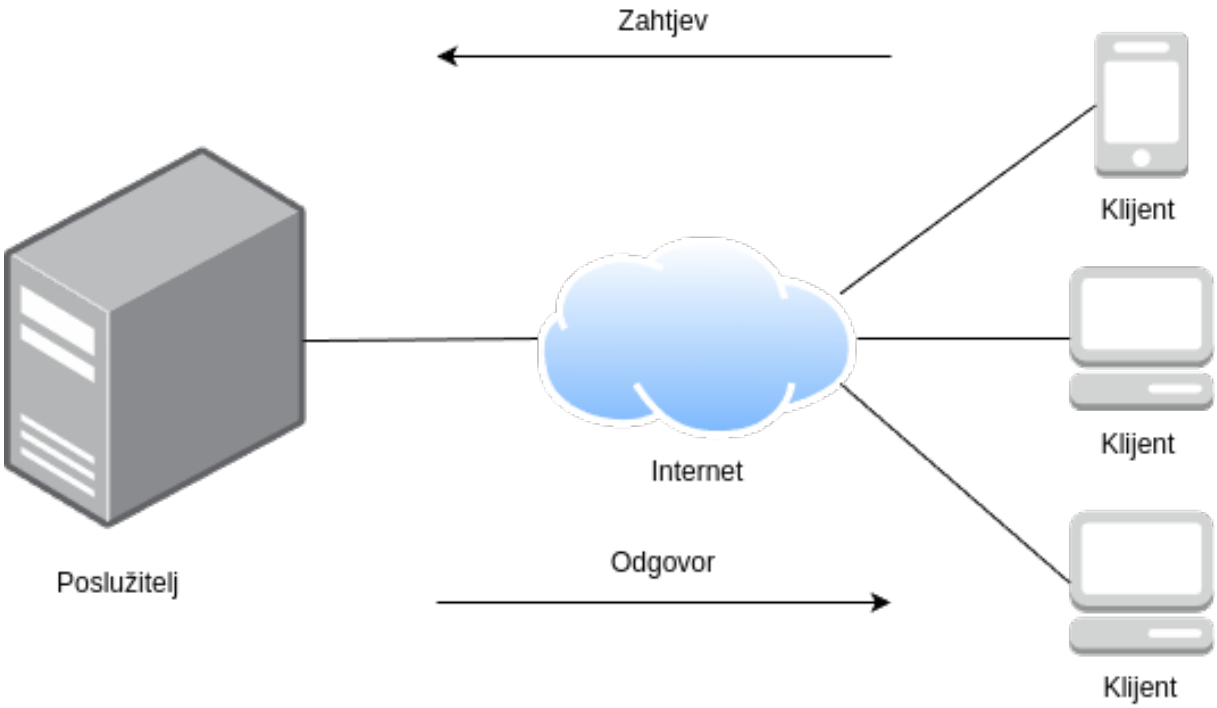
Pismeni dio rada napisan je pomoću alata LaTeX unutar internet aplikacije Overleaf. U svrhu izrade dijagrama i grafičkih prikaza korišten je alat Draw.io.

Za izradu praktičnog dijela korišten je uređivač koda NeoVim unutar kojeg su instalirani svi potrebni jezični protokoli (engl. *Language Server Protocol, LSP*) koji omogućavaju sve funkcionalnosti koje su dostupne unutar popularnijih uređivača koda poput programa Visual Studio Code. Potpunu konfiguraciju za NeoVim moguće je pronaći na osobnom GitHub repozitoriju.

Također cijela aplikacija je implementirana na operacijskom sustavu NixOS koji je također poznat zbog svoje prenosivosti i zbog toga što koristi Nix koji je deklarativni programski jezik točnije koristi paradigmu funkcijskog programiranja. Prednost tome je sigurnost da će aplikacija raditi na bilo kojem poslužitelju na kojem je instaliran NixOS i koristi konfiguraciju unutar koje je implementirana. Potpunu konfiguraciju je moguće također pronaći na osobnom GitHub repozitoriju te je moguće vidjeti kako su instalirani alati koji se koriste poput rebar3 što je standardni alat za izgradnju Erlang aplikacija. Također se koristi alat rustup za instalaciju svih alata koji su potrebni da bi se mogla implementirati Rust aplikacija koja se prevodi u WebAssembly te je instaliran Node.js pomoću kojega je moguće kreirati Angular projekte.

3. Klijent-poslužitelj arhitektura

Klijent-poslužitelj arhitektura je temeljni model koji se koristi u distribuiranom računarstvu. Pripada distribuiranim informacijskim sustavima iz razloga što se dvije glavne komponente arhitekture klijent i poslužitelj nalaze na više fizičkih ili virtualnih lokacija. U takvim sustavima klijent predstavlja računalo ili uređaj koji traži podatke te šalje zahtjeve dok pružatelj te zahtjeve prima, obrađuje ih te šalje nazad klijentu u obliku odgovora na zahtjev te dodatno u sebi sadrži podatke koje je klijent zatražio. U većini slučajeva pružatelj je centraliziran na jednu lokaciju te prima zahtjeve sa klijenata koji se nalaze na više lokacija, ali također je moguće i poslužitelja implementirati na više udaljenih računala koja međusobno komuniciraju. Ovakva arhitektura dizajnirana je kako bi se optimiziralo korištenje resursa te povećala skalabilnost sustava.[1]



Slika 1: Klijent-poslužitelj arhitektura (Izvor: Autorski rad)

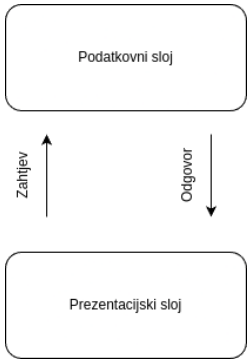
Prvi put se pojavljuje 1980-ih godina zato što su osobna računala postala naprednija te se pojavila mogućnost da se dio opterećenja s poslužitelja prebaci na klijente.[1] Time se omogućilo da se na klijentima izvršavaju osnovne pa čak i naprednije funkcionalnosti koje se prije nisu mogle izvršavati nego se sve odvijalo na poslužiteljima. To se može vidjeti čak i na web aplikacijama pošto u današnje vrijeme većina aplikacija koristi prikazivanje na strani klijenta umjesto prikazivanja na strani poslužitelja što prije nije bilo moguće zbog nedostatka resursa na osobnim računalima.

Neki od najvećih nedostataka ovakve arhitekture su transformiranje podataka pošto implementacija baze podataka ne mora treba određivati modele koji se koriste na strani klijenta te u većini slučajeva se kreiraju upiti pridruživanja (engl. *join*) pomoću kojih se podaci iz baze podataka transformiraju u podatke koji se koriste na klijentu. Nadalje jedan od problema je taj

što se svaki zahtjev obrađuje u zasebnoj dretvi te u tim trenucima može lako doći do pogreške koja kod nekih alata ili tehnologija može rezultirati prestankom rada aplikacije te prestankom rada aplikacijskog sloja. [2]

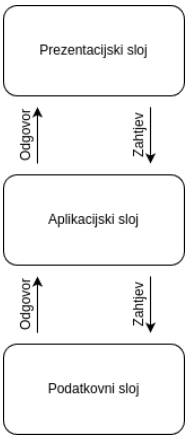
3.1. Vrste klijent-poslužitelj arhitekture

U početcima korištenja klijent-poslužitelj arhitekture postojao je samo jedan način implementacije, a to je dvoslojna arhitektura od koje je zapravo i došlo do raspodjele sustava na zadnju stranu (engl. *backend*) i prednju stranu (engl. *frontend*). Takva arhitektura je kao što i sam naziv govori podijeljena na dva sloja, a to su podatkovni sloj unutar kojega se nalazi baza podataka i sva potrebna poslovna logika te sloj klijenta unutar kojega se nalazi prezentacijski dio. U tom primjeru klijent komunicira izravno s poslužiteljem što zapravo povećava performanse sustava sve dok ne dođe do većeg broja korisnika pri čemu poslužitelj troši previše resursa kako bi svi klijenti mogli dobiti željene podatke.[3]



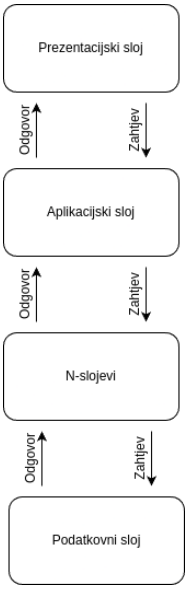
Slika 2: Dvoslojna arhitektura (Izvor: Autorski rad)

Sljedeća vrsta jest troslojna arhitektura unutar koje se uvodi srednji sloj ili tako zvani aplikacijski sloj te ostala dva sloja ostaju ista, a to su podatkovni i prezentacijski sloj. To je u današnje vrijeme i najčešći način implementacije klijent-poslužitelj arhitekture zato što se dodatno povećavaju performanse prilikom velikog broja korisnika. Dodatno uvedeni sloj se također može implementirati na većem broju udaljenih računala tako da je dodatno moguće odrediti koji klijent na kojem udaljenom računalu troši resurse, a prilikom toga baza podataka ostaje jedini izvor istine (engl. *single source of truth*).[3] Pošto operacije nad bazom podataka nisu skupe kao i obrada podataka na aplikacijskom sloju nije potrebno uvoditi više baza podataka dok je kod većih sustava poželjno imati poslužitelja implementiranih na više udaljenih računala pri čemu je onda također potrebno implementirati upravitelja opterećenja (engl. *load balancer*) koji određuje s kojim aplikacijskim slojem komunicira prezentacijski sloj.



Slika 3: Troslojna arhitektura (Izvor: Autorski rad)

Zadnja vrsta jest N-slojna arhitektura koja proširuje troslojnu arhitekturu tako da se uvode dodatni slojevi od kojih je svaki zadužen za specifične zadatke. Primjer takve arhitekture bi bio sustav koji dodatno uz aplikacijski sloj ima i sigurnosni sloj koji se nalazi na potpuno drugim udaljenim računalima. Najveća prednost im je zapravo razvoj takve arhitekture u velikim poduzećima pošto je svaki sloj neovisan o drugom sloju te ih je moguće razvijati u izolaciji. To omogućuje poduzećima podjelu programera u puno specifičnije timove čime se zapravo pojednostavljuje posao programera.[3]



Slika 4: N-slojna arhitektura (Izvor: Autorski rad)

4. Funkcijska paradigma programiranja

Funkcijsko programiranje je vrlo zanimljiva tema zato što se temelji na lambda računu kojega je osmislio Alonzo Church 1936. godine.[4] Za razliku od imperativnih programskih jezika unutar kojih se programira s redoslijedom naredbi koje mijenjaju stanje, funkcijski programski jezici pripadaju deklarativnim programskim jezicima unutar kojih se programira specificirajući što želimo, a ne kako.[5] Neki od najpoznatijih čistih funkcijskih programskih jezika su: LISP, Haskell, Scala i Erlang. Također postoje i programski jezici koji nisu čisto funkcionalni ali podržavaju sve koncepte funkcijskog programiranja, a neki od poznatijih su: Python, JavaScript i Rust.

4.1. Koncepti funkcijskog programiranja

Iako nisu svi funkcijski programski jezici potpuno jednaki postoji nekoliko koncepata zbog koji se smatraju funkcijskim jezicima, a to su:

- **Čiste funkcije (engl. *Pure functions*)** - su one funkcije koje za ulazne vrijednosti uvijek vraćaju jednaki izlaz te nema nikakvih popratnih efekata što znači da funkcija ne smije raditi promjene na varijablama izvan sebe. Na taj način postiže se lakše razumijevanje i testiranje koda.
- **Nepromjenjivost (engl. *Immutability*)** - označava da se varijable ne smiju mijenjati nakon što su stvorene već je potrebno stvoriti novu kopiju s izmijenjenim vrijednostima. Zbog svojstva nepromjenjivosti nemoguće je doći do pogreške zbog neočekivane promjene stanja
- **Funkcije višeg reda (engl. *Higher-order functions*)** - su funkcije koje se mogu dodjeljivati varijablama, prosljeđivati kao argument i vraćati kao rezultat drugih funkcija (engl. *currying*).
- **Rekurzije (engl. *Recursion*)** - su funkcije koje pozivaju same sebe te na taj način zamjenjuju imperativne petlje poput for i while.
- **Kompozicija (engl. *Composition*)** - je pristup programiranju gdje se funkcije grade od više manjih funkcija isto kao i kompozicija funkcija u matematici. Kompozicija potiče ponovno korištenje koda i modularnost.[6]

4.2. Lambda račun

Prvo je bitno napomenuti kako izgleda sintaksa lambda račun $\lambda x, y.xc$ gdje:

- λ - predstavlja početak funkcije,

- **.** - pomoću znaka točke se dijele ulazne i izlazne varijable što znači da se ispred točke nalaze ulazne, a iza točke izlazne,
- **X** - predstavlja vezanu varijablu zato što je ulazna i izlazna vrijednost iz funkcije,
- **Y** - predstavlja slobodnu varijablu zato što je samo ulazna varijabla,
- **C** - predstavlja konstantu

4.2.1. Osnovni koncepti lambda računa

Nakon upoznavanja sa sintaksom lambda računa jednostavnije se mogu objasniti neke od bazičnih koncepata na koje se zapravo svodi cijeli lambda račun. Varijable su već spomenute tako da je poznato da postoje ulazne i izlazne. Sljedeći bitan koncept je apstrakcija. Za apstrakciju se može reći da u funkciju izdvaja dio koda koji će se moći ponovno koristiti (engl. *reusable*). Primjer apstrakcije u lambda računu izgleda ovako:

$$\lambda x.x + 1$$

ta funkcija prima vezanu varijablu x i vraća $x + 1$ te predstavlja funkciju za inkrementaciju.[6] Sljedeći bitan koncept je koncept aplikacije odnosno primjena funkcije koji se odnosi zapravo na predavanje vrijednosti funkciji, jednostavnije rečeno dodjela vrijednosti ulaznoj varijabli ili varijablama. Aplikacija se zapisuje u sljedećem obliku:

$$(\lambda x.x + 1)2$$

gdje se može primijetiti da se ponovno koristi funkcija za inkrementaciju, ali ovaj put se ulaznoj varijabli x dodjeljuje vrijednost 2 te je izlaz iz ove funkcije jednak 3. Nakon osnovnih koncepata prelazi se na kompleksnije koncepte koji se temelje na procesu redukcije.

4.2.2. Evaluacija

Prvo je potrebno objasniti proces redukcije kako bi se jednostavnije shvatili. Redukcija predstavlja pojednostavljivanje složenog izraza dok ne dosegnu minimalni oblik. Tako na primjer izraz

$$(1 * 9) + (3 * 4)$$

redukcijom se može dovesti do jedne vrijednosti koja je 21 i to ne na jedan način već na više načina pošto sustavi redukcije zadovoljavaju Church-Rosser svojstvo koje govori da je dobivanje normalne forme neovisno o redoslijedu izraza.[7] Znajući to do rješenja je moguće doći na

sljedeće načine:

$$\begin{aligned}(1*9)+(3*4) &\rightarrow 9+(3*4)\\ &\rightarrow 9+12\\ &\rightarrow 21\\ (1*9)+(3*4) &\rightarrow (1*9)+12\\ &\rightarrow 9+12\\ &\rightarrow 21\end{aligned}$$

Nakon upoznavanja s procesom redukcije jednostavnije je opisati što je zapravo evaluacija lambda računa. To je postupno pojednostavljivanje lambda računa uz dodatna pravila koja se koriste, a to su:

- $\alpha - \mathbf{konverzija}$ koja predstavlja preimenovanje varijabli što znači da se izraz $\lambda x.x$ može pretvoriti u izraz $\lambda y.y$ bez promjene značenja izraza,
- $\beta - \mathbf{redukcija}$ ujedno i najvažnije pravilo koje se odnosi na zamjenu varijable u tijelu funkcije s argumentom na koji se funkcija aplicira. Za potrebe primjera moguće je ponovno koristiti funkciju za inkrementaciju koja glasi ovako $(\lambda x.x+1)2$, ali β-redukcijom se reducira na $2+1$ što iznosi 3 [6]

4.2.3. Currying

Currying u funkcijskom programu predstavlja iteriranje aplikacije funkcije te omogućava funkcije s više argumenata.[4] Kada se priča o funkcijskom programiranju to zapravo predstavlja "odgađanje" apliciranja funkcije, a postiže se na način da funkcija koja prima argument vraća drugu funkciju koja također prima argument. Na taj način se također može postići znatno veća apstrakcija u funkcijskom programu. Primjer izraza izgleda ovako:

$$\lambda x.\lambda y.x+y$$

koji se zbog jednostavnosti prikaz prikazuje kao funkcija s više argumenata koji su odvojeni znakom ",".

$$\lambda x,y.x+y$$

Kako bi se aplicirali brojeve 1 i 2 na ovu funkciju to je moguće napraviti pomoću ovakvog izraza

$$\begin{aligned}F(12) &\rightarrow \lambda x,y.x+y(12)\\ &\rightarrow 1+2\\ &\rightarrow 3\end{aligned}$$

5. Tehnologije korištene za implementaciju praktičnog dijela

5.1. Erlang

Erlang je funkcijski programski jezik kojega su razvili Joe Armstrong, Robert Virding i Mike Williams u Ericssonovom laboratoriju za računarstvo 1986. godine. Zbog kompleksnih potreba u telekomunikacijskom sektoru došlo je do potrebe za programskim jezikom čije aplikacije moraju imati iznimno veliko vrijeme dostupnosti (engl. *uptime*) te istovremeno održavati veliki broj konekcija. Postao je vrlo popularan zbog konkurentnosti (engl. *concurrency*) i otpornosti na pogreške te se vrlo brzo počeo koristiti i u ostalim industrijama, a ne samo u telekomunikacijama.

