Razvoj web aplikacije za šahovski klub

Ban, Antonio

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:741627

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-12**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

Sveučilište u Rijeci – Odjel za informatiku

Diplomski studij Informatika – nastavnički smjer

Antonio Ban

Razvoj web aplikacije za šahovski klub

Diplomski rad

Mentor: izv. prof. dr. sc. Marina Ivašić-Kos

Rijeka, 7.9.2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 17. lipnja 2020.

Zadatak za diplomski rad

Pristupnik: Antonio Ban

Naziv diplomskog rada: **Razvoj web aplikacije za šahovski klub**

Naziv diplomskog rada na engleskom jeziku: **Web application development for Chess Club**

Sadržaj zadatka:

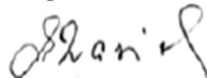
Opisati osnovnu arhitekturu klijent-server sustava te osnovne karakteristike serverless web aplikacije. Proučiti Amazon Web Services (AWS) servise i njihove resursi te ukratko opisati njihovu implementacija u razvoju backend dijela serverless web aplikacije.

Predložiti i opisati arhitekturu serverless web aplikacije razvijene za potrebe šahovskog kluba. Odabrati odgovarajući programski jezik i biblioteke za razvoj backend i frontedn dijela aplikacije.

Napraviti prototip web aplikacije za šahovski klub koja je responzivna, ima više administratora te omogućuje dodavanje i brisanje članova, kreiranje i spremanje članaka i podataka o internim turnirima. Opisati ključne korake implementacije.

Mentor:

Izv. prof. dr. sc. Marina Ivašić-Kos



Voditeljica za diplomske radove:

Izv. prof. dr. sc. Ana Meštrović

Zadatak preuzet: 17.06.2020.

(potpis pristupnika)

Sadržaj

1. Uvod 5
2. Opis projekta 6
3. Amazon Web Services 7
3.1. Amazon Simple Storage Service (S3) 7
3.2. Amazon Cognito 8
3.3. AWS Identity & Access Management 9
3.4. Amazon DynamoDB 11
3.5. Amazon API Gateway 11
3.6. AWS Lambda 13
3.7. AWS Cloudfront 14
3.8. AWS Cloudformation 15
4. Implementacija servisa za web aplikaciju za šahovski klub 16
4.1. Arhitektura 16
4.2. Mreža i pristup 17
4.3. Autentifikacija i sigurnost 19
4.4. API, Lambda i baza podataka 20
4.5. Resursi 22
5. Korištene tehnologije 23
5.1. JavaScript 23
5.2. React 23
5.2.1. Kreiranje React aplikacije 24
5.2.2. Razvoj frontenda 24
5.2.3. Arhitektura frontenda 25
5.2.4. Komponente 27
5.2.5. Životni tijek komponente 31
5.2.6. App.js 32
5.2.7. Redux 33
5.2.8. Slanje HTTP zahtjeva 36
5.2.9. Autentifikacija 38
5.2.10. Material UI 39
5.2.11. Slike zaslona 40

5.3. AWS Serverless Application Model (SAM) ... 45
5.3.1. REST API ... 46
5.3.2. Flask i FlaskRESTful ... 47
5.3.3. Popis ruta i metoda ... 52
6. Budući rad na projektu ... 53
7. Zaključak ... 54
8. Popis literature ... 55
Popis ilustracija ... 57

Sažetak

U ovom diplomskom radu teoretski su opisani različiti Amazon Web Services (AWS) servisi i njihovi resursi te njihova implementacija u razvoju serverless web aplikacije za šahovski klub. U razvoju web aplikacije osim AWS servisa za razvoj backend dijela korišten je programski jezik Python i biblioteka FlaskRESTful, a za frontend dio web aplikacije korišten je programski jezik JavaScript i React biblioteka kako bi se kreirala single-page-application (SPA) korisničko sučelje.

Ključne riječi

Amazon, AWS, Python, Web, Application, JavaScript, React

1. Uvod

U prošlosti je razvoj web aplikacije bio spor i često skup proces, zbog potrebne konfiguracije, testiranja te samog održavanja aplikacije i hardvera na kojem se nalazi. Danas zbog pojave mikro servisa koje pružaju, između ostalih Amazon, Google i Microsoft, hardverske resurse kao uslugu, cjelokupni proces je znatno olakšan i ubrzan. Web aplikacija, razvijena pomoću ovakvih usluga, koristi takozvanu *serverless* arhitekturu iako sam naziv sugerira da ovakva aplikacije ne koristi server, to nije u potpunosti točno, naziv se više odnosi na činjenicu da se osoba koja kreira aplikaciju ne mora brinuti o samom održavanju servera.

Cilj diplomskog rada je prikazati razvoj *serverless* web aplikacije pomoću Amazon Web Services servisa i opisati potrebnu arhitekturu resursa, različite mogućnosti implementacije i pregled samog programskog koda.

Rezultat diplomskog rada je web aplikacija za šahovski klub, pomoću koje će se pružati osnovne informacije o klubu, omogućiti administratoru da dodaje i briše članove kluba, da kreira nove članke te organizira interne turnire.

2. Opis projekta

Svrha ovog projekta je obnoviti postojeću web stranicu kluba koja je statična stranica u PHP-u. Trenutna stranica je samo informativne prirode te ne podržava pristup administratoru koji bi mogao upravljati članovima ili novostima, već se bilo kakvo dodavanje informacija na stranicu vrši kroz ažuriranje i učitavanje samog projekta na server.

Šahovski klub djeluje 20 godina te se ponosi mlađim članovima koji postižu zavidne rezultate. Zahtjevi koje je postavio klub su: responzivnost stranice (mogućnost korištenja web aplikacije na mobilnim uređajima), dodavanje i brisanje članova, kreiranje i spremanje članaka, kreiranje i spremanje podataka o internim turnirima (bazirani na švicarskom sistemu) te mogućnost više administratora web aplikacije.

Web aplikaciji mogu pristupiti dvije vrste korisnika – javni korisnik i administrator stranice. Javni korisnik može pristupiti aplikaciji te pročitati informacije o klubu, vidjeti popis aktivnih članova i popis održanih turnira, dok administrator (koji se mora registrirati i prijaviti) može vršiti kreaciju novih članova kluba, brisanje starih članova kluba, kreiranje članaka s novostima te kreiranje turnira za članove kluba. Osim toga administrator može odobriti ili odbiti zahtjev za registraciju novog administratora.

Programski kod aplikacije je podijeljen u dva dijela, frontend i backend. Frontend kod sadrži kod aplikacije koji određuje što će se prikazati korisniku unutar preglednika te sadrži definiciju metoda i zahtjeva prema backend dijelu aplikacije. Backend dio koda sadrži klase i funkcije koje manipuliraju s podatcima koji se nalaze u bazi podataka.

3. Amazon Web Services

Amazon Web Services (AWS 2020) je dio tvrtke Amazon, koja poslužuje platformu na kojoj individualne osobe ili tvrtke mogu koristiti različite servise koje nudi, kao što su na primjer: virtualna računala, prostor za pohranu podataka te ostali servisi. Korištenje usluga se naplaćuje s obzirom na količinu upotrebe, performanse i različite ostale karakteristike. Za potrebe razvoja web aplikacije za šahovski klub iskoristio se AWS Free Tier, račun koji sadrži većinu servisa, no s nekim ograničenjima. Tako na primjer AWS Lambda može primiti jedan milion zahtjeva na mjesec dana besplatno, no ako se prijeđe preko tog limita, svaki zahtjev se naplaćuje $0.20 po jednom milionu zahtjeva. Slični su uvjeti za ostale servise. Pošto se web aplikacija izrađuje za lokalni šahovski klub, ne očekuje se da će se prijeći preko ograničenja postavljena od AWS Free Tier računa. U nastavku slijedi pregled korištenih servisa te njihovih mogućnosti.

3.1. Amazon Simple Storage Service (S3)

Amazon Simple Storage Service (Amazon S3) (AWS S3 2020) je servis za spremanje objekata koji nudi korisnicima različitih potreba. Pod objektima se podrazumijevaju različite datoteke kao što su tekstualni dokumenti, slike, video zapisi i slično. Veličina datoteka koje se učitavaju mogu biti veličina od 1 bajta do 5 terabajta, nema ograničenja na količinu podataka koji se spremaju.

