# iOS aplikacija za digitalizaciju narudžbi i jelovnika u ugostiteljskim objektima

**Mecanović, Benjamin**

**Master's thesis / Diplomski rad**

**2021**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:200:052920

*Rights / Prava:* In copyright / Zaštićeno autorskim pravom.

*Download date / Datum preuzimanja:* **2024-07-14**

*Repository / Repozitorij:*

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU**
**FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA**

**Sveučilišni studij**

# iOS APLIKACIJA ZA DIGITALIZACIJU NARUDŽBI I JELOVNIKA U UGOSTITELJSKIM OBJEKTIMA

**Diplomski rad**

**Benjamin Mecanović**

**Osijek, 2021.**

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

**Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit**

**Osijek, 03.09.2021.**

**Odboru za završne i diplomske ispite**

# Imenovanje Povjerenstva za diplomski ispit

| | |
|---|---|
| **Ime i prezime studenta:** | Benjamin Mecanović |
| **Studij, smjer:** | Diplomski sveučilišni studij Računarstvo |
| **Mat. br. studenta, godina upisa:** | D-1072R, 06.10.2019. |
| **OIB studenta:** | 70081140856 |
| **Mentor:** | Izv. prof. dr. sc. Ivica Lukić |
| **Sumentor:** | |
| **Sumentor iz tvrtke:** | |
| **Predsjednik Povjerenstva:** | Izv. prof. dr. sc. Mirko Köhler |
| **Član Povjerenstva 1:** | Izv. prof. dr. sc. Ivica Lukić |
| **Član Povjerenstva 2:** | Robert Šojo |
| **Naslov diplomskog rada:** | iOS aplikacija za digitalizaciju narudžbi i jelovnika u ugostiteljskim objektima |
| **Znanstvena grana rada:** | **Informacijski sustavi (zn. polje računarstvo)** |
| **Zadatak diplomskog rada:** | Napravili iOS aplikaciju koja će gostima ugostiteljskih objekata pomoći kod kreiranja narudžbi. Korisnik u ugostiteljskom objektu za svojim stolom može pomoću aplikacije skenirati QR koj koji će mu otvoriti jelovnik te da pomoću iste može naručiti. Također kreiranje jelovnika/qr kodova bit će omogućeno korisnicima koji su prijavljeni u aplikaciju kao vlasnici nekog objekta, a djelatnici mogu vidjeti i realizirati narudžbe. |
| **Prijedlog ocjene pismenog dijela ispita (diplomskog rada):** | Izvrstan (5) |
| **Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:** | Primjena znanja stečenih na fakultetu: 3 bod/boda<br>Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda<br>Jasnoća pismenog izražavanja: 2 bod/boda<br>Razina samostalnosti: 3 razina |
| **Datum prijedloga ocjene mentora:** | 03.09.2021. |

| | |
|---|---|
| Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija: | Potpis: |
| | Datum: |

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

# IZJAVA O ORIGINALNOSTI RADA

**Osijek, 21.09.2021.**

| | |
|---|---|
| **Ime i prezime studenta:** | Benjamin Mecanović |
| **Studij**: | Diplomski sveučilišni studij Računarstvo |
| **Mat. br. studenta, godina upisa:** | D-1072R, 06.10.2019. |
| **Turnitin podudaranje [%]:** | 3% |

Ovom izjavom izjavljujem da je rad pod nazivom**: iOS aplikacija za digitalizaciju narudžbi i jelovnika u ugostiteljskim objektima**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

# SADRŽAJ

**1. UVOD ..... 1**

**1.1. Zadatak diplomskog rada ..... 1**

**2. PREGLED PODRUČJA ..... 2**

**2.1. Glovo ..... 2**

**2.2. Wolt ..... 3**

**2.3. Leggiero ..... 3**

**3. OPIS PRIMIJENJENE TEHNOLOGIJE ..... 4**

**3.1. Opis primijenjene tehnologije poslužiteljske aplikacije ..... 4**

3.1.1. TypeScript programski jezik ..... 4

3.1.2. WebStorm IDE ..... 4

3.1.3. DataGrip IDE ..... 5

3.1.4. Amazon S3 ..... 6

3.1.5. NestJS ..... 7

3.1.6. Docker ..... 12

3.1.7. Google Cloud Run ..... 13

3.1.8. Mailtrap ..... 13

**3.2. Opis primijenjene tehnologije iOS aplikacije ..... 14**

3.2.1. Swift programski jezik ..... 14

3.2.2. Xcode IDE ..... 14

3.2.3. SwiftUI ..... 15

3.2.4. Combine ..... 16

3.2.5. Resolver ..... 16

3.2.6. Arhitektura ..... 16

**4. REALIZACIJA APLIKACIJE ..... 19**

**4.1. Realizacija poslužiteljske aplikacije ..... 19**

4.1.1. Konfiguracija ..... 19

4.1.2. Pomoćni moduli ..... 21

4.1.3. Auth modul ..... 24

4.1.4. User i restaurant modul ..... 26

4.1.5. Offers module ..... 28

4.1.6. Orders module ..... 30

4.1.7. Invitations module ..... 31

4.1.8. Post razvoj ..... 32
**4.2. Realizacija iOS aplikacije ..... 35**
4.2.1. Application direktorij ..... 35
4.2.2. DI direktorij ..... 36
4.2.3. Networking direktorij ..... 37
4.2.4. Data direktorij ..... 38
4.2.5. Repositories direktorij ..... 39
4.2.6. Services direktorij ..... 40
4.2.7. Utilities direktorij ..... 42
4.2.8. Resources direktorij ..... 42
4.2.9. UI direktorij ..... 42
**5. IZGLED I NAČIN RADA APLIKACIJE ..... 45**
5.1.1. Registracija i prijava ..... 45
5.1.2. Pretraživanje ..... 49
5.1.3. Skener QR koda ..... 51
5.1.4. Korisnički račun ..... 54
5.1.5. Ponuda ..... 59
**6. ZAKLJUČAK ..... 61**
**LITERATURA ..... 62**
**ABSTRACT ..... 64**
**ŽIVOTOPIS ..... 65**
**PRILOZI ..... 66**

# 1. UVOD

Tema ovog diplomskog rada i izrada aplikacije, u nastavku *Menuely*, proizašla je kao potreba za modernizacijom ugostiteljskih obrta i metoda naručivanja, odnosno izvršavanja narudžbi u istima. U situacijama velikih gužvi unutar ugostiteljskih obrta nerijetko se dogodi da osoblje krivo ili uopće ne izvrši traženu narudžbu. Iz toga proizlaze dva glavna problema. Prvi predstavlja gubitak vremena između prihvaćanja i izvršavanja narudžbe, odnosno vrijeme koje osoblje troši na obilazak mušterija. Drugi problem predstavlja moguće preopterećenje osoblja narudžbama što će u konačnici dovesti do velikog čekanja ili krivo izvršenih narudžbi. Navedeni problemi bit će riješeni u okviru ovog diplomskog rada na način da osoblje neće imati potrebu ići do mušterija i kreirati narudžbe osobno. Narudžbe će kreirati mušterije putem aplikacije prilikom čega će osoblje na svojoj aplikaciji samo zaprimati narudžbe te ih izvršavati. Na taj način riješen je i problem preopterećenja osoblja jer će svaki član osoblja imati uvid u sve trenutno otvorene narudžbe, njihovom trenutnom statusu jesu li i od koga prihvaćene kao i točnu listu naručenih proizvoda.

U prvom poglavlju *Menuely* je uspoređen sa sličnim aplikacijama na tržištu te se govori o samoj motivaciji izrade. Drugo poglavlje sadržava detaljne informacije o tehnologijama koje su korištene za izradu iOS, odnosno poslužiteljske aplikacije. Treće poglavlje, kao i prethodno razdvojeno je na iOS i poslužiteljsku aplikaciju te je u njemu opisan način izrade aplikacije pomoću korištenih tehnologija uz predloške koda. U četvrtom poglavlju će se pomoću slika i detaljnih opisa prikazat izgled, navigacija kroz aplikaciju i funkcionalnosti iste.

## 1.1. Zadatak diplomskog rada

Zadatak diplomskog rada je izrada mobilne aplikacije na iOS platformi u okviru koje je kreirana i poslužiteljska aplikacija. Aplikacija ima mogućnost prijave korisnika i ugostiteljskih obrta kao zasebnih entiteta. Ugostiteljski obrt može kreirati, uređivati i brisati jelovnike, kategorije i proizvode koje žele imati u ponudi. Također može pozivati korisnike na rad u njihovom obrtu te uređivati podatke vezane za sami obrt. Korisnik ima mogućnosti skeniranja QR koda putem kojeg se otvara jelovnik za navedeni kod, mogućnost pregleda ugostiteljskih obrta s njihovim pripadajućim jelovnicima, kreiranje i pregled narudžbi. Korisnik također kao i ugostiteljski obrt može uređivati vlastite podatke. Ukoliko je korisnik prihvatio poziv određenog ugostiteljskog obrta, smatra se zaposlenikom tog obrta te dobiva mogućnost prihvaćanja narudžbi za navedeni obrt.

# 2. PREGLED PODRUČJA

Veliki broj srednjih i velikih ugostiteljskih obrta već ima određeni oblik aplikacije koja se odnosi na taj obrt. Većina tih aplikacija ima vrlo malo funkcionalnosti za stvarnu primjeru te su to većinom aplikacije reklamnog karaktera. Današnji korisnici su pretrpani brojem raspoloživih aplikacija te često neće imati motivaciju preuzeti novu aplikaciju za svaki ugostiteljski obrt koji posjete. *Menuely* taj problem rješava na način da pruža mogućnost registracije bilo kojeg ugostiteljskog obrta čime korisnik putem jedne aplikacije može pregledati ponudu i osnovne informacije svih registriranih ugostiteljskih obrta. Korisničko sučelje aplikacije je iznimno jednostavno što omogućava lakoću korištenja. Korisnici mogu uspoređivati ponude pojedinih ugostiteljskih obrta te odlučiti unaprijed koji im najbolje odgovara. Isto tako ugostiteljski obrti mogu pregledavati ponude drugih ugostiteljskih obrta te na taj način prilagođavati svoju ponudu kako bi ostali konkurentni na tržištu.

Većina ugostiteljskih obrta ima tiskane jelovnike unutar kojih prikazuju ponudu. Nedostatak je smanjena fleksibilnost izmijene ponuda. Digitalizacijom navedenog smanjuju se troškovi uzajamnog tiskanja jelovnika i omogućava se slobodno uređivanje ponude gdje se u nekoliko dodira ponuda ažurira svim korisnicima aplikacije. Aplikacije također nudi i mogućnost kreiranja većeg broja jelovnika koje je moguće mijenjati ovisno o pogodnostima i poslovanju ugostiteljskog obrta. Jedna od motivacija izrade ove aplikacije bila je i nagli porast popularnosti aplikacija za dostavu kao što su Glovo i Wolt. Te dvije aplikacije na elegantan način rješavaju problem dostavljanja proizvoda za veliki broj ugostiteljskih obrta, ali i dostavu potrepština iz trgovina. Sličan koncept koristi i ova aplikacija pri čemu je glavna razlika u tome što *Menuely* rješava problem narudžbi unutar lokacije obrta dok dvije spomenute aplikacije rješavaju problem kućnih dostava. Uvođenje načina plaćanja, kao što prethodno spomenute dvije aplikacije koriste, uvelike bi povećalo praktičnost i korisnost *Menuely* aplikacije. U nastavku će biti opisane aplikacije sa sličnim konceptima koje su služile kao motivacija za izradu.

## 2.1. Glovo

Prema [1], Glovo je aplikacija koja spaja korisnike, poslovanja i dostavljače kako bi dostava proizvoda unutar grada bila moguća. Aplikacija je izrađena u svrhu transformacije načina opskrbljivanja potrepštinama prilikom čega gradovi postaju pristupačniji. Korisnici naručuju

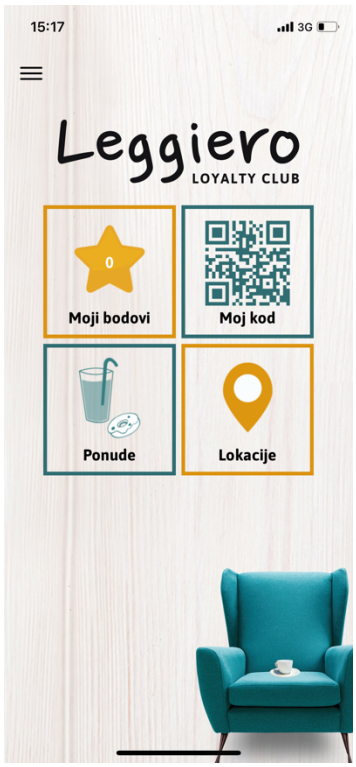
potrepštine prilikom čega dostavljači dobavljaju navedeno na adresu korisnika. Aplikacija omogućava plaćanje gotovinom i kartični način plaćanja.

## 2.2. Wolt

Wolt je također aplikacija za dostavu potrepština koja spaja korisnike, poslovanja i dostavljače. Prema [2], glavne prednosti Wolt aplikacije naspram drugih sličnih aplikacija su efikasnost u estimacijama dostave i korisničko iskustvo prilikom korištenja. Wolt osim kartičnog načina plaćanja i plaćanja gotovinom prihvaća Apple Pay i Google Pay.

## 2.3. Leggiero

Prema [3], Leggiero je aplikacija kluba lojalnosti (eng. *loyalty club*) koja omogućuje jednostavno sakupljanje i iskorištavanje bodova. Svaki korisnik aplikacije ima QR kod koji osoblje prilikom svake narudžbe može skenirati čime se korisniku pridjeljuju nagradni bodovi. Aplikacija također nudi prikaz ponude i lokacije svih ugostiteljskih obrta, Slika 2.1.




Slika 2.1 Prikaz sučelja Leggiero aplikacije

# 3. OPIS PRIMIJENJENE TEHNOLOGIJE

U ovom poglavlju su opisane tehnologije i alati korišteni za izradu iOS i poslužiteljske aplikacije.

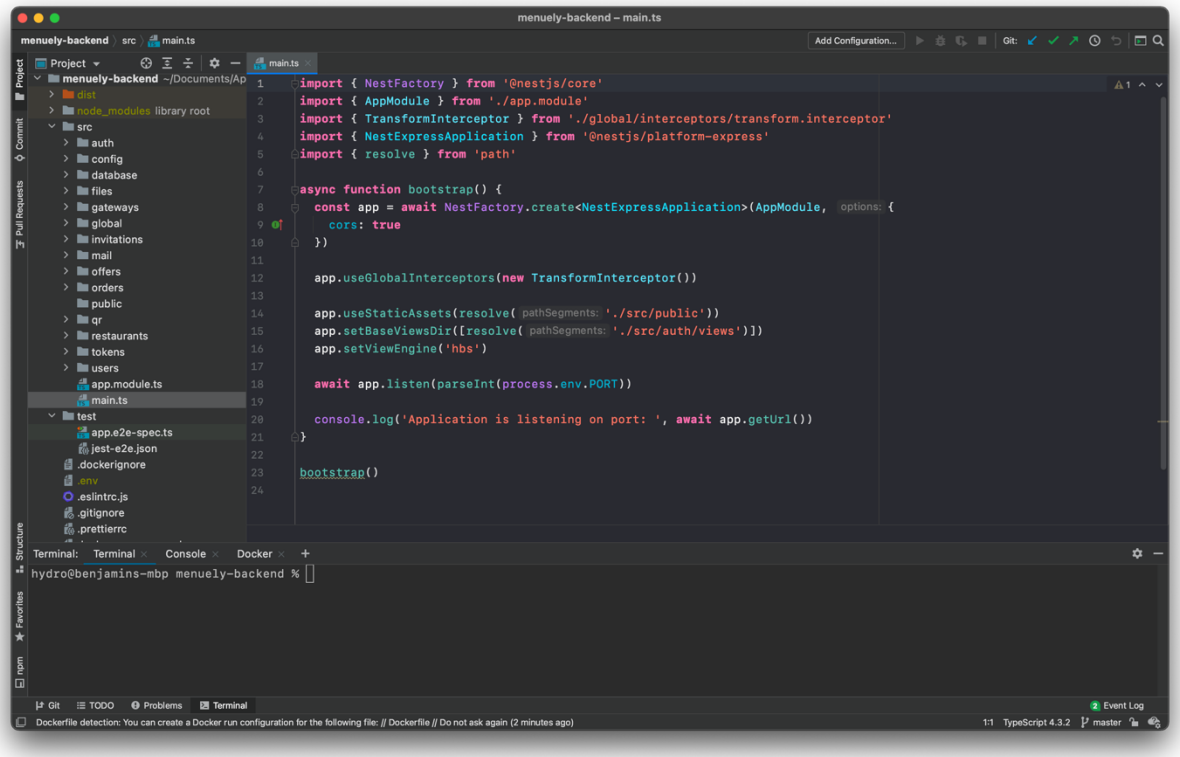
## 3.1. Opis primijenjene tehnologije poslužiteljske aplikacije

### 3.1.1. TypeScript programski jezik

Prema [4], TypeScript je zapravo JavaScript sa sintaksom za tipove podataka. Predstavlja strogi sintaktički nadskup (eng. *strict syntactical superset*) JavaScript programskog jezika što omogućava naizmjenično pisanje i kombiniranje tih dvaju jezika. TypeScript pridodaje sintaksu JavaScript programskom jeziku u svrhu užeg povezivanja s uređivačem teksta. Stoga većinu grešaka koje nastanu možemo otkriti prije samog izvođenja programa u odnosu na JavaScript. Prilikom prevođenja TypeScript kod se pretvara u JavaScript što omogućava izvođenje na svim mjestima gdje se i JavaScript može izvoditi, primjerice unutar web preglednika, Node.js ili Deno okruženja za izvođenje (eng. *runtime environment*).