Unutar jednog Erlang projekta svaki dokument sa ekstenzijom .erl zapravo predstavlja modul. Unutar svakog dokumenta na početku je potrebno navesti naziv modula koji mora biti isti kao i naziv dokumenta. Na module u Erlangu može se gledati kao na klase u OOP[1]. Funkcije koje se implementiraju unutar modula mogu biti javne ili privatne, ali ne na način kao u OOP već jednostavnim izvođenjem funkcija koje trebaju biti javne. Privatne funkcije se ne izvode van već ih javne koriste te ih se može nazvati internim funkcijama.

```
module(novi_modul).
export([...]).
```

5.1.1. Konkurentnost

Kao što je navedeno jedna od najzanimljivijih funkcionalnosti Erlanga je konkurentnost za koju je potrebno reći da je različita od paralelnosti iz razloga što se procesi ne odvijaju u isto vrijeme. Unutar Erlanga koristi se model laganih procesa koji su na razini jezika te oni mogu međusobno komunicirati pomoću razmjenjivanja poruka. Pošto se procesi pokreću unutar Erlang virtualnog računala (engl. *Erlang Virtual Machine, Erlang VM*) te rade u memoriji moguće ih je pokrenuti milijune unutar jedne instance čak i na računalu sa jednom jezgrom. Razlog tome je to što Erlang VM vremenski raspoređuje izvršavanje tih procesa tako da se samo čini da se oni paralelno izvršavaju.[8] Za kreiranje procesa koristi se naredba:

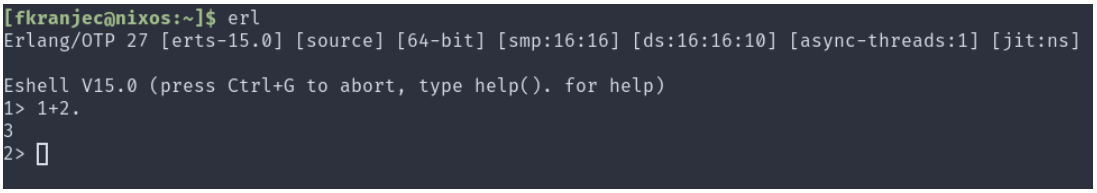
```
spawn(Module, Name, Args).
```

Funkcija *spawn* prima kao argument cijeli modul, naziv procesa te dodatne argumente. Kako je rečeno da se za modul može reći da je klasa isto tako za proces određenog modula može se reći da je to objekt ili instanca te klase koja komunicira putem poruka s ostalim procesima.[9]

[1] Objektno orijentirano programiranje

5.1.2. Erlang ljuska

Unutar Erlang-a većinu aplikacija moguće je testirati unutar emulatora koji se može pokrenuti pomoću naredbe *erl* unutar terminala. Nakon pokrenute Erlang ljuske moguće je unutra pisati funkcije ili učitavati potpune module te ih koristiti. Sintaksa za učitavanje modula je *c(modul)*..



Slika 5: Erlang ljuska (Izvor: Autorski rad)

Kao što je moguće vidjeti na slici 5 unutar ljuske je moguće izvršavati i aritmetičke operacije za koje se odmah dobije rezultat što također jako podsjeća na prolog unutar kojega također točka predstavlja terminator kojim se označava kraj funkcija.

5.1.3. Sintaksa

Sintaksa Erlang-a inspirirana je sintaksom prologa te je vrlo jednostavna. Kako bi se jednostavnije prikazala sintaksa prvo je potrebno navesti tipove podataka koji su ugrađeni u jezik kojih nema puno:

- **Brojevi** - podržani su i cijeli brojevi (engl. *integer*) i decimalni brojevi (engl. *float*) te također brojevi u drugim bazama pomoću izraza *Baza#Vrijednost*,
- **Invarijabilne varijable** - predstavljaju varijable kojima je moguće dodijeliti vrijednost. Varijable se uvijek moraju nazivati velikim početnim slovo zato što postoje atomi koji se definiraju malim početnim slovom,
- **Atomi** - kao što je navedeno definiraju se malim početnim slovom te prestavljaju konstante unutar koda koje se najčešće koriste uz podatke kako bi ih opisali,
- **Boolean** - kao i u većini jezika postoje true i false pomoću kojih se određuje istinitost nekog izraza,
- **Uređeni n-terac (engl. *tuple*)** - kao što sam naziv govori to je skup od n elemenata pomoću kojega se strukturiraju podaci. Tuple se može prepoznati prema sintaksi *{Element1, Element2, ..., ElementN}*,
- **Liste** - kao i u većini jezika i u Erlangu postoje liste, ali treba biti oprezan pošto se tu i stringovi prikazuju kao liste brojeva tako da je u nekim situacijama lagano napraviti pogrešku prilikom ispisivanja liste,
- **Bit sintaksa** - unutar Erlang-a je također moguće koristiti i binarne podatke te ima vrlo jednostavnu sintaksu kako upravljati njima. Binarni podaci se mogu prepoznati prema sintaksi *«"podatak"»* koja u ovom slučaju predstavlja binarni string.[8]

Za početak prikazati će se jednostavna funkcija za inkrementaciju:

```
inkrementiraj(X) ->
                X+1.
```

Naravno odmah je moguće prikazati i apstrakciju o kojoj se pričalo u prethodnom poglavlju te se može implementirati na ovaj način:

```
inkrementiraj(X) ->
                X+1.

inkrementiraj_broj_2() ->
                inkrementiraj(2).
```

Također jedna od bitnijih funkcionalnosti Erlang-a su zaštitari (engl. *guards*) te spajanje po uzorcima (engl. *pattern matching*). Pattern matching se najčešće koristi kada se funkcija treba drugačije izvršavati za drugačije slučajeve te se tu mogu dosta dobro iskoristiti atomi kako bi se prepoznalo koju funkciju izvršiti.

```
inkrementiraj(jedan, X) ->
                X+1;
inkrementiraj(dva, X) ->
                X+2;
inkrementiraj(tri, X) ->
                X+3.
```

Također je sličnu stvar moguće implementirati pomoću guard-ova, ali na malo drugačiji način koji bi izgledao ovako

```
inkrementiraj(A, X) when A =:= jedan ->
              X+1;
inkrementiraj(A, X) when A =:= dva ->
              X+2;
inkrementiraj(A, 3) when A =:= tri ->
              X+3.
```

Kod implementacije sa Guard-ovima može se vidjeti da se koristi puno više standardnog koda (engl. *boilerplate code*), ali postoje slučajevi u kojem je bolja za koristiti kao na primjer kada treba provjeriti ako je osoba punoljetna. Također kada se priča o pattern matching-u potrebno je napomenuti na koji način se liste mogu zapisivati.

```
prvi([H | _]) ->
           H.
drugi([_,X | _]) ->
           X.
```

Kao što se može vidjeti listu je moguće podijeliti na više dijelova te se varijable koje se ne koriste označavaju sa _. Do problema dolazi kada je potrebno vratiti zadnji element liste kada se ne zna koliko lista ima elemenata. Rješenje tome su rekurzije.

Unutar Erlang-a ne postoje petlje poput for i while kao ni u drugim čisto funkcijskim jezicima već se petlje implementiraju pomoću rekurzija. Rekurzije su funkcije koje pozivaju same sebe unutar tijela funkcije. Kako je već napomenuto da je sintaksa inspirirana prologom tako i u Erlang-u postoje sidra, što su funkcije koje određuju izlaz iz petlje i implementiraju se pomoću pattern matching-a. Znajući to ovako se može implementirati funkcija koja vraća zadnji element iz liste:

```
zadnji([X]) -> X;
zadnji([_ | T]) ->
            zadnji(T).
```

Koristeći rekurzije vrlo lagano može doći do opterećivanja memorije što se ne može vidjeti na prethodnom primjeru, ali vrlo lako se može objasniti na primjeru funkcije za određivanje duljine liste.[10]

```
duljina([]) -> 0;
duljina([_ | T]) ->
            1 + duljina(T).
```

Ako se preda funkciji lista s četiri elementa funkcija će za svaki korak dodati novo zbrajanje koje će se u memoriji izršavati na ovaj način:

$$\begin{aligned}
duljina([1,2,3,4]) &= duljina([1|[2,3,4]]) \\
&= 1 + duljina([2|[3,4]]) \\
&= 1 + 1 + duljina([3|[4]]) \\
&= 1 + 1 + 1 + duljina([4|[]]) \\
&= 1 + 1 + 1 + 1 + duljina([]) \\
&= 1 + 1 + 1 + 1 + 0 \\
&= 1 + 1 + 1 + 1 \\
&= 1 + 1 + 2 \\
&= 1 + 3 \\
&= 4
\end{aligned}$$

Kod manjih listi ovako nešto je dovoljno dobro, ali kada dođu velike liste dolazi do problema zato što bi se čuvalo jako puno brojeva u memoriji za vrlo jednostavan izračun. Kako bi se taj problem riješio postoje repne rekurzije (engl. *tail recursions*) gdje se uvodi dodatna varijabla akumulator unutar koje se čuva nova vrijednost i prenosi u sljedeći korak rekurzije. Algoritam za dohvaćanje duljine liste koristeći tail rekurziju može se implementirati na sljedeći način:

```
duljina(L) -> duljina(L, 0).

duljina([], Acc) -> Acc;
duljina([_ | T], Acc) ->
            duljina(T, Acc + 1).
```

Funkciji je potrebno proslijediti akumulator tako da je implementirana apstrakcija koja prima samo listu te originalnoj funkciji prosljeđuje listu i postavlja vrijednost akumulatora na 0. Na ovaj način u svakom koraku će se unutar memorije nalaziti samo jedan izraz (engl. *term*). [10]

5.1.4. Mnesia

Mnesia je distribuirani sustav za upravljanje bazom podataka posebno implementiran za korištenje unutar Erlang-a. Kako su potrebe aplikacija postajale sve veće i zahtjevale sve veću dostupnost i otpornost na kvarove razvijena je kako bi se zadovoljile takve potrebe. Implementirana je kao ključ-vrijednost baza podataka što znači da pripada u skupinu nerelacijskih (engl. *noSQL*) baza podataka.[11]

Kako bi se kreirale tablice prvo je potrebno razumijeti koncept zapisa (engl. record) u Erlang-u pošto se Mnesia temelji na zapisima. Zapisi su jedna od struktura koje se koriste za razvoj aplikacija u Erlang-u.[10] Sintaksa definiranja zapisa izgleda ovako:

```
-record(osoba, {ime, prezime, prijatelji=[]}).
```

Ovim zapisaom definira se zapis osoba koja sadrži ime, prezime i listu prijatelja. Kako Mnesia nije sigurna na tipove (engl. *type safe*) atributi nemaju specificirane tipove već je moguće staviti bilo koji tip te je potrebno s oprežnošću implementirati bazu podataka. Zapisi se također mogu koristiti i kao ulazni argumenti i također se može raditi pattern matching nad više zapisa što omogućuje dodatnu jednostavnost implementacije složenijih aplikacija.

Zapisi se pretvaraju u tablice pomoću funkcije koju nudi Mnesia i zove se *create_table*. Unutar te funkcije moguće je odrediti koja će se vrsta tablice koristiti. Mnesia ima 4 vrste tablica, a to su:

- **set** - tablica koja omogućuje da svaka prva vrijednost u zapisu mora biti jedinstvena. U slučaju zapisa osobe to je ime zato što je prvo navedeno,
- **ordered_set** - isto funkcionira kao i set samo što su elementi poredani prema veličini,
- **bag** - to je tablica unutar koje može postojati više ključeva s istom vrijednošću sve dok ne postoji zapis koji ima sve vrijednosti jednake. Tako je na primjer moguće zapisati 1,1 i 1,2, ali nije moguće ponovno zapisati vrijednost 1,1.
- **duplicate_bag** - isto kao i bag samo što dodatno omogućuje potpuno iste zapise da budu više puta zapisani

Također je moguće postaviti polja koja se indeksiraju, ali bitno je znati da se prvo navedeno polje uvijek indeksira tako da ga nije potrebno navoditi.[11] Uz indeksiranje potrebno je odrediti kako će se zapisi čuvati. Mnesia ima dva načina čuvanja podataka, a to je na disku ili u memoriji, a opcije koje je moguće koristiti su:

- **ram_copies** - podaci se čuvaju samo u memoriji,
- **disc_only_copies** - podaci se čuvaju samo na disku,
- **disc_copies** - podaci se čuvaju i na disku i u memoriji.[11]

Odabirom postavki za tablicu potrebno je sve proslijediti u funkciju te dodatno postaviti čvor na kojem se kreira shema i čuvaju podaci.

```
mnesia:create_table(osoba,
                    [{attributes, record_info(fields, osoba)},
                    {index, []},
                    {disc_copies, node()},
                    {type, set}]).
```

Kako bi se u bazu podataka zapisali podatci ili se dohvatili postoje funkcije koje to dopuštaj. Neke od tih funkcija se moraju izvršavati unutar transakcija. Transakcije nude sigurnost da će tablice nakon izvršavanja operacija nad njima ostati u konzistentnom stanju.[10] Koristeći transakcije moguće je izvršiti više operacija nad bazom podataka unutar jedne anonimne funkcije te ako samo jedna od tih operacija ne uspije cijela transakcija će vratiti pogrešku. Kako bi se zapisala vrijednost u bazu podataka koristeći transakciju potrebno je napraviti ovakvu implementaciju:

```
dodaj_osobu(Ime, Prezime) ->
    Fun = fun() ->
            mnesia:write(#osoba{ime=Ime,
                                prezime=Prezime,
                                prijatelji=[]})
          end,
    mnesia:transaction(Fun).
```

Podaci se mogu dohvaćati na više način, ali najpoznatiji su: *mnesia:read/1* i *mnesia:select/2*. Jednostavniji je *mnesia:read* kojemu je potrebno proslijediti tuple koji se sastoji od naziva tablice i vrijednost po kojoj se pretražuje, ali da je postavljena prilikom kreiranja tablice kao indeks.

```
dohvati_osobu(Ime) ->
    Fun = fun() ->
            mnesia:read({osoba, Ime})
          end,
    mnesia:transaction(Fun).
```

Problem kod *mnesia:read* je to što se može pretraživati samo po indeksiranim poljima, ali *mnesia:select* omogućuje dohvaćanje prema specifikaciji sparivanja (engl. *match specification*) te vraća listu svih elemenata koji zadovoljavaju specifikaciju. Specifikacije je malo teže za

razumjeti, ali postoji funkcija unutar Erlang-a koja omogućuje pretvaranje funkcija u specifikaciju što olakšava implementaciju.

```
dohvati_osobu_prema_prezimenu(Prezime) ->
    Fun = fun() ->
            Match = ets:fun2ms(fun(#osoba{ime=I,prezime=P,
                prijatelji=Pr}) when P =:= Prezime->
                                        {I,P,Pr}
                                end),

            mnesia:select(osoba, Match)
          end,
    mnesia:transaction(Fun).
```

Operacija brisanja podataka prima potpuno isti argument kao i funkcija čitanja, a može se implementirati na ovaj način:

```
obrisi_osobu(Ime) ->
    Fun = fun() ->
            mnesia:delete({osoba, Ime})
          end,
    mnesia:transaction(Fun).
```

5.1.5. Cowboy

Cowboy je minimalistički, ali moćan HTTP poslužitelj koji se koristi za implementaciju aplikacijskih programskih sučelja u Erlang-u. Dizajn ovog okvira se fokusira na modularnost i visoke performanse te omogućuje obrađivanje velikog broja istovremenih zahtjeva.[12]

Ključne komponente okvira Cowboy su:

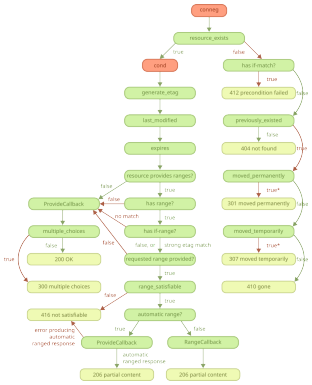
- **Usmjerivač (engl. Router)** - odgovoran je za usmjeravanje zahtjeva koji dolaze od klijenata na upravljače koji su zaduženi za upravljanje na toj putanji,
- **Upravljači (engl. Handlers)** - moduli koji obrađuju HTTP zahtjeve te se sastoje od funkcija koje određuju kako se koja od HTTP metoda obrađuje.
- **Petljajući upravljači (engl. Loop handlers)** - koriste se za implementaciju mrežnih utičnica (engl. WebSocket) pomoću kojih se podržavaju dugotrajne veze između klijenta i HTTP poslužitelja,
- **Srednji sloj (engl. Middleware)** - služi za obradu zahtjeva prije nego što stignu do upravljača koji je zadužen za taj zahtjev.[12]

Prilikom implementacije HTTP poslužitelja prvo je potrebno implementirati usmjerivač te postaviti putanje na kojima će biti dostupni upravljači.