Datoteke se spremaju u *buckete* (kantice), a na kantice se može gledati kao mapu s podatcima. Ime kantice je globalno ograničeno, što u prijevodu znači da ime svake kantice mora biti unikatno na globalnoj razini. Razlog tome je što se S3 kantica ime koristi kao URL za web stranice koje se poslužuju. Tako statična stranica koja se poslužuje s S3 kantice ima URL oblika test-bucket.s3.amazonaws.com.

Što se tiče sigurnosti pristupu S3 kantici, postoje dvije vrste zaštite. Jedna je ACL (Access Control List) u kojem se može definirati tko ima pristup toj kantici te koje operacije smije vršiti nad podatcima. Druga vrsta zaštite je Bucket Policy – dokument u kojem se mogu definirati različite restrikcije kao što je restrikcija pristupa po IP adresi ili s koje domene dolazi zahtjev prema S3 kantici.

3.2. Amazon Cognito

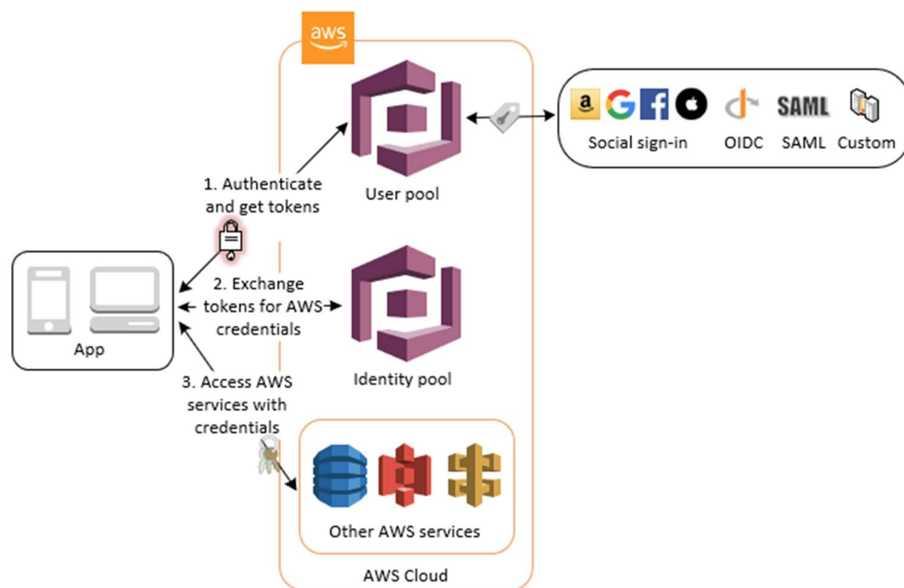
Amazon Cognito je Web Identity Federation servis koji korisnicima daje potrebne tokene ili dozvole kako bi pristupili različitim resursima unutar AWS oblaka (AWS Cognito 2020). Unutar Cognito servisa razlikujemo Cognito User Pool i Identity Pool.

User Pool je namijenjen korisnicima koji će koristiti ovaj servis za registraciju i prijavu na web aplikacije, kod registracije korisničko ime i lozinka su spremljeni unutar samog Cognito servisa. Kod uspješne prijave na Cognito korisnik dobije JSON Web Token ili JWT(Slika 1).

```
{
"sub": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
"aud": "xxxxxxxxxxxxexample",
"email_verified": true,
"token_use": "id",
"auth_time": 1500009400,
"iss": "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_example",
"cognito:username": "janedoe",
"exp": 1500013000,
"given_name": "Jane",
"iat": 1500009400,
"email": "janedoe@example.com"
}
```

Slika 1 - Cognito JWT Token

Identity Pool je namijenjen za kreaciju kratkoročnih ključeva za pristup određenim servisima. Kombinacija ova dva servisa na web aplikaciji izgleda ovako: korisnik se prijavi u Cognito User Pool, primljeni JWT token, šalje Cognito Identity Poolu koji korisniku prosljeđuje dozvolu u formi IAM role koja će biti objašnjena u nastavku (Slika 2 (AWS Cognito 2020)).



Slika 2 - Izmjena tokena za dozvolu pristupa resursima

3.3. AWS Identity & Access Management

Identity & Access Management (IAM) je servis s kojim je moguće unutar jednog AWS računa postaviti: korisnike, role i grupe (AWS IAM 2020).

Glavna stavka ovog servisa je IAM Policy. To je dokument JSON oblika u kojem je definirano čemu koji korisnik, grupa ili rola smije pristupiti.

Pod korisnikom se podrazumijeva jedna ili više osoba koja će koristiti resurse na AWS računu. Root korisnik (vlasnik glavnog računa) može kreirati nove korisnike te ograničiti pristup tom korisniku određenom servisu, mogućnosti servisa ili određenom resursu. Tako jedna tvrtka može imati samo jedan AWS račun i kreirati korisnika za svakog od zaposlenika te ograničiti pristup servisima s obzirom koji su im potrebni.

Grupa je kolekcija korisnika. Korisnik unutar grupe će naslijediti ograničenja postavljena za tu grupu.

Rola se dodjeljuje određenom AWS resursu kao što je na primjer AWS Lambda, ako je u *policy* role zapisano da rola nema pristup tablici *TestnaTablica*, AWS Lambda neće moći pristupiti podatcima koji se nalaze unutar te tablice.

3.4. Amazon DynamoDB

DynamoDB je NoSQL baza podataka (AWS DynamoDB 2020). NoSQL podrazumijeva bazu podataka koja nije relacijska, a koristi zapis podataka u ključ-vrijednost obliku. Također, za ovakvu bazu podataka nije potrebno definirati schemu koja definira svaku tablicu i stupce što omogućuje fleksibilnost kod spremanja podataka.

Za razliku od relacijskih baza podataka, kod rasta količine podataka ne postoji znatan pad performansi (TechTerm 2020). Tako DynamoDB poslužuje upite s brzinama ispod 10 milisekundi. Podatci spremljeni pomoću servisa DynamoDB se spremaju na SSD diskove što pridonosi samoj brzini odrađivanja zahtjeva.

3.5. Amazon API Gateway

Amazon API Gateway je servis pomoću kojeg je moguće kreirati resurs API (AWS API Gateway 2020). API ili *Application Programming Interface* je sučelje za programiranje aplikacija koje određuje interakciju između različitih softvera, definira vrste zahtjeva koje podržava, kako ih poslati, koje formate podataka podržava i koje konvencije treba pratiti (Wikipedia API 2020).

Dakle, API Gateway je servis koji podržava kreaciju API koji se koristi kao početna točka pristupu AWS resursima. Tako kroz API Gateway možemo pristupiti različitim resursima kao što su DynamoDB, AWS Lambda, EC2 instance ili bilo kojem drugom resursu. *Endpoint* je resurs API Gatewaya, a moguće je dodijeliti više *endpointa* svakom API, a svakom od tih *endpointa* konačna destinacija može biti različit resurs.

Također, dodavanjem HTTPS metoda *endpointu* možemo kreirati *Restful* API koji će koristiti web aplikacija kako bi pristupila određenim podatcima. *Restful* API je detaljnije objašnjen u poglavlju 5.3.1. API Gateway podržava verzioniranje API, to jest, moguće je imati više verzija samog API, recimo produkcijsku i development verziju.

Bitna stavka API Gatewaya je keširanje odgovora određenog *endpointa*. Recimo da korisnik 1 napravi GET zahtjev na određeni *endpoint* koji čeka odgovor od AWS Lambde, taj odgovor će se proslijediti zatim korisniku 1, ako korisnik 2 napravi identičan zahtjev na isti *endpoint*, API

Gateway neće čekati odgovor od AWS Lambde, već će proslijediti keširani odgovor. Ovakva funkcionalnost smanjuje vrijeme čekanja za odgovor kod korisnika.