### 3.1.2. WebStorm IDE

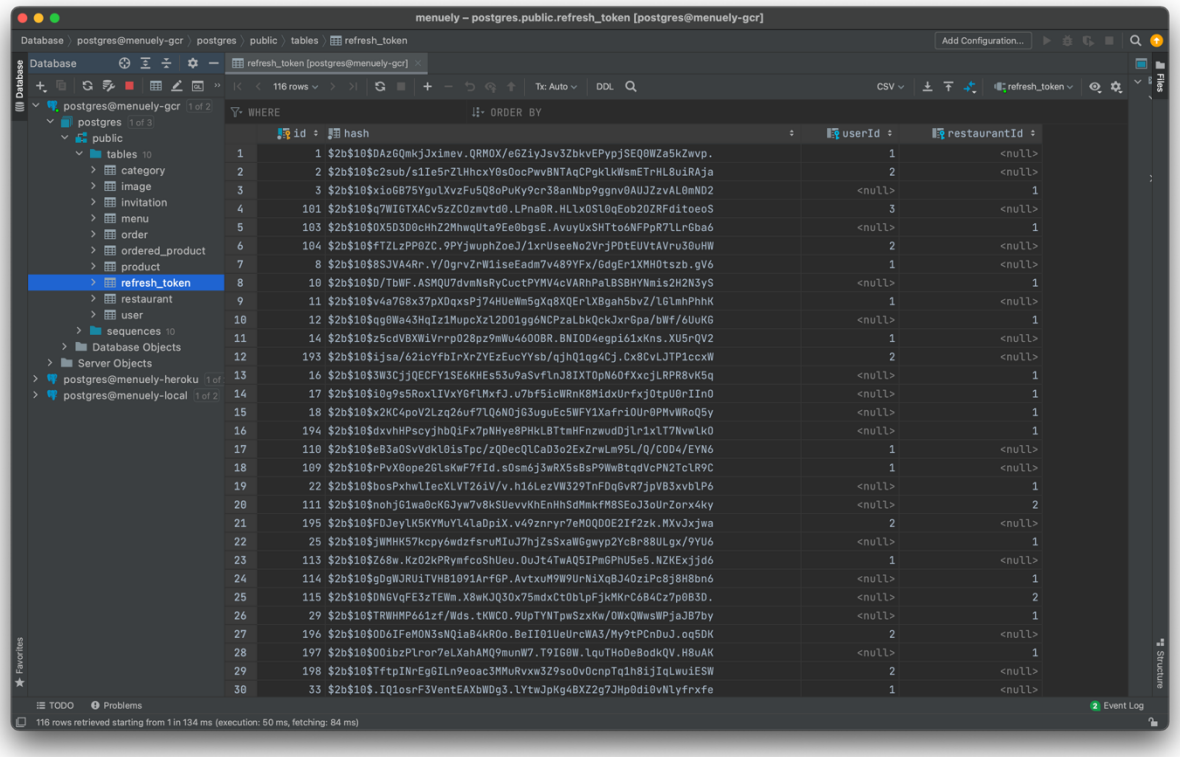
Prema [5], WebStorm IDE je integrirano razvojno okruženje za JavaScript i povezane tehnologije. Predstavlja pametan uređivač koji podržava automatsko dovršavanje koda, otkriva greške i redundancije prilikom čega daje preporuku rješenja i omogućava brzo i sigurno refaktoriranje (eng. *refactor*) koda. Unutar sebe ima ugrađene dodatne alate poput HTTP klijenta, terminala, alata za gradnju (eng. *build tools*), statičku analizu koda i kontrolu verzija kao što je GIT. Sadrži i napredan sustav pretraživanja pomoću kojega se mogu pretraživati imena datoteka, tipova ili simbola prilikom čega se prikazuje i broj pojavljivanja istih. Rad je moguće dijeliti s drugima prilikom čega sudionici u stvarnom vremenu mogu uređivati kod. Slika 3.1 prikazuje izgled integriranog okruženja.




Slika 3.1 Prikaz WebStorm IDE sučelja

### 3.1.3. DataGrip IDE

Prema [6], DataGrip predstavlja integrirano razvojno okruženje za SQL. Sadrži pametnu konzolu za upite koja pruža lokalnu povijest naredbi koje su korištene što sprječava gubitak rada. Uz pomoć efikasne navigacije između shema moguće je odlazak na bilo koju tablicu, pogled ili proceduru putem imena iste koristeći odgovarajuću naredbu ili direktno iz SQL koda. Pametno dovršavanje koda svjesno je tablica, stranih ključeva i objekata unutar baze podataka. Također podržava refaktoriranje SQL datoteka i shema prilikom čega će se sve reference određenog pojma automatski promijeniti na svim mjestima korištenja. Kao i WebStorm IDE ima integriran sustav za kontrolu verzija.




Slika 3.2 Prikaz DataGrip IDE sučelja

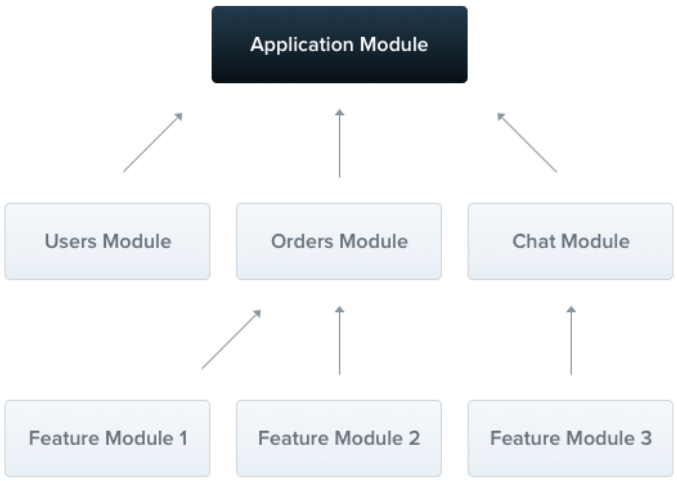
### 3.1.4. Amazon S3

Amazon Simple Storage Service ili skraćeno Amazon S3 predstavlja uslugu za pohranu objekata koji pruža skalabilnost, dostupnost podataka, sigurnost i performanse koji su vodeći u industriji [7]. Korisnici različitih veličina i industrija mogu koristiti uslugu kako bi pohranili i zaštitili bilo koju količinu podataka za niz slučajeva uporabe kao što su jezera podataka, web stranice, mobilne aplikacije, sigurnosno kopiranje i vračanje, arhiviranje i analizu velikih podataka (eng. *Big Data*). Amazon S3 nudi značajke upravljanja koje su jednostavne za upotrebu te omogućavaju visok stupanj organizacije i konfiguriranja kontrole pristupa podataka kako je navedeno u [4].

### 3.1.5. NestJS

NestJS ili skraćeno Nest je okvir (eng. *framework*) za izgradnju učinkovitih, skalabilnih Node.js poslužiteljskih aplikacija. Građen je na temelju ExpressJS okvira. Koristi progresivni (eng. *progressive*) JavaScript s punom podrškom za TypeScript te koristi kombinaciju objektno orijentiranog programiranja (eng. *Object Oriented Programming*) ili skraćeno OOP, funkcionalnog programiranja (eng. *Functional Programming*) ili FP i funkcionalnog reaktivnog programiranja (eng. *Functional Reactive Programming*) odnosno FRP. Nest je uvelike nadahnut Angular okvirom za razvoj web aplikacija te za razliku od brojnih drugih JavaScript okvira pruža arhitekturu aplikacije koja razvojnim programerima i timovima omogućava stvaranje aplikacija koje su modularne, skalabilne, lako održive i imaju visok stupanj testiranja što je navedeno u [8].

Glavni dio arhitekture Nest aplikacije predstavljaju moduli (eng. *module*). Svaka aplikacija mora sadržavati barem jedan modul, glavni modul odnosno *Application Module*. *Application Module* predstavlja početnu točku putem koje okvir stvara graf aplikacije (eng. *application graph*), što prikazuje Slika 3.3, odnosno internu strukturu podataka cijele aplikacije koja je potrebna za rješavanje veza između međusobno povezanih modula i njihovih međusobnih ovisnosti. Modul se sastoji od četiri dijela: *controllers*, *providers*, *imports* i *exports*.




Slika 3.3 Graf Nest aplikacije

Unutar *controllers* liste nalaze se svi *controlleri* za navedeni modul. Svaki pojedini *controller* je zadužen za rukovanje nadolazećih zahtjeva (eng. *request*) i vraćanje odgovora (eng. *response*). Definira rute (eng. *route*) s odgovarajućim krajnjim točkama (eng. *endpoint*) za koje je zadužen, zajedno s HTTP metodama za svaku pojedinu. *Controller* ne bi trebao sadržavati logiku unutar sebe već bi obavljanje radnji trebao delegirati na odgovarajući *provider*.

*Provider* predstavlja jedan od osnovnih koncepta u Nest okviru. Većina osnovnih Nest klasa mogu se tretirati kao *provider*, a to su: servis (eng. *service)*, repozitorij (eng. *repository*), *factory*. Glavna ideja *providera* je mogućnost injekcije ovisnosti (eng. *dependancy injection*), o čemu će nešto više biti rečeno u nastavku pod skraćenim nazivom DI. DI omogućava da objekti mogu kreirati razne međusobne veze jedni s drugima, a sami posao instanciranja (eng. *instancing*) tih veza obavlja Nest sustav izvođenja (eng. *runtime system*).

*Exports* predstavlja podskup *providera* koje pruža modul i koji moraju biti dostupni drugim modulima koji navedeni modul navedu unutar svoje *imports* liste.

*Imports* je lista vanjskih modula čiji će se *provider* objekti navedeni u *exports* listi svakog pojedinog modula koristiti u trenutnom modulu.

Osim iznimno sofisticirane arhitekture, Nest je građen uzimajući u obzir dekoratere (eng. *decorator*), novu jezičnu značajku koja je dostupna u JavaScript i TypeScript jezicima od ES2016 standarda. Dekorater predstavlja izraz koji vraća funkciju i može primiti metu (eng. *target*), naziv i *property descriptor* kao argumente. Mogu se primijeniti tako da se znak „@“ koristi kao prefiks imena samog dekoratera i postavi iznad onoga na što se želi primijeniti. Mogu se definirati nad klasama, metodama i atributima. Slika 3.4 prikazuje neke od brojnih dekoratera dostupnih unutar Nest okvira.

```
@Request(), @Req()

@Response(), @Res()

@Next()

@Session()

@Param(param?: string)

@Body(param?: string)

@Query(param?: string)

@Headers(param?: string)
```

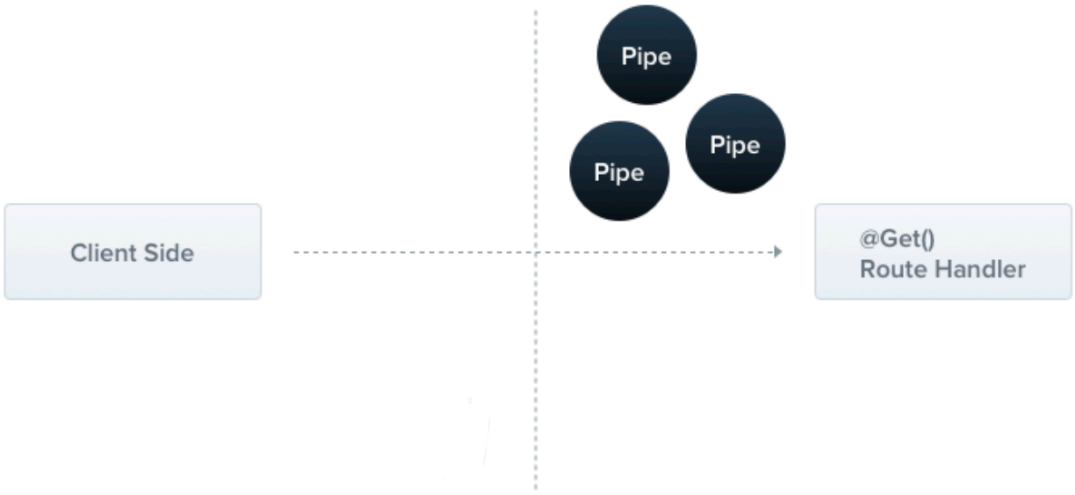
Slika 3.4 Prikaz nekih dekoratera

*Dependancy injection* jedna je od bitnih značajki NestJS okvira. Temelji se na principu inverzije ovisnosti (eng. *Inversion of Control*), odnosno IoC. IoC nalaže kako klasa ne bi trebala statički konfigurirati svoje ovisnosti već bi je trebala konfigurirati neka druga klasa izvana. Ukoliko imamo klase A, B i C, prilikom čega klasa B ovisi o klasi A, a klasa C ovisi o klasi B. Prilikom instanciranja klase C prvo je potrebno instancirati klasu A jer je njena instanca potrebna za instanciranje klase B. Nakon što je B instanciran tek tada je moguće instancirati i klasu C. Nest DI sustav nam omogućava da izbjegnemo ponavljanje tog postupka prilikom svakog korištenja instance određene klase. Korištenjem @Injected dekoratera nad *provider* klasom Nest sustavu izvođenja se govori da je on zadužen za njeno instanciranje i konfiguriranje, a samu instancu te klase možemo tada koristiti navođenjem imena i tipa podatka na željenom mjestu.

Nest okvir osim gore navedenih klasa koje su bitne za razumijevanje same arhitekture sadrži još nekoliko predložaka klasa od kojih svaka ima specifični slučaj upotrebe. Neke od tih klasa su:

- *Pipe*
- *Guard*
- *Interceptor*

*Pipe* je klasa koja je označena s @Injected dekoraterom. Trebala bi implementirati „PipeTransform“ sučelje (eng. *interface*). Postoje dva slučaja primijene. Prvi se odnosi na pretvorbu podataka kao što je pretvorba iz *string* u *integer* podatak. Drugi slučaj se odnosi na validaciju ulaznog toka podataka, koji ukoliko je ispravan prolazi dalje nepromijenjen, a u slučaju greške baca iznimku. U oba slučaja *pipe* radi nad argumentima koje obrađuje rukovatelj rutama (eng. *controller route handler*). Slika 3.5 prikazuje mjesto pozivanja *pipea*.




Slika 3.5 Mjesto pozivanja *pipea*

*Guard* je klasa koje je označena s @Injectable dekoraterom. Trebala bi implemetirati „CanActivate“ sučelje. *Guard* ima isključivo jednu odgovornost. On određuje hoće li *route handler* obraditi zahtjev ili ne u ovisnosti o određenom uvjetu kao što je dopuštenje (eng. *permission*) ili uloga. Najčešće se koristi u svrhu autorizacije i autentifikacije korisnika. Slika 3.6 prikazuje *guard* u kontekstu zahtjeva.

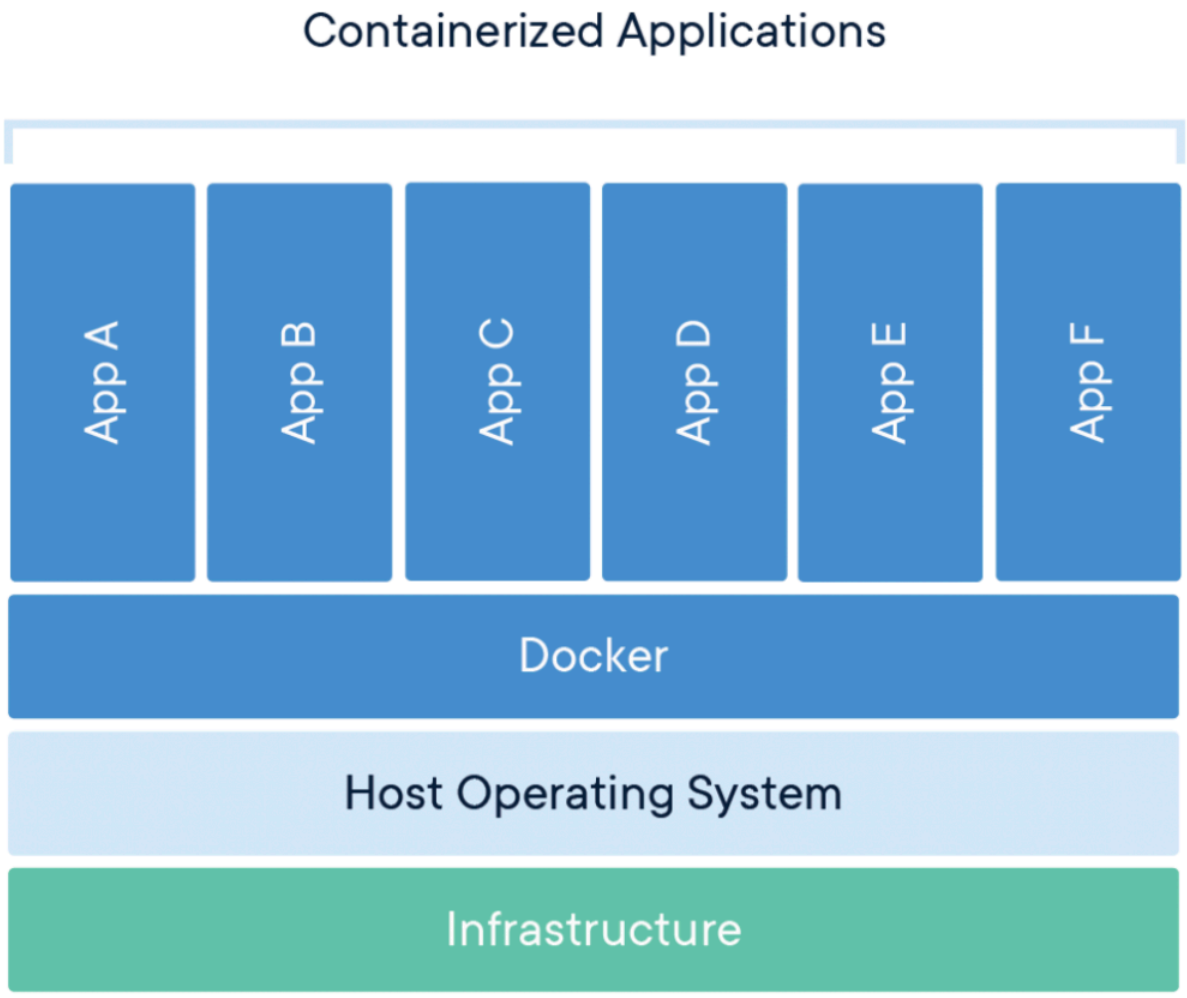
Slika 3.6 Prikaz *guarda* u kontekstu zahtjeva

*Interceptor* je klasa označena s @Injected dekoraterom. Trebala bi implementirati „NestInterceptor“ sučelje. Interceptori imaju brojne korisne primjene koje su nadahnute aspektno orijentiranim programiranjem (eng. *Aspect Oriented Programming*) ili skraćeno AOP. *Interceptor* omogućava dodavanje dodatne logike prije i nakon izvođenja određene metode. Mogu transformirati rezultat koji određena funkcija vrati. Osim rezultata funkcije mogu transformirati i iznimke koje određena funkcija baci. Pomoću njih se može proširiti ponašanje određene funkcije ili potpuno pregaziti u ovisnosti o nekom uvjetu. Slika 3.7 prikazuje *interceptor* u kontekstu zahtjeva.

Slika 3.7 Prikaz *interceptora* u kontekstu zahtjeva

### 3.1.6. Docker

Prema [9], Docker je usluga koja koristi virtualizaciju na razini operacijskog sustava prilikom čega se programska podrška isporučuje u paketima koji se nazivaju kontejneri (eng. *container*). Kontejneri su izolirani jedni od drugih, a unutar sebe sadrže vlastitu programsku podršku, biblioteke i konfiguracijske datoteke. Komunikacija između izoliranih kontejnera je moguća uspostavom komunikacijskih kanala. Svi kontejneri dijele usluge istog operacijskog sustava zbog čega koriste manje resursa u odnosu na virtualne strojeve. Docker omogućava izvođenje aplikacije u obliku kontejnera neovisno o operacijskom sustavu i infrastrukturi zbog čega je postao *de facto* standard za stvaranje i dijeljenje kontejnerskih aplikacija lokalno i u oblaku. Slika 3.8 prikazuje izgled Docker kontejnera.