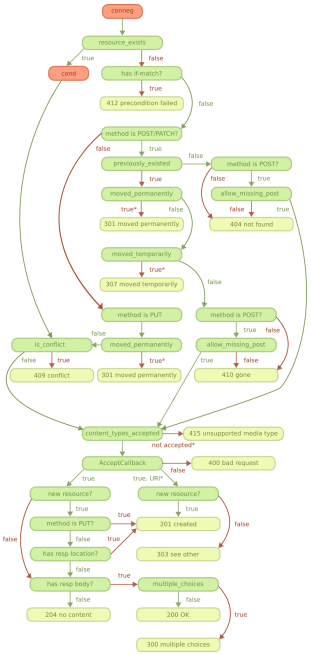
Unutar usmjerivača specificirano je ime za upravljač tako da je potrebno kreirati modul pod tim imenom kako bi Cowboy koristio taj modul. Unutar upravljača potrebno je implementirati

```
usmjerivac() ->
    cowboy_router:compile([{'_',[
        {"/", pozdrav, []}
    ]}]).
```

funkcije koje Cowboy redoslijedom poziva prilikom svakog zahtjeva koji pristigne. Ovisno koji zahtjev pristigne upravljač poziva funkcije koje se izvršavaju specifični redoslijedom.



Slika 6: Funkcije i redoslijed kod GET i HEAD zahtjeva (Izvor:Cowboy user Guide)



Slika 7: Funkcije i redoslijed kod POST, PUT i PATCH zahtjeva (Izvor:Cowboy user Guide)

Najosnovniju strukturu upravljača moguće je implementirati na sljedeći način:

```
-module(pozdrav).
-export([init/2]).

init(Req, State) ->
    {ok, Reply} = cowboy_req:reply(200,
                            #{<<"content-type">> => <<"text/plain">> },
                            <<"Pozdrav␣Svijete!">>, Req),
    {stop, Reply, State}.
```

Nakon što je implementiran upravljač potrebno je implementirati i funkciju za pokretanje servera unutar koje se specificira na kojem portu će se HTTP poslužitelj pokretati te koji se usmjerivač korist na tom poslužitelju.

```
start() ->
    {ok, _} = cowboy:start_http(http_slusac, 100,
            #{port => 5000},
            #{env => #{dispatch => usmjerivac()}}).
```

5.2. WebAssembly

U današnje vrijeme aplikacije na internetu postaju sve kompleksnije i zahtjevaju sve više resursa. Razlog tome je popularnost novih okvira za implementaciju web aplikacija koji se orijentiraju prema prikazivanja na strani klijenta (engl. *client side rendering*) pomoću kojega se koriste resursi klijenta, a ne poslužitelja kao kod prikazivanja na strani poslužitelja (engl. *server side rendering*). Kao rješenje tih problema kreiran je WebAssembly 2017. godine od strane W3C zajednica kao otvoreni standard.

WebAssembly se fokusira na visoke performanse, sigurnost i prenosivost. Pošto je WebAssembly kod niske razine vrlo je blizak strojnom kodu te se unutar preglednika izvršava u gotovo nativnim performansama.[13] Također je bitno napomenuti da se WebAssembly kod pokreće u sigurnom okruženju unutar preglednika te je izoliran od aplikacije što omogućuje dodatnu sigurnost prilikom izvođenja aplikacija, ali isto tako moguće ga je pokrenuti na svim modernim preglednicima neovisno o platformi te čak postoje mogućnosti pokretanja unutar nativnih mobilnih aplikacija.[14]

5.2.1. Razvoj WebAssembly aplikacija

WebAssembly nije namjenjen da se piše ručno već da se prevodi iz jezika poput C, C++ ili Rust-a. Na taj način se programerima olakšava razvoj takvih aplikacija i čitljivost koda pošto je strojni kod vrlo teško za održavati i testirati. Nakon pisanja koda u nekom od navedenih jezika potrebno je prevesti kod u strojni kod pomoću alata ovisno od programskoj jezika u kojem je napisan. Tako se za prevođenje iz C ili C++ koristi Emscripten te za prevođenje iz Rust-a se koristi wasm-pack. Pokretanjem tih alata kod se prevodi u binarnu datoteku sa ekstenzijom

.wasm te ga je moguće koristiti unutar web aplikacija na način da učita u aplikaciju te instancira WebAssembly koji je nativni dio preglednika.[15]

5.2.2. Poznate primjene WebAssemblya

Neke od napoznatijih primjena WebAssemblya u današnje vrijeme su većinom za aplikacije koje se prije nisu mogle pokretati u preglednici kao na primjer:

- **Uređivanje slika** - aplikacije poput Photoshop-a koriste WebAssembly kako bi ostavarile bolje performanse unutar preglednika, ali i iz razloga zato što implementiranjem funkcionalnosti u nativnoj aplikaciji na jednostavan način mogu iste funkcionalnosti prenesti na web samo prevođenjem koda,
- **Igrice** - zbog svojih performansi WebAssembly omogućuje sve naprednije igrice unutar preglednika, ali isto tako dopušta implementiranje igrica pomoću nativnog OpenGL-a umjesto korištenja WebGL-a,
- **Video konferencije** - unutar video poziva postoje kompleksnije funkcionalnosti koje se implementiraju kao na primjer dodavanje filtera na video ili onemogućavanje pozadinskih zvukova.

5.2.3. Rust

Rust je moderan programski jezik stvoren od stane Mozzila Researcha 2010. godine. Kreiran je sa fokusom na sigurnost i performanse te je postao popularan zbog rješavanja velikog broja pogrešaka do kojih dolazi u jezicima poput C i C++.[16] Većina tih problema su problem s upravljanjem memorijom kao na primjer segfaults, buffer overflows ili use-after-free. Rust je sve te probleme riješio pomoću nekoliko inovativnih koncepata:

- **Sigurno upravljanje memorijom** - jedan od najvažnijih koncepata jest sustav vlasništva koji osigurava sigurnost upravljanja memorijom. Uz kombinaciju s provjerama tijekom prevođenja, greške poput dvostrukog oslobađanja memorije ne mogu se dogoditi,
- **Neizmjenjivost (engl. Immutability)** - unutar Rust-a nije moguće direktno izmijeniti niti proslijediti varijablu već je potrebno koristiti njihove reference osim ako nije direktno navedeno da je dopušteno,
- **Cargo** - upravitelj paketa i alata za izgradnju aplikacija pomoću kojega je vrlo jednostavno dodati novi paket unutar projekta te postoji repozitorij sa svim paketima koje je moguće koristiti,
- **Sustav za tipiziranje** - Rust koristi statički sustav za tipiziranje što mu uz neke od naprednijih značajki poput pattern matching-a omogućuje pisanje fleksibilnog koda.[16]

Rust ima dosta jednostavnu sintaksu koja podsjeća na TypeScript i na C++ u nekim situacijama. Varijable se unutar Rust-a mogu deklarirati kao promjenjive ili nepromjenjive te im je moguće dodijeliti tip.

```
let x = 1;
let mut y = 2;
let z:i32 = 3;
```

U navedenom primjeru moguće je vidjeti da se pomoću naredbe *mut* dopušta varijabli da ju je moguće mijenjati, a pomoću sintakse *varijabla:tip = vrijednost* moguće je dodijeliti tip varijabli.[17] Funkcije se u Rust-u zapisuju na sljedeći način:

```
pub fn zbroji(x: i32, y: i32) -> i32{
    a + b
}
```

Iz primjera se može vidjeti da je za svaki argument potrebno postaviti tip te također pomoću $\rightarrow$ se postavlja tip podataka koji funkcija vraća. Bitno je napomenuti da unutar funkcija u Rust-u nije potrebno pisati ključnu riječ *return* već se vraća posljednja naredba koja na kraju nema *;*.

Kao što je navedeno vlasništvo je najbitniji koncept u Rust-u te posuđivanje vrijednosti koje se može postići pomoću oznaka *&* i *&mut*.[17]

```
fn main() {
    let x = 1;
    let y = &x;
    println!("{}", y);
    let z = x;
    println!("{}", x);
}
```

Pomoću prethodnog primjera jednostavno je objasniti posuđivanje varijabli zato što je u primjeru prva naredba za ispisivanje dopuštena pošto je u varijablu y posuđena vrijednost varijable x, ali druga naredba za ispisivanje nije dopuštena zato što se vlasništvo s varijable x prenijelo na varijablu z.

5.2.3.1. Reqwest i wasm-bindgen

Biblioteke su glavni dio Rust razvojnog okruženja te služe kako bi se na jednostavniji način implementirale neke od funkcionalnosti. Svaka biblioteka ima svoju ulogu tako i reqwest koji služi kao REST klijent pomoću kojega se dohvaćaju podaci s REST poslužitelja. Dizajniran je da bude što jednostavniji za korištenje i da podržava sinkrono i asinkrono programiranje.[18]

```
fn dohvati() -> Result<(), reqwest::Error> {
    let odgovor = reqwest::get("http://localhost:5000/putanja")
        .await.unwrap().text().unwrap();
    Ok(odgovor)
}
```

Primjer prikazuje jednostavnu funkciju koja dohvaća podatke s putanje koju REST pružatelj mora nuditi putem usmjerivača. Funkcija vraća odgovor od pružatelja koji je u većini

slučajeva neki tip podataka koji se zatim mora obraditi.

Wasm-bindgen je biblioteka koja olakšava interoperabilnost između Rust koda prevedenog u WebAssembly i JavaScript-a. Unutar nje se većinom nalaze dekoratori pomoću kojih se specificira koje se funkcije ili tipovi prevode u WebAssembly.[19]

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
fn dohvati() -> Result<(), reqwest::Error> {
    let odgovor = reqwest::get("http://localhost:5000/putanja")
        .await.unwrap().text().unwrap();
    Ok(odgovor)
}
```

Dodavanjem dekoratora omogućilo se prevođenje prethodne funkcije u WebAssembly te korištenje unutar JavaScript koda čime se sva logika dohvaćanja podataka prebacila na jedno mjesto i izdvojila od samog projekta web aplikacije.

5.3. Angular

U svijetu razvoja web aplikacija svakim danom se pojavljuju novi okviri pomoću kojih se implementiraju aplikacije. Angular je jedan od okvira koji je uspio opstati zbog konstantnih nadogradnji koje Google-ov tim radi na njemu. Prvi put je predstavljen kao AngularJS 2010. godine, ali je potpuno obnovljen 2016. godine te nadograđivan sve do današnje verzije 18 unutar koje su uvedeni funkcijske paradigme programiranja. Pomoću Angulara moguće je napraviti složene, skalabilne i modularne aplikacije koje je jednostavno za održavati.[20]

Angular se fokusira na razvoj komponenti koje je moguće ponovno koristiti te olakšati razvoj aplikacija na način da se smanjuje opseg programiranja što je aplikacija veća.[20] U prethodnim verzijama koristili su se moduli kako bi svaka komponenta bila odvojena i mogla se koristiti u drugim komponentama, ali današnji Angular koristi samostalne komponente (engl. *standalone components*) koje izbacuju module pritom zadržavajući modularni razvoj pošto se svaka samostalna komponenta može koristiti u drugim komponentama neovisno jesu li one samostalne ili ne.

Orijentacija na komponente je okviru otvorila mogućnost za obostranim vezama podataka (engl. *Two-Way Data Binding*) što omogućuje automatiziranu promjenu podataka unutar modela svakom akcijom napravljenom na sučelju te dodatno u roditeljskim komponentama. Jedna od velikih prednosti Angular-a je injektiranje zavisnost (engl. *dependency injection*) koje u kombinaciji sa servisima omogućuju međusobnu komunikaciju komponenti koje nisu povezane.[20] Time se izbacuju iz priče biblioteke poput NgRx koje pružaju reaktivno upravljanje stanjima iako se u nekim aplikacijama još uvijek koriste.

Za razliku od drugih okvira Angular olakšava strukturu projekata tako što je moguće pomoću jednostavnih naredbi koje se nalaze u Angular razvojnom okruženju kreirati nove komponente, servise, zaštitare, cjevovode i presretače. Također je vrlo jednostavno implementirati

biblioteke unutar Angular-a pošto nudi mogućnost generiranja projekta u obliku biblioteke koju je moguće objaviti na centralni repozitorij biblioteka npm.

Neke od najbitnijih gradivnih jedinica Angular-a su:

- **Komponente** - unutar komponenti programeri implementiraju izgled same komponente pomoću HTML-a i CSS-a te akcije koje se događaju unutar te komponente poput klika na gumb ili pisanja u polje za tekst pomoću TypeScript-a,
- **Servisi** - služe za implementiranje poslovne logike te komuniciranje između komponenti koje je neophodno kako bi aplikacija funkcionirala te se razvila modularnost prilikom implementacije. Jedan primjer toga bi bio servis za upravljanje prijavljenim korisnikom. Kao što je navedeno servise je moguće injektirati u komponente što znači da svaka komponenta u aplikaciji u svakom trenutku može znati koji korisnik je prijavljen.
- **Usmjerivač** - slično kao i kod usmjerivača kod Cowboy-a i unutar Angular-a se koristi kako bi postavio koja komponenta se prikazuje na kojoj putanji. Usmjerivač u Angular-u je malo napredniji pošto može imati višestruko ugniježdene putanje te se može kreirati nekoliko navigacija unutar samo jedne stranice.[20]

Za implementaciju unutar samih komponenti i servisa koristi se TypeScript koji je idealan pošto koristi tipove podataka koji ne postoje u JavaScript-u. Također zbog TypeScript-a moguće je koristiti dinamičko programiranje što omoguće kreiranje apstrakcija za komponente i algoritme tako da mogu raditi nad većim broj podataka što još dodatno povećava modularnost.

Također je bitno napomenuti da unutar Angular-a postoji mogućnost implementiranja progresivnih web aplikacija (engl. *PWA*) koje omogućuju dodatne funkcionalnosti poput notifikacija korisniku aplikacije o novijim verzijama, komunikaciju između više otvorenih prozora jedne aplikacije u pregledniku te dodatno početno učitavanje podataka takozvana ljuska aplikacije (engl. *app shell*).[20]

Najčešće primjene Angulara moguće je vidjeti u aplikacijama unutar poduzeća zato što je moguće izolirati poslovnu logiku od dizajna i strukture aplikacije što olakšava razvoj unutar poduzeća, e-trgovine unutar kojih je potrebna reaktivnost te dinamičke promjene modela te aplikacije unutar kojih je potrebno ažuriranje podataka u stvarnom vremenu poput aplikacija za dopisivanje.

6. Implementacija klijent-poslužitelj arhitekture

U svrhu izrade praktičnog dijela implementirana je arhitektura klijent-poslužitelj koristeći Mnesi-u kao bazu podataka, Cowboy kao aplikacijsko programsko sučelje i Angular za implementaciju klijentske aplikacije. Jedna specifična stvar unutar ove implementacije je to što je sva komunikacija klijenta s poslužiteljem implementirana pomoću programskog jezika Rust, a zatim preveden u WebAssembly. Na taj način implementiran je unificirani sustav za dohvaćanje podataka koji je moguće koristiti na velikom broj projekata i to ne samo web aplikacijama već i mobilnima. Ne samo da je cijeli dio dohvaćanja podataka implementiran već i svi mogući tipovi koje poslužitelj može vratiti što još dodatno pojednostavljuje implementacije.

Baza podataka i HTTP poslužitelj implementirani su koristeći Erlang što znači da je sve implementirano funkcijskom paradigmom programiranja. Funkcijska paradigma programiranja također se koristi i u Rust projektu i u Angular projektu zato što oba jezika podržavaju funkcijsko programiranje.

Svrha aplikacije jest platforma za e-učenje koja je povezana s fakultetima. Unutar platforme nalaze se fakulteti koji se sastoje od katedri, a katedre se sastoje od kolegija. Unutar kolegija postoje sekcije koje se pune sadržajem. Ideja je da sekcije imaju polimorfni sadržaj koji može biti poveznica, dokument ili lekcija. Svakom fakultetu pridruženi su korisnici, dok su katedrama pridruženi djelatnici uz tip djelatnika što omoguće distinkciju između voditelja katedre i djelatnika na katedri. Unutar kolegija također se dodjeljuju djelatnici uz tip djelatnika te se dodjeljuju studenti koji su upisani na taj kolegij. Djelatnici i Studenti su svi zapisani unutar iste tablice, ali ih se može raspoznati prema atributu uloga.

6.1. Implementacija baze podataka

Unutar baze podataka kreirane su "glavne" tablice: *Fakultet*, *Katedra*, *Kolegij*, *Korisnik*, *Sekcija*, *Sadrzaj*. Kako bi se zapravo uspostavila poslovna logika sustava kreirane su pomoćne ili tako zvane spajajuće (engl. *join*) tablice unutar kojih se zapisuju ključevi dviju ili više tablica koje spajaju. Svaka od tih tablica kreirana je pomoću zapisa unutar Erlang-a:

```
-record(db_fakultet,
        {id :: id(),
         logo :: binary(),
         naziv :: binary(),
         skraceno :: binary(),
         adresa :: adresa(),
         opis :: binary(),
         lokacija :: {number(), number()}}).
-record(adresa,
        {grad :: binary(),
         ulica :: binary(),
         postanski_broj :: number(),
         drzava :: binary(),
         kucni_broj :: binary()}).
-record(db_fakultet_korisnik,
```

```
        {id_korisnik :: korisnik_ref(), id_fakultet :: fakultet_ref()}).
-record(db_fakultet_katedra,
        {id_katedra :: katedra_ref(), id_fakultet :: fakultet_ref()}).
-record(db_katedra, {id :: id(), naziv :: binary(), opis :: binary()}).
-record(db_katedra_djelatnik,
        {id_katedra :: katedra_ref(), id_djelatnik :: korisnik_ref(), tip ::
            tip_djelatnika()}).
-record(db_katedra_kolegij, {id_kolegij :: kolegij_ref(), id_katedra :: katedra_ref
    ()}).
-record(db_korisnik,
        {id :: id(),
         ime :: binary(),
         prezime :: binary(),
         oib :: integer(),
         slika :: binary(),
         lozinka :: lozinka(),
         uloga :: student | profesor | dekan | asistent,
         email :: binary(),
         opis :: binary(),
         dodatno :: student() | djelatnik()}).
-record(student, {nadimak :: binary()}).
-record(djelatnik, {kabinet :: binary(), vrijeme_konzultacija = [] :: [datum_vrijeme
    ()]}).
-record(db_djelatnik_kolegij,
        {id :: {korisnik_ref(), kolegij_ref()}, status :: status_djelatnika()}).
-record(db_student_kolegij, {id :: {korisnik_ref(), kolegij_ref()}, ocjene :: [{}]})
    .
-record(db_kolegij,
        {id :: id(), slika :: binary(), naziv :: binary(), skraceno :: binary()}).
-record(db_kolegij_sekcija, {id_kolegij :: kolegij_ref(), id_sekcija :: sekcija_ref
    ()}).
-record(db_sekcija,
        {id :: id(), naziv :: binary(), opis :: binary(), vidljivo :: boolean()}).
-record(db_sekcija_sadrzaj, {id_sadrzaj :: sadrzaj_ref(), id_sekcija :: sekcija_ref
    ()}).
-record(db_sadrzaj,
        {id :: id(),
         naziv :: binary(),
         tip :: dokument | lekcija | poveznica | kviz,
         vrijednost :: dokument() | lekcija() | poveznica() | kviz_ref()}).
-record(dokument, {referenca :: binary(), vrijeme_kreiranja :: datum_vrijeme()}).
-record(lekcija,
        {sadrzaj :: binary(), slika :: binary(), vrijeme_kreiranja :: datum_vrijeme
            ()}).
-record(poveznica, {referenca :: binary(), vrijeme_kreiranja :: datum_vrijeme()}).
```

Također prednost spajajućih tablica je u tome što se zapravo unutar njih nalazi cijela poslovna logika aplikacije pošto je u njima zapisano na primjer koji student je na kojem kolegiju, ili koje sekcije pripadaju kojem kolegij. Svaka od tih tablica kreirana je drugačije zato što u nekim slučajevima kao student na fakultetu potrebno je ograničiti da student može biti samo na jednom fakultetu dok na primjer student može sudjelovati na više kolegija i na svakom od njih ima svoje ocjene. To je postignuto pravilnim odabirom tipa tablice set ili bag te po potrebi

kreiranjem kombiniranog ključa.

```
    create_table(db_katedra, set, Nodes, record_info(fields, db_katedra), []),
    create_table(db_katedra_kolegij, set, Nodes, record_info(fields,
        db_katedra_kolegij), []),
    create_table(db_katedra_djelatnik,
                bag,
                Nodes,
                record_info(fields, db_katedra_djelatnik),
                [#db_katedra_djelatnik.id_djelatnik])
```

Unutar istog projekta implementirane su operacije za čitanje, zapisivanje, uređivanje i brisanje podataka unutar svake tablice, ali postoji problem kod čitanja podataka. Koristeći spajajuće tablice unutar glavnih tablica ne postoje zapisi iz drugih tablica kao na primjer čitanjem iz tablice kolegija ne čitaju se sekcije pošto su te dvije tablice spojene putem spajajuće tablice. Rješenje tom problemu je da se unutar svakog modula tablica implementira učitavanje s razinama prikaza. Na taj način zapravo aplikacija koja koristi tu funkciju odlučuje kakav prikaz želi.

```
-module(kolegij).

-include_lib("database/include/records.hrl").
-include_lib("stdlib/include/ms_transform.hrl").

-export([dodaj/3, dohvati/1, dohvati/2, dohvati/0, obrisi/1, uredi/4]).

dohvati() ->
    Fun = fun(Kolegij, Acc) -> [ucitaj(core, Kolegij) | Acc] end,
    mnesia:transaction(fun() -> mnesia:foldl(Fun, [], db_kolegij) end).

dohvati(Id) ->
    Fun = fun() ->
              case mnesia:read({db_kolegij, Id}) of
                  [Kolegij] -> ucitaj(full, Kolegij);
                  [] -> {error, "Kolegij ne postoji"}
              end
          end,
    mnesia:transaction(Fun).

dohvati(Type, Id) ->
    Fun = fun() ->
              case mnesia:read({db_kolegij, Id}) of
                  [Kolegij] -> ucitaj(Type, Kolegij);
                  [] -> {error, "Kolegij ne postoji"}
              end
          end,
    mnesia:transaction(Fun).

dodaj(Naziv, Skraceno, Slika) ->
    Id = ?ID,
    Fun = fun() ->
              case mnesia:write(#db_kolegij{id = Id,
                                            naziv = Naziv,
```

```
                                        slika = Slika,
                                        skraceno = Skraceno})
            of
                ok -> {ok, Id};
                _ -> {error, "Transakcija␣šneuspjena"}
            end
        end,
    mnesia:transaction(Fun).

obrisi(Id) ->
    Fun = fun() ->
            djelatnik_kolegij:obrisi_kolegij(Id),
            student_kolegij:obrisi_kolegij(Id),
            kolegij_sekcija:obrisi_kolegij(Id),
            mnesia:delete({db_kolegij, Id})
        end,
    mnesia:transaction(Fun).

uredi(Id, Naziv, Skraceno, Slika) ->
    Fun = fun() ->
            case mnesia:write(#db_kolegij{id = Id,
                                        naziv = Naziv,
                                        slika = Slika,
                                        skraceno = Skraceno})
            of
                ok -> {ok, Id};
                _ -> {error, "Transakcija␣šneuspjena"}
            end
        end,
    mnesia:transaction(Fun).

ucitaj(core, R) ->
    transform_kolegij(R);
ucitaj(sekcije, R) ->
    M0 = transform_kolegij(R),
    kolegij_sekcija:ucitaj_sekcije(M0);
ucitaj(djelatnici, R) ->
    M0 = transform_kolegij(R),
    djelatnik_kolegij:ucitaj_djelatnike(M0);
ucitaj(studenti, R) ->
    M0 = transform_kolegij(R),
    student_kolegij:ucitaj_studente(M0);
ucitaj(full, R) ->
    M0 = transform_kolegij(R),
    M1 = kolegij_sekcija:ucitaj_sekcije(M0),
    M2 = djelatnik_kolegij:ucitaj_djelatnike(M1),
    student_kolegij:ucitaj_studente(M2).

transform_kolegij(#db_kolegij{id = Id,
                              naziv = Naziv,
                              slika = Slika,
                              skraceno = Skraceno}) ->
    #{id => Id,
```

```
      slika => Slika,
      naziv => Naziv,
      skraceno => Skraceno}.
```

Pisanje u spajajuće tablice ostaje isto kao i kod glavnih tablica, ali unutar ovih tablica se čitanje odvija drugačije. Pošto su to u većini slučajeva samo dva ključa koji spajaju tablice potrebno je na primjer pročitati sve korisnike koji su na nekom fakultetu ili pročitati sve djelatnike na katedri i sve katedre djelatnika. Iz tog razloga se tu koriste dohvaćanja iz modula od glavnih tablica. Ovakva implementacija potencijalno može doći do beskonačne petlje dohvaćanja podataka, ali zato je pažljivo napravljeno da se to ne bi dogodilo.

```
dodaj_korisnika_na_fakultet(IdKorisnik, IdFakultet) ->
    Fun = fun() ->
              mnesia:write(#db_fakultet_korisnik{id_korisnik = IdKorisnik,
                  id_fakultet = IdFakultet})
          end,
    mnesia:transaction(Fun).

dohvati_korisnike(IdFakultet) ->
    Fun = fun() ->
              Match =
                  ets:fun2ms(fun(#db_fakultet_korisnik{id_fakultet = Fakultet,
                                                       id_korisnik = Korisnik})
                                 when Fakultet =:= IdFakultet ->
                                 Korisnik
                             end),
              case mnesia:select(db_fakultet_korisnik, Match) of
                  K ->
                      Korisnici =
                          lists:map(fun(Korisnik) ->
                                        {atomic, Result} = korisnik:dohvati_korisnika(
                                            core, Korisnik),
                                        Result
                                    end,
                                    K),
                      Korisnici;
                  _ -> []
              end
          end,
    mnesia:transaction(Fun).

dohvati_fakultet(IdKorisnik) ->
    Fun = fun() ->
              case mnesia:read({db_fakultet_korisnik, IdKorisnik}) of
                  [Obj] ->
                      {atomic, Fakultet} =
                          fakultet:dohvati(core, Obj#db_fakultet_korisnik.id_fakultet
                              ),
                      Fakultet;
                  _ -> undefined
              end
          end,
```

```
    mnesia:transaction(Fun).
```

Isto tako funkcije za učitavanje funkcioniraju drugačije nego kod glavnih tablica pošto se unutar ovih samo nadodaje podatke na već postojeću mapu i to se vraća nazad kako bi se unutar glavnih tablica mogla vratiti nova mapa s nadodanim poljima.

```
ucitaj_korisnike(#{id := Id} = M) ->
    {atomic, Korisnici} = dohvati_korisnike(Id),
    M#{korisnici => Korisnici}.

ucitaj_fakultet(#{id := Id} = M) ->
    {atomic, Fakultet} = dohvati_fakultet(Id),
    case Fakultet of
        undefined ->
            M;
        _ ->
            M#{fakultet => Fakultet}
    end.
```

Kako bi baza podataka konstantno mogla biti napunjena prilikom testiranja i brisanja podataka kreiran je mock modul pomoću kojeg je moguće ponovno napuniti bazu ako se prilikom razvoja obriše.

```
    ucitaj_mock() ->
            ucitaj(fakulteti()),
            ucitaj(katedre()),
            ucitaj(kolegiji()),
            ucitaj(sekcije()),
            ucitaj(sadrzaj()),
            ucitaj_korisnike(),
            ucitaj(fakultet_katedra_keys()),
            ucitaj(katedra_kolegij_keys()),
            ucitaj(kolegij_student_keys()),
            ucitaj(kolegij_djelatnik_keys()),
            ucitaj(fakultet_korisnik_keys()),
            ucitaj(katedra_djelatnik_keys()),
            ucitaj(kolegij_sekcija_keys()),
            ucitaj(sekcija_sadrzaj_keys()).

    ucitaj(L) ->
            lists:foreach(fun(E) -> mnesia:dirty_write(E) end, L).

    fakultet_katedra_keys() ->
            [#db_fakultet_katedra{id_katedra = 1, id_fakultet = 1},
             #db_fakultet_katedra{id_katedra = 2, id_fakultet = 1},
             #db_fakultet_katedra{id_katedra = 3, id_fakultet = 1},
             #db_fakultet_katedra{id_katedra = 4, id_fakultet = 1},
             #db_fakultet_katedra{id_katedra = 5, id_fakultet = 1}].

    fakulteti() ->
            [#db_fakultet{id = 1,
                          naziv = <<"Fakultet␣Organizacije␣i␣Informatike">>,
                          opis =
```

```
                                <<"SUZG␣FOI␣djeluje␣60␣godina,␣što␣je␣za␣
                                    studij␣suvremenih␣tehnologija␣"
                                  "dugo␣razdoblje.␣Tijekom␣tog␣razdoblja␣
                                     Fakultet␣obrazuje␣čnajstrunije␣"
                                  "kadrove␣u␣čpodruju␣informacijskih␣znanosti␣
                                     (informatike)␣i␣"
                                  "informacijskih␣tehnologija,␣kao␣i␣ekonomije
                                     ,␣organizacije,␣"
                                  "komunikologije␣i␣drugih␣srodnih␣čpodruja."/
                                     utf8>>,
                        skraceno = <<"FOI"/utf8>>,
                        logo = <<"foi.png">>,
                        lokacija = {46.30772093396054, 16.33808609928215},
                        adresa = adresa()},
        #db_fakultet{id = 2,
                        naziv = <<"Fakultet␣Elektrotehnike␣i␣čRaunarstva"/utf8>>,
                        opis = <<"Opis␣Fakulteta"/utf8>>,
                        skraceno = <<"FER"/utf8>>,
                        logo = <<"fer.jpg">>,
                        lokacija = {31.44, 116.44},
                        adresa = adresa()},
        #db_fakultet{id = 3,
                        naziv = <<"Ekonomski␣Fakultet␣Zagreb">>,
                        opis = <<"Opis␣Fakulteta">>,
                        skraceno = <<"EFZG"/utf8>>,
                        logo = <<"efzg.png">>,
                        lokacija = {31.44, 116.44},
                        adresa = adresa()},
        #db_fakultet{id = 4,
                        naziv = <<"Filozofski␣Fakultet␣Zagreb">>,
                        opis = <<"Opis␣Fakulteta">>,
                        skraceno = <<"FFZG"/utf8>>,
                        logo = <<"ffzg.jpg">>,
                        lokacija = {31.44, 116.44},
                        adresa = adresa()},
        #db_fakultet{id = 5,
                        naziv = <<"Fakultet␣Kemijskog␣žIninjerstva␣i␣Tehnologije"/
                            utf8>>,
                        opis = <<"Opis␣Fakulteta">>,
                        skraceno = <<"FKIT"/utf8>>,
                        logo = <<"fkit.jpg">>,
                        lokacija = {31.44, 116.44},
                        adresa = adresa()}].
```

Unutar sustava svi korisnici se dijele na studente i djelatnike, ali se nalaze unutar iste tablice jedina je razlika što će biti spremljeno unutar atributa dodatno i koji tip ima korisnik. Iz tog razlog prilikom zapisivanja korisnika potrebno je napraviti dodatne funkcije koje unutar sebe kreiraju zapis *dodatno* te s tim zapisom pozivaju funkciju za kreiranje korisnika. Također direktno kreiranje korisnika je u tom slučaju privatna funkcija isto kao i direktno uređivanje korisnika zato što će se uvijek uređivati ili student ili djelatnik. Ista implementacija napravljena je i za uređivanje korisnika dok se kod dohvaćanja ovisno o tipu odlučuje kakav tip zapisa se

pruža korisniku funkcije.