Još jedna od bitnih značajki API Gatewaya, posebno kod razvoja web aplikacije je *cross origin resource sharing*, no za to je potrebno znati što je *same origin policy* (Same-origin policy 2020). *Same origin policy* se može vidjeti kod web preglednika koji dopušta skriptama, koje se nalaze na jednoj stranici, pristup podatcima, koji se nalaze na drugoj stranici, ako imaju jednak izvor (eng. origin), to jest, ako im je ime domene jednak. Ovo pravilo postoji kako bi se spriječili napadi drugih web stranica – poznato pod imenom *cross-site scripting*.

Problem na koji se nailazi je da web aplikacija, koja se nalazi u S3 kantici, ima jednu domenu, API Gateway drugu, a ako koristimo i AWS Cloudfront (poglavlje 3.7.), pak treću. Zbog tog problema postoji mehanizam *cross origin resource sharing* ili CORS. To je mehanizam koji dopušta restriktiranim resursima pristup sa drugih domena. Primjer CORS mehanizma je: ako web stranica napravi zahtjev na URL prvo će poslati OPTIONS zahtjev, zatim server vraća odgovor u kojem je popis domena koje mogu pristupiti te, ako stranica nije na domeni, koja je unutar tog popisa, stranica će kao odgovor dobiti grešku. Tako stranica s domene *domain-a.com* može poslati GET zahtjev na server koji se nalazi na *domain-a.com* domeni, no u slučaju da stanica pošalje GET zahtjev na *domain-b.com* prvo će se poslati OPTIONS zahtjev u kojem će se provjeriti je li *domain-a.com* zapisana na serveru unutar popisa dozvoljenih izvora(eng. origin). Ako se ta domena nalazi unutar popisa, server će na OPTIONS zahtjev odgovoriti sa status kodom 200 te će obraditi GET zahtjev, u suprotnom OPTIONS zahtjev će biti odbijen sa status kodom 401 – *Unauthorized*.

3.6. AWS Lambda

Kako bi web aplikacija bila *serverless* potrebno je koristiti AWS Lambda servis (AWS Lambda 2020). Nekada su se backend operacije odvijale na serverima koje je bilo ili potrebno nabaviti ili unajmiti. Danas se preferira korištenje oblaka za ovakve potrebe zbog smanjenja vremena za konfiguraciju web servera, baze podataka, vatrozida i ostalih potrebnih servisa.

Amazon je 2006. godine predstavio EC2 instancu s kojom bi mogao pristupiti virtualnom računalu te time promijenio cjelokupnu arhitekturu potrebnu za različite vrste razvoja aplikacija.

AWS Lambda je zapravo komputacijski servis u kojem je moguće kreirati Lambda funkcije. Lambda funkcije se najčešće koriste kako bi se pokrenuo određeni kod zapisan u lambda funkciji s obzirom na određeni događaj. Česta je upotreba pokretanja Lambda funkcije kod HTTP poziva na API Gateway kod web aplikacija, pokretanje kod dodavanja zapisa u DynamoDB tablicu ili kod učitavanja datoteke na S3 kanticu. Te funkcije se pokreću na serverima koji se nalaze u data centrima, a na developeru je samo da se brine o programskom kodu.

AWS Lambda se pokreće s obzirom na događaje (event). Ti događaji se nazivaju okidačima (eng. triggerima) koji pokreću Lambdu koja zatim izvrši kod koji je zapisan unutar Lambda funkcije. Takav okidač može biti recimo promjena nekog zapisa unutar određene DynamoDB tablice, učitavanje datoteke na S3 kanticu ili poziv na određeni resurs API Gatewaya. Izvršavanje jedne Lambde, također, može biti okidač druge Lambde.

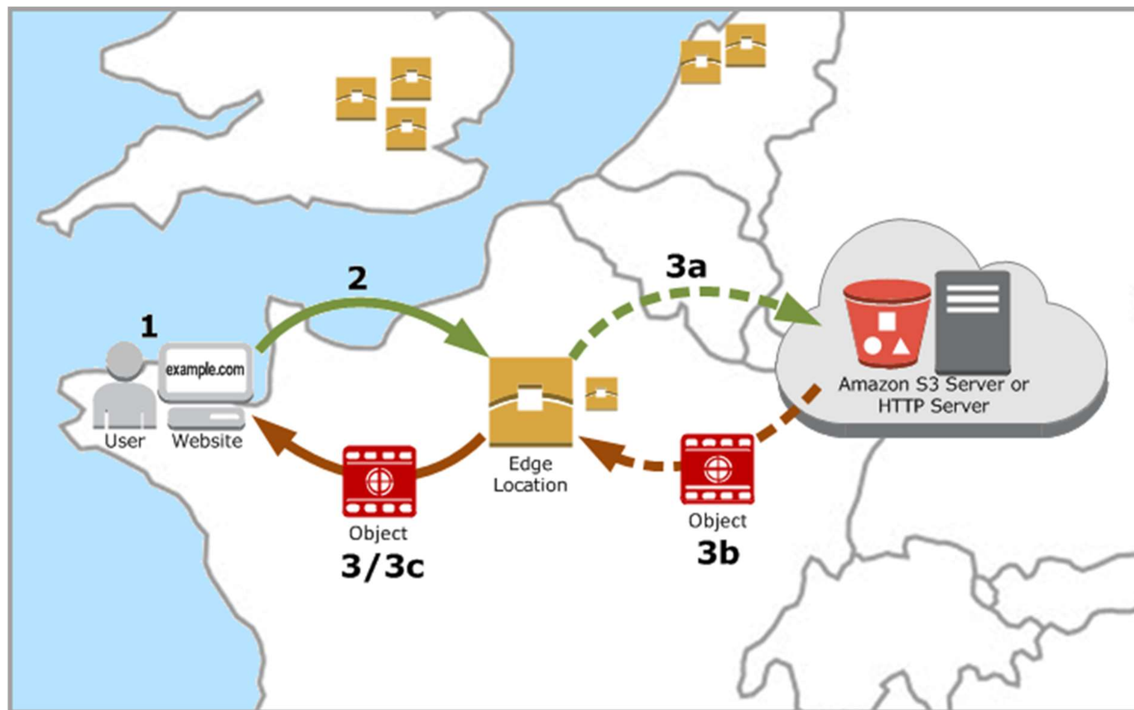
Prednost AWS Lambde je skalabilnost. To podrazumijeva da, ako dva korisnika naprave zahtjev na API Gateway resurs u isto vrijeme, taj zahtjev će pokrenuti izvršavanje Lambda funkcije, ali će se potpuno odvojeno pokrenuti izvršavanje te funkcije. Naravno, to nije neko poboljšanje u slučaju dva korisnika, ali ako par tisuća korisnika radi isti zahtjev u isto vrijeme, AWS Lambda će pokrenuti istu Lambda funkciju par tisuća puta u isto vrijeme.

AWS Lambda podržava nekoliko programskih jezika, a to su: Node.js, Java, Python, C#, Go i Powershell.

3.7. AWS Cloudfront

AWS Cloudfront je *content delivery network* ili CDN. To je sistem distribuiranih servera ili mreža koja pruža korisniku web stranice i ostale web sadržaje bazirane na geografijskoj lokaciji korisnika i izvoru web stranice (AWS Cloudfront 2020).

CDN omogućuje, ako se web aplikacija poslužuje iz, recimo, Londona, a korisnik koji pokušava pristupiti stranici se nalazi u SAD-u, korisnik ne mora pristupati direktno izvoru stranice u Londonu i čekati prijenos podataka. CDN koristi *Edge* lokacije - to je lokacija gdje će se sadržaj web stranice keširati. Osim pojma *Edge* lokacije, druga bitna stavka je distribucija, a to je ime koje je dodijeljeno CDN-u, odnosno to je popis *Edge* lokacija koje sadrži (Slika 3 (AWS Cloudfront 2020)).