Slika 3.8 Prikaz kontejnera

### 3.1.7. Google Cloud Run

Prema [10], Google Cloud Run predstavlja uslugu za razvoj i postavljanje (eng. *deployment*) visoko skalabilnih kontejnerskih aplikacija na potpuno upravljanoj *serverless* [1]platformi. Omogućava postavljanje i pokretanje kontejnerske aplikacije u nekoliko naredbi. Koristi metodu *pay-per-use* kod koje se usluga naplaćuje samo kada se aplikacija aktivno koristi. Apstrahira upravljanje infrastrukturom tako što automatski skalira računalne resurse ovisno o prometu.

### 3.1.8. Mailtrap

Mailtrap je usluga za sigurno testiranje email poruka poslanih iz razvojne i staging [2]okoline. Mailtrap hvata email unutar virtualnog *inoboxa* u svrhu testiranja i optimiziranja email kampanja prije slanja stvarnim korisnicima.

[1] *Serverless* predstavlja metodu pružanja poslužiteljskih usluga na temelju korištenja
[2] *Staging* okolina predstavlja okolinu nakon razvoja u kojoj se simulira stvarna okolina, odnosno *production* okolina, u kojoj će se programska podrška izvoditi

## 3.2. Opis primijenjene tehnologije iOS aplikacije

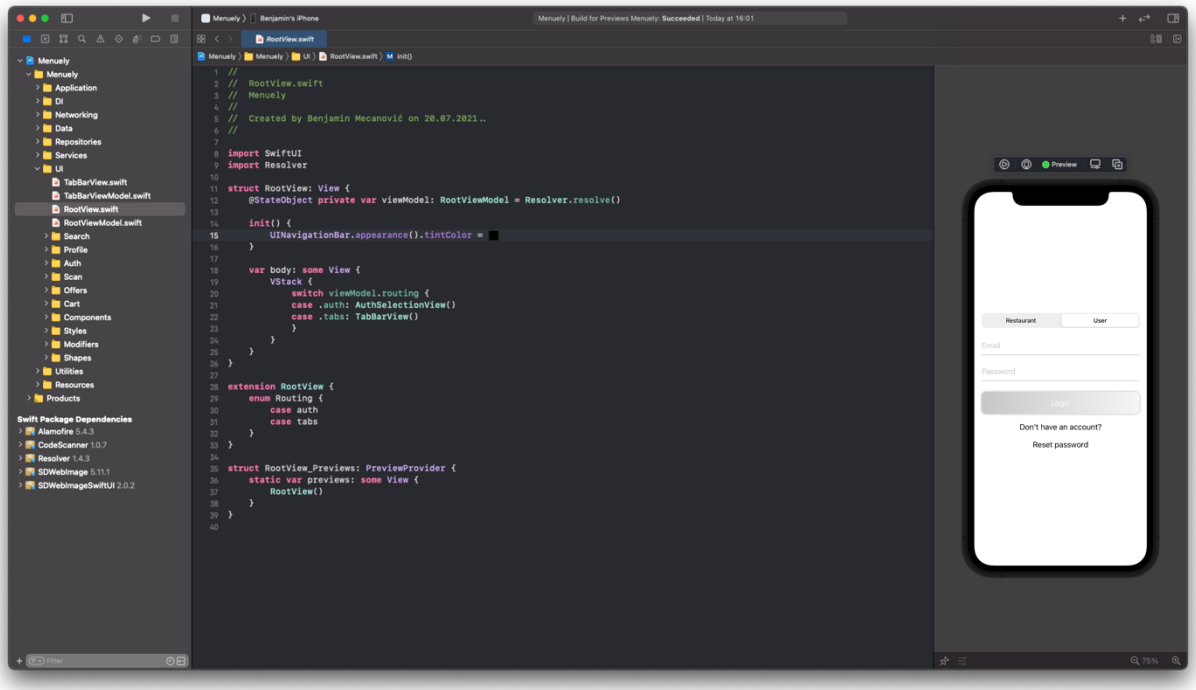
### 3.2.1. Swift programski jezik

Swift je moćan i intuitivan programski jezik za izgradnju iOS, iPadOS, macOS, tvOS i watchOS aplikacija. Jezik podržava brojne paradigme programiranja kao što su funkcionalno, protokolno i objektno orijentirano programiranje. To je moderan jezik koji ima čistu, a u isto vrijeme izražajnu sintaksu što omogućava lakoću čitanja i održavanja koda. Točka zarez je neobavezna čime se dodatno potiče nepotrebna uporaba simbola u svrhu još lakšeg čitanja. Kao što i samo ime kaže, Swift je brz i moćan što je bio jedan od glavnih zahtjeva od samog začeća jezika. Zahvaljujući visokim performansama LLVM prevoditelja, Swift kod se transformira u optimizirani izvorni kod čime se dobivaju najbolje performanse modernog sklopovlja. Jedna od glavnih značajki koje Swift nudi je sigurnost. Varijable su uvijek inicijalizirane prije korištenja, a upravljanje memorijom je automatsko. Još jedna sigurnosna značajka je da prema zadanim postavkama Swift objekti nikada ne mogu biti *nil*[3]. Ukoliko želimo koristiti podatak koji je *nil* Swift prevoditelj će nas u tome spriječiti. Gore navedeno je moguće pronaći na [11].

### 3.2.2. Xcode IDE

Prema [12], Xcode IDE je središte razvojnog iskustva na Apple platformi koje omogućava razvoj Mac, iPhone, iPad, Apple Watch i Apple TV aplikacija. Prilikom izgradnje sučelja, *Assistant editor* intuitivno prikazuje odgovarajući izvorni kod u odvojenom prozoru. Sučelje je moguće graditi jednostavnim povlačenjem elemenata sučelja iz izbornika što ga čini iznimno intuitivnim za korištenje. Prilikom pisanja pronalazi pogreške i nudi potencijalna rješenja za iste. Xcode je povezan s Apple developer web stranicom zbog čega se usluge kao što su Apple Pay mogu omogućiti direktno. Koristi napredan sustav pretraživanja pomoću kojega se mogu pretraživati imena datoteka, tipova ili simbola. Također omogućava refaktoriranje prilikom čega se mogu preimenovati sva pojavljivanja određenog imena unutar projekta. Direktno je povezan s alatom za kontrolu verzija GIT. Nakon završetka razvoja aplikacije, Xcode omogućava direktno slanje aplikacije na App Store. Slika 3.9 prikazuje izgled Xcode korisničkog sučelja.

[3] Nil, odnosno *null* u drugim programskih jezicima, govori da promatrani podatak nema pridijeljenu vrijednost




Slika 3.9 Prikaz Xcode IDE korisničkog sučelja

### 3.2.3. SwiftUI

SwiftUI je okvir za razvoj korisničkih sučelja na deklarativan (eng. *declarative*) način. Deklarativno programiranje je paradigma u kojoj se navodi krajnji izgled i ponašanje umjesto navođenja svih potrebnih koraka kako doći do krajnjeg rezultata što predstavlja imperativno programiranje. Jedna od glavnih prednosti deklarativnog programiranja je sinkronizacija pogleda (eng. *view*) zbog čega SwiftUI poglede možemo definirati kao funkciju svog trenutnog stanja (eng. *state*) gdje stanje predstavlja podatak o kojem pogled ovisi što prikazuje jednadžba 3.1.

$$view = f(state) \tag{3.1.}$$

Jednadžba 3.2. govori da se promjenom stanja automatski osvježava i pogled koji ovisi o tom stanju zbog čega je pogled uvijek u sinkronizaciji s podacima koje prikazuje. O samom osvježavanju pogleda se brine okvir.

$$view' = f(state')$$ (3.2.)

SwiftUI osim olakšanog načina razvoja nudi i mogućnost razvoja korisničkih sučelja za iOS, iPadOS, macOS, tvOS i watchOS istovremeno.

### 3.2.4. Combine

Combine je okvir za rukovanje asinkronih događaja spajanjem operatora za obradu događaja. Prema [13], okvir nudi deklarativno Swift aplikacijsko programsko sučelje (eng. *Application Programming Interface*), odnosno API, za obradu podataka tokom vremena. Podaci mogu predstavljati brojne asinkrone događaje. Combine deklarira objavljivače (eng. *publishers*) koji izlažu vrijednosti koji se mogu mijenjati tokom vremena i pretplatnike (eng. *subscribers*) koji te vrijednosti mogu osluškivati.

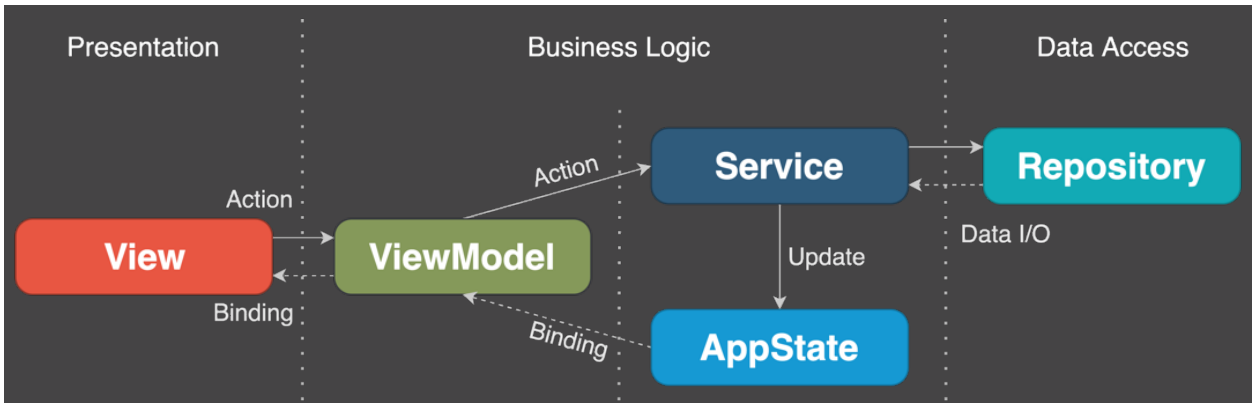
### 3.2.5. Resolver

Resolver je dependancy injection okvir pisan u Swift 5.2 programskom jeziku za iOS platformu. Omogućava registraciju svih potrebnih servisa na jednom mjestu prilikom čega se svaki pojedini servis direktno može koristiti bez dodatne konfiguracije, o samoj konfiguraciji svakog pojedinačnog servisa brine se okvir, korisnik samo navodi naziv i tip servisa koji želi koristiti. Dodatne prednosti korištenja *dependancy injection-a* obrazložene su u poglavlju 3.1.5 .

### 3.2.6. Arhitektura

Arhitektura aplikacije uvelike je nadahnuta člankom pod nazivom „Clean Architecture for SwiftUI“ koji je moguće pronaći na [14]. Slika 3.10 predstavlja konceptualni prikaz arhitekture koji je razložen na tri sloja (eng. *layer*) :

- Podatkovni sloj (eng. *Data Access Layer*)
- Sloj poslovne logike (eng. *Business Logic Layer*)
- Prezentacijski sloj (eng. *Presentation Layer*)




Slika 3.10 Konceptualni prikaz arhitekture

Podatkovni sloj se odnosi na podatke koji su potrebni aplikaciji i na njihovo dohvaćanje neovisno radi li se to putem mreže s udaljenog poslužitelja ili lokalno iz baze podataka. Unutar ovog sloja se nalaze repozitoriji. Repozitorij može biti lokalni za rad s bazom podataka ili udaljeni za rad s poslužiteljem. Jedina zadaća repozitorija je dohvaćanje i vraćanje traženog podatka. Repozitorij nema doticaja, niti smije imat doticaja s ostalim slojevima. Repozitorij bi trebao imati svoj protokol (eng. *protocol*[4]) koji nasljeđuje i putem kojeg se koristi umjesto putem konkretne instance klase repozitorija čime si olakšavamo pisanje potencijalnih testova.

Sloj poslovne logike predstavlja dio aplikacije unutar kojeg se nalaze pravila koja se brinu o razmijeni podataka između prezentacijskog i podatkovnog sloja. Pravila koja se brinu o razmijeni podataka derivirana su iz zahtjeva aplikacije. Nalaže međusobnu interakciju poslovnih objekata i provodi metode pomoću kojih se ti objekti pristupaju i ažuriraju. Sloj poslovne logike ove aplikacije se sastoji od tri dijela. AppState odnosno Store predstavlja središnji dio aplikacije tzv. jedini izvor istine (eng. *single source of truth)*, odnosno sadrži stanje cijele aplikacije. To je koncept koji je preuzet iz Redux oblikovnog obrasca. AppState se sastoji od podataka (eng. *data*), Routing i System dijela. Unutar podatkovnog dijela se nalaze podaci koji se koriste na više mjesta aplikacije. Routing predstavlja podatke koji su potrebni kako bi se premjestili s jednog na drugi zaslon. Unutar System dijela se nalaze podaci koji govore o trenutnom stanju samog sustava kao što su odlazak aplikacije u pozadinu i prikaz tipkovnice. Kao i repozitorij, servis bi trebao imati svoj protokol koji nasljeđuje. Servis dobiva zahtjeve za obavljanje određene zadaće kao što su

[4] Protokol (eng. *protocol*) je koncept sličan sučelju (eng. *interface*) u većini drugih objektno orijentiranih programskih jezika, unutar protokola se definiraju pravila koja, ukoliko se protokol naslijedi, moraju poštovati.

dohvaćanje podataka putem komunikacije s repozitorijem pri čemu podatke ne vraća direktno. Umjesto toga podaci se prosljeđuju AppStateu ili poveznici (eng. *binding*[5]*)* koje će iskoristiti ViewModel pri ćemu ViewModel postaje vlasnikom tih podataka. ViewModel predstavlja međusloj između prezentacijskog dijela i servisa i AppStatea. Svaki ViewModel je u vlasništu svog pogleda pri čemu enkapsulira poslovnu logiku lokalnu za taj pogled. Promatra promijene unutar AppStatea opskrbljujući svoj pogled ažuriranim podacima putem poveznice čime ViewModel iz perspektive pogleda predstavlja izvor istine.

Prezentacijski sloj se sastoji od SwiftUI pogleda koji ne sadržavaju poslovnu logiku. Prilikom inicijalizacije pogledu se predaje referenca na svoj odgovarajući ViewModel putem *dependancy injection-a* što ujedno predstavlja i varijablu stanja za konkretni pogled. Pogled također može imati i lokalne varijable stanja koje se uglavnom koriste za animacije. Metode se pokreću korisnikovom interakcijom ili putem događaja životnog ciklusa pogleda (eng. *lifecycle events*).

[5] Poveznica (eng. *binding*) predstavlja podatak koji je u vlasništvu nekog tipa izvan tipa unutar kojeg je deklariran

# 4. REALIZACIJA APLIKACIJE

Naredno poglavlje će detaljno opisati način izrade poslužiteljske i iOS aplikacije koristeći tehnologije koje su opisane u prethodnom poglavlju popraćeno isječcima koda koju su od posebne važnosti. Kao i prethodno poglavlje bit će podijeljeno na poslužiteljski i iOS dio.

## 4.1. Realizacija poslužiteljske aplikacije

Poslužiteljska aplikacija podijeljena je na module unutar kojeg je izolirana specifična funkcionalnost prilikom čega moduli imaju međusobnu interakciju.

### 4.1.1. Konfiguracija

Aplikacija je podijeljena na dvije okoline, *development* i *production*. *Development* okolina služi prilikom razvoja i ispravljanja grešaka pri čemu se koristi lokalna baza podataka. *Production* okolina s druge strane predstavlja okolinu aplikacije kada je aplikacija postavljena na stvarni poslužitelj, u ovom slučaju Google Cloud Run, prilikom čega je dostupna putem mreže. U svrhu konfiguracije tih dvaju okolina korištena je .env datoteka. Navedena datoteka dobro je poznata praksa u Node.js svijetu, a sadržava informacije kao što su informacije za spajanje na bazu podataka, šifre koje se koriste prilikom enkripcije određenih podataka kao što su tokeni, pristupni podaci za Amazon S3 i pristupne podatke za Mailtrap uslugu. Slika 4.1 prikazuje konfiguracijsku datoteku baze podataka unutar koje se iz .env datoteke uzimaju svi potrebni podaci.

```
1   import { registerAs } from '@nestjs/config'
2
3   export default registerAs( token: 'database', configFactory: () => ({
4     cloudSqlConnectionName: process.env.CLOUD_SQL_CONNECTION_NAME,
5     databaseHost: process.env.DATABASE_HOST,
6     databasePort: parseInt(process.env.DATABASE_PORT),
7     databaseUser: process.env.DATABASE_USER,
8     databasePassword: process.env.DATABASE_PASSWORD,
9     databaseName: process.env.DATABASE_NAME
10  }))
```

Slika 4.1 Prikaz database.config.ts datoteke

Za potrebe ove aplikacije korišten je TypeORM modul koji omogućava interakciju s bazom podataka korištenjem TypeScript programskog jezika. U svrhu konfiguracije i uspostave veze s bazom podataka TypeORM zahtjeva predaju određenih postavki koje prikazuje Slika 4.2 . Metoda createTypeOrmOptions određuje ovisno o trenutnoj okolini koje će se postavke koristiti.

```
17    private readonly productionOptions: TypeOrmModuleOptions = {
18      type: 'postgres',
19      host: '/cloudsql/' + this.databaseConfiguration.cloudSqlConnectionName,
20      username: this.databaseConfiguration.databaseUser,
21      password: this.databaseConfiguration.databasePassword,
22      database: this.databaseConfiguration.databaseName,
23      autoLoadEntities: true,
24      synchronize: true
25    }
26
27    private readonly developmentOptions: TypeOrmModuleOptions = {
28      type: 'postgres',
29      host: this.databaseConfiguration.databaseHost,
30      port: this.databaseConfiguration.databasePort,
31      username: this.databaseConfiguration.databaseUser,
32      password: this.databaseConfiguration.databasePassword,
33      database: this.databaseConfiguration.databaseName,
34      autoLoadEntities: true,
35      synchronize: true
36    }
37
38    createTypeOrmOptions(
39      connectionName?: string
40    ): Promise<TypeOrmModuleOptions> | TypeOrmModuleOptions {
41      return this.appConfiguration.nodeEnv === Environment.PRODUCTION
42        ? this.productionOptions
43        : this.developmentOptions
44    }
45  }
```