```
dodaj_studenta(Ime, Prezime, Oib, Lozinka, Email, Opis, Nadimak) ->
    Dodatno = #student{nadimak = Nadimak},
    dodaj_korisnika(Ime, Prezime, Oib, Lozinka, Email, Opis, student, Dodatno).

dodaj_djelatnika(Ime, Prezime, Oib, Lozinka, Email, Opis, Kabinet) ->
    Dodatno = #djelatnik{kabinet = Kabinet, vrijeme_konzultacija = []},
    dodaj_korisnika(Ime, Prezime, Oib, Lozinka, Email, Opis, djelatnik, Dodatno).

dodaj_korisnika(Ime, Prezime, Oib, Lozinka, Email, Opis, Uloga, Dodatno) ->
    Id = ?ID,
    Salt = crypto:strong_rand_bytes(16),
    Hash = crypto:hash(sha256, <<Salt/binary, Lozinka/binary>>),
    Fun = fun() ->
            case mnesia:index_read(db_korisnik, Oib, #db_korisnik.oib) == []
                 andalso mnesia:index_read(db_korisnik, Email, #db_korisnik.email)
                     == []
            of
                false -> {error, "Korisnik postoji"};
                true ->
                    case mnesia:write(#db_korisnik{id = Id,
                                                   ime = Ime,
                                                   prezime = Prezime,
                                                   oib = Oib,
                                                   lozinka = {Hash, Salt},
                                                   opis = Opis,
                                                   slika = <<"21104.png">>,
                                                   email = Email,
                                                   uloga = Uloga,
                                                   dodatno = Dodatno})
                    of
                        ok -> {ok, Id};
                        _ -> {error, "Transakcija šneuspjena"}
                    end
            end
        end,
    mnesia:transaction(Fun).
```

Također se iz isječka može vidjeti da se lozinka sprema kao tuple hash-a i soli koji je korištena za kriptiranje te lozinke. Na taj način se ne koristi uvijek ista sol već svaki korisnik ima svoju nasumično generiranu.

6.2. Implementacija HTTP poslužitelja

Unutar projekta za HTTP poslužitelj baza podataka je ovisnost (engl. *dependency*) što znači da pokretanjem servera pokreće se i baza podataka te se funkcije iz baze podataka mogu direktno koristiti. Pružatelj se također ponaša kao statički pružatelj zato što se unutar njega spremaju slike koje se kasnije mogu koristiti unutar web aplikacije.

Prvo se implementira usmjerivač kako bi poslužitelj znao na kojoj putanje se izvršava

koji upravljač.

```
   -module(server_http).

-export([start/0, stop/0]).

start() ->
    Dispatch =
        cowboy_router:compile([{'_',
                                [{"/home/", server_home_handler, []},
                                 {"/faculty/[:id]", server_faculty_handler, []},
                                 {"/faculty/department/",
                                     server_faculty_department_handler, []},
                                 {"/faculty/user/", server_faculty_user_handler, []}
                                     ,
                                 {"/department/[:id]", server_department_handler, []
                                     },
                                 {"/department/worker/",
                                     server_department_worker_handler, []},
                                 {"/department/course/",
                                     server_department_course_handler, []},
                                 {"/quiz/[:id]", server_quiz_handler, []},
                                 {"/quiz/student", server_quiz_student_handler, []},
                                 {"/student/course/[[:student]/[:course]]",
                                  server_student_course_handler,
                                  []},
                                 {"/student/", server_student_handler, []},
                                 {"/worker/", server_worker_handler, []},
                                 {"/worker/course/[[:worker]/[:course]]",
                                  server_worker_course_handler,
                                  []},
                                 {"/course/[:id]", server_course_handler, []},
                                 {"/course/section/", server_course_section_handler,
                                      []},
                                 {"/section/[:id]", server_section_handler, []},
                                 {"/section/content/",
                                     server_section_content_handler, []},
                                 {"/content/[:id]", server_content_handler, []},
                                 {"/user/[:id]", server_user_handler, []},
                                 {"/question/[:id]", server_question_handler, []},
                                 {"/login/", server_login_handler, []},
                                 {"/upload/", server_upload_handler, []},
                                 {"/assets/[...]", cowboy_static, {priv_dir, server,
                                      "assets"}},
                                 {"/jwt/refresh", server_jwt_refresh_handler, []}]}]
                                     ),

    cowboy:start_clear(server_http_listener,
                       [{port, 5000}],
                       #{middlewares => [server_cors_middleware, cowboy_router,
                           cowboy_handler],
                         env => #{dispatch => Dispatch}}).

stop() ->
```

```
    ok = cowboy:stop_listener(server_http_listener).
```

Neke putanje su striktno ograničene dok se drugima može proslijediti još dodatan parametar koji se koristi u većini slučajeva za dohvaćanje specifičnog resursa. Može se primijetiti da je implementiran upravljač za svaku tablicu koja je kreirana te još nekoliko dodatnih. Uz upravljače implementiran je i jedan dodatan srednji sloj koji otklanja probleme s primanjem HTTP zahtjeva i slanjem HTTP odgovora. To je poznati problem kod razvoja web aplikacija te je samo potrebno presresti zahtjev i promijeniti zaglavlje da dopušta pristup resursima iz druge domene.

```
execute(#{headers := #{<<"origin">> := HeaderVal}} = Req, Env) ->
    handle_cors_request(HeaderVal, Req, Env);
execute(Req, Env) ->
    {ok, Req, Env}.

handle_cors_request(Origin, #{method := Method} = Req, Env) ->
    Req2 = cowboy_req:set_resp_header(<<"access-control-allow-origin">>, Origin, Req
        ),
    Req3 = cowboy_req:set_resp_header(<<"vary">>, <<"Origin">>, Req2),
    case Method of
        <<"OPTIONS">> ->
            Req4 =
                cowboy_req:set_resp_header(<<"access-control-allow-methods">>,
                                           <<"GET, DELETE, PUT, POST, PATCH">>,
                                           Req3),
            Req5 = cowboy_req:set_resp_header(<<"access-control-allow-headers">>, <<
                "*">>, Req4),
            Req6 = cowboy_req:set_resp_header(<<"access-control-max-age">>, <<"0">>,
                 Req5),
            Req7 = cowboy_req:reply(200, Req6),
            {stop, Req7};
        _ ->
            {ok, Req3, Env}
    end.
```

Sljedeće na redu je implementacija upravljača za svaku putanju. Kako bi se implementacija olakšala kreirane su funkcije koje to olakšavaju. Pošto je većina putanja zaštićena, što znači da je potrebno proslijediti token koji se generira prilikom prijave u aplikaciju, nema smisla ponovno pisati istu funkciju unutar svakog upravljača. Također je odmah implementirana funkcija koja šalje odgovore pogreške te funkcija koja šalje ispravne odgovore. Posljednja funkcija koja je implementirana jest apstrakcija prilikom komuniciranja s bazom podataka koja ovisno o odgovoru baze zna kakav odgovor treba poslati klijentu. To je moguće napravit samo zato što je baza implementirana na način da sve funkcije vraćaju iste strukture koje je onda moguće usporediti pomoću pattern matching-a.

```
send_response(Req, Data, State) ->
    Body = json:encode(#{data => Data}),
    Req2 = cowboy_req:set_resp_body(Body, Req),
    Reply = cowboy_req:reply(200, Req2),
    {stop, Reply, State}.
```

```
err(Code, Reason, Req, State) ->
    Formatted = iolist_to_binary(io_lib:format("~p", [Reason])),
    Body = json:encode(#{data => Formatted}),
    Req2 = cowboy_req:set_resp_body(Body, Req),
    Reply = cowboy_req:reply(Code, Req2),
    {stop, Reply, State}.

auth(Req, State) ->
    case cowboy_req:header(<<"authorization">>, Req) of
        undefined ->
            {{false, <<"Bearer token_type=\"JWT\"">>}, Req, State};
        <<"Bearer ", Token/binary>> ->
            case jwt_manager:verify_access_token(Token) of
                {ok, _} ->
                    {true, Req, State};
                {error, _} ->
                    {{false, <<"Bearer token_type=\"JWT\"">>}, Req, State}
            end
    end.

delete(Req, State, Fun) ->
    case utils:gather_json(Req) of
        {ok, Map, Req2} ->
            case Fun(maps:get(<<"id">>, Map)) of
                {atomic, ok} ->
                    send_response(Req2, <<"ok">>, State);
                {aborted, Reason} ->
                    err(400, Reason, Req, State)
            end;
        _ ->
            err(400, "Db Error", Req, State)
    end.

response(Req, State, Fun) ->
    case Fun() of
        {atomic, Result} ->
            case Result of
                {error, Reason} ->
                    err(403, Reason, Req, State);
                {ok, Id} ->
                    send_response(Req, Id, State);
                _ ->
                    send_response(Req, Result, State)
            end;
        {aborted, Reason} ->
            err(403, Reason, Req, State)
    end.
```

Kako je navedeno da se prilikom svakog zahtjeva provjerava autentikacija korisnika pomoću tokena potrebno je implementirati tokene. Unutar ovog sustava korišteni su JWT tokeni kojima je moguće postaviti vrijeme trajanja, provjeravati im valjanost te najbitnije zapisati po-

datke. Svaka od tih funkcionalnosti implementirana je u modulu *jwt_manager*.

```
-define(ACCESS_TOKEN_SECRET, <<"ABXw3mqeOO40n/SzaU31lD2ENJePJ1Mr9ZQE1ZBBQAU=">>).
-define(ACCESS_TOKEN_LIFETIME, 60 * 24 * 60 * 60).

generate_tokens(Id) ->
    CurrentTime =
        calendar:datetime_to_gregorian_seconds(
            calendar:universal_time())
        - calendar:datetime_to_gregorian_seconds({{1970, 1, 1}, {0, 0, 0}}),
    AccessTokenExp = CurrentTime + ?ACCESS_TOKEN_LIFETIME,
    AccessClaims = #{<<"sub">> => Id, <<"exp">> => AccessTokenExp},
    AccessToken = jwerl:sign(AccessClaims, hs256, ?ACCESS_TOKEN_SECRET),

    AccessToken.

verify_access_token(Token) ->
    jwerl:verify(Token, hs256, ?ACCESS_TOKEN_SECRET).
```

Tokeni se generiraju prilikom prijave u aplikaciju koja je sama svoj upravljač i nalazi se na putanji */login* te prima samo HTTP POST zahtjev. Kao što je već navedeno i prikazano na slici 7 svaka vrsta zahtjeva prolazi kroz druge funkcije. Prilikom implementacije putanje za POST zahtjev potrebno je dopustiti metodu POST, u slučaju ovog sustava prihvaćati JSON koji se zatim obrađuje. Iz zahtjeva se čita tijelo te se uzimaju vrijednosti koje je poslao klijent i prosljeđuju se u funkciju iz baze podataka. Ovisno o ishodu funkcije ili se vraća odgovor kao pogreška s razlogom ili se generira token te se on vraća kao odgovor na zahtjev.

```
-module(server_login_handler).

-behaviour(cowboy_handler).

-export([init/2, allowed_methods/2, content_type_provided/2, content_types_accepted
    /2,
         from_json/2, to_json/2, charsets_provided/2]).

init(Req, State) ->
    {cowboy_rest, Req, State}.

allowed_methods(Req, State) ->
    {[<<"POST">>], Req, State}.

content_types_accepted(Req, State) ->
    {[{{<<"application">>, <<"json">>, []}, from_json}], Req, State}.

content_type_provided(Req, State) ->
    {[{{<<"application">>, <<"json">>, []}, to_json}], Req, State}.

charsets_provided(Req, State) ->
    {[<<"utf-8">>], Req, State}.

from_json(Req, State) ->
    json_request(Req, State).
```

```
to_json(Req, State) ->
    json_request(Req, State).

json_request(Req, State) ->
    case utils:gather_json(Req) of
        {error, Reason, _} ->
            request:err(400, Reason, Req, State);
        {ok, Map, Req2} ->
            run_post_request(Map, Req2, State)
    end.

run_post_request(Map, Req, State) ->
    case korisnik:prijava(
             maps:get(<<"email">>, Map), maps:get(<<"password">>, Map))
    of
        {atomic, Result} ->
            case Result of
                {ok, #{id := Id}} ->
                    {AccessToken, RefreshToken} = jwt_manager:generate_tokens(Id),
                    request:send_response(Req,
                                          #{access_token => AccessToken},
                                          State);
                {error, Reason} ->
                    request:err(404, Reason, Req, State)
            end;
        {aborted, _} ->
            request:err(500, "Database error", Req, State)
    end.
```

Na sličan način se implementiraju HTTP GET zahtjevi jedina razlika je to što unutar njih nije potrebno čitati podatke iz tijela zahtjeva već se čitaju proslijeđeni parametri iz putanje. Naravno putanja ne mora imati nikakve parametar te se u tom slučaju dohvaća lista svih elemenata iz tablice za koju je ta putanja implementirana. Kada parametar postoji onda se dohvaća točno jedan podataka iz tablice.

```
-module(server_user_handler).

-behaviour(cowboy_handler).

-export([init/2, allowed_methods/2, charsets_provided/2, is_authorized/2, to_html/2,
         delete_resource/2]).

init(Req, State) ->
    {cowboy_rest, Req, State}.

allowed_methods(Req, State) ->
    {[<<"GET">>, <<"DELETE">>], Req, State}.

is_authorized(Req, State) ->
    request:auth(Req, State).

charsets_provided(Req, State) ->
    {[<<"utf-8">>], Req, State}.
```

```
delete_resource(Req, State) ->
    request:delete(Req, State, fun(Id) -> korisnik:obrisi_korisnika(Id) end).

to_html(Req, State) ->
    html_request(Req, State).

html_request(Req, State) ->
    case utils:gather_html(Req) of
        {error, Reason, _} ->
            request:err(400, Reason, Req, State);
        {ok, #{id := Id}, Req2} ->
            request:response(Req2,
                             State,
                             fun() -> korisnik:dohvati_korisnika(kolegiji,
                                 binary_to_integer(Id))
                             end);
        {ok, _, Req2} ->
            request:response(Req2, State, fun() -> korisnik:dohvati_korisnike() end)
    end.
```

Unutar većine upravljača uz HTTP GET i POST dopušteni su HTTP DELETE, PATCH i PUT zahtjevi pomoću kojih se brišu i dodaju podaci. Razlog tome je što je pomoću Cowboy-a moguće prepoznati koji zahtjev dolazi te za svaki odraditi ono što je potrebno.

```
-module(server_faculty_handler).