Slika 3 - Prikaz HTTP zahtjeva Cloudfront distribuciji

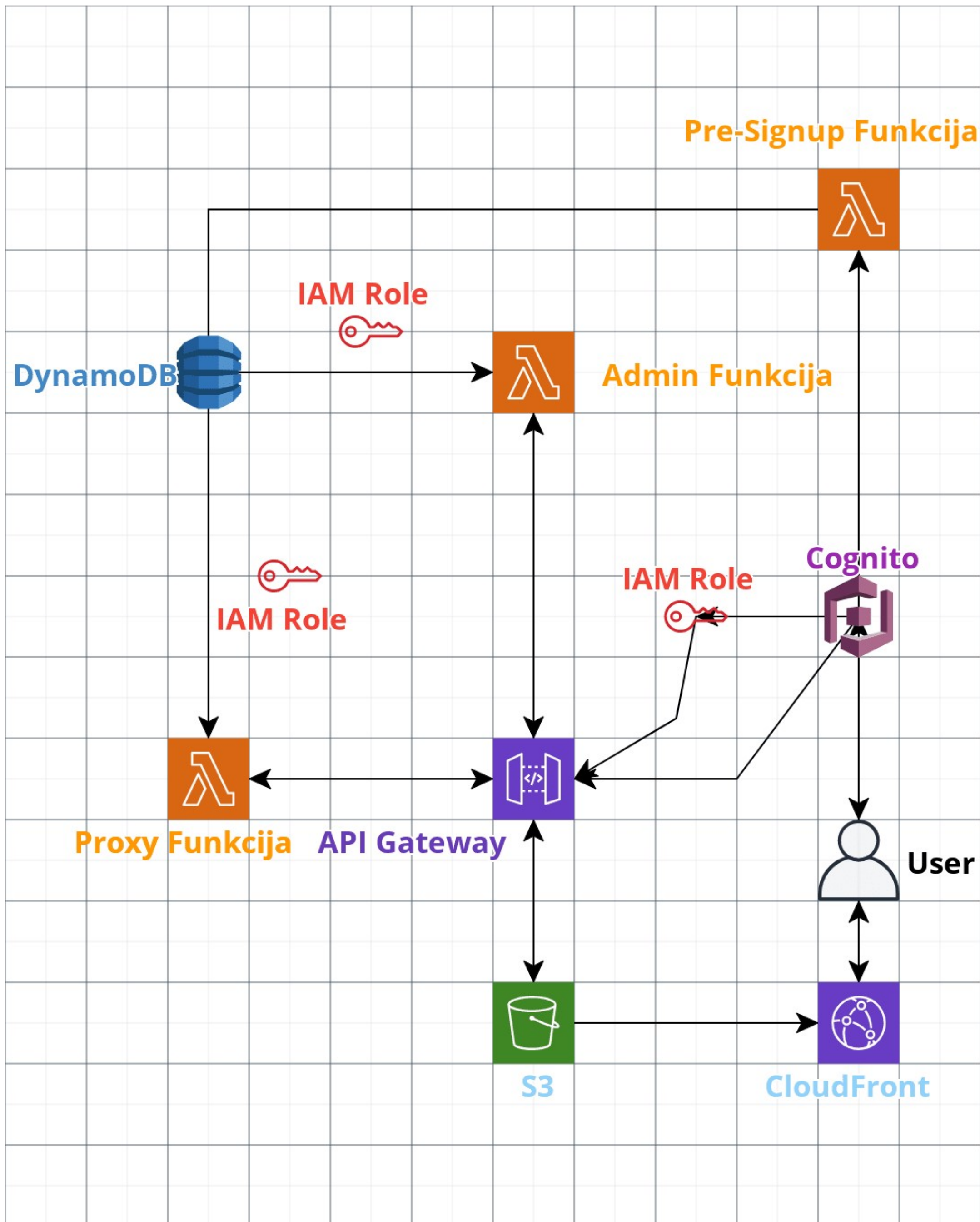
Korištenjem AWS Cloudfronta korisnik koji radi zahtjev na web aplikaciju iz Londona će poslati zahtjev na *Edge* lokaciju, a, ako sadržaj nije već keširan, *Edge* lokacija će preuzeti taj sadržaj iz Londona te će ostati tamo keširan ovisno o tome kako je konfiguriran AWS Cloudfront, najčešće 48 h ili 72 h. Postoje dvije vrste distribucija, web distribucija koja služi za serviranje web sadržaja i RTMP distribucije koje se pretežno koriste za Adobe tj. za posluživanje medijskih streamova.

3.8. AWS Cloudformation

AWS Cloudformation je servis koji omogućuje upravljanje, konfiguraciju arhitekture AWS resursa u obliku koda (AWS Cloudformation 2020). To je ostvareno pomoću Cloudformation templatea – dokument u YAML ili JSON obliku u kojem se definiraju AWS resursi poput AWS Lambde, Cognito User Poola, DynamoDB tablice te ostalih. Ovaj dokument AWS Cloudformation procesira te poziva potrebne API pozive kako bi se kreirao takozvani *stack*. *Stack* je reprezentacija svih resursa koji su kreirani pomoću ovog dokumenta. Sve ove resurse moguće je kreirati i kroz konzolu, no kreacija resursa na ovaj način donosi benefite transporta arhitekture s jednog AWS računa na drugi ili transport arhitekture u drugu regiju AWS računa.

4. Implementacija servisa za web aplikaciju za šahovski klub

4.1. Arhitektura



Slika 4 - Arhitektura AWS servisa i resursa

Na slici 4. je prikaz AWS servisa i resursa koji su korišteni za izradu web aplikacije za šahovski klub. U S3 kantici nalaziti će se programski kod frontend dijela aplikacije, a sadržaj kantice se poslužuje pomoću Cloudfront distribucije. Korisnik se može registrirati pomoću Cognito User Pool koji pomoću IAM role dozvoljava pristup resursu API Gateway-a. Kada se korisnik

registrira Cognito pokreće *pre-signup*-lambda koja korisnikov zahtjev zapisuje u DynamoDB bazu podataka. Korisnik preko web aplikacije može poslati HTTP zahtjev na API Gateway koji zatim određuje pokreće li se Lambda *Proxy Funkcija ili Admin Funkcija* s obzirom je li korisnik prijavljen u Cognito User Pool. Lambda funkcije s obzirom koji je HTTP zahtjev poslan pokreću backend programski kod te vraća kao rezultat određene podatke iz DynamoDB baze podataka (podatci o turnirima, članovima ili novostima).

Svaka ikona predstavlja jedan resurs koji je potrebno kreirati kako bi web aplikacija podržavala sve potrebne funkcionalnosti. Implementacija istih je objašnjena u nastavku.

4.2. Mreža i pristup

Prvi resurs koji je potrebno postaviti je S3 kantica. U S3 kanticu nazvanu „sk-rjecina-app" nalazi se programski kod frontend dijela web aplikacije. Kantica je konfigurirana da poslužuje datoteke koje se nalaze u njoj kao web stranicu te je definirana ruta na index.html datoteku koja će biti prikazana korisniku kada pristupi URL-u ove S3 kantice.

Kako bi se osiguralo da nitko drugi osim vlasnika kantice ne bi mogao mijenjati sadržaj, potrebno je postaviti ograničenje unutar S3 *bucket policy*. Na slici (Slika 5) je prikazan JSON zapis *policy-a* u kojem je definirana dozvola GET metode na ovoj kantici.

Bucket policy editor ARN: arn:aws:s3:::sk-rjecina-app
Type to add a new policy or edit an existing policy in the text area below.

```
1  {
2      "Version": "2012-10-17",
3      "Statement": [
4          {
5              "Sid": "PublicReadGetObject",
6              "Effect": "Allow",
7              "Principal": "*",
8              "Action": "s3:GetObject",
9              "Resource": "arn:aws:s3:::sk-rjecina-app/*"
10         }
11     ]
12 }
```

Slika 5 - Bucket policy

Time je dozvoljen pristup bilo kojem korisniku da čita datoteke s ove S3 kantice, to jest kada korisnik otvori URL ove kantice metoda dozvoljena je GET metoda kako bi pristupio

index.html stranici. Zatim je potrebno definirati CORS konfiguraciju u kojoj dozvoljavamo pristup samo s domene s koje se poslužuje (Slika 6).