Slika 4.2 Prikaz type-orm-config.service.ts datoteke

Polaznu točku Nest aplikacije predstavlja datoteka main.ts gdje se kreira ApplicationModule koji unutar sebe sadrži sve ostale module aplikacije. Datoteka main.ts također je korištena za registraciju TransformInterceptora koji se poziva prije nego se odgovor određenog zahtjeva vrati klijentu. TransformInterceptor „presreće“ odgovor pri čemu ga naknadno formatira pridodajući informacije kao što su statusCode i message u slučaju greške. U svrhu izrade email predložaka

koristi se Handlebars *template engine*. Handebars se također registrira unutar main.ts datoteke zajedno s lokacijom direktorija unutar kojeg će se predlošci nalaziti. Gore navedeno prikazuje Slika 4.3.

```
async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule, options: {
    cors: true
  })

  app.useGlobalInterceptors(new TransformInterceptor())

  app.useStaticAssets(resolve( pathSegments: './src/public'))
  app.setBaseViewsDir([resolve( pathSegments: './src/auth/views')])
  app.setViewEngine('hbs')

  await app.listen(parseInt(process.env.PORT))

  console.log('Application is listening on port: ', await app.getUrl())
}

bootstrap()
```

Slika 4.3 Prikaz main.ts datoteke

### 4.1.2. Pomoćni moduli

Pomoćni moduli se odnose na module koji pomažu ostalim modulima tako što pružaju određeni servis. Kao i ostali moduli enkapsuliraju specifičnu funkcionalnost, a glavna razlika je što ne pružaju krajnje točke putem kojih se klijent može spojiti. Pomoćni moduli korišteni u aplikaciji su:

- Tokens module
- Files module
- Mail module

Tokens module se koristi za kreiranje *access tokena*[6], *refresh tokena* [7]i *verification tokena*[8]. Komunicira s repozitorijem koji omogućava stvaranje i brisanje tokena. Servis ovog modula pruža metode kao što su signToken koja kao argumente prihvaća payload i jwtSignType. Payload se odnosi na podatak koje će se enkriptirati unutar tokena, najčešće korisnikov id, dok jwtSignType nalaže tip tokena koji je potrebno napraviti.

```
23    signToken(payload: JwtPayload, jwtSignType: JwtSignType): string {
24      let options: JwtSignOptions
25
26      if (jwtSignType === JwtSignType.ACCESS) {
27        options = {
28          secret: this.tokensConfiguration.accessTokenSecret,
29          expiresIn: this.tokensConfiguration.accessTokenExpiration
30        }
31      }
32
33      if (jwtSignType === JwtSignType.REFRESH) {
34        options = {
35          secret: this.tokensConfiguration.refreshTokenSecret,
36          expiresIn: this.tokensConfiguration.refreshTokenExpiration
37        }
38      }
39
40      if (jwtSignType === JwtSignType.VERIFICATION) {
41        options = {
42          secret: this.tokensConfiguration.verificationTokenSecret,
43          expiresIn: this.tokensConfiguration.verificationTokenExpiration
44        }
45      }
46
47      const token = this.jwtService.sign(payload, options)
48
49      return token
50    }
```

Slika 4.4 Definicija signToken metode

[6] Access token predstavlja token s određenim vremenom trajanja koji poslužitelj vraća korisniku nakon uspješne prijave

[7] Refresh token predstavlja token koji za razliku od *access tokena* ima dulje vrijeme trajanja uz pomoć kojeg poslužitelj korisniku može pridijeliti novi *access token*.

[8] Verification token predstavlja token koji se koristi za verifikaciju email adrese korisničkog računa

Files module služi za prijenos, kreiranje i brisanje datoteka, u ovom slučaju slika, korištenjem Amazon S3 usluge, kao i kreiranje entiteta prenesenih datoteka koji se mogu zapisati u bazi podataka. Slika 4.5 prikazuje uploadFile metodu koja prihvaća imageFileParams parametar uz pomoć kojeg se datoteka prenosi na Amazon S3 koji kao rezultat vraća uploadResult koji sadrži URL do udaljenog resursa i njegov naziv koji se uz pomoć repozitorija koriste za kreiranje instance Image entiteta u bazi podataka.

```
async uploadImage(imageFileParams: ImageFileParams): Promise<Image> {
  const { name, mime, buffer } = imageFileParams

  if (!name || !mime || !buffer) {
    throw new BadRequestException( objectOrError: 'ImageFileParams can not be empty')
  }

  if (!Object.values<string>(ImageMimeType).includes(mime)) {
    throw new UnsupportedMediaTypeException()
  }

  const params = {
    Bucket: this.filesConfiguration.awsS3BucketName,
    Body: buffer,
    Key: `${uuid()}-${name}`,
    ACL: 'public-read'
  }

  const uploadResult = await this.s3.upload(params).promise()

  if (!uploadResult) {
    throw new ConflictException( objectOrError: 'File upload failed')
  }

  const createImageParams: CreateImageParams = {
    url: uploadResult.Location,
    name: uploadResult.Key
  }

  return this.imagesRepository.createImage(createImageParams)
}
```

Slika 4.5 Definicija upoadFile metode

Mail module se koristi za slanje email poruka korisnicima putem Mailtrap usluge. Sadržaj poruka kreiran je korištenjem Handlebars *template enginea*. Modul omogućava slanje poruka za

resetiranje zaporke, verifikaciju i liste generiranih QR kodova. Slika 4.6 prikazuje metodu za slanje poruke za resetiranje zaporke.

```
11    async sendResetPassword(
12      resetPasswordEmailParams: ResetPasswordEmailParams
13    ): Promise<void> {
14      const { email, name, password } = resetPasswordEmailParams
15
16      await this.mailerService.sendMail( sendMailOptions: {
17        to: email,
18        subject: 'Password Reset - Menuely Support',
19        template: './reset-password',
20        context: {
21          name,
22          password
23        }
24      })
```

Slika 4.6 Definicija sendResetPassword metode

### 4.1.3. Auth modul

Svrha Auth modula je registracija, autentikacija, autorizacija i verifikacija korisnika i ugostiteljskih obrta unutar sustava. AuthController sadrži metode koje se pozivaju ovisno o krajnjoj točki kojoj je pristupljeno. Bitno je naglasiti da AuthController svojim rutama dodaje prefiks „auth“, a popis svih ruta prikazuje Tablica 4.1.

Tablica 4.1 Popis AuthController ruta

| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| registerUser | /register/user | POST |
| loginUser | /login/user | POST |
| registerRestaurant | /register/restaurant | POST |
| loginRestaurant | /login/restaurant | POST |
| refreshToken | /refresh-token | POST |
| resetUserPassword | /reset-password/user | POST |
| resetRestaurantPassword | /reset-password/restaurant | POST |
| verifyUser | /verify/user | GET |
| verifyRestaurant | /verify/restaurant | GET |
| resendUserVerification | /resend-verification/user/:id | GET |
| resendRestaurantVerification | /resend-verification/restaurant/:id | GET |
| logout | /logout | DELETE |

Unutar modula definirani su *guardovi* koje koriste ostali moduli prilikom pristupa svojih ruta u svrhu zaštite. Kako bi *guard* štitio pristup ovisno o nekom uvjetu potrebno je definirati specifičnu strategiju (eng. *strategy*). U svrhu kreiranja strategija koristi se biblioteka Passport. Slika 4.7 prikazuje definiciju UserAccessJwtStrategy strategije. Bitno je naglasiti da UserAccessJwtStrategy nasljeđuje PassportStrategy sučelje. Sučelje nalaže definiranje metode validate. Slika 4.8 prikazuje definiciju UserAccessJwtAuthGuard *guarda* gdje se tipu AuthGuard kao parametar predaje željena strategija. Pri pozivu *guarda* pokreće se predana strategija koja u ovom konkretnom slučaju automatski provjerava postojanje i ispravnost *access tokena* . Ukoliko je token ispravan poziva se metoda validate unutar koje se može definirati naknadna logika, kao što je pronalazak korisnika vezanog za pročitani token. Metoda validate podatak vraća u *request*.

```
@Injectable()
export class UserAccessJwtStrategy extends PassportStrategy(
  Strategy,
  StrategyType.USER_ACCESS_JWT
) {
  constructor(
    private usersService: UsersService,
    @Inject(tokensConfig.KEY)
    private readonly tokensConfiguration: ConfigType<typeof tokensConfig>
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: tokensConfiguration.accessTokenSecret
    })
  }

  async validate(payload: JwtPayload): Promise<User> {
    const { id } = payload
    const user = await this.usersService.findUser( searchCriteria: { id })

    return user
  }
}
```

Slika 4.7 Definicija UserAccessJwtStrategy strategije

```
@Injectable()
export class UserAccessJwtAuthGuard extends AuthGuard(
  StrategyType.USER_ACCESS_JWT
) {}
```

Slika 4.8 Definicija UserAccessJwtAuthGuard *guarda*

### 4.1.4. User i restaurant modul

User i restaurant moduli služe za obavljanje radnji specifičnih za svoje odgovarajuće entitete. Rute obrađuje UsersController odnosno RestaurantsController pri čemu postavljaju prefiks „users“ odnosno „restaurants“. Tablica 4.2 i Tablica 4.3 prikazuju popis svih dostupnih ruta unutar navedena dva modula.

Tablica 4.2 Popis UsersController ruta

| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| getAuthenticatedUserProfile | /me | GET |
| getUser | /:id | GET |
| getUsers | / | GET |
| updateUserProfile | /me/profile | PATCH |
| updateUserPassword | /me/password | PATCH |
| updateUserEmai | /me/email | PATCH |
| updateUserImage | /me/image | PATCH |
| quitEmployer | /me/quit-employer | PATCH |
| deleteUser | /me | DELETE |

Tablica 4.3 Popis RestaurantsController ruta

| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| getAuthenticatedRestaurantProfile | /me | GET |
| getRestaurant | /:id | GET |
| getRestaurants | / | GET |
| getEmployees | /me/employees | GET |
| updateRestaurantProfile | /me/profile | PATCH |
| updateRestaurantPassword | /me/password | PATCH |
| updateRestaurantEmail | /me/email | PATCH |
| updateRestaurantImage | /me/image | PATCH |
| fireEmployee | /me/fire-employee | PATCH |
| deleteRestaurant | /me | DELETE |

U poglavlju 4.1.3 prikazana je definicija UserAccessJwtAuthGuard *guarda* koji se konkretno koristi nad getAuthenticatedUserProfile metodom za rukovanje rute kao što prikazuje Slika 4.9 .

```
32    @Get( path: 'me')
33    @UseGuards(UserAccessJwtAuthGuard)
34    getAuthenticatedUserProfile(
35      @AuthenticatedEntity() user: User
36    ): UserProfileResponseDto {
37      return this.usersService.formatUserProfileResponse(user)
38    }
```

Slika 4.9 Definicija getAuthenticatedUserProfile metode

### 4.1.5. Offers module

Offers module sadrži CRUD [9]operacije za jelovnike, kategorije i proizvode. Rutama rukuje OffersController koji svim rutama pridjeljuje prefiks „offers“. Tablica 4.4 prikazuje sve ruta unutar modula. OffersController komunicira s OffersService servisom koji uz pomoć repozitorija za jelovnike, kategorije i proizvode obavlja operacije s bazom podataka.

[9] CRUD skraćenica od engleskih riječi *create*, *read*, *update*, *delete*

Tablica 4.4 Prikaz OffersController ruta

| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| getMenu | menus/:id | GET |
| getsMenus | /menus | GET |
| createMenu | /menus | POST |
| updateMenu | /menus/:id | PATCH |
| deleteMenu | /menus/:id | DELETE |
| getCategory | /categories/:id | GET |
| getCategories | /cateogires | GET |
| createCategory | /cateogires | POST |
| updateCategory | /categoires/:id | PATCH |
| deleteCategory | /cateogires/:id | DELETE |
| getProduct | /products/:id | GET |
| getProducts | /products | GET |
| createProduct | /products | POST |
| updateProduct | /products/:id | PATCH |
| deleteProduct | /products/:id | DELETE |

```
@EntityRepository(Menu)
export class MenusRepository extends Repository<Menu> {
  async findMenu(id: number): Promise<Menu> {
    const menu = await this.findOne(id, options: { relations: ['qrCodeImages'] })

    return menu
  }

  async findMenus(restaurantId: number): Promise<Menu[]> {
    const query = this.createQueryBuilder( alias: 'menu')

    if (restaurantId) {
      query.where( where: 'menu.restaurantId = :restaurantId', parameters: { restaurantId })
    }

    const menus = await query
      .leftJoinAndSelect( property: 'menu.qrCodeImages', alias: 'qrCodeImages')
      .leftJoinAndSelect( property: 'menu.restaurant', alias: 'restaurant')
      .getMany()

    return menus
  }

  createMenu(createMenuParams: CreateMenuParams): Menu {
    const { name, description, currency, restaurantId } = createMenuParams

    const menu = new Menu()
    menu.name = name
    menu.description = description
    menu.currency = currency
    menu.restaurantId = restaurantId

    return menu
  }
```

Slika 4.10 Prikaz MenusRepository repozitorija

### 4.1.6. Orders module

Modul za dohvaćanje, pravljenje i prihvaćanje narudžbi. OrdersController je zadužen za rukovanje rutama kojima kao prefiks postavlja „orders“ . Sve rute modula prikazuje Tablica 4.5. Rute su zaštićene UserAccessJwtAuthGuard *guardom*. Na rutama kao što su getRestaurantOrder i getRestaurantOrders dodatno se provjerava ima li korisnik koji pristupa resursu prava na njihov pristup, odnosno je li korisnik koji pristupa narudžbama ugostiteljskog obrta postavljen kao zaposlenik u istom.

Tablica 4.5 Prikaz OrdersController ruta

| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| getUserOrder | :id/user | GET |
| getUserOrders | /user | GET |
| getRestaurantOrder | :id/restaurant | GET |
| getRestaurantOrders | /restaurant | GET |
| createOrder | /create | POST |
| acceptOrder | /accept | POST |

### 4.1.7. Invitations module

Invitations module sadrži rute za kreiranje, prihvaćanje, pravljenje, odbijanje i uklanjanje pozivnica za rad u ugostiteljskom obrtu. Rute prikazuje Tablica 4.6. Rukovanje rutama obavlja InvitationsController koji svima dodaje prefiks „invitations“. Ruta getInvitations koristi AccessJwtAuthGuard koji predstavlja kombinaciju UserAccessJwtAuthGuard i RestaurantAccessJwtAuthGuard *guardova*. AccessJwtAuthGuard također koristi i ruta rejectInvitation koja u slučaju korisnika predstavlja rutu za odbijanje poziva, a u slučaju ugostitelja rutu za uklanjanje poziva.

Tablica 4.6 Prikaz InvitationsController ruta

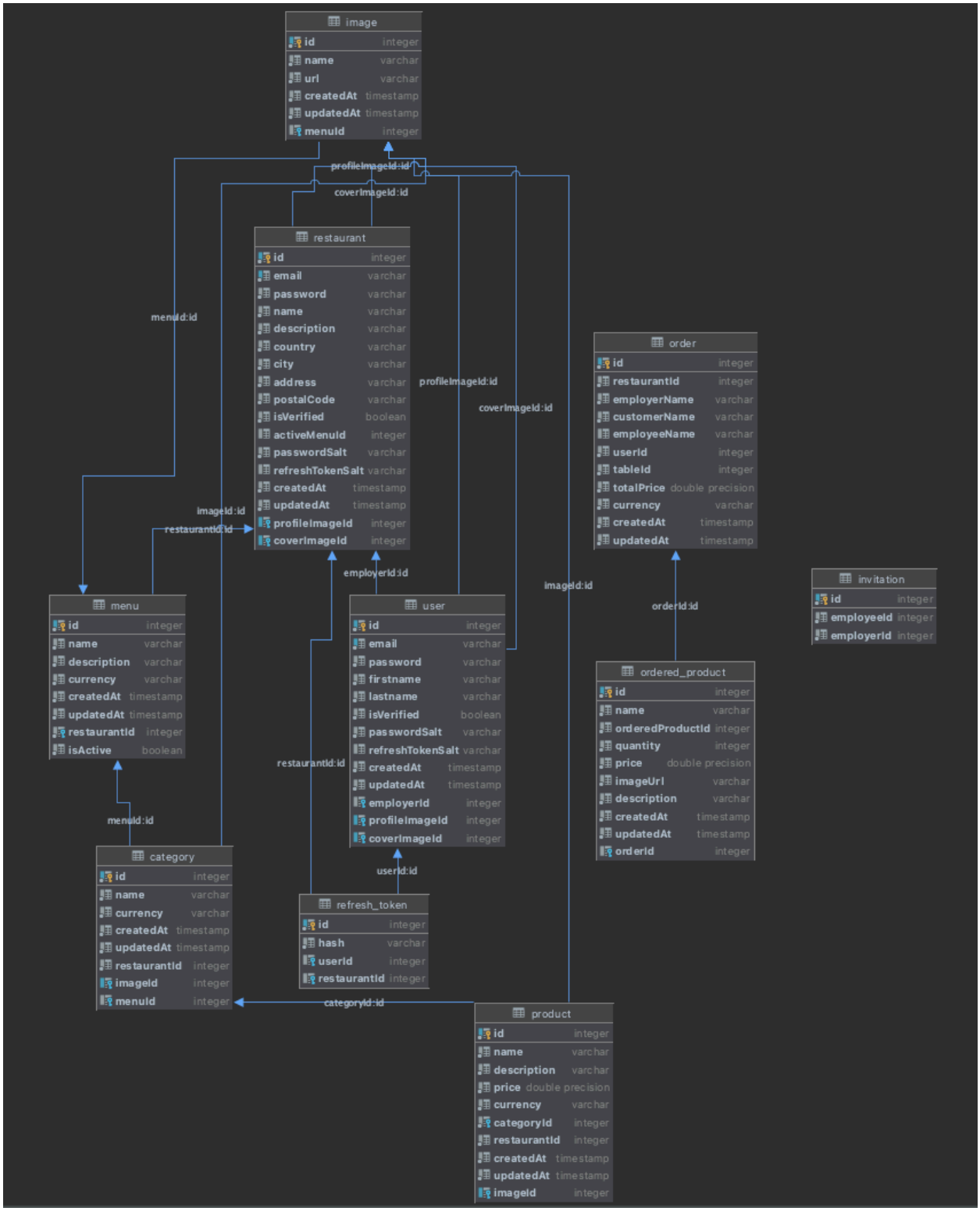
| Naziv rute | Krajnja točka | HTTP metoda |
|---|---|---|
| getInvitations | / | GET |
| acceptInvitation | /accept | POST |
| createInvitation | /create | POST |
| rejectInvitation | /reject | POST |

```
5   @Injectable()
6   export class AccessJwtAuthGuard extends AuthGuard( type: [
7     StrategyType.USER_ACCESS_JWT,
8     StrategyType.RESTAURANT_ACCESS_JWT
9   ]) {}
```