-export([init/2, allowed_methods/2, charsets_provided/2, is_authorized/2,
         content_type_provided/2, content_types_accepted/2, to_html/2,
             delete_resource/2,
         from_json/2]).

init(Req, State) ->
    {cowboy_rest, Req, State}.

allowed_methods(Req, State) ->
    {[<<"GET">>, <<"PATCH">>, <<"PUT">>, <<"DELETE">>], Req, State}.

is_authorized(Req, State) ->
    request:auth(Req, State).

content_types_accepted(Req, State) ->
    {[{{<<"application">>, <<"json">>, []}, from_json}], Req, State}.

content_type_provided(Req, State) ->
    {[{{<<"application">>, <<"json">>, []}, to_html}], Req, State}.

charsets_provided(Req, State) ->
    {[<<"utf-8">>], Req, State}.

delete_resource(Req, State) ->
    request:delete(Req, State, fun(Id) -> fakultet:obrisi(Id) end).
```

```
to_html(Req, State) ->
    html_request(Req, State).

from_json(Req, State) ->
    json_request(Req, State).

html_request(Req, State) ->
    case utils:gather_html(Req) of
        {error, Reason, _} ->
            request:err(400, Reason, Req, State);
        {ok, #{id := Id}, Req2} ->
            request:response(Req2, State, fun() -> fakultet:dohvati(
                binary_to_integer(Id)) end);
        {ok, _, Req2} ->
            request:response(Req2, State, fun() -> fakultet:dohvati() end)
    end.

json_request(Req, State) ->
    case utils:gather_json(Req) of
        {error, Reason, _} ->
            request:err(400, Reason, Req, State);
        {ok, Map, Req2} ->
            gather_method(Map, Req2, State)
    end.

gather_method(Map, Req, State) ->
    case cowboy_req:method(Req) of
        <<"PUT">> ->
            run_put_request(Map, Req, State);
        <<"PATCH">> ->
            run_patch_request(Map, Req, State)
    end.

run_put_request(#{<<"naziv">> := Naziv, <<"adresa">> := #{<<"ulica">> := _} = Adresa
    },
                Req,
                State) ->
    request:response(Req, State, fun() -> fakultet:dodaj(Naziv, Adresa) end);
run_put_request(_, Req, State) ->
    request:err(400, <<"Wrong␣keys">>, Req, State).

run_patch_request(#{<<"id">> := Id,
                    <<"naziv">> := Naziv,
                    <<"adresa">> := #{<<"ulica">> := _} = Adresa},
                  Req,
                  State) ->
    request:response(Req, State, fun() -> fakultet:uredi(Id, Naziv, Adresa) end);
run_patch_request(_, Req, State) ->
    request:err(400, <<"Wrong␣keys">>, Req, State).
```

Na početku je spomenuto da je moguće spremati slike na poslužitelja pomoću HTTP poslužitelja. Za taj slučaj također je kreiran upravljač na putanji */upload* koji je malo kompleksniji

zato što je potrebno čitati višedjelne (engl. *multipart*) podatke iz tijela. Ideja je da klijent unutar multipart podatka prosljeđuje dokument u binarnom obliku koji se zatim obrađuju i spremaju na poslužitelju unutar direktorija *priv/assets/*. Nakon uspješnog spremanja vraća se HTTP odgovor klijentu unutar kojega se nalazi novi naziv dokumenta koji je spremljen na poslužitelju kako bi se mogao dodjeliti objektu kojem je potreban i zatim napraviti uređivanje podataka.

```
-module(server_upload_handler).

-behaviour(cowboy_handler).

-export([init/2, allowed_methods/2, content_types_accepted/2, charsets_provided/2,
         is_authorized/2, from_multipart/2]).

init(Req, State) ->
    {cowboy_rest, Req, State}.

allowed_methods(Req, State) ->
    {[<<"POST">>], Req, State}.

is_authorized(Req, State) ->
    request:auth(Req, State).

content_types_accepted(Req, State) ->
    {[{{<<"multipart">>, <<"form-data">>, []}, from_multipart}], Req, State}.

charsets_provided(Req, State) ->
    {[<<"utf-8">>], Req, State}.

from_multipart(Req0, State) ->
    case cowboy_req:read_part(Req0) of
        {ok, Headers, Req1} ->
            case cow_multipart:form_data(Headers) of
                {data, _} ->
                    {ok, _, _} = cowboy_req:read_part_body(Req1);
                {file, FieldName, _, _} ->
                    {Req3, File} = stream_file(Req1, State, FieldName),
                    NewState = State ++ [File],
                    from_multipart(Req3, NewState)
            end;
        {done, Req} ->
            [{Extension, File}] = State,
            Res = handle(File, Extension),
            request:send_response(Req, Res, State)
    end.

handle(File, Extension) ->
    handle_data(Extension, File).

generate_paths(Extension) ->
    Id = integer_to_binary(erlang:unique_integer([positive])),
    Separator = <<".">>,
    Path = <<"priv/assets/">>,
    FileName = <<Id/binary, Separator/binary, Extension/binary>>,
```

```
    WritePath = <<Path/binary, FileName/binary>>,
    {WritePath, FileName}.

write_assets(WritePath, File) ->
    case file:write_file(WritePath, File) of
        ok ->
            ok;
        {error, _} ->
            error
    end.

handle_data(Extension, File) ->
    {WritePath, FileName} = generate_paths(Extension),
    write_assets(WritePath, File),
    FileName.

stream_file(Req0, State, FieldName) ->
    case cowboy_req:read_part_body(Req0) of
        {ok, _LastBodyChunk, Req} ->
            {Req, {FieldName, _LastBodyChunk}};
        {more, _BodyChunk, Req} ->
            stream_file(Req, State, FieldName)
    end.
```

6.3. Implementacija HTTP klijenta

Kako bi se WebAssembly što bolje iskoristio kreiran je projekt unutar kojega se odvija sva komunikacija s poslužiteljem. Na taj način projekt unutar kojega se nalazi web aplikacija ne zna ništa niti i poslužitelju na kojega se zahtjevi šalju niti na koji način se šalju. Ovakva implementacije je jedna od najmodernijih implementacija web aplikacija iz razloga prenosivosti.

Za potrebe ovoga dijela kreiran je projekt u Rust-u unutar kojega postoje samo dva dokumenta. Dokument s tipovima koje poslužitelj vraća te dokument s funkcijama koje se pozivaju iz web aplikacije. Kako bi se implementirani tipovi mogli koristiti unutar Angular aplikacije potrebno je dodati dekoratore koji mu to omogućuju te je potrebno za svaku strukturu napisati i implementaciju. Pošto se unutar nekih struktura neki od atributa neće uvijek koristiti implementirani su kao opcionalni atributi.

```
#[wasm_bindgen]
#[derive(Serialize, Deserialize, Clone)]
pub struct Korisnik {
    id: i32,
    ime: String,
    prezime: String,
    slika: String,
    oib: i32,
    uloga: String,
    email: String,
    opis: String,
    dodatno: Dodatno,
```

```
    #[serde(skip_serializing_if = "Option::is_none")]
    kolegiji: Option<Vec<Kolegij>>,
    #[serde(skip_serializing_if = "Option::is_none")]
    fakultet: Option<Fakultet>,
    #[serde(skip_serializing_if = "Option::is_none")]
    katedre: Option<Vec<Katedra>>,
    #[serde(skip_serializing_if = "Option::is_none")]
    tip: Option<String>,
}

#[wasm_bindgen]
impl Korisnik {
    #[wasm_bindgen(getter)]
    pub fn id(&self) -> i32 {
        self.id.clone()
    }

    #[wasm_bindgen(setter)]
    pub fn set_id(&mut self, id: i32) {
        self.id = id;
    }
    #[wasm_bindgen(getter)]
    pub fn ime(&self) -> String {
        self.ime.clone()
    }

    #[wasm_bindgen(setter)]
    pub fn set_ime(&mut self, ime: String) {
        self.ime = ime;
    }
    #[wasm_bindgen(getter)]
    pub fn slika(&self) -> String {
        self.slika.clone()
    }

    #[wasm_bindgen(setter)]
    pub fn set_slika(&mut self, slika: String) {
        self.slika = slika;
    }

    #[wasm_bindgen(getter)]
    pub fn prezime(&self) -> String {
        self.prezime.clone()
    }
    #[wasm_bindgen(setter)]
    pub fn set_prezime(&mut self, prezime: String) {
        self.prezime = prezime;
    }

    #[wasm_bindgen(getter)]
    pub fn oib(&self) -> i32 {
        self.oib.clone()
    }
```

```
#[wasm_bindgen(setter)]
pub fn set_oib(&mut self, oib: i32) {
    self.oib = oib;
}

#[wasm_bindgen(getter)]
pub fn uloga(&self) -> String {
    self.uloga.clone()
}
#[wasm_bindgen(setter)]
pub fn set_uloga(&mut self, uloga: String) {
    self.uloga = uloga;
}

#[wasm_bindgen(getter)]
pub fn email(&self) -> String {
    self.email.clone()
}
#[wasm_bindgen(setter)]
pub fn set_email(&mut self, email: String) {
    self.email = email;
}

#[wasm_bindgen(getter)]
pub fn opis(&self) -> String {
    self.opis.clone()
}
#[wasm_bindgen(setter)]
pub fn set_opis(&mut self, opis: String) {
    self.opis = opis;
}
#[wasm_bindgen(getter)]
pub fn dodatno(&self) -> Dodatno {
    self.dodatno.clone()
}
#[wasm_bindgen(setter)]
pub fn set_dodatno(&mut self, dodatno: Dodatno) {
    self.dodatno = dodatno;
}

#[wasm_bindgen(getter)]
pub fn kolegiji(&self) -> Vec<Kolegij> {
    self.kolegiji.clone().expect("Kolegiji")
}

#[wasm_bindgen(setter)]
pub fn set_kolegiji(&mut self, kolegiji: Vec<Kolegij>) {
    self.kolegiji = Some(kolegiji);
}
#[wasm_bindgen(getter)]
pub fn fakultet(&self) -> Fakultet {
    self.fakultet.clone().expect("Fakultet")
}
```

```
    #[wasm_bindgen(setter)]
    pub fn set_fakultet(&mut self, fakultet: Fakultet) {
        self.fakultet = Some(fakultet);
    }
    #[wasm_bindgen(getter)]
    pub fn katedre(&self) -> Vec<Katedra> {
        self.katedre.clone().expect("Katedra")
    }

    #[wasm_bindgen(setter)]
    pub fn set_katedre(&mut self, katedre: Vec<Katedra>) {
        self.katedre = Some(katedre);
    }
    #[wasm_bindgen(getter)]
    pub fn tip(&self) -> String {
        self.tip.clone().expect("Tip")
    }

    #[wasm_bindgen(setter)]
    pub fn set_tip(&mut self, tip: String) {
        self.tip = Some(tip);
    }
}

#[wasm_bindgen]
#[derive(Serialize, Deserialize, Clone)]
pub struct Dodatno {
    #[serde(skip_serializing_if = "Option::is_none")]
    nadimak: Option<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    kabinet: Option<String>,
    #[serde(skip_serializing_if = "Option::is_none")]
    vrijeme_konzultacija: Option<Vec<String>>,
}

#[wasm_bindgen]
impl Dodatno {
    #[wasm_bindgen(getter)]
    pub fn nadimak(&self) -> String {
        self.nadimak.clone().expect("nadimak")
    }

    #[wasm_bindgen(setter)]
    pub fn set_nadimak(&mut self, nadimak: String) {
        self.nadimak = Some(nadimak);
    }

    #[wasm_bindgen(getter)]
    pub fn kabinet(&self) -> String {
        self.kabinet.clone().expect("kabinet")
    }
```

```
    #[wasm_bindgen(setter)]
    pub fn set_kabinet(&mut self, kabinet: String) {
        self.kabinet = Some(kabinet);
    }
    #[wasm_bindgen(getter)]
    pub fn vrijeme_konzultacija(&self) -> Vec<String> {
        self.vrijeme_konzultacija
            .clone()
            .expect("vrijeme␣konzultacija")
    }

    #[wasm_bindgen(setter)]
    pub fn set_vrijeme_konzultacija(&mut self, vrijeme_konzultacija: Vec<String>) {
        self.vrijeme_konzultacija = Some(vrijeme_konzultacija);
    }
}
```

Nakon kreiranja svih tipova sljedeći korak jest implementiranje samih zahtjeva pomoću biblioteke Reqwest. Za komunikaciju s poslužiteljem potrebne su četiri vrste zahtjeva te je za svaki napravljena apstrakcija kako bi se lakše implementirali konkretni zahtjevi. Svi zahtjevi osim GET zahtjeva šalju tijelo unutar kojega se nalazi JSON objekt, ali pošto svaka putanja prima drugačiji objekt potrebno je dio koda koji kreira objekt izdvojiti u konkretne funkcije

```
pub fn create_get_request(
    uri: &str,
    token: String,
) -> impl Future<Output = Result<reqwest::Response, reqwest::Error>> {
    let client = reqwest::Client::new();
    client.get(uri).bearer_auth(token).send()
}

pub fn create_post_request(
    uri: &str,
    object: Value,
    token: String,
) -> impl Future<Output = Result<reqwest::Response, reqwest::Error>> {
    let client = reqwest::Client::new();
    client
        .post(uri)
        .header("Content-Type", "application/json")
        .json(&object)
        .bearer_auth(token)
        .send()
}

pub fn create_patch_request(
    uri: &str,
    object: Value,
    token: String,
) -> impl Future<Output = Result<reqwest::Response, reqwest::Error>> {
    let client = reqwest::Client::new();
    client.patch(uri).json(&object).bearer_auth(token).send()
}
```

```
pub fn create_delete_request(
    uri: &str,
    object: Value,
    token: String,
) -> impl Future<Output = Result<reqwest::Response, reqwest::Error>> {
    let client = reqwest::Client::new();
    client.delete(uri).json(&object).bearer_auth(token).send()
}

pub fn parse_network_err() -> Result<JsValue, JsValue> {
    let err = types::MyError::new(500, "Network␣error");
    Err(to_value(&err).unwrap())
}

pub async fn parse_data<T>(response: reqwest::Response) -> Result<JsValue, JsValue>
where
    T: DeserializeOwned + serde::Serialize,
{
    let body = response.text().await.unwrap();
    let json_data: types::Response<T> = match serde_json::from_str::<types::Response
        <T>>(&body) {
        Ok(data) => data,
        Err(_) => return Err(JsValue::from_str("šPogreka␣u␣parsiranju")),
    };
    Ok(to_value(&json_data.data).unwrap())
}

pub async fn parse_vec_data<T>(response: reqwest::Response) -> Result<JsValue,
    JsValue>
where
    T: DeserializeOwned + serde::Serialize,
{
    let body: String = match response.text().await {
        Ok(res) => res,
        Err(_) => return Err(JsValue::from_str("šPogreka␣u␣text")),
    };
    let json_data: types::VecResponse = match serde_json::from_str::<types::
        VecResponse>(&body) {
        Ok(data) => data,
        Err(_) => return Err(JsValue::from_str("šPogreka␣u␣parse")),
    };
    Ok(to_value(&json_data.data).unwrap())
}
```

Zbog korištenja funkcije *create_get_request* pojednostavljuje se implementacija zahtjeva za dohvaćanje za konkretne slučajeve te su razlike minimalne.

```
#[wasm_bindgen]
pub async fn dohvati_katedre() -> Result<JsValue, JsValue> {
    let result = match create_get_request("http://localhost:5000/deparment", "".
        to_string()).await {
        Ok(response) => parse_data::<Vec<types::Katedra>>(response).await,
```

```
        Err(_) => parse_network_err(),
    };
    result
}

#[wasm_bindgen]
pub async fn dohvati_katedru(id: i32) -> Result<JsValue, JsValue> {
    let result = match create_get_request(
        &format!("http://localhost:5000/department/{}", id),
        "".to_string(),
    )
    .await
    {
        Ok(response) => parse_data::<types::Katedra>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}
#[wasm_bindgen]
pub async fn dohvati_kolegije(token: String) -> Result<JsValue, JsValue> {
    let result = match create_get_request("http://localhost:5000/course", token).
        await {
        Ok(response) => parse_data::<Vec<types::Kolegij>>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}

#[wasm_bindgen]
pub async fn dohvati_kolegij(id: i32, token: String) -> Result<JsValue, JsValue> {
    let result =
        match create_get_request(&format!("http://localhost:5000/course/{}", id),
            token).await {
            Ok(response) => parse_data::<types::Kolegij>(response).await,
            Err(_) => parse_network_err(),
        };
    result
}
```

Isto kao što funkcija za dohvaćanje zahtjeva ima svoje konkretne implementacije tako i sve ostalo funkcije imaju isto.