CORS configuration editor ARN: arn:aws:s3:::sk-rjecina-app
Add a new cors configuration or edit an existing one in the text area below.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
3  <CORSRule>
4      <AllowedOrigin>https://d155rsjf9cbkoj.cloudfront.net/home</AllowedOrigin>
5      <AllowedMethod>PUT</AllowedMethod>
6      <AllowedMethod>POST</AllowedMethod>
7      <AllowedMethod>DELETE</AllowedMethod>
8      <AllowedHeader>*</AllowedHeader>
9  </CORSRule>
10 </CORSConfiguration>
11
```

Slika 6 - Konfiguracija CORS

Domena je definirana pomoću AWS Cloudfronta. Kod kreacije web distribucije definiran je izvor web aplikacije (Slika 7) i ruta do index.html datoteke. AWS Cloudfront zatim kreira *Edge* lokacije u kojem će se podatci koje zahtjeva korisnik keširati te na kraju dodjeljuje domenu s kojom je moguće pristupiti web aplikaciji u obliku <id>.cloudfront.net domene.

Create Origin | Edit | Delete

	Origin Domain Name and Path	Origin ID
☐	sk-rjecina-app.s3-website.eu-central-1.amazonaws.com	S3-Website-sk-rjecina-app.s3-website.eu-central-1.amazonaws.com

Slika 7 - Izvor web aplikacije

4.3. Autentifikacija i sigurnost

Nakon učitavanja programskog koda web aplikacije na S3 kanticu i postavljanja Cloudfront distribucije, bilo koji korisnik može pristupiti web aplikaciji, no time trenutno ima i pristup svim ostalim AWS resursima koji su korišteni u izradi ove aplikacije.

Kako bi to ograničili koriste se servisi Cognito i IAM. Kod registracije korisnik s web aplikacije šalje zahtjev Cognito User Poolu za registraciju.

Korisnik kroz formu na web aplikaciji šalje podatke o e-mailu, korisničkom imenu i lozinki direktno u Cognito User Pool. Za lozinku je postavljena restrikcija koja zahtjeva minimalno osam znakova, najmanje jedno malo slovo i najmanje jedan broj.

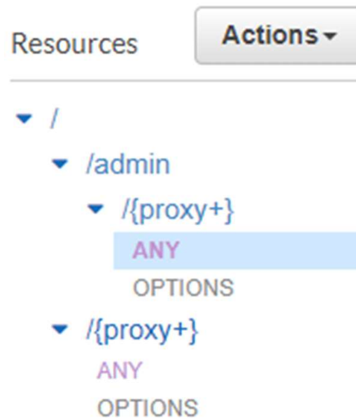
U Cognito servisu definiran je okidač (eng. trigger) da prije nego što se zapiše zahtjev za registraciju u sam Cognito servis pokrene „pre-signup-lambda“ koja će korisnika zapisati u DynamoDB tablicu. Ova funkcionalnost služi kako bi odobravanje ili odbijanje ovog zahtjeva za registraciju mogao provesti administrator web aplikacije. Ako je korisnik odobren od strane administratora, on se može prijaviti na portal.

Kod prijave korisnik dobije set ključeva koji se povezuju s 'AdminRola' rolom koja definira pristup svim servisima i metodama.

Osim ograničenja kojim resursima korisnik može pristupiti, ako je prijavljen, ograničene su i funkcionalnosti unutar frontend dijela web aplikacije. Tako su određene stranice i ostale funkcionalnosti, kao što je na primjer kreacija članaka, članova i turnira dostupne samo prijavljenim korisnicima

4.4. API, Lambda i baza podataka

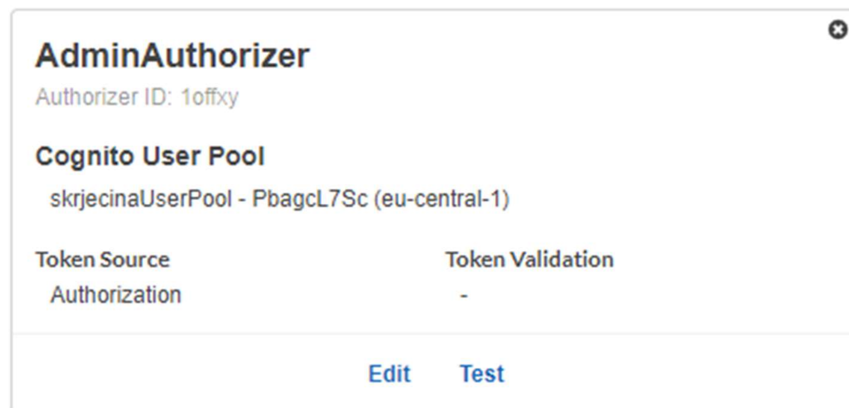
API Gateway je servis koji se, kako je već spomenuto, koristi za kreaciju API. Taj API definira pristup ostalim servisima koji su potrebni za rad web aplikacije (Slika 8).



Slika 8 - Struktura API

Proxy+ definiran *endpoint* prosljeđuje zaprimljeni zahtjev na AWS Lambda funkcije. Definirana su dva *endpointa*: /{proxy+} i /admin/{proxy+}.

Prvi *endpoint* prosljeđuje zahtjeve Lambda funkciji *'PublicFunction'* koja je namijenjena korisnicima koji nisu prijavljeni u web aplikaciju. Tako će zahtjev na /clanovi vratiti kao odgovor listu svih aktivnih članova.



Slika 9 - Autorizacija pomoću Cognita

Na tom *endpointu* je definirana zaštita pomoću Cognita (Slika 9). Korisnik koji šalje ovaj zahtjev unutar *headera* tog zahtjeva treba priključiti privremeni token koji je dobio od Cognito Identity Poola.

Ovim se sprječava mogućnost zagušivanja *endpointa* sa zahtjevima i time povećanje troška. Drugi *endpoint* je namijenjen administratorima web aplikacije. Tako, recimo, zahtjev na /admin/clanovi, za razliku od /clanovi, vraća listu aktivnih i neaktivnih članova kluba.

Pristup ovom *endpointu* korisnik dobije pomoću ključeva koje dobije od Cognito User Poola, koji definiraju mogućnost pristupa ovom *endpointu*.

Nakon što API proslijedi zahtjev na AWS Lambdu, pokreće se jedna od dvije kreirane funkcije: *'PublicFunction'* ili *'AdminFunction'*. Unutar programskog koda nalaze se rute (Slika 10) koje definiraju koja će se klasa i metoda unutar klase pokrenuti kod određenog zahtjeva. Tako na primjer zahtjev GET na endpoint /clanovi će pokrenuti Lambda funkciju *'PublicFunction'* koja unutar sebe ima definiranu klasu *Clanovi*, a ta klasa ima definiranu metodu GET.

```
api.add_resource(api_public.Turniri, '/turniri')
api.add_resource(api_public.Novosti, '/novosti')
api.add_resource(api_public.Clanovi, '/clanovi')
api.add_resource(api_admin.AdminTurniri, '/admin/turniri')
api.add_resource(api_admin.AdminNovosti, '/admin/novosti')
api.add_resource(api_admin.AdminClanovi, '/admin/clanovi')
api.add_resource(api_admin.AdminAdmini, '/admin/admini')
```

Slika 10 - Rute

4.5. Resursi

Svi potrebni resursi su kreirani pomoću Cloudformation template-a, koji se zatim učitava preko AWS Cloudformation servisa te kreira definirane resurse.

Kreirani resursi su:

1. S3 bucket
2. AWS Cognito
    a. User Pool
    b. Identity Pool
3. AWS IAM
    a. Admin Role
    b. Public Role
4. AWS Cloudfront
    a. Web distribucija
5. AWS Lambda
    a. Pre-Signup Funkcija
    b. Proxy Funkcija
    c. Admin Funkcija
6. AWS DynamoDB
    a. PortalClanovi
    b. PortalAdmin
    c. PortalNovosti
    d. PortalTurniri