Slika 4.11 Definicija AccessJwtAuthGuard *guarda*

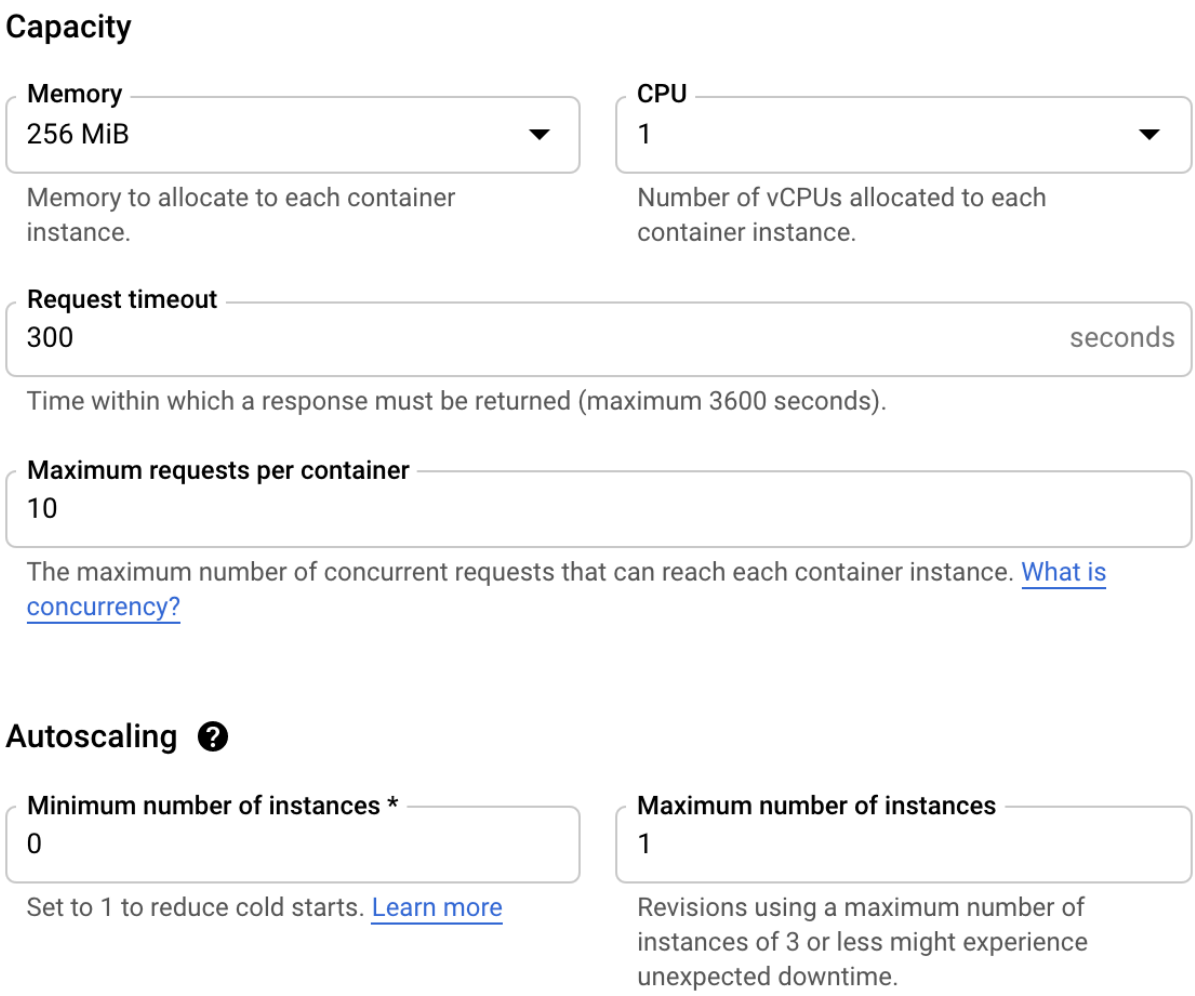
### 4.1.8. Post razvoj

Nakon razvoja aplikacije postignuta je struktura baze podataka s odgovarajućim vezama među entitetima koju prikazuje Slika 4.12 .



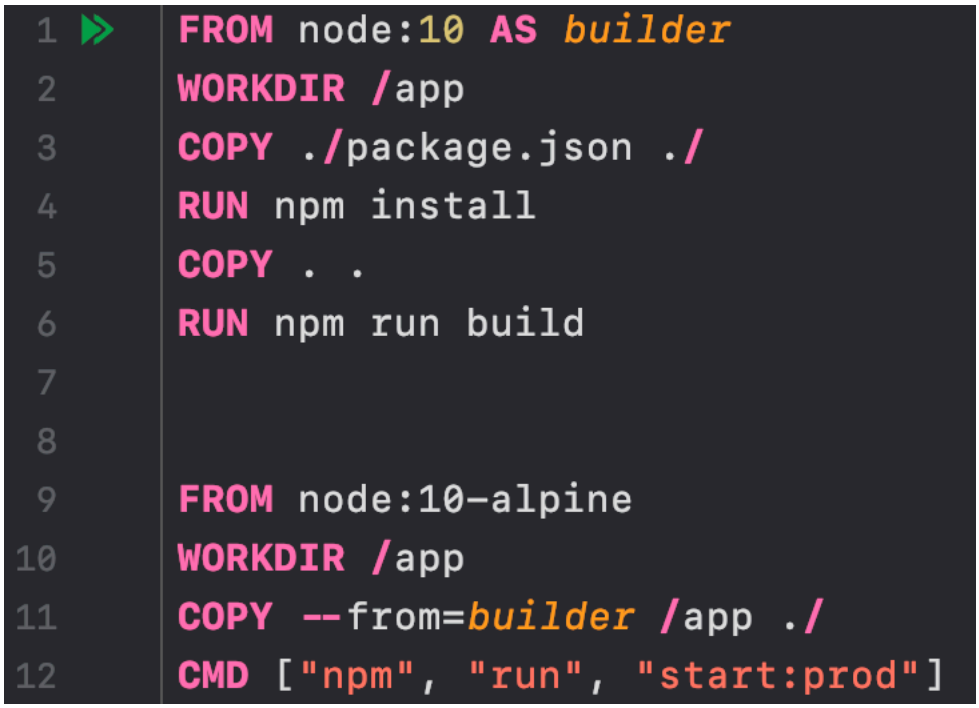

Slika 4.12 UML dijagram baze podataka

Prethodno prikazani UML dijagram kreiran je uz pomoć DataGrip Diagrams alata. Nadalje, aplikaciju je potrebno postaviti na Google Cloud Run pri čemu je prethodno trebalo kreirati korisnički račun. Unutar Google Cloud Run konzole potrebno je kreirati projekt kojem pridjeljujemo količinu računalnih resursa koje će projekt imati na raspolaganju. Slika 4.13 prikazuje neke od odabranih računalnih resursa kao što su memorija i broj virtualnih procesorskih jezgri.




Slika 4.13 Prikaz korištenih GCR računalnih resursa

Unutar korijenskog direktorija projekta definirana je Dockerfile datoteka. Datoteka sadrži instrukcije potrebne za kreiranje Docker kontejnerske slike koja će sadržavati aplikaciju zajedno sa svim potrebnim ovisnostima koje aplikacija zahtjeva. Sadržaj Dockerfile datoteke prikazuje Slika 4.14.




Slika 4.14 Sadržaj Dockerfile datoteke

Za potrebe pokretanja Dockerfile datoteke potrebno je instalirati Docker Desktop aplikaciju koja usporedno instalira i Docker CLI[10]. U svrhu postavljanja kontejnerske slike na Google Cloud Run instaliran je gcloud CLI koji je potrebno konfigurirati s prethodno kreiranim korisničkim računom. Nakon konfiguracije korištenjem jedne komande moguće je stvoriti i prenijeti kontejnersku sliku na Google Cloud Run. Komandu prikazuje Slika 4.15 . Nakon prijenosa aplikacija je dostupna za postavljanje putem Google Cloud Run konzole. Prenesenu kontejnersku sliku naknadno je potrebno samo objaviti čime poslužiteljska aplikacija postaje dostupna za korištenje.

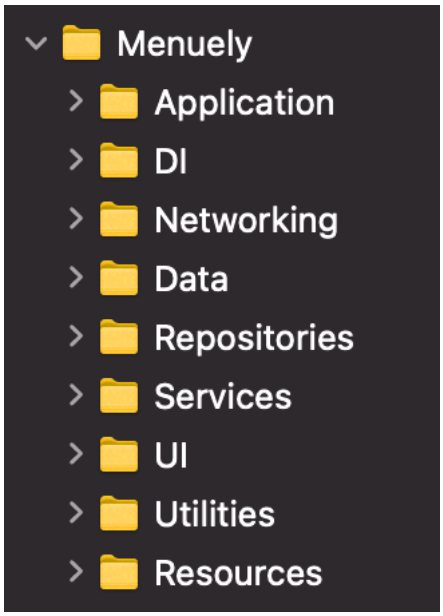
```
gcloud builds submit --tag gcr.io/menuely/menuely
```

Slika 4.15 Komanda za stvaranje i prijenos kontejnerske slike

[10] CLI skraćenica engleskog izraza za tekstualno sučelje (eng. *Command Line Interface*)

## 4.2. Realizacija iOS aplikacije

Aplikacijska struktura podijeljena je na nekoliko direktorija koji unutar sebe grupiranju dijelove aplikacije sa sličnim funkcionalnostima. Svaki direktorij predstavlja značenjem odvojenu cjelinu. Ukupno se aplikacija sastoji od devet direktorija koji će bit opisani u narednim poglavljima.




Slika 4.16 Struktura aplikacije

### 4.2.1. Application direktorij

Unutar Application direktorija nalaze se datoteke koje sadrže funkcionalnosti vezane za aplikaciju kao cjelinu. MenuelyApp datoteka predstavlja strukturu koja služi kao polazna točka iOS aplikacije građene pomoću SwiftUI okvira. Unutar nje se inicijalizira početni zaslon aplikacije RootView kao što Slika 4.17 prikazuje. Sadrži i ApplicationEventsHandler datoteku koja je zadužena za promatranje sustavnih događaja (eng. *system events*) kao što su odlazak aplikacije u pozadinu ili otvaranje tipkovnice. Promijene koje se dogode spremaju se u AppState. Unutar ovog direktorija nalazi se i AppState čija je svrha objašnjena u poglavlju 3.2.6.

```
@main
struct MenuelyApp: App {
    @Injected private var applicationEventsHandler: ApplicationEventsHandler

    var body: some Scene {
        WindowGroup {
            RootView()
                .modifier(PopoversViewModifier())
                .modifier(RootViewAppearance())

        }
    }
}
```

Slika 4.17 Definicija MenuelyApp strukture

### 4.2.2. DI direktorij

DI direktorij sadrži datoteke unutar kojih se registriraju i konfiguriraju svi servis, repozitorij, *view model* i *utility* tipovi za korištenje putem *dependancy injection-a*. U svrhu registriranja tipova i obavljanja *dependancy injection-a* koristi se Resolver biblioteka. Slika 4.18 prikazuje registraciju servis tipova. Bitno je naglasiti da se tipovi registriraju putem protokola, a ne kao konkretni tipovi, gdje kao primjer UsersService() predstavlja instancu tipa koja se ključnom riječi „as“ pretvara u protokol UsersServicing koji UsersService nasljeđuje. Takav način registracije tipova omogućava primjerice kreiranje UsersMockService tipa koji će se koristiti u svrhu testiranja i ukoliko taj tip također nasljeđuje UsersServicing protokol ostatak koda nije potrebno mijenjati kako bi se novi tip s testnim podacima koristio.

```
extension Resolver {
    public static func registerServices() {
        register { AuthRequestInterceptor() as RequestInterceptor }
        register { NetworkClient(session: AF, interceptor: resolve()) as Networking }
        register { UsersService() as UsersServicing }
        register { RestaurantsService() as RestaurantsServicing }
        register { AuthService() as AuthServicing }
        register { MenusService() as MenusServicing }
        register { CategoriesService() as CategoriesServicing }
        register { ProductsService() as ProductsServicing }
        register { CartService(appState: resolve()) as CartServicing }.scope(.shared)
        register { OrdersService() as OrdersServicing }
        register { InvitationsService() as InvitationsServicing }
    }
}
```

Slika 4.18 Registracija servis tipova

### 4.2.3. Networking direktorij

Komunikaciju s udaljenim poslužiteljem omogućavaju datoteke unutar Networking direktorija. NetworkClient, koji prikazuje Slika 4.19, izgrađen je pomoću Alamofire biblioteke. Alamofire je jedna od najkorištenijih biblioteka na iOS platformi koja nudi brojne *out-of-the-box* funkcionalnosti kao što su validacija odgovora. NetworkClient predstavlja servis koji kao parametre prihvaća Alamofire Session i interceptor atribute. Interceptor predstavlja klasu koja „presreće“ odgovore poslužitelja. Ukoliko je odgovor došao s pogreškom koja ima statusni kod 401 koji predstavlja da je *access token* trenutnog korisnika istekao, interceptor pokreće metodu za obnavljanje *access tokena*. NetworkClient definira i request metodu koja predstavlja osnovnu metodu za slanje zahtjeva poslužitelju . Request metodu koriste repozitoriji za definiranje svojih specifičnih zahtjeva.

```swift
class NetworkClient: Networking {
    private let session: Session
    private let interceptor: RequestInterceptor

    init(session: Session, interceptor: RequestInterceptor) {
        self.session = session
        self.interceptor = interceptor
    }

    func request<Value>(endpoint: URLRequestConvertible) -> AnyPublisher<Value, Error> where Value: Decodable {
        session.request(endpoint, interceptor: interceptor)
            .cURLDescription(calling: { curl in
                print(curl)
            })
            .validate()
            .publishDecodable(type: Value.self)
            .tryMap {
                guard let code = $0.response?.statusCode else {
                    throw NetworkError.unexpectedResponse
                }
                guard HTTPCodes.success.contains(code) else {
                    throw NetworkError.httpCode(code)
                }
                guard let data = $0.data else {
                    throw DataError.missing
                }
                return data
            }
            .decode(type: Value.self, decoder: JSONDecoder())
            .handleEvents(receiveOutput: { decodable in
                print("Response: ", decodable)
            })
            .eraseToAnyPublisher()
    }
}
```

Slika 4.19 Definicija NetworkClient klase

Networking direktorij sadrži i APIConfigurable datoteku. APIConfigurable sadrži protokol za definiranje zahtjeva. Protokol koriste repozitoriji za definiranje svojih specifičnih zahtjeva. Slika 4.20 prikazuje APIConfigurable protokol zajedno s obaveznim parametrima.

```swift
protocol APIConfigurable: URLRequestConvertible {
    var path: String { get }
    var method: HTTPMethod { get }
    var headers: [String: String]? { get }
    var queryRequestable: QueryRequestable? { get }
    var bodyRequestable: BodyRequestable? { get }
    var multipartFormDataRequestable: MultipartFormDataRequestable? { get }
}
```

Slika 4.20 Definicija APIConfigurable protokola

### 4.2.4. Data direktorij

Dana direktorij sadrži sve proizvoljno definirane tipove podataka koji se koriste unutar aplikacije. Tipovi su podijeljeni u tri poddirektorija: Request, Response i Models. Request poddirektorij sadrži strukture koje se koriste za stvaranje zahtjeva. Slika 4.21 prikazuje LoginBodyRequest strukturu koja se koristi prilikom prijave korisnika. Nasljeđuje BodyRequestable protokol koji omogućava automatsku pretvorbu strukture u JSON [11] *body* poruku HTTP zahtjeva.

```swift
struct LoginBodyRequest: BodyRequestable {
    let email: String
    let password: String
}
```

Slika 4.21 Definicija LoginBodyRequest strukture

Unutar Response poddirektorija se nalaze strukture koje su modelirane prema odgovorima koje poslužitelj vraća. Pomoću njih se dobiveni podaci pretvaraju u konkretne modele podataka. Primjer takve strukture prikazuje Slika 4.22. Decodable protokol koji struktura nasljeđuje govori da se odgovor poslužitelja automatski može pretvoriti u navedenu strukturu.

[11] JSON skraćeno od JavaScript Object Notation predstavlja format za razmjenu podataka

```
43  struct UserLoginResponse: Decodable {
44      enum CodingKeys: String, CodingKey {
45          case authenticatedUser = "data"
46          case statusCode
47      }
48
49      let statusCode: Int
50      let authenticatedUser: AuthenticatedUser
51  }
```

Slika 4.22 Definicija UserLoginResponse strukture

Models poddirektorij sadrži konkretne strukture modela koji predstavljaju podatke potrebne poslovnoj logici. U prethodnom primjeru AuthenticatedUser je model čiju definiciju prikazuje Slika 4.23.

```
10  struct AuthenticatedUser: Codable, Equatable {
11      let user: User
12      var auth: Tokens
13  }
```

Slika 4.23 Definicija AuthenticatedUser strukture

### 4.2.5. Repositories direktorij

Repositories direktorij sadrži repozitorije aplikacije. Uloga repozitorija objasnjena je u poglavlju 3.2.6. Svaki repozitorij ima definiraj protokol koji klasa repozitorija nasljeđuje. Protokol navodi sve metode koje će korisnici klase moći pristupiti. Repozitorij klasa sadrži instancu NetworkClient klase putem Networking protokola kao i definicije svih funkcija koje protokol repozitorija navodi. Gore navedeno vidljivo je na primjeru UsersRemoteRepository klase koju prikazuju Slika 4.24 i UsersRemoteRepositing protokola koju prikazuje Slika 4.25.

```swift
protocol UsersRemoteRepositing {
    func getUser(with id: PathParameter) -> AnyPublisher<UserResponse, Error>
    func getUsers(with queryRequestable: QueryRequestable?) -> AnyPublisher<UsersListResponse, Error>
    func getUserProfile() -> AnyPublisher<UserResponse, Error>
    func uploadImage(with multipartFormDataRequestable: MultipartFormDataRequestable) -> AnyPublisher<Discardable, Error>
    func updateUserProfile(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error>
    func updateUserPassword(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error>
    func updateUserEmail(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error>
    func quitEmployer() -> AnyPublisher<Discardable, Error>
    func deleteUserProfile() -> AnyPublisher<Discardable, Error>
}
```

Slika 4.24 Definicija UsersRemoteRepositing protokola

```swift
class UsersRemoteRepository: UsersRemoteRepositing {
    @Injected private var networkClient: Networking

    func getUser(with id: PathParameter) -> AnyPublisher<UserResponse, Error> {
        networkClient.request(endpoint: Endpoint.user(id))
    }

    func getUsers(with queryRequestable: QueryRequestable?) -> AnyPublisher<UsersListResponse, Error> {
        networkClient.request(endpoint: Endpoint.users(queryRequestable))
    }

    func getUserProfile() -> AnyPublisher<UserResponse, Error> {
        networkClient.request(endpoint: Endpoint.userProfile)
    }

    func uploadImage(with multipartFormDataRequestable: MultipartFormDataRequestable) -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.upload(multipartFormDataRequestable))
    }

    func updateUserProfile(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.updateUserProfile(bodyRequest))
    }

    func updateUserPassword(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.updateUserPassword(bodyRequest))
    }

    func updateUserEmail(with bodyRequest: BodyRequestable) -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.updateUserEmail(bodyRequest))
    }

    func quitEmployer() -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.quitEmployer)
    }

    func deleteUserProfile() -> AnyPublisher<Discardable, Error> {
        networkClient.request(endpoint: Endpoint.delete)
    }
}
```