```
#[wasm_bindgen]
pub async fn dodaj_poveznicu(
    naziv: String,
    poveznica: String,
    token: String,
) -> Result<JsValue, JsValue> {
    let obj = serde_json::json!({
        "naziv": naziv,
        "tip": "poveznica",
        "vrijednost":{
            "referenca": poveznica
```

```
        }
    });
    let result = match create_post_request("http://localhost:5000/content", obj,
        token).await {
        Ok(response) => parse_data::<i32>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}

#[wasm_bindgen]
pub async fn uredi_poveznicu(
    id: i32,
    naziv: String,
    poveznica: String,
    token: String,
) -> Result<JsValue, JsValue> {
    let obj = serde_json::json!({
        "id": id,
        "naziv": naziv,
        "tip": "poveznica",
        "vrijednost":{
            "referenca": poveznica
        }
    });
    let result = match create_patch_request("http://localhost:5000/content", obj,
        token).await {
        Ok(response) => parse_data::<i32>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}

#[wasm_bindgen]
pub async fn obrisi_sadrzaj(id: i32, token: String) -> Result<JsValue, JsValue> {
    let obj = serde_json::json!({
        "id": id,
    });
    let result = match create_delete_request("http://localhost:5000/content", obj,
        token).await {
        Ok(response) => parse_data::<String>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}
```

Pošto je planirano spremati dokumente na poslužitelja potrebno je i taj zahtjev implementirati tako da se apsolutno sva komunikacija s poslužiteljem izolira unutar WebAssembly klijenta. Najveća razlika je u tome što se dokument mora obraditi te zatim nakon toga poslati u obliku multipart zahtjeva.

```
#[wasm_bindgen]
pub async fn upload_file(file: web_sys::File, token: String) -> Result<JsValue,
```

```
    JsValue> {
    let client = reqwest::Client::new();

    let array_buffer = JsFuture::from(file.array_buffer()).await?;
    let uint8_array = Uint8Array::new(&array_buffer);
    let bytes = uint8_array.to_vec();
    let filename = file.name();

    let extension = get_extension(&filename).unwrap_or("unknown");

    let part_file = reqwest::multipart::Part::bytes(bytes).file_name(filename.clone
        ());

    let form = reqwest::multipart::Form::new().part(extension.to_string(), part_file
        );

    let result = match client
        .post("http://localhost:5000/upload")
        .bearer_auth(token)
        .multipart(form)
        .send()
        .await
    {
        Ok(response) => parse_data::<String>(response).await,
        Err(_) => parse_network_err(),
    };
    result
}
```

Nakon implementacije svih zahtjeva potrebno je ovaj projekt izgraditi pomoću naredbe *wasm-pack build –target web*. Pomoću te naredbe prevodi se Rust u WebAssembly te se generiraju svi potrebni dokumenti kako bi se koristio unutar Angular aplikacije.

6.4. Implementacija web aplikacije

Kako je sva logika oko dohvaćanja podataka izolirana unutar WebAssembly-a potrebno je prevedene dokumente implementirati unutar Angular-a. To se radi tako da se unutar direktorija *assets* kopira sve što se dobilo izgradnjom Rust projekta. Nakon toga potrebno je namjestiti da se aplikacija pokreće tek kada se WebAssembly instancira kako bi se mogli slati zahtjevi na poslužitelja. Prije toga potrebno je kreirati servis koji omogućuje instanciranje WebAssembly-a.

```
import { Injectable } from '@angular/core';
import { InitOutput } from '../../../assets/pkg/client';
import init from '../../../assets/pkg/client';

export type Response<T> = {
  data: T;
};

@Injectable({
  providedIn: 'platform',
```

```
})
export class WasmService {
  public wasm: any;

  constructor() {}

  async loadWasmModule() {
    if (!this.wasm) {
      this.wasm = await import('../../../assets/pkg/client');
      await init({ wasm: this.wasm });
    }
    return this.wasm;
  }
}
```

Kako bi se WebAssembly instancirao prije samog pokretanja aplikacije potrebno je unutar konfiguracije aplikacije postaviti funkciju za inicijalizaciju te trenutak u kojem se inicijalizira što je *APP_INITIALIZER*. Pošto unutar aplikacije postoje putanje kojima mogu pristupiti samo prijavljeni korisnici također se nakon same inicijalizacije modula odmah koristi funkcija koja šalje zahtjev na poslužitelja te ovisno o odgovoru ili postavlja korisnika kao trenutnog ili briše token iz lokalnog spremnik i prosljeđuje korisnika na prijavu.

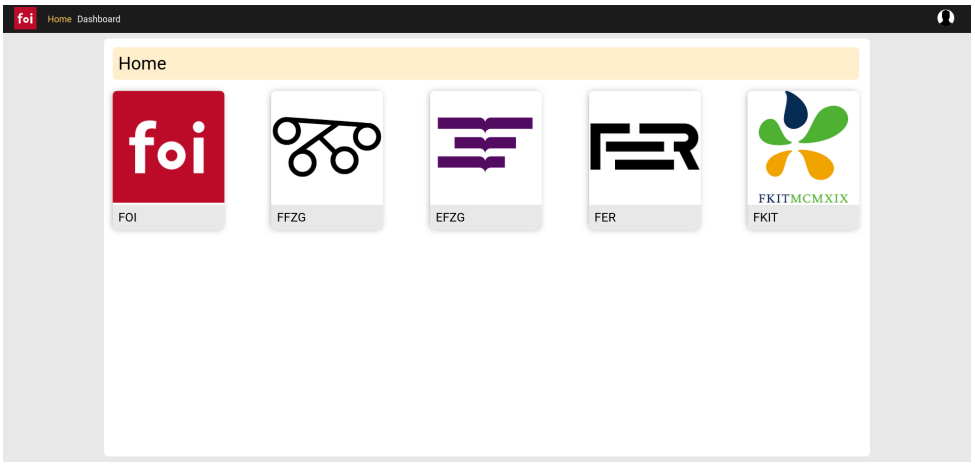
```
export function initializeWasm(
  wasmService: WasmService,
  userService: UserService,
  tokenService: TokenService,
  router: Router,
) {
  return () =>
    wasmService.loadWasmModule().then((e) =>
      userService
        .dohvati_korisnika()
        .then((k) => {
          userService.user.set(k);
        })
        .catch((err) => {
          tokenService.removeAccessToken();
          router.navigate(['/login']);
        }),
    );
}

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withComponentInputBinding()),
    provideAnimations(),
    provideToastr(),
    WasmService,
    {
      provide: APP_INITIALIZER,
      useFactory: initializeWasm,
      deps: [WasmService, UserService, TokenService, Router],
```

```
      multi: true,
    },
  ],
};
```

Unutar aplikacije neke putanje mogu vidjeti i ne prijavljeni korisnici kao što su popis fakulteta, pregled jednog fakulteta i pregled katedre. Sve ostale stranice zaštičene su pomoću Angular zaštitniki unutar kojih se provjerava je li korisnik prijavljen ili nije.

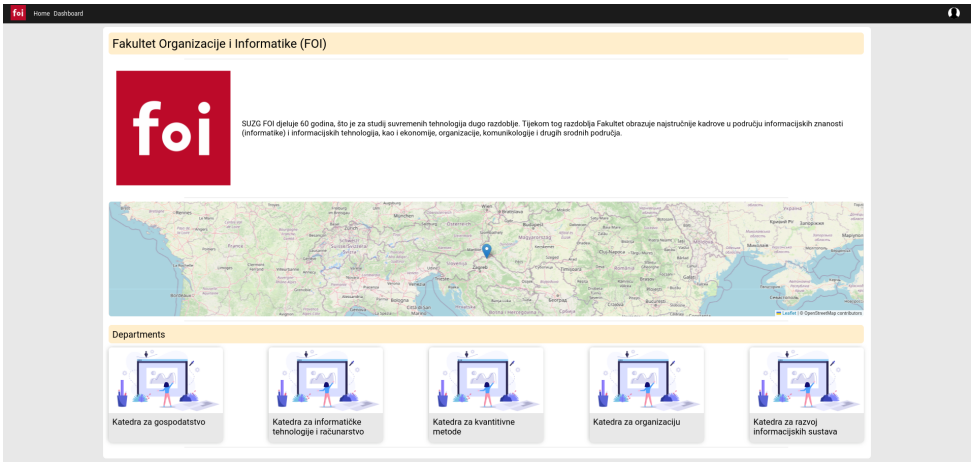
```
export const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent, title: 'Home' },
  { path: 'login', component: LoginComponent, title: 'Login' },
  { path: 'faculty/:id', component: FacultyComponent, title: 'Faculty' },
  { path: 'department/:id', component: DepartmentComponent, title: 'Faculty' },
  {
    path: 'dashboard',
    loadComponent: () => import('./features').then((c) => c.DashboardComponent),
    title: 'Dashboard',
    canActivate: [authGuard],
  },
  {
    path: 'courses',
    title: 'Courses',
    loadComponent: () => import('./features').then((c) => c.CoursesComponent),
    canActivate: [authGuard],
  },
  {
    path: 'profile/:id',
    title: 'Profile',
    loadComponent: () => import('./features').then((c) => c.ProfileComponent),
    canActivate: [authGuard],
  },
  {
    path: 'course/:id',
    title: 'Course',
    loadComponent: () => import('./features').then((c) => c.CourseComponent),
    canActivate: [authGuard],
  },
  {
    path: 'content/:id',
    title: 'Content',
    loadComponent: () => import('./features').then((c) => c.ContentComponent),
    canActivate: [authGuard],
  },
];
```

Na početnoj stranici aplikacije moguće je vidjeti listu svih fakulteta koji su integrirani u aplikaciju te klik na svaki od njih vodi na stranicu fakulteta.



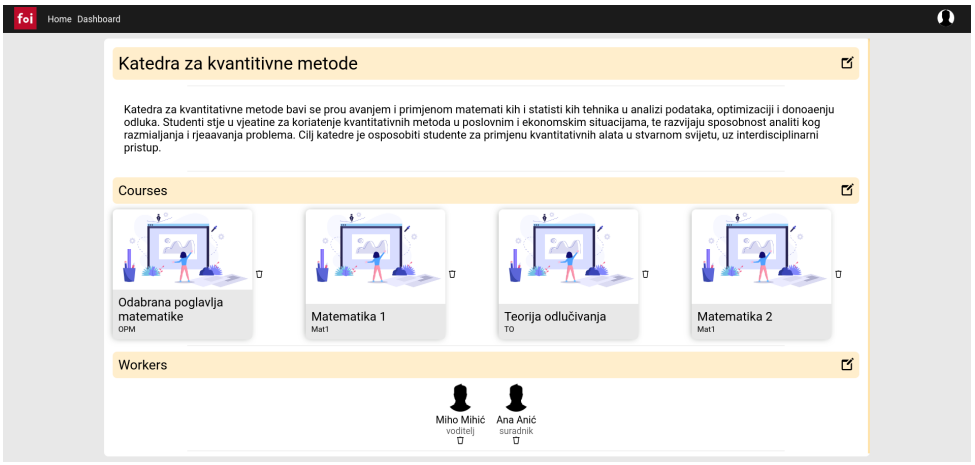
Slika 8: Ekran početne stranice (Izvor: Autorski rad)

Na stranici fakulteta nalazi se pregled osnovnih informacija o fakultetu poput opisa, slike, lokacije i popisa katedri koje postoje na fakultetu. Fakultet je moguće uređivati, ali to može raditi samo korisnik koji je ujedno i dekan na tom fakultetu. Također klik na katedru vodi na stranicu katedre.



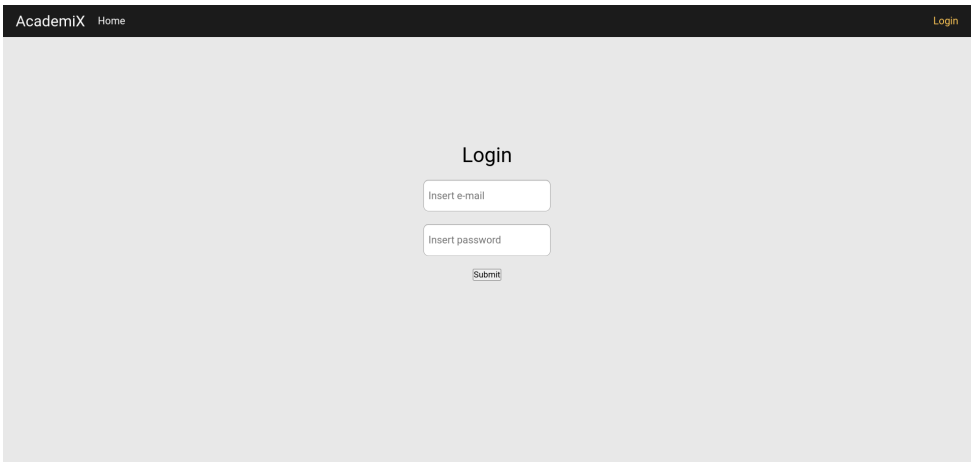
Slika 9: Ekran fakulteta (Izvor: Autorski rad)

Stranica katedre sastoji se od prikaza opisa katedre, kolegija koji su na katedri i radnika na katedri. Ova stranica specifična je zato što je klikove moguće napraviti jedino ako je korisnik prijavljen te je također moguće uređivati osnovne informacije o katedri, dodavati i brisati kolegije te dodavati i brisati djelatnike na kolegiju. Sve akcije dopuštene se samo voditelju te katedre dok se drugim korisnicima niti ne prikazuju.



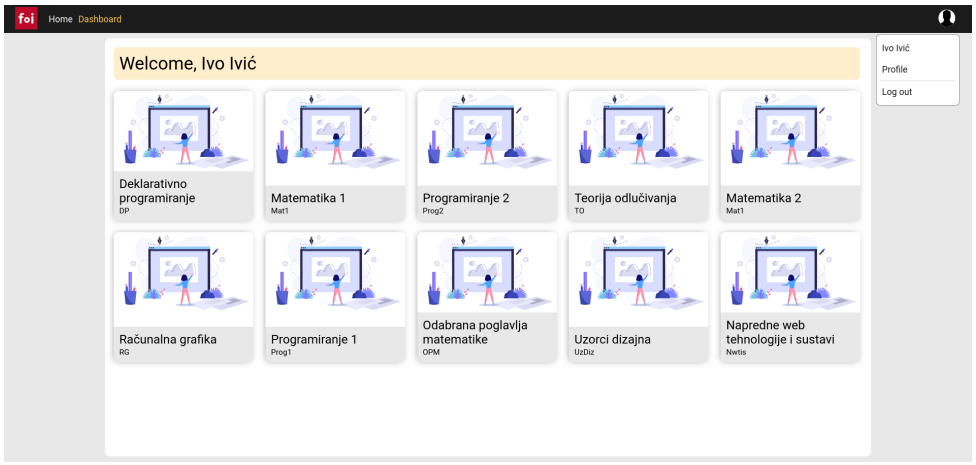
Slika 10: Ekran katedre (Izvor: Autorski rad)

Nakon ekrana koji su dostupni svima postoje i ekrani koji su dopušteni samo prijavljenim korisnicima. Kako bi im se pristupilo potrebno je prijaviti se u aplikaciju. Zanimljiva stvar je da se nakon prijave mijenja i ikona aplikacije pošto se u tom trenutku doznaje kojem fakultetu pripada korisnik. Također nakon prijave na navigaciji se pojavljuje novi gumb koji vodi na stranicu unutar koje se nalaze svi kolegiji na kojima je prijavljeni korisnik.



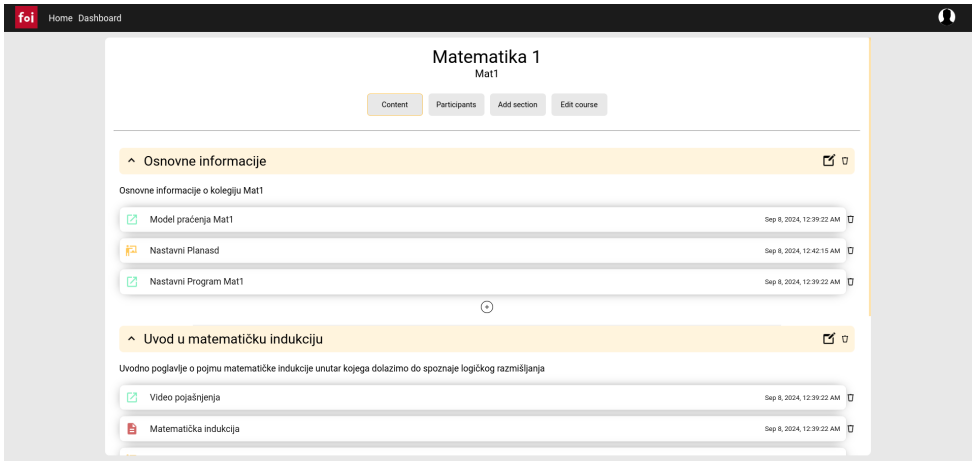
Slika 11: Ekran prijave (Izvor: Autorski rad)

Ako je prijava uspješna korisnika se vodi na stranicu s kolegijima te se na navigaciji uz novog gumba pojavljuje i slika korisnika koja se ponaša kao izbornik unutar kojega je moguće posjetiti stranicu profila i odjaviti se iz aplikacije. Klikom na bilo koji od kolegija korisnika se usmjerava na stranicu tog kolegija unutar koje se dohvaća objekt povezujuće tablice.



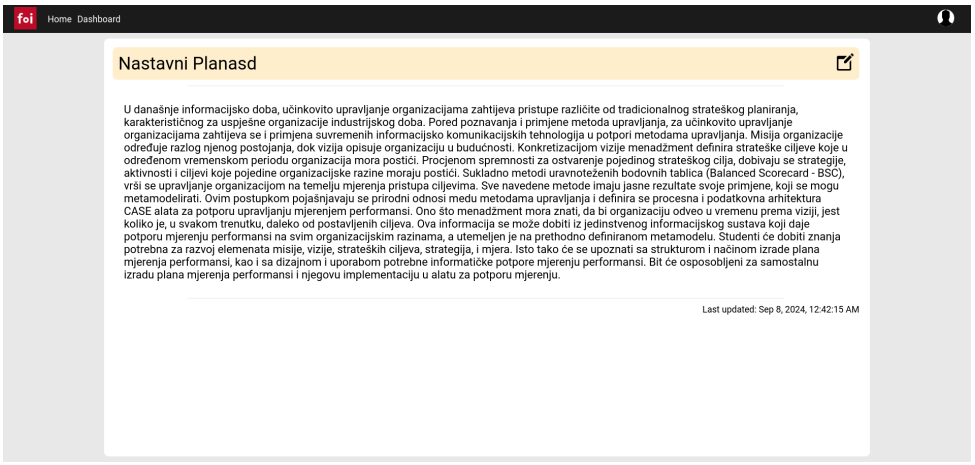
Slika 12: Ekran glavne stranice (Izvor: Autorski rad)

Na stranicama kolegija moguće je pregledati sve sekcije i sadržaje unutar sekcija koji se nalaze na tom kolegiju, a da im je postavljena vidljivost na istinu. Na ovoj stranici također drugačiji pogled ima student od djelatnika pošto djelatnik može uređivati cijeli kolegij te se na stranicu dodaje gumb za dodavanje sekcije, dodavanje sadržaja u sekciju i uređivanje sekcije. Svaki sadržaj može imati tri različite vrste: poveznica, dokument i lekcija. Klikom na poveznicu otvara se ta poveznica u novom prozoru što je isto kao i kod dokumenta, ali klikom na lekciju otvara se ekran tog sadržaja.



Slika 13: Ekran kolegija (Izvor: Autorski rad)

Ekran sadržaja je vrlo jednostavan i sastoji se samo od naslova, teksta i vremena zadnjeg uređivanja. Isto kao i na ekranu kolegija na ovom ekranu je moguće urediti sadržaj ali samo djelatnicima koji su na tom kolegiju.



Slika 14: Ekran sadržaja (Izvor: Autorski rad)

Sljedeći ekran jest profil korisnika unutar kojega se nalaze podaci o korisniku, o fakultetu korisnika i kolegijima na kojim je korisnik. Također ako je korisnik djelatnik onda se prikazuju i katedre na kojima je djelatnik. Na ovom ekranu se prikazuju i svi ostali profili korisnika, ali gumb za uređivanje se dodaje samo ako je otvoren profil trenutno prijavljenog korisnika.

Slika 15: Ekran korisničkog profila (Izvor: Autorski rad)

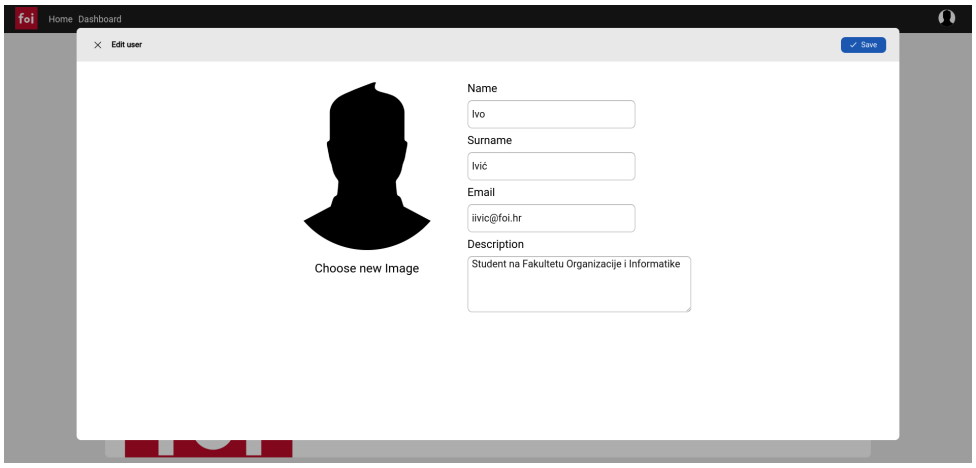
Svi ekrani za uređivanje se otvaraju u dodatnom dijalogu (engl. *modal*) koji je implementiran na vrlo zanimljiv način. Kao što je spomenuto Angular se fokusira na komponente te je iz tog razloga kreirana komponenta za dijalog koja prima podatke, alat koji se prikazuje unutar njega, naslov te dodatne ulazne vrijednosti ako su potrebne. Ova komponenta je posebno zanimljiva zato što se koristi dinamičko instanciranje komponenti koje korisnik, u ovom slučaju programer, proslijedi. To znači da se unutar HTML dokumenta navode samo obrubi dijaloga dok se unutarnja komponenta može razvijati u izolaciji. Zbog takve implementacije moguće je implementirati funkciju koja otvara dijaloge te vraća referencu na otvoreni dijalog i dopušta daljnje upravljanje nad njim kao što je slušanje na izlazne vrijednosti.