```yaml
PortalClanovi:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: "PortalClanovi"
    AttributeDefinitions:
      - AttributeName: id
        AttributeType: S
    KeySchema:
      - AttributeName: id
        KeyType: HASH
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
```

Slika 11 - Definicija DynamoDB tablice

DynamoDB tablici je definirano ime, atribut ključa i ograničenje čitanja i pisanja (Slika 11). Ograničenje se odnosi na maksimalni broj zahtjeva za čitanje i pisanje po sekundi prije nego DynamoDB servis, kao odgovor, vrati grešku zagušenja.

5. Korištene tehnologije

5.1. JavaScript

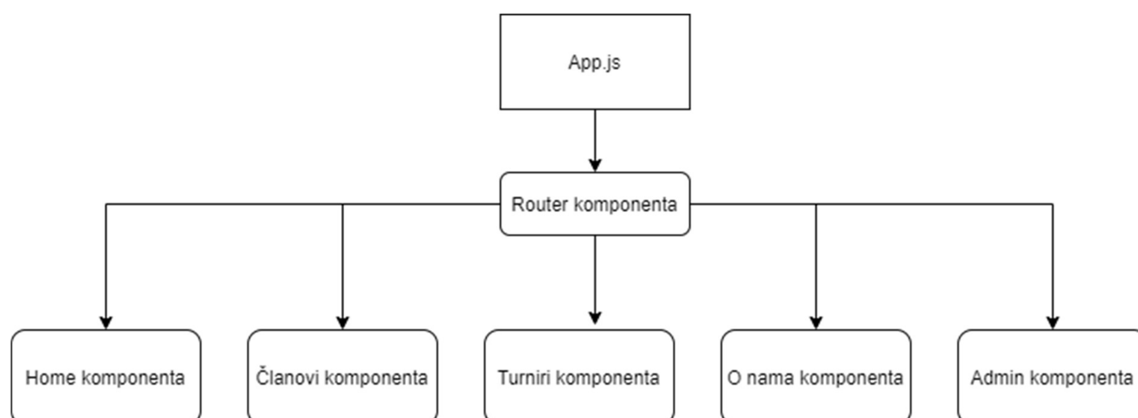
JavaScript(JS) je interpretirani ili na vrijeme kompilirani programski jezik. Najpoznatije korištenje je kao skriptni jezik za web stranice koji podržava objektno orijentirano programiranje. Standard pisanja koda u JavaScript-u je ECMAScript, a posljednja verzija ECMAScript-a je 6, pa se sintaksa pisanja u toj verziji češće naziva ES6 (JavaScript 2020).

5.2. React

Za razliku od Angular frameworka, React je biblioteka izrađena kao open-source projekt Facebooka 2013. godine (The Next Web 2020). Ako bi gledali prema MVC modelu, React je *view* komponenta tog modela.

React je deklarativna i fleksibilna JavaScript biblioteka za kreiranje korisničkog sučelja. Ona dozvoljava developeru razvoj kompleksnog korisničkog sučelja pomoću malih izoliranih dijelova koda koji se nazivaju komponentama (ReactJS Tutorial 2020). Pomoću komponenti određujemo što će biti prikazano na ekranu, a kod promjene podataka pokreće se funkcija *render*, koja iscrtava komponente na ekranu.

Korištenjem React frameworka moguće je razviti SPA – *single-page application*. To je web aplikacija, ili stranica, koja ima interakciju s preglednikom tako da dinamički ažurira trenutnu web stranicu s podatcima koje prima s web servera (Wikipedia SPA 2020). Web aplikacija kreirana u sklopu ovog diplomskog rada je SPA (Slika 12).



Slika 12 - SPA arhitektura web aplikacije

5.2.1. Kreiranje React aplikacije

Kako bi se kreirala React aplikacija na računalu potrebno je imati instaliranu verziju 8.10 Node.js i npm verziju 5.6 (ReactJS 2020) te zatim u terminalu pokrenuti naredbu *npx create-react-app ImeAplikacije*. Tada se kreira projekt pomoću Babela i webpacka. Babel pretvara ECMAScript kod u kompatibilnu verziju JavaScripta (BabelJS 2020) dok webpack kreira graf ovisnosti, to jest, mapira module projekta i kreira jedan ili više snopova (eng. bundles). (Webpack 2020)

5.2.2. Razvoj frontenda

Programski kod frontend dijela aplikacije definira što će korisnik vidjeti na ekranu kod pristupanja web aplikaciji. Definiraju se komponente, pogledi(eng. views), HTTP zahtjevi prema API Gateway resursu i registracija i autentifikacija korisnika.

Programski kod je razvijen pomoću IDE Visual Studio Code koji je dostupan za Windows, macOS i Linux operativne sustave. Podržava razvoj u JavaScript, TypeScript i Node.js programskim jezicima uz mogućnost proširenja za jezike poput C++, C#, Python, PHP i ostale (Visual Studio Code 2020).

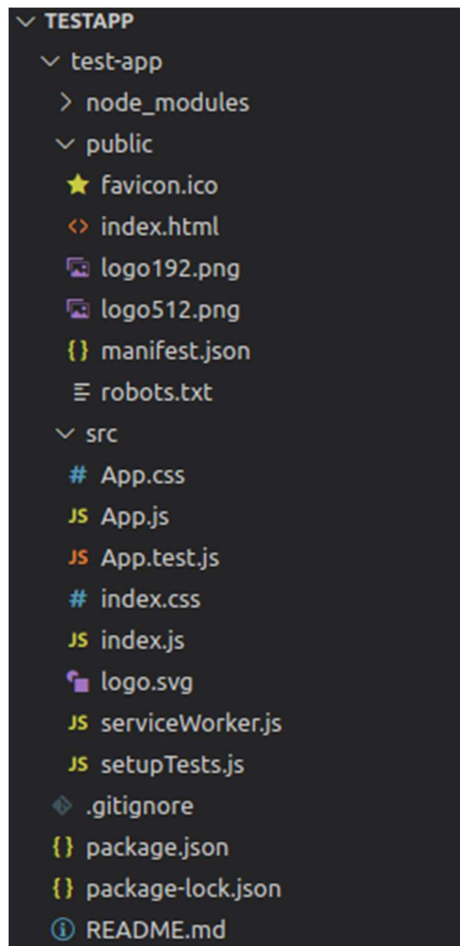
5.2.3. Arhitektura frontenda

Nakon završetka kreacije aplikacije kreiran je pripremljeni projekt s određenim datotekama i mapama (Slika 13).

Mapa *public* je mjesto gdje se nalaze statične datoteke. Ako se datoteka ne uvozi u *javascript* datoteke i mora zadržati originalno ime, treba se spremiti u ovu mapu.

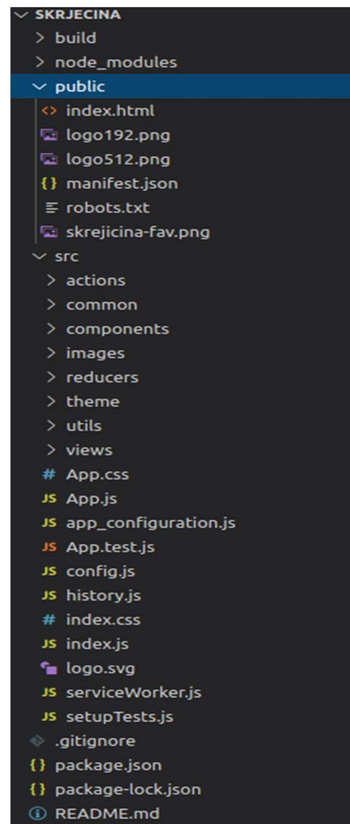
Mapa node*_modules* je mapa u koju se spremaju paketi instalirani pomoću npm-a.

Mapa *src* je mjesto gdje se nalaze dinamične datoteke, to jest, *javascript* datoteke koje mijenjaju sadržaj.



Slika 13 - Početna struktura

Ovo je početni kostur arhitekture frontend dijela web aplikacije, a na slici 14 je prikazan konačan raspored datoteka i mapa unutar projekta.



Slika 14 - Konačna struktura

Unutar *src* mape dodane su mape: *actions*, *common*, *componentes*, *images*, *reducers*, *theme*, *utils* i *views*.

Mapa *actions* sadrži datoteke u kojima su definirani HTTP zahtjevi prema backend dijelu web aplikacije. Unutar *common* mape nalazi se datoteka s definiranim bojama.

Mapa *components* sadrži komponente koje su korištene za prikaz pojedinih dijelova pogleda web aplikacije. Mapa *images* i *theme* sadrže slike i definiciju boja koje su korištene unutar komponenti. Mapa *utils* sadrži funkcije koje su često korištene. Primjer takve funkcije bila bi funkcija koja radi konverziju formata vremena.

Mapa *reducers* sadrži datoteke koji koristi biblioteka Redux, koji je objašnjen u poglavlju 5.2.7.

Posljednja kreirana mapa je *views*. Ona sadrži *javascript* datoteke koje sadrže kao komponente koje predstavljaju jednu stranicu web aplikacije.

5.2.4. Komponente

Kako je već spomenuto komponente su dijelovi koda koji su izolirani i postoje samostalno. Svaku od komponenti moguće je gledati kao zasebnu klasu sa svojim atributima i funkcijama koje ta klasa koristi.

Komponente ne moraju biti klase, komponenta može biti i funkcija koja vraća druge komponente s obzirom na primljene vrijednosti.

Vrijednosti koje su spremljene unutar jedne komponente moguće je proslijediti „dijete" komponenti u obliku objekta s ključ-vrijednost elementima. Objekt *props* je vrijednost koju komponenta prima od komponente roditelja, a može sadržavati vrijednosti, funkcije ili druge komponente.

5.2.4.1. Funkcijska komponenta

Primjer koda funkcijske komponente prikazan je na slici 15. Ova komponenta je funkcija koja vraća modalni prozor, definiran kao posebna komponenta *Dialog*, te prima jedan argument – *props*.

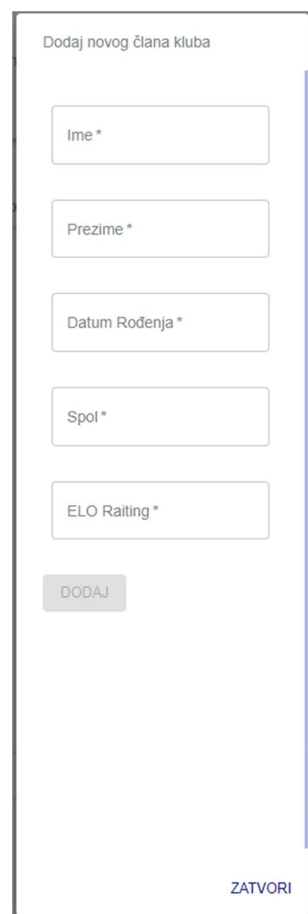
```
import React from "react";
import Button from "@material-ui/core/Button";
import Dialog from "@material-ui/core/Dialog";
import DialogActions from "@material-ui/core/DialogActions";
import DialogContent from "@material-ui/core/DialogContent";
import DialogTitle from "@material-ui/core/DialogTitle";