Slika 4.25 Definicija UsersRemoteRepository repozitorija

### 4.2.6. Services direktorij

Sve servis klase nalaze se u Services direktoriju. Kao i za repozitorije, svrha servisa objasnjena je u poglavlju 3.2.6. Osim toga svaki servis ima protokol koji klasa servisa nasljeđuje. Protokol navodi sve metode koje će korisnici klase moći pristupiti, vidljivo na primjeru UsersServicing protokola kojeg prikazuje Slika 4.26. Servis sadrži instancu AppStatea kao i instance repozitorija koji su potrebni. Repozitoriji su predstavljeni protokolima, a ne konkretnim instancama. Svaki servis ima inicijaliziranu i svoju lokalnu CancelBag instancu. CancelBag klasa implementirana je uz pomoć Combine okvira i omogućava prestanak odašiljanja promjena objavljivača. Prestanak

odašiljanja može se eksplicitno zaustaviti metodom cancel() nad CancelBag instancom ili automatski prilikom deinicijalizacije roditeljskog tipa.

```
protocol UsersServicing {
    func getUser(with id: PathParameter, user: LoadableSubject<User>)
    func getUsers(with queryRequestable: QueryRequestable?, users: LoadableSubject<[User]>)
    func getUserProfile(user: LoadableSubject<User>)
    func uploadImageAndGetUserProfile(with multipartFormDataRequestable: MultipartFormDataRequestable, user:
        LoadableSubject<User>)
    func uploadImage(with multipartFormDataRequestable: MultipartFormDataRequestable, imageResult:
        LoadableSubject<Discardable>)
    func updateUserProfile(with bodyRequest: BodyRequestable, updateProfileResult: LoadableSubject<Discardable>)
    func updateUserPassword(with bodyRequest: BodyRequestable, updatePasswordResult: LoadableSubject<Discardable>)
    func updateUserEmail(with bodyRequest: BodyRequestable, updateEmailResult: LoadableSubject<Discardable>)
    func quitEmployer(quitEmployerResult: LoadableSubject<Discardable>)
    func delete(deletionResult: LoadableSubject<Discardable>)
}
```

Slika 4.26 Definicija UsersServicing protokola

```
    @Injected private var appState: Store<AppState>
    @Injected private var remoteRepository: UsersRemoteRepositing
    @Injected private var localRepository: LocalRepositing

    let cancelBag = CancelBag()
```

Slika 4.27 Atributi UsersService servisa

Slika 4.28 prikazuje definiciju getUser metode za dohvaćanje korisnika ovisno o njegovom id parametru. Drugi parametar predstavlja poveznica koja će se predati na mjestu poziva metode čime će tip koji poziva metodu postati vlasnikom podatka unutar poveznice, u ovom slučaju instance User klase. U tijelu metode vidljiv je poziv setIsLoading metode koja naznačuje početak dohvaćanja podataka koji može trajati određeno vrijeme, odnosno koji je asinkron. Prethodno se može iskoristiti kako bi pogled prikazao određeni indikator aktivnosti prilikom čega se korisnika obavještava da je radnja u tijeku. Just<Void> predstavlja prazni objavljivač. Narednim ulančavanjem metoda odnosno Combine operatora, objavljivaču se dodaje mogućnost odašiljanja pogreške u obliku Error tipa. Operator flatMap praznom Just objavljivaču pridodaje objavljivač vraćen pozivom metode getUser koji kao rezultat može vratiti UserResponse objekt. UserResponse objekt unutar sebe sadrži User objekt koji se izolira uz pomoć operatora map. Operator sinkToLoadable navedeni user objekt sprema unutar user parametra metode, a konačni rezultat se sprema unutar cancelBag instance uz pomoć operatora za spremanje (eng. *store operator*).

```
33      func getUser(with id: PathParameter, user: LoadableSubject<User>) {
34          user.wrappedValue.setIsLoading(cancelBag: cancelBag)
35
36          Just<Void>
37              .withErrorType(Error.self)
38              .flatMap { [remoteRepository] in
39                  return remoteRepository.getUser(with: id)
40              }
41              .map { $0.user }
42              .sinkToLoadable { user.wrappedValue = $0 }
43              .store(in: cancelBag)
44      }
```

Slika 4.28 Definicija getUser metode

### 4.2.7. Utilities direktorij

Unutar Utilities direktorija nalaze se pomoćni tipovi koje drugi tipovi koriste radi obavljanja određene zadaće. Osim zasebno odvojenih tipova u obliku struktura i klasa, nalaze se i ekstenzije (eng. *extension*) koje već postojećim tipovima proširuju funkcionalnosti.

### 4.2.8. Resources direktorij

Resources direktorij sadrži resurse aplikacije kao što su ikone ili pozadinske slike, palete boja i fontovi.

### 4.2.9. UI direktorij

UI direktorij sadrži sve poglede koji se koriste unutar aplikacije zajedno njihovim modifikatorima (eng. *modifier*) i stilovima (eng. *style*). Poddirektoriji se mogu podijeliti na dvije skupine: pomoćne i glavne. Pomoćni se sastoji od Components, Styles i Modifiers poddirektorija. Components poddirektorij sadrži sve poglede koji se često koriste na više zaslona. Primjer komponente je SearchCell koji se koristi prilikom pretraživanja korisnika i ugostiteljskih obrta, definiciju navedenom pogleda prikazuje Slika 4.29. Styles sadrži predefinirane stilove koji se mogu primijeniti nad određenim pogledom. Modifiers poddirektorij sadrži modifikatore pogleda koji kada se nad određenim pogledom primjene ostaju primijenjeni i na svim dječjim pogledima inicijalnog pogleda. Slika 4.30 prikazuje ActivityViewModifier zajedno s njegovim ViewModelom. ActivityViewModifier animira zamagljenje zaslona pri odlasku aplikacije u pozadinu. ViewModel osluškuje AppState te promijene sprema u lokalnu isActive varijablu. ActivityViewModifier unutar body metode postavlja količinu zamagljenja ovisno o isActive atributu ViewModela te promijene animira.

```swift
struct SearchCell: View {

    let title: String
    let imageURL: URL?


    var body: some View {
        HStack(spacing: 0) {
            WebImage(url: imageURL)
                .resizable()
                .placeholder {
                    Image(.person)
                        .resizable()
                        .aspectRatio(contentMode: .fit)
                        .padding(.all, 5)
                        .frame(width: 80, height: 80, alignment: .center)
                        .background(Color(  ))
                }
                .aspectRatio(contentMode: .fill)
                .frame(width: 80, height: 80)
                .background(Color(  ))
                .clipped()

            VStack(spacing: 0) {
                Text(title)
                    .font(.body).bold()
                    .frame(maxWidth: .infinity)
            }
            .frame(maxWidth: .infinity)
            .padding(.all, 5)
        }
        .frame(height: 80)
    }
}
```

Slika 4.29 Definicija SearchCell pogleda

```
struct ActivityViewModifier: ViewModifier {

    @StateObject private var viewModel: ViewModel = Resolver.resolve()

    func body(content: Content) -> some View {
        content
            .blur(radius: viewModel.isActive ? 0 : 10)
            .animation(.easeInOut(duration: 0.2), value: viewModel.isActive)
    }
}

extension ActivityViewModifier {
    class ViewModel: ObservableObject {
        @Injected private(set) var appState: Store<AppState>
        @Published var isActive: Bool = false
        private let cancelBag = CancelBag()

        init() {
            appState.map(\.application.isActive)
                .removeDuplicates()
                .assign(to: \.isActive, on: self)
                .store(in: cancelBag)
        }
    }
}
```

Slika 4.30 Definicija ActivityViewModifier modifikatora

Glavne poddirektorije predstavljaju Auth, Profile, Search, Scan, Offers i Cart direktorij unutar kojih se nalaze pogledi zaslona aplikacije zajedno s njihovim odgovarajućim ViewModelima. Auth poddirektorij sadrži poglede koji sastavljaju zaslone za registraciju, prijavu i slanje emaila za resetiranje zaporke korisnika i ugostiteljskih obrta. Pogledi koji sastavljaju zaslone korisničkog profila korisnika i ugostiteljskih obrta nalaze se unutar Profile poddirektorija. Također se nalaze i zasloni koji omogućavaju promjenu korisničkih postavki profila. Search poddirektorij sadrži poglede koji grade zaslone za pretraživanje korisnika i ugostiteljskih obrta kao i zaslone za prikaz detalja istih. Scan poddirektorij sadrži ScanView i ScanViewModel koji sadrži QR kod skener putem kojeg se mogu otvoriti jelovnici i raditi narudžbe unutar određenog ugostiteljskog obrta. Offers poddirektorij sadrži zaslone za prikaz, uređivanje i brisanje jelovnika, kategorija i proizvoda. Obavljanje narudžbi omogućava CartView i CartViewModel unutar Cart poddirektorija. Izgled gore navedenih zaslona bit će detaljno prikazan u narednom poglavlju.

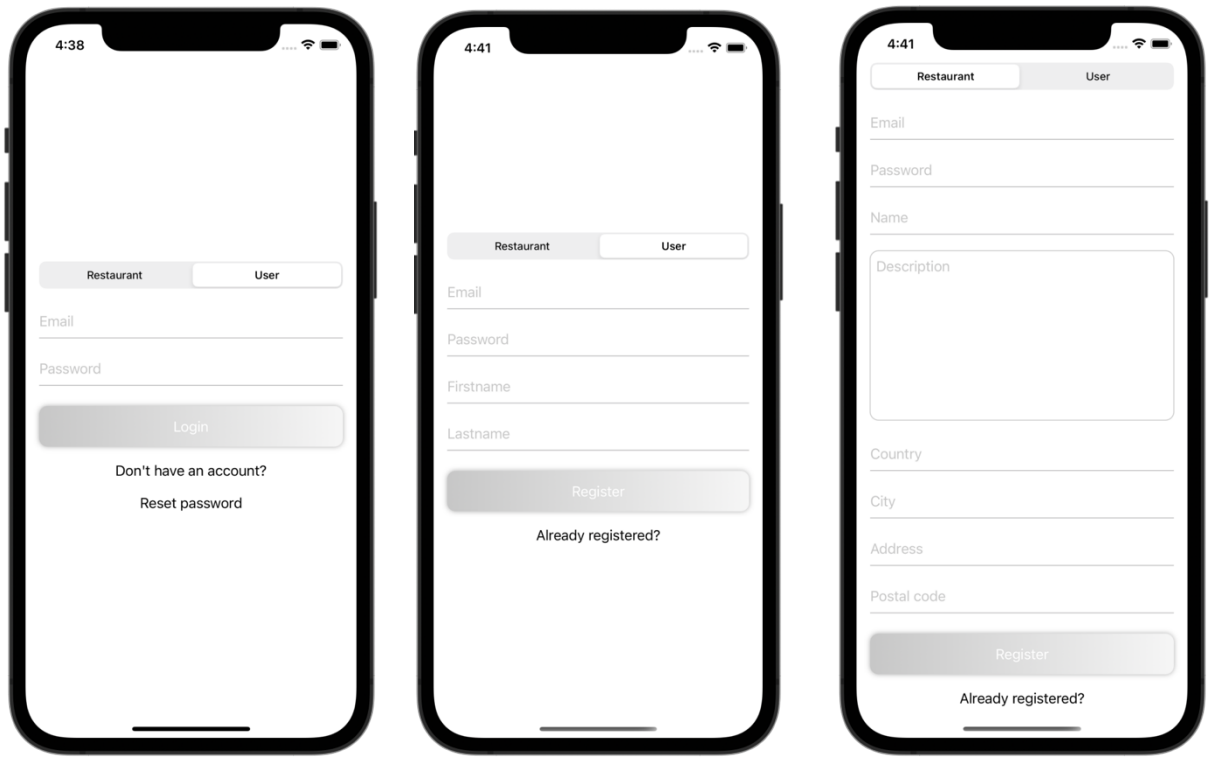
# 5. IZGLED I NAČIN RADA APLIKACIJE

Unutar ovog poglavlja objasnit će se način rada aplikacije uz pomoć predložaka za vizualizaciju korisničkog sučelja.

### 5.1.1. Registracija i prijava

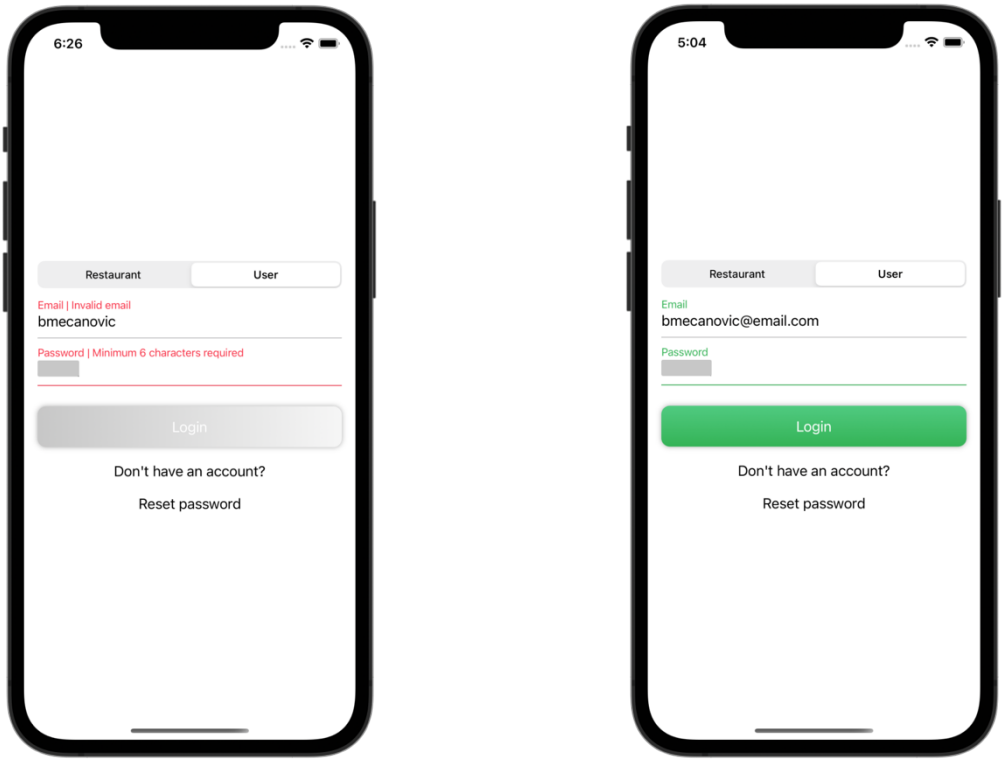
Prilikom ulaska u aplikaciju prikazuje se početni zaslon. Na početnom zaslonu moguće je birati između prijave i registracije korisnika ili ugostiteljskog obrta. Pritiskom na gumb „Don't have an account?“ odlazi se na sučelje za registraciju, a izgled registracijskog sučelja ovisi o odabiru korisnika ili obrta. Isto tako, pritiskom na gumb „Already registered?“ vraća se nazad na sučelje za prijavu. Gore navedeno vidljivo je na slikama koje prikazuje Tablica 5.1.

Tablica 5.1 Zasloni za prijavu i registraciju

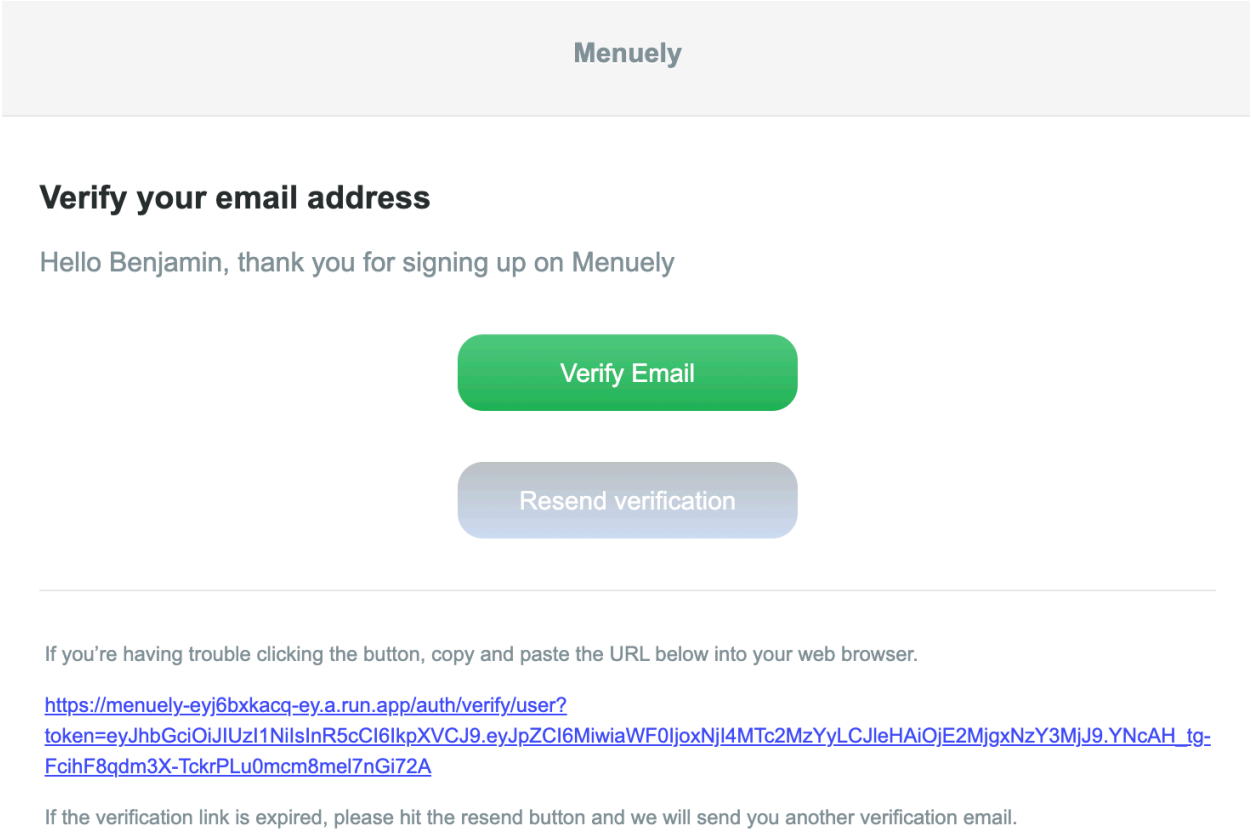



Polja za unos teksta obavještavaju korisnika ovisno o ispravnosti unosa. Tablica 5.2 prikazuje izgled formi s ispravnim unosom (slika lijevo) i neispravnim unosom (slika desno).