```
export function openModal<T>(
  dialog: Dialog,
  data: ModalData<T>,
): DialogRef<unknown> {
```

```
  const ref = dialog.open(ModalComponent, {
    width: '',
    height: '',
    hasBackdrop: true,
    data: data,
  });
  return ref;
}

export type ModalData<T> = {
  data: T;
  tool: { view: Type<unknown> };
  title: string;
  inputs: { [key: string]: any };
};

@Component({
  selector: 'app-modal',
  standalone: true,
  imports: [],
  templateUrl: './modal.component.html',
  styleUrl: './modal.component.scss',
})
export class ModalComponent<T> {
  readonly injectedData = inject<ModalData<T>>(DIALOG_DATA);
  public data = model<T>(this.injectedData.data);
  public tool = input(this.injectedData.tool);
  public title = input(this.injectedData.title);
  public inputs = input(this.injectedData.inputs);

  private vcr = viewChild('view', { read: ViewContainerRef });
  public query = output<T>();

  constructor() {
    afterNextRender(() => {
      if (this.vcr()) {
        let c = this.vcr().createComponent(this.tool().view);
        c.setInput('data', this.data());
        if (this.inputs()) {
          Object.keys(this.inputs()).forEach((key) => {
            c.setInput(key, this.inputs()[key]);
          });
        }
        c.instance['query'].subscribe((o: T) => {
          this.data.set(o);
        });
      }
    });
  }

  close(): void {
    this.query.emit(undefined);
  }
```

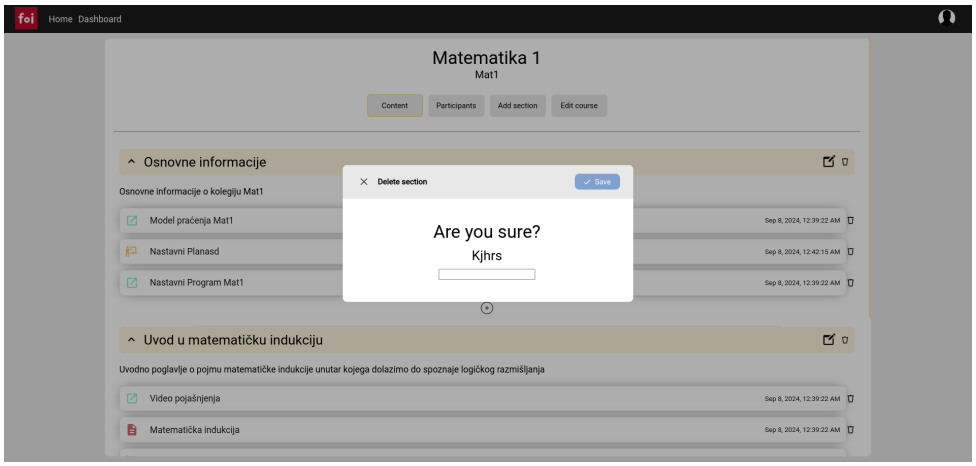
```
  save(): void {
    this.query.emit(this.data());
  }
}
```

Dijalog za uređivanje korisničkog profila je samo jedan od takvih ekrana, ali svi rade na istom principu. Ovaj način implementacije jako je zanimljivo pošto nudi ponovno iskoristivu komponentu koja radi nad bilo kojim tipom podataka. Jedino je bitno da ulazni podaci koji se proslijede u modal odgovaraju ulaznim podacima komponente koja im se prosljeđuje.



Slika 16: Ekran uređivanja korisničkog profila (Izvor: Autorski rad)

Unutar aplikacije je naravno omogućeno i brisanje podataka, ali to je akcije za koju su potrebne dodatne sigurnosne radnje poput potvrđivanja je li korisnik siguran da to želi napraviti. Unutar ove aplikacije implementirana je funkcionalnost unutar koje korisnik prvo mora unesti pet nasumičnih slova kako bi mu se omogućilo potvrđivanje brisanja podataka.



Slika 17: Ekran brisanja elemenata (Izvor: Autorski rad)

7. Zaključak

Unutar ovog diplomskog rada objašnjena je arhitektura klijent poslužitelj i navedene su njene vrste. Nadalje objašnjena je funkcijska paradigma programiranja kroz glavne koncepte koji se koriste unutar takve vrste programiranje te pomoću lambda računa na kojem se temelji. Uz osnovne koncepte također su objašnjene sve tehnologije od kojih se posebno ističu Erlang i WebAssembly koji su bili glavna motivacije izrade ovog rada.

Cilj rada bio je prikazati jednu od najmodernijih arhitekutra sustava koja se može koristiti u današnje vrijeme koja je postala jako popularna zbog svoje fleksibilnosti, ponovne iskoristivosti te prenosivosti. Implementacijom HTTP klijenta unutar WebAssembly-a omogućava se izoliranje jedne od najbitnijih funkcionalnosti današnjih aplikacija što omogućuje detaljno i jednostavno testiranje tog dijela koda. Osim izolacije slanja zahtjeva jako je bitno napomenuti da se isti kod koji se koristi unutar jedne aplikacije može koristiti i unutar više njih koje ne moraju biti samo web aplikacije već mogu biti i mobilne. To znači da više nikada nije potrebno implementirati HTTP klijenta ako dođe do potrebe implementacije novih aplikacija.

Najveći problem kod ovog sustava jest implementiranje dodatne autentifikacije pomoću osvježavajućih tokena. Pošto unutar Angulara postoje HTTP presretači moguće je presresti svaki od tih zahtjeva te ako je poslužitelj vratio pogrešku kod autentifikacije poslati zahtjev za osvježavanjem tokena te ponovno poslati klonirani zahtjev koji nije prošao prvi put. Tako nešto nije podržano unutar Rust-ove biblioteke Reqwest već je potrebno implementirati takve presretače.

Kod implementacije baze podataka mislim da bi bilo bolje da su se svi modeli još dodatno povezivali na fakultet pošto bi na taj način postojala mogućnost ponovnog korištenja katedri, kolegija, sekcija i sadržaja. To je jedan od problema unutar aplikacije pošto kada se na primjer obriše katedra, kolegije sa katedre nije više moguće dodati na novu katedru ili na neku drugu katedru što stvara takozvane viseće objekte (engl. *dangling objects*).

Nakon implementacije baze podataka primjetio sam da se moglo napraviti još nekoliko apstrakcija koje bi omogućile da postoji samo četiri funkcije pomoću kojih se izvršavaju operacije nad cijelom bazom podataka. Ideja je da se na primjer u funkciju za dohvaćanje proslijedi dodatan argument koji specificira nad kojom bazom treba raditi operaciju. Također sam primjetio da se i unutar HTTP poslužitelja mogla napraviti slična apstrakcija. To nam samo govori da je Erlang zapravo jako dobar jezik pomoću kojeg je moguće implementirati sustav koji je vrlo jednostavno za koristiti.

Ovu temu sam izabrao iz razloga što je WebAssembly nova tehnologija koja nije još potpuno istražena te sam htio testirati što je sve zapravo moguće s njom pošto nisam našao niti jedan projekt koji koristi WebAssembly kao HTTP klijenta. Još jedna motivacija jest učenje novog jezika Erlang-a, unutar kojega se koristi funkcijsko programiranje, kako bih mogao tu paradigmu primjeniti i unutar drugih jezika koji ju podržavaju.

Popis literature

[1] A. S. Tanenbaum i D. J. Wetherall, *Computer Networks*. Pearson Education, 2011.

[2] G. Coulouris, J. Dollimore, T. Kindberg i G. Blair, „Distributed Systems: Concepts and Design (5th ed.)," *Addison-Wesley*, 2011.

[3] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[4] H. Barendregt i E. Barendsen, „Introduction to lambda calculus," *Programming Methodology Group*, sv. 85, 2000.

[5] R. Toal, „Classifying Programming Languages," Loyola Marymount University, teh. izv.

[6] S. Thompson, *Type Theory and Functional Programming*. Computing Laboratory, University Kent, 1999.

[7] M. Schatten, „Funkcijsko programiranje i lambda račun," Fakultet organizacije i informatike, teh. izv.

[8] J. Armstrong, *Programming Erlang Software for a Concurrent World*. The Pragmatic Programmers, 2013.

[9] Erlang documentation, dostupno na: https://www.erlang.org/doc/readme.html.

[10] F. Hebert, *Learn You Some Erlang for Great Good!* No Starch Press, 2013., dostupno na: https://learnyousomeerlang.com/.

[11] Mnesia documentation, dostupno na: https://www.erlang.org/doc/apps/mnesia/mnesia.html.

[12] Cowboy User Guide, dostupno na: https://ninenines.eu/docs/en/cowboy/2.12/guide/.

[13] WebAssembly Core Specification, W3C, 2018, dostupno na: https://www.w3.org/TR/wasm-core-1/.

[14] WebAssembly documentation, dostupno na: https://developer.mozilla.org/en-US/docs/WebAssembly.

[15] B. Sletten, *WebAssembly The Definitive Guide*. O'Reilly Media, 2021.

[16] S. Klabnik i C. Nichols, *The Rust Programming Language, 2nd Edition*. No Starch Press, 2022.

[17] The Rust Programming Language, dostupno na: https://doc.rust-lang.org/1.8.0/book/README.html.

[18] Reqwest documentation, dostupno na: https://docs.rs/reqwest/latest/reqwest/.

[19] The 'wasm-bindgen' Guide, dostupno na: https://rustwasm.github.io/wasm-bindgen/.

[20] Angular documentation, dostupno na: https://angular.dev/.

Popis slika