function AlertDialog(props) {
  const { children, open, handleClose, title } = props;

  const renderSecondaryButton = () => {
    if (
      props.hasOwnProperty("handleSecondary") &&
      props.hasOwnProperty("secondaryText")
    ) {
      return (
        <Button onClick={props.handleSecondary} color="primary">
          {props.secondaryText}
        </Button>
      );
    } else {
      return "";
    }
  };

  return (
    <Dialog
      open={open}
      onClose={handleClose}
      aria-labelledby="alert-dialog-title"
      aria-describedby="alert-dialog-description"
    >
      <DialogTitle id="alert-dialog-title">{title}</DialogTitle>
      <DialogContent id="alert-dialog-description">{children}</DialogContent>
      <DialogActions>
        {renderSecondaryButton()}
        <Button onClick={handleClose} color="primary">
          {"Zatvori"}
        </Button>
      </DialogActions>
    </Dialog>
  );
}

export default AlertDialog;
```

Slika 15 - Funkcijska komponenta

Unutar ovog objekta se nalaze 4 elementa – *children* koji predstavlja komponentu koja će se prikazivati unutar modalnog prozora, *open* – *boolean* vrijednost koja označava kada se komponenta prikazuje, to jest, ako je vrijednost true, modalni prozor će se prikazati.

Funkcija *handleClose* mijenja vrijednost *open* varijable u roditelj komponenti, a pokreće se ako korisnik klikne na gumb Zatvori i title – string, koji će se prikazati kao naslov modalnog prozora.

Vizualni prikaz ove komponente se može vidjeti na Slika 16. Vrijednost *children* elementa je u ovom slučaju forma u kojoj se zapisuju podatci o članu kluba.



Slika 16 - Modalni prozor

5.2.4.2. Komponenta klase

Druga vrsta komponente je komponenta klase. U pogledu što se prikazuje na ekranu korisnika, nema razlike s funkcijskom komponentom, no ovakva komponenta nam omogućuje spremanja

vrijednosti unutar stanja (eng. *state*). Stanje je objekt u kojem se spremanju vrijednosti koje pripadaju komponenti.

Stanje se inicijalizira u konstruktoru klase, a ,ako se neka od vrijednosti unutar stanja promijeni, pokreće se *render* funkcija te se time ažurira trenutni prikaz (W3Schools 2020).

Za primjer ovakve komponente uzet je programski kod forme koja je bila *children* element u prethodnom poglavlju.

Na slici Slika 17 je prikaz klase *NoviClan* i konstruktora te klase u kojem se definira stanje pomoću funkcije *defaultState*. Ta funkcija vraća objekt koji postavlja početne vrijednosti kako bi se prikazala komponenta.

```
class NoviClan extends Component {
  constructor(props) {
    super(props);

    this.state = this.defaultState();
  }

  defaultState() {
    return {
      values: {
        ime: "",
        prezime: "",
        datum: "",
        spol: "",
        raiting: "",
      },
      isValid: false,
      loading: false,
    };
  }
```

Slika 17 - Klasa NoviClan

Unutar *render* funkcije definiran je *props* koji ova komponenta prima (Slika 18). Funkcija vraća polje za tekst, kojemu je definirana vrijednost *values.ime*. Ta vrijednost je referenca na stanje ove komponente.

```
render() {
  const { classes, loading, message, removeMessage } = this.props;
  const { values, isValid } = this.state;

  return (
    <div className={classes.root}>
      <Box m={1}>
        <TextField
          variant="outlined"
          margin="normal"
          required
          fullWidth
          value={values.ime}
          name="ime"
          label="Ime"
          id="ime"
          onChange={(event) => {
            this.inputHandler("ime", event.target.value);
          }}
        />
      </Box>
```

Slika 18 - render funkcija

Kada dođe do promjene unutar ovog polja za tekst, poziva se funkcija *inputHandler* koja prima argumente *ime* i *event.target.value*. Na slici (Slika 19) je prikazana funkcija. Prvi argument služi kako bi se odredio koji od polja za tekst je primio novu vrijednost, to jest, koju vrijednost unutar stanja ove komponente treba ažurirati. Drugi argument je vrijednost koju je korisnik upisao u polje za tekst.

```
inputHandler = (field, value) => {
  const newState = { ...this.state };

  newState.submitError = null;
  newState.values[field] = value;
  if (
    newState.values.ime !== "" &&
    newState.values.prezime !== "" &&
    newState.values.datum !== "" &&
    newState.values.spol !== "" &&
    newState.values.raiting !== ""
  ) {
    newState.isValid = true;
  }
  this.setState(newState);
};
```

Slika 19 - inputHandler funkcija

Na početku funkcije radi se kopija trenutnog stanja, a zatim postavlja vrijednost koju je korisnik upisao, kao vrijednost atributa stanja *values.ime* u slučaju da je prvi argument ime. Unutar ove funkcije također se radi provjera da su sva polja popunjena, to jest, različita je vrijednost svakog

atributa od početne vrijednosti koja je definirana kao prazan *string*. Ako su sva polja popunjena vrijednost atributa *isValid* se postavlja na *true*, a gumb Dodaj je onemogućen dok vrijednost ovog atributa u stanju nije različit od *false*. Zatim se novo stanje postavlja pomoću funkcije *this.setState* koja za argument prima objekt koji opisuje novo stanje komponente.

5.2.5. Životni tijek komponente

Osim funkcija koje je programer razvio za komponentu, postoje funkcije koje su definirane kao funkcije životnog tijeka (eng. Lifecycle). Ove funkcije se pozivaju u određeno vrijeme postojanja komponente na ekranu.

Tako funkcija *componentDidMount*() se pokreće kod prvog prikazivanja komponente, a koristi se kada komponenta treba asinkrono primiti podatke s backenda ili je potrebno postaviti početne vrijednosti stanja.

Funkcija *componentDidUpdate*() se poziva prije svakog poziva *render* funkcije. Implementacija ove metode prikazana je na slici (Slika 20). Funkcija prima argument *prevProps*, koji predstavlja objekt *props*, koji je komponenta imala nakon posljednjeg završetka izvođenja funkcije *render*. Unutar funkcije izvode se provjere ako su određene vrijednosti unutar *prevProps* objekta različite od trenutnih te se time određuje hoće li se pokrenuti dio koda koji će prikazati *pop-up* poruku korisniku. Ova funkcija je korisna za provjeru rezultata asinkronih poziva na backend, to jest, je li HTTP zahtjev završio uspješno ili se dogodila greška.