Tablica 5.2 Prikaz formi s ispravnim i neispravnim unosom

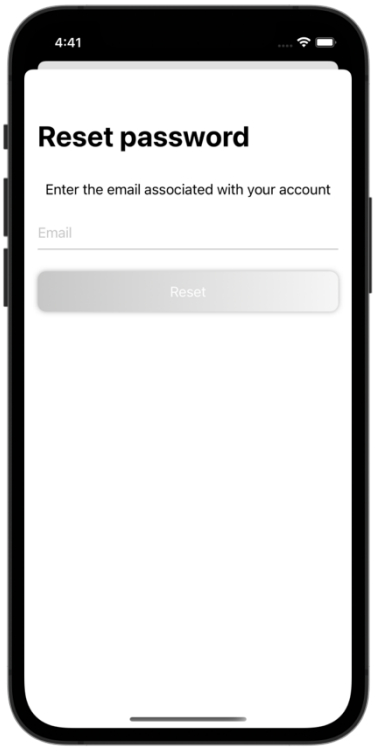



Ako su unosi ispravni gumb za registraciju i prijavu će postati aktivan. Nakon uspješne registracije korisniku se šalje email poruka putem koje je potrebno email adresu potvrditi. Sadržaj tog emaila prikazuje Slika 5.1. Bitno je naglasiti da poveznica za potvrdu emaila ima određeno vrijeme trajanje prije nego postane nedostupna. U slučaju da je nedostupna pritiskom na gumb „Resend verification“ šalje se novi email za potvrdu.

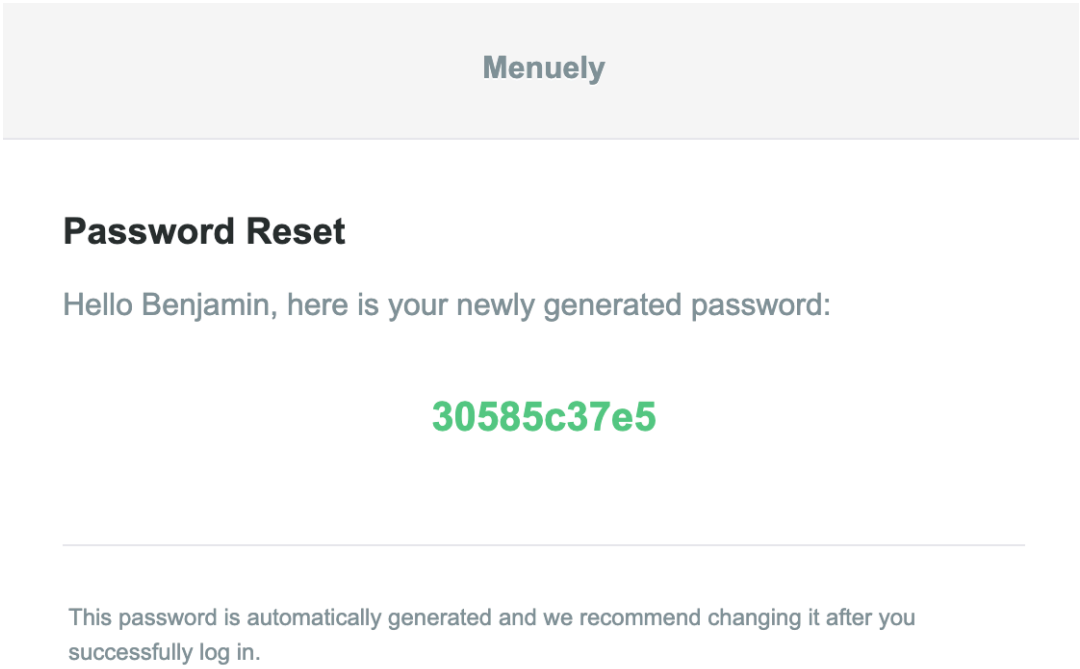



Slika 5.1 Sadržaj emaila za potvrdu

Zasloni za prijavu osim odlaska na registracijsko sučelje omogućavaju i resetiranje zaporke korisnika. Pritiskom gumba „Reset password“ otvara se dijaloški okvir s mogućnosti unosa email adrese koje prikazuje Slika 5.2. Ispravnim unosom email adrese i pritiskom na gumb „Reset“ šalje se nova automatski generirana zaporka na navedenu email adresu. Slika 5.3 prikazuje sadržaj emaila za resetiranje zaporke.




Slika 5.2 Dijaloški okvir za resetiranje zaporke

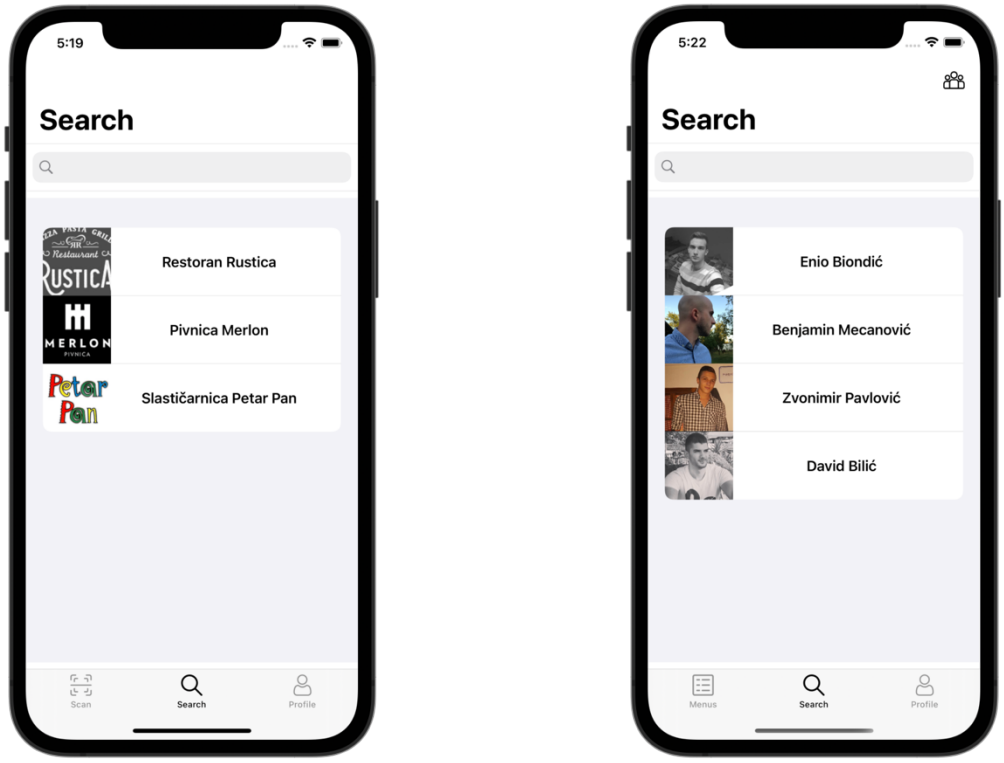



Slika 5.3 Sadržaj emaila za resetiranje zaporke

### 5.1.2. Pretraživanje

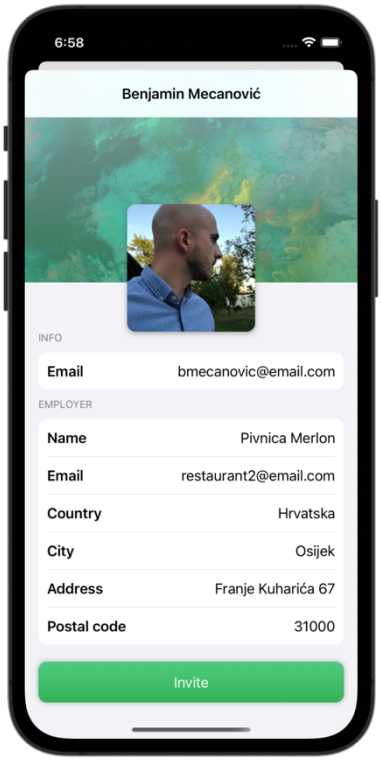
Nakon uspješne prijave početni zaslon postaje zaslon za pretraživanje. U slučaju korisnika pretražuju se ugostiteljski obrti dok se u slučaju ugostiteljskog obrta pretražuju korisnici. Ukoliko je prijavljen ugostiteljski obrt, zaslon za pretraživanje sadrži u gornjem desnom kutu gumb za prikaz liste trenutnih zaposlenika obrta.

Tablica 5.3 Prikaz zaslona za pretraživanje




Pritiskom na ćeliju unutar liste otvara se zaslon s detaljima. Slika 5.4 prikazuje detalje ugostiteljskog obrta prilikom čega korisnik može pogledati ponudu istog. Ugostiteljski obrt ima mogućnost slanja pozivnice za rad korisniku pritiskom na gumb „Invite“ kao što Slika 5.5 prikazuje.

Slika 5.4 Prikaz detalja ugostiteljskog obrta



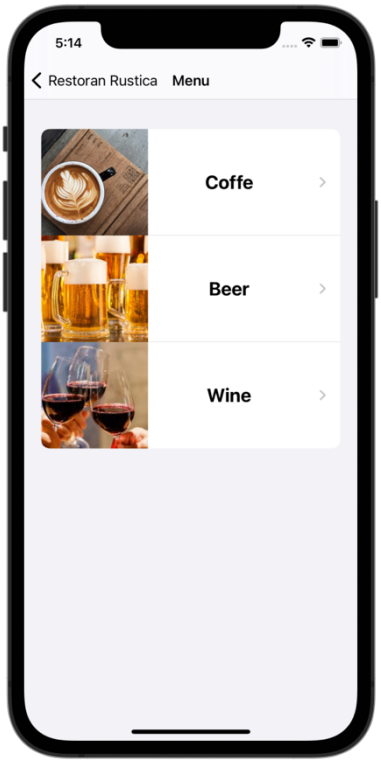

Slika 5.5 Prikaz detalja korisnika

### 5.1.3. Skener QR koda

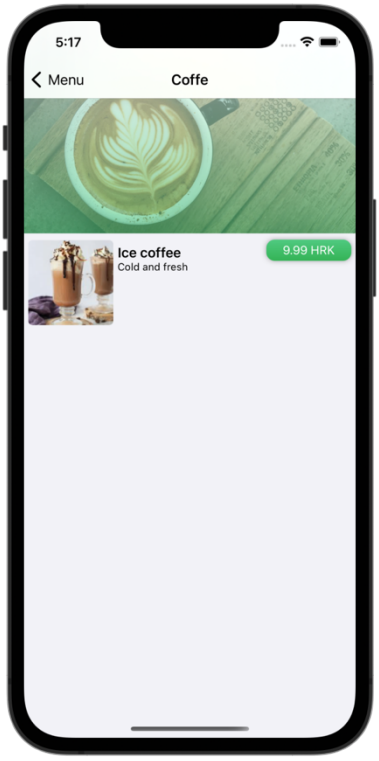
Zaslon sa skenerom QR koda, Slika 5.6 , omogućava skeniranje QR kodova čime se otvara zaslon kao što prikazuje Slika 5.4. Razlika je što se skeniranjem u pozadini kreira prazna košarica za stvaranje narudžbi.

Slika 5.6 Zaslon QR kod skenera

Pritiskom na gumb „Open menu“ otvara se trenutna ponuda ugostiteljskog obrta. Korisnik ima mogućnost prolaska kroz ponudu i dodavanja proizvoda u košaricu. Gore navedeno prikazuju Slika 5.7 i Slika 5.8.

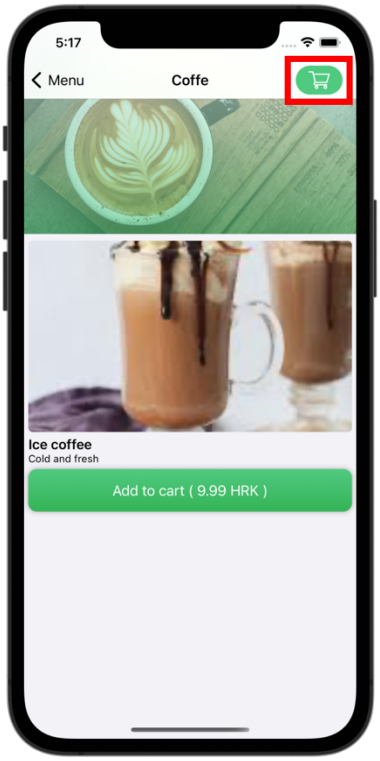



Slika 5.7 Prikaz kategorija



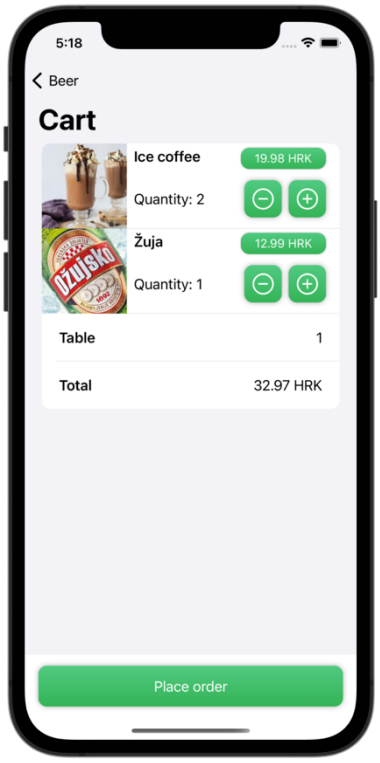

Slika 5.8 Prikaz proizvoda

Dodavanjem barem jednog proizvoda u košaricu u gornjem desnom kutu otkriva se gumb s ikonom košarice. Gumb postaje vidljiv na zaslonu o detaljima ugostiteljskog obrta kao i na zaslonu s kategorijama i proizvodima što prikazuje Slika 5.9.




Slika 5.9 Prikaz zaslona s proizvodima kada košarica nije prazna

Pritiskom na gumb košarice otvara se zaslon košarice, Slika 5.10. Na zaslonu su vidljivi trenutno dodani proizvodi s njihovim odgovarajućim količinama i cijenama. Količinu je moguće uređivati pomoću + i – gumbova. Osim toga prikazan je i redni broj stola na kojem se nalazio QR kod koji je skeniran kao i ukupna cijena narudžbe. Narudžba se može izvršiti pritiskom na gumb „Place order“.

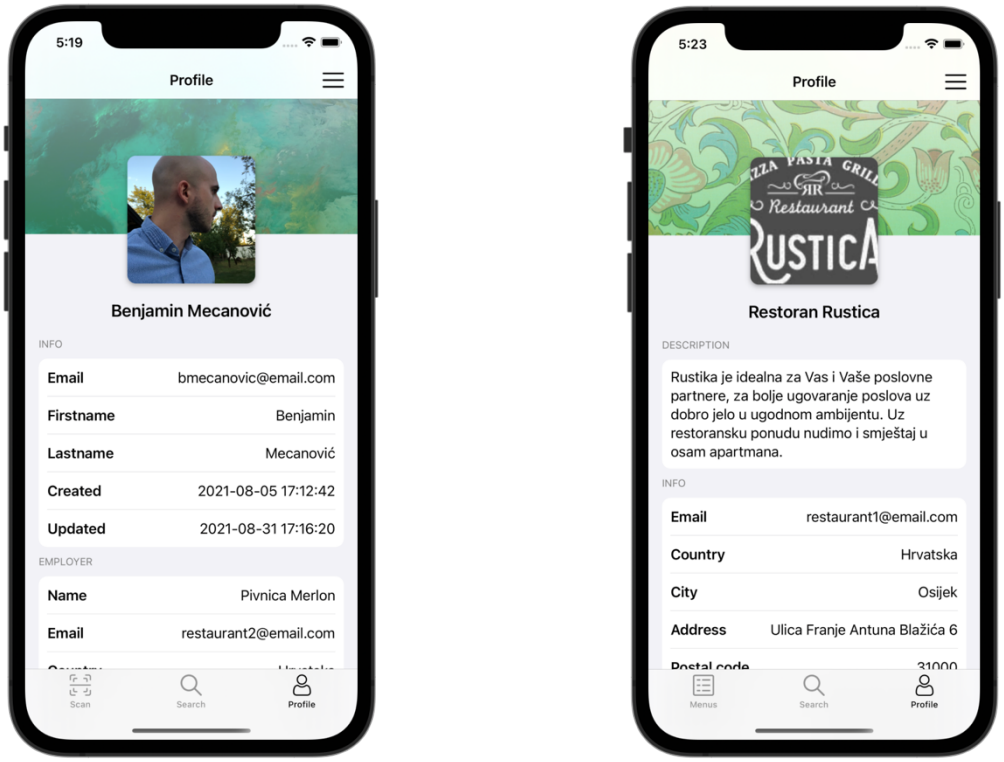



Slika 5.10 Prikaz zaslona košarice

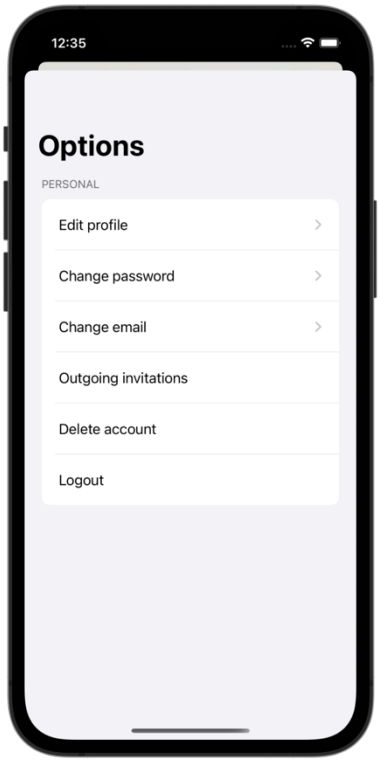
### 5.1.4. Korisnički račun

Zaslon korisničkog računa prikazuje osnovne informacije o korisniku, odnosno restoranu, ovisno tko je prijavljen. Korisniku se osim osobnih podataka prikazuju i podaci o poslodavcu ukoliko je zaposlen. Tablica 5.4 prikazuje zaslone korisničkog računa korisnika (lijevo) i ugostiteljskog obrta (desno).

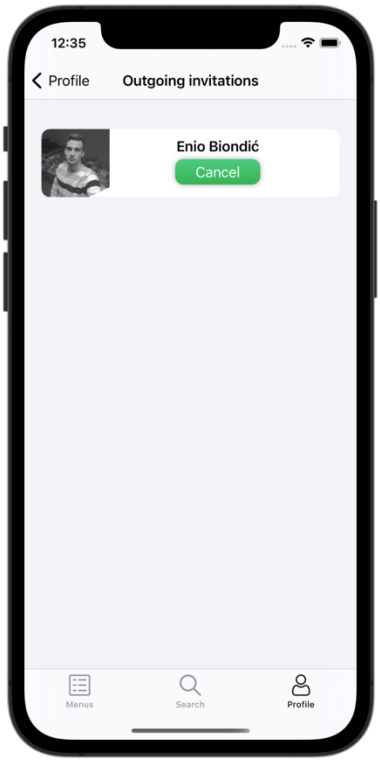
Tablica 5.4 Prikaz zaslona korisničkih računa




Osim osnovnih informacija, u gornjem desnom kutu se nalazi gumb za prikazivanje mogućnosti (eng. *options*). Pritiskom na gumb prikazuje se zaslon s mogućnostima koji se razlikuje ovisno o prijavi korisnika ili ugostiteljskog obrta. Slika 5.11 prikazuje zaslon s mogućnostima ugostiteljskog obrta. Na zaslonu se nalaze poveznice koje vode na zaslone za uređivanje računa. Osim toga moguće se odjaviti, obrisati račun i pregledati listu poslanih pozivnica potencijalnim zaposlenicima restorana kao što prikazuje Slika 5.12. Poslane pozivnice moguće je otkazati pritiskom na gumb „Cancel“.

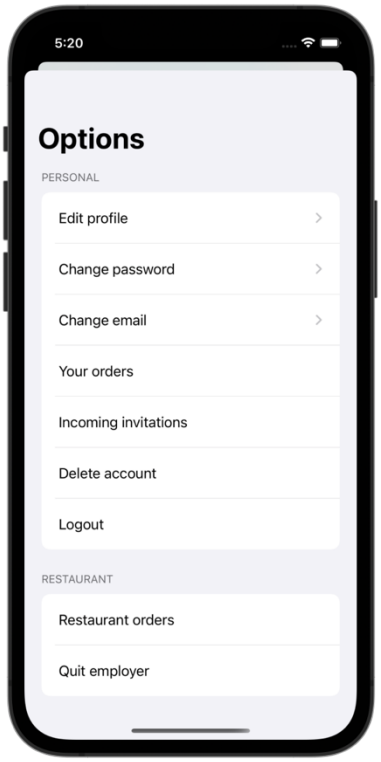



Slika 5.11 Prikaz zaslona s mogućnostima ugostiteljskog obrta



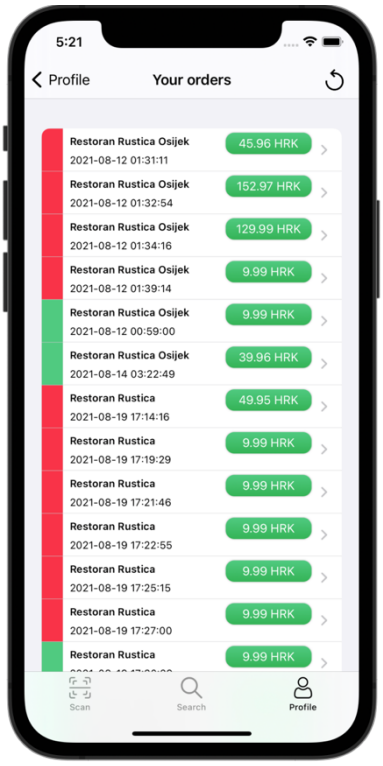

Slika 5.12 Prikaz zaslona s listom poslanih pozivnica

Glavnu razliku između zaslona s mogućnostima ugostiteljskog obrta i korisnika je što korisnik ukoliko je zaposlen dobiva odvojenu sekciju unutar koje se nalaze mogućnosti vezane za poslodavca što prikazuje Slika 5.13.

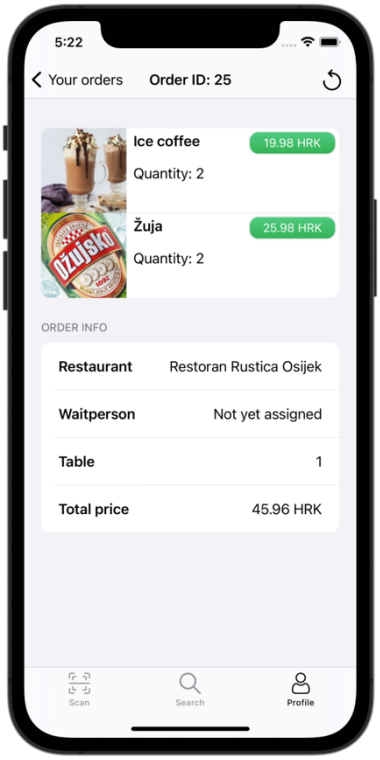



Slika 5.13 Prikaz zaslona s mogućnostima korisnika

Pritiskom na poveznicu „Quit employer“ otvara se prikaz s upozorenjem. Ukoliko korisnik odgovori potvrdo na upozorenje, prestaje biti zaposlenikom trenutnog ugostiteljskog obrta čime se ujedno i briše sekcija kao i prikazane informacije na profilu vezane za istog. Poveznica „Restaurant orders“ prikazuje zaslon s listom svih aktivnih i neaktivnih narudžbi koje je ugostitelj zaprimio. Slično tomu, Slika 5.14 prikazuje „Your orders“ zaslon s listom svih narudžbi koje je korisnik plasirao ugostiteljskim obrtima. Bitno je naglasiti da crveno označene narudžbe predstavljaju narudžbe koje osoblje još nije prihvatilo. Pritiskom na neku od narudžbi otvara se zaslon s detaljima narudžbe kao što prikazuje Slika 5.15.




Slika 5.14 Prikaz zaslona s listom korisnikovih narudžbi

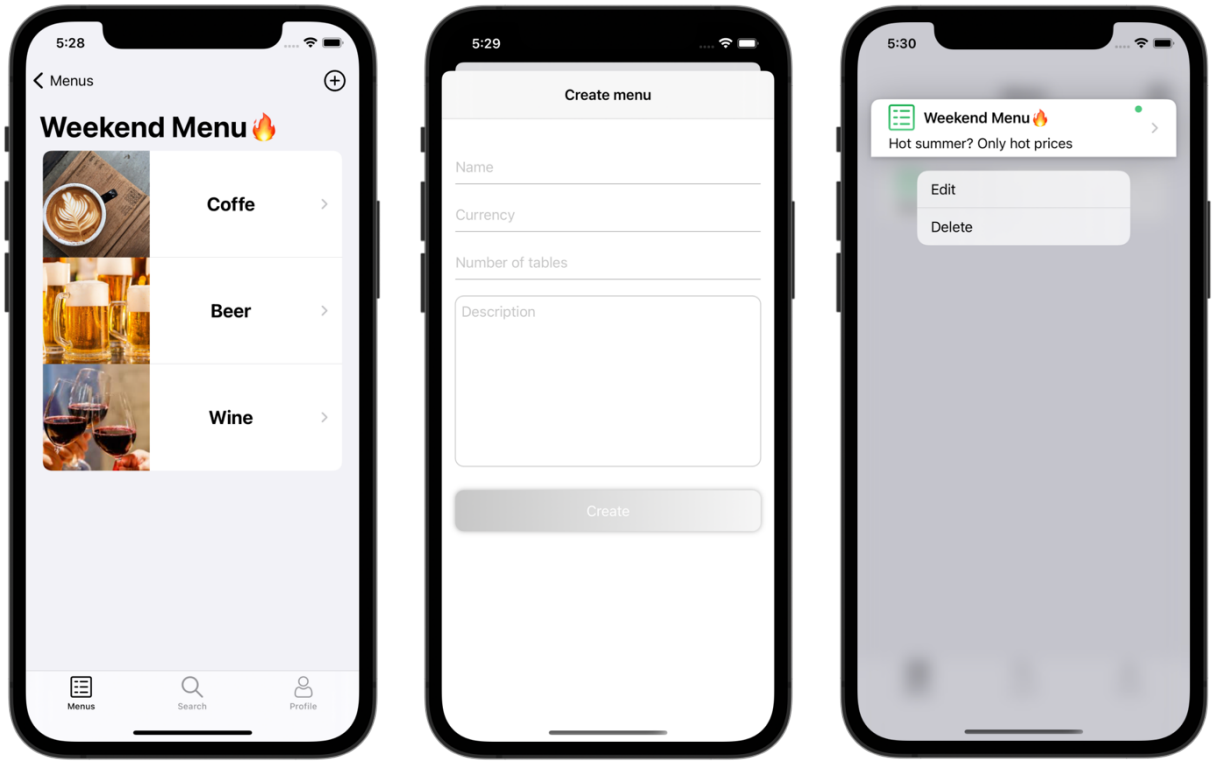



Slika 5.15 Prikaz zaslona s detaljima narudžbe

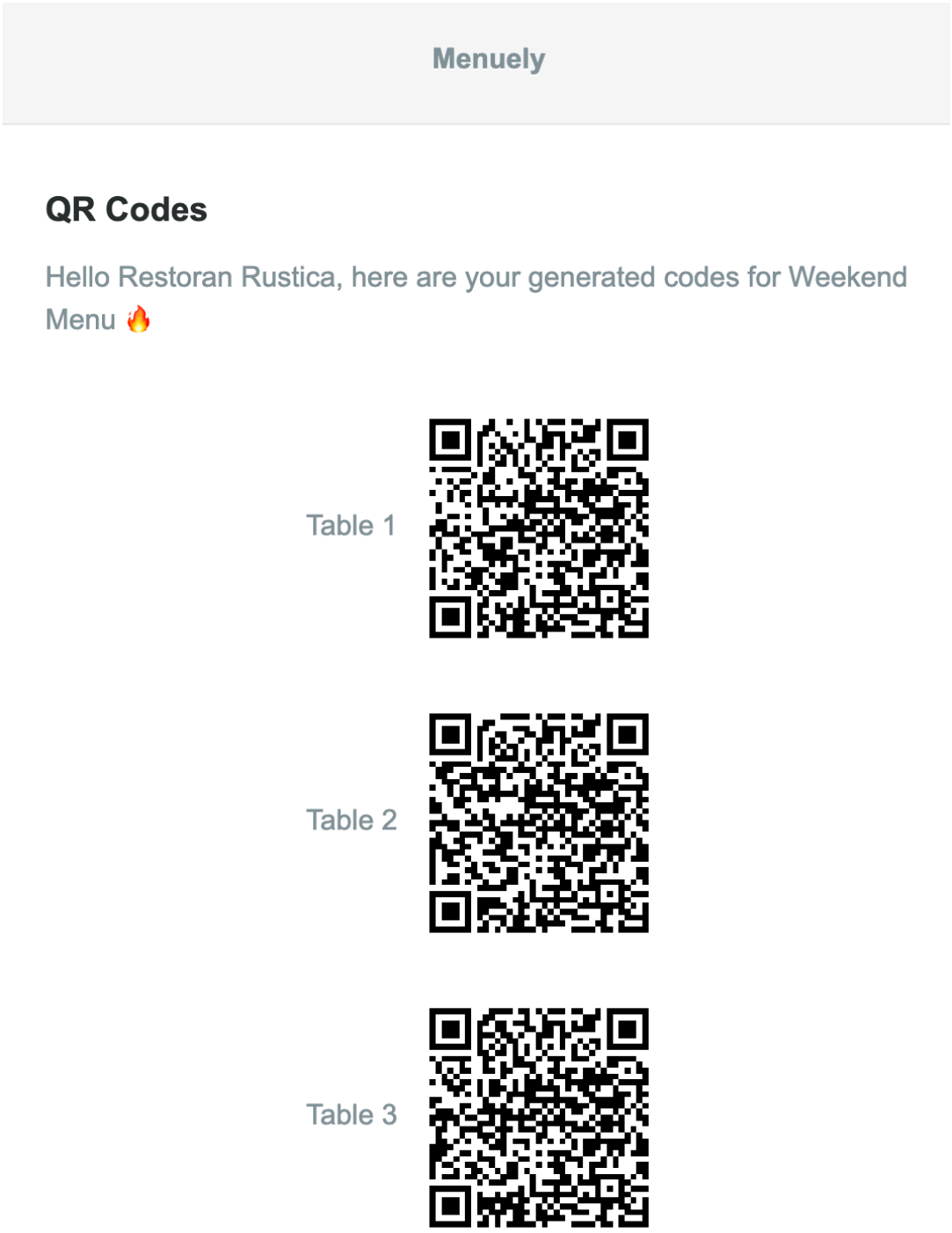
### 5.1.5. Ponuda

Zasloni za prikaz ponude sastoje se od zaslona za prikaz liste jelovnika, kategorija i proizvoda. Omogućavaju njihovo dodavanje, uređivanje i brisanje. Tablica 5.5 prikazuje zaslon s listom jelovnika (lijevo), zaslon za izradu jelovnika (sredina) i kontekstualni izbornik (eng. *context menu*) za uređivanje i brisanje jelovnika (desno). Kontekstualni izbornik se prikazuje prilikom dugog pritiska (eng. *long press*) na ćeliju jelovnika.

Tablica 5.5 Zasloni za izradu, uređivanje, brisanje i prikaz jelovnika




Nakon uspješnog popunjavanja forme za izradu jelovnika i pritiskom gumba „Create“ jelovnik se kreira i pri tome se šalje email poruka koja sadrži QR kodove. Izgled email poruke prikazuje Slika 5.16.




Slika 5.16 Sadržaj emaila s QR kodovima

Zasloni za kategorije i proizvode izgledaju kao što prikazuju Slika 5.7 i Slika 5.8 u poglavlju 5.1.3 uz dodatne mogućnosti kreiranja, uređivanja i brisanja kao i prethodno navedeni jelovnici.

# 6. ZAKLJUČAK

Realizirana je aplikacija koja korisnicima omogućava pregled ponude i izvršavanje narudžbi unutar svih registriranih ugostiteljskih obrta. Time se kod korisnika povećava vjerojatnost korištenja *Menuely* aplikacije naspram preuzimanja velikog broja individualnih aplikacija za svaki pojedini posječeni ugostiteljski obrt. Ograničenje ovog pristupa je smanjena mogućnost prilagodbe aplikacije svakom pojedinom ugostiteljskom obrtu. Uzimajući u obzir i poslužiteljski dio, aplikacija koristi veliki broj najmodernijih tehnologija dostupnih na tržištu. Korištenje novih tehnologija ima i svoje nedostatke, a najbolje se očituju u tome što je potreban uređaj s iOS 14.5 ili novijom verzijom na koju nemaju svi pristup. Ograničenjem iOS verzije efektivno je smanjen broj korisnika što u poslovnom pogledu nije idealno. Zahvaljujući modularnosti, razdvajanjem koda na logičke cjeline i primjenom arhitekture aplikacija je skalabilna i lako održiva. Korištenjem protokola postavljeni su temelji za pisanje testova koji su poželjni ukoliko bi se aplikacija proširivala. Korisničko iskustvo bi se uvelike moglo poboljšati korištenjem *push notifikacija* [12]i *internet socketa*[13]. *Push notifikacije* bi omogućile obavještavanje korisnika o narudžbama dok je aplikacija ugašena, a *socket* dok je aplikacija upaljena, čime bi se izbjegla potreba ručnog osvježavanja statusa narudžbi. Poboljšanje korisničkog iskustva postiglo bi se i implementacijom prijave putem društvenih mreža dok bi izrada *dashboard* web aplikacije uvelike povećala praktičnost kreiranja ponude ugostiteljskih obrta. Uzimajući sve navedeno u obzir, aplikacija ima dobro postavljene temelje i uz dodatnu doradu bi mogla poslužiti početku modernizacije ugostiteljskih obrta.

[12] *Push notifikacija* (eng. *push notification*) predstavlja poruku koju središnji poslužitelj može poslati korisnicima aplikacije

[13] *Internet socket* je krajnja točka unutar dvosmjerne komunikacijske veze između dva programa na mreži

# LITERATURA


[1] Glovo, https://about.glovoapp.com/en, posljednji pristup: 2.9.2021.

[2] Wolt, https://wolt.com/en/about, posljednji pristup: 2.9.2021.

[3] Leggiero, https://www.leggiero.hr, posljednji pristup: 2.9.2021.

[4] TypeScript programski jezik, https://www.typescriptlang.org, posljednji pristup: 24.8.2021.

[5] WebStorm IDE, https://www.jetbrains.com/webstorm, posljednji pristup: 24.8.2021.

[6] DataGrip IDE, https://www.jetbrains.com/datagrip, posljednji pristup: 24.8.2021.

[7] Amazon S3, https://aws.amazon.com/s3, posljednji pristup: 24.8.2021.

[8] NestJS, https://docs.nestjs.com/, posljednji pristup: 24.8.2021.

[9] Docker, https://www.docker.com, posljednji pristup: 24.8.2021.

[10] Google Cloud Run, https://cloud.google.com/run, posljednji pristup: 24.8.2021.

[11] Swift programski jezik, https://developer.apple.com/swift, posljednji pristup: 25.8.2021.

[12] Xcode IDE, https://developer.apple.com/xcode/ide, posljednji pristup: 25.8.2021.

[13] Combine, https://developer.apple.com/documentation/combine, posljednji pristup: 25.8.2021.

[14] SwiftUI Clean Architecture, https://nalexn.github.io/clean-architecture-swiftui, posljednji pristup: 25.8.2021.

[15] Stack Overflow, https://stackoverflow.com, posljednji pristup: 23.8.2021.

[16] Medium, https://medium.com, posljednji pristup: 23.8.2021.

[17] Hacking with Swift, https://www.hackingwithswift.com, posljednji pristup: 23.8.2021.

# SAŽETAK

U okviru ovog diplomskog rada izrađena je iOS mobilna i poslužiteljska aplikacija za modernizaciju ugostiteljskih obrta. Aplikacija kao glavne značajke nudi kreiranje ponuda ugostiteljskih obrta i izvršavanje narudžbi korisnika istih. Rješava probleme gubitka vremena prilikom prihvaćanja i izvršavanja narudžbi te preopterećenja osoblja prilikom velikih gužvi.

**Ključne riječi:** iOS, mobilna aplikacija, poslužiteljska aplikacija, ugostiteljstvo

# ABSTRACT


## iOS APPLICATION FOR ORDERS AND MENU DIGITIZATION OF HOSPITALITY FACILITIES

As part of this thesis, an iOS mobile and server application for the modernization of hospitality facilities was developed. The main features of this application are the creation of offers for hospitality facilities and the execution of customer orders. The application solves the problems of time waste between accepting and executing orders as well as overloading staff during large crowds.



**Keywords:** iOS, mobile application, server application, hospitality

# ŽIVOTOPIS

Benjamin Mecanović rođen je 14. siječnja 1997. godine u Vinkovcima. Od 2004. do 2012. godine pohađa Osnovnu Školu Zrinskih u Nuštru. Godine 2012. upisuje Opću Gimnaziju Matije Antuna Reljkovića u Vinkovcima. Godine 2016. završava srednju školu polaganjem ispita državne mature te upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, preddiplomski studij računarstva. 2019. godine završava preddiplomski studij te upisuje diplomski studij programskog inženjerstva.

# PRILOZI

[1] Github repozitorij iOS aplikacije, https://github.com/hydro1337x/menuely-ios

[2] Github repozitorij poslužiteljske aplikacije, https://github.com/hydro1337x/menuely-backend