```
componentDidUpdate(prevProps) {
  if (prevProps.error !== this.props.error && this.props.error) {
    this.setState({ requestLoading: false });
    this.props.pushMessage({ message: this.props.error, type: "error" });
    this.props.clearError();
  }
  if (prevProps.success !== this.props.success && this.props.success) {
    this.props.pushMessage({
      message: this.props.success,
      type: "success",
    });
    this.props.clearSuccess("ClanoviPost");
    this.setState({ requestLoading: false });
    this.setState(this.defaultState());
  }
}
```

Slika 20 - componentDidUpdate funkcija

5.2.6. App.js

Kod kreiranja React aplikacije u *root* direktoriju se kreiraju *indeks.html* i *App.js* datoteke. Datoteka *index.html* predstavlja polazišnu točku za web aplikaciju, dok je unutar *App.js* datoteke definirana *root* komponenta koja sadrži sve ostale komponente koje su kreirane u web aplikaciji.

Unutar *App.js* datoteke definirane su javne i privatne komponente, na vrhu *render* funkcije je dodana komponenta navigacije (zbog toga se na vrhu stranice prikazuje navigacija), a zatim pomoću *Router* komponente su definirane rute, to jest, za koju rutu će se prikazivati koja komponenta (Slika 21).

```
<Router history={history}>
  <Navigation />

  <Switch style={{ display: "flex" }}>
    <Redirect exact from="/" to="/home" />
    <PrivateRoute
      component={Admin}
      exact
      path="/admini"
      authStatus={authenticated}
    />

    <PublicRoute
      component={SignUp}
      exact
      path="/sign-up"
      authStatus={authenticated}
    />
```

Slika 21 - definiranje ruta

Tako će se za rutu /admini prikazati privatna komponenta koja će, kako je prikazano na Slici 22, ili prikazati *Admin* komponentu ili će korisnika preusmjeriti na /sign-in rutu s obzirom na to je li korisnik prijavljen ili nije.

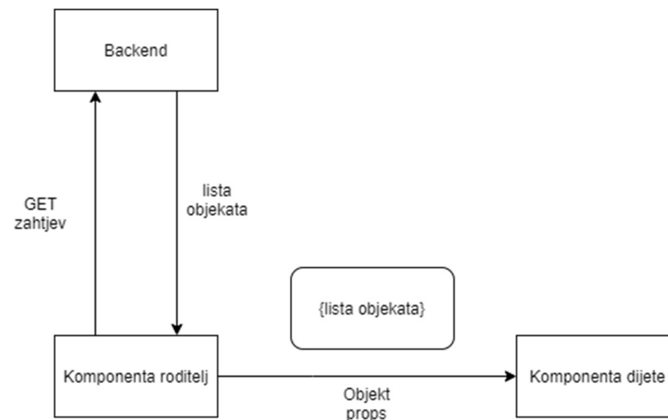
```
const PrivateRoute = ({ component: ReactComponent, authStatus, ...rest }) => {
  return (
    <Route
      {...rest}
      render={(props) => {
        return typeof authStatus === "undefined" || authStatus === false ? (
          <Redirect
            to={{
              pathname: "/sign-in",
              state: {
                from: props.location,
              },
            }}
          />
        ) : (
          <ReactComponent {...props} />
        );
      }}
    />
  );
};
```

Slika 22 - Definicija privatne rute

5.2.7. Redux

Kako je već navedeno, komponente imaju svoje stanje te mogu prosljeđivati stanje komponentama koje su njihova djeca. S takvim pristupom u izradi web aplikacije u React-u dolazi se do problema upravljanja stanja.

S tim problemom se suočava kada se želi iskoristiti podatke koje je jedna komponenta dobila kroz HTTP zahtjev na backend u drugoj komponenti. Ako je komponenta dijete komponente koja je napravila poziv, to je moguće jednostavno napraviti slanjem podataka pomoću objekta *props* (Slika 23), no, ako komponenta nije dijete, tada dolazi do problema.



Slika 23 - Prosljeđivanje odgovora dijete komponenti

Problem se može riješiti procesom podizanja stanja, u kojem je u komponenti roditelju definirana funkcija koja primljeni argument postavlja kao svoje stanje, a funkciju prosljeđuje u komponentu dijete kroz objekt *props*. No takvo rješenje je dostatno za jedno podizanje stanja, no, ako se komponenta nalazi duboko u hijerarhiji aplikacije, potreban je drugačiji pristup.

Za rješavanje tog problema koristi se biblioteka Redux. Ona omogućava da se web aplikaciji kreira objekt zvan *store.* Taj objekt je moguće gledati kao globalno stanje cijele web aplikacije. Dakle, ako postoji varijabla spremljena unutar *store* objekta, sve komponente unutar aplikacije mogu pristupiti toj vrijednosti. Potrebno je mapirati *store* u *props* objekt komponente pomoću funkcije *mapStateToProps*() koja kao argument prima *store* objekt (Slika 24).

```
function mapStateToProps(state) {
  return {
    authenticated: state.auth.authenticated,
    success:
      state.data.successClanoviPost === undefined
        ? {}
        : state.data.successClanoviPost,
    error: state.data.error === undefined ? {} : state.data.error,
    message: state.common.message,
  };
}

NoviClan = connect(mapStateToProps, {
  createClan,
  getClanovi,
  clearSuccess,
  removeMessage,
  pushMessage,
  clearError,
})(NoviClan);

export default withStyles(styles)(withRouter(NoviClan));
```

Slika 24 - funkcija mapStateToProps

Ova funkcionalnost je u web aplikaciji iskorištena kako bi se spremili podatci koji su dobiveni zahtjevima na backend. Kada se napravi GET zahtjev za popis članova, rezultat će se spremiti u Redux *store*.

Kako bi se to postiglo, potrebno je mapirati vrijednosti unutar objekta *store* sa vrijednostima određenih zahtjeva. Svaki HTTP zahtjev je definiran kao *action* koji ima definiran *type*. *Type* je proizvoljno imenovana konstanta koja je zapisana u datoteci *types.js* (Slika 25).

```
1   // COMMON TYPES
2   export const PUSH_NOTIFICATION = "PUSH_NOTIFICATION";
3   export const REMOVE_MESSAGE = "REMOVE_MESSAGE";
4
5
6   // AUTH TYPES
7   export const AUTH_USER = "AUTH_USER";
8   export const AUTH_MFA = "AUTH_MFA";
9   export const AUTH_NEW_PASSWORD_REQUIRED = "AUTH_NEW_PASSWORD_REQUIRED";
10  export const UNAUTH_USER = "UNAUTH_USER";
11  export const REGISTER_USER = "REGISTER_USER";
12  export const REGISTER_USER_CONFIRM = "REGISTER_USER_CONFIRM";
13  export const REGISTER_MFA = "REGISTER_MFA";
14  export const REGISTER_USER_ERROR = "REGISTER_USER_ERROR";
15  export const FORGOT_PASSWORD = "FORGOT_PASSWORD";
16  export const FORGOT_PASSWORD_CONFIRM = "FORGOT_PASSWORD_CONFIRM";
17  export const AUTH_ERROR = "AUTH_ERROR";
18  export const AUTH_ERROR_CLEAR = "AUTH_ERROR_CLEAR";
19  export const IS_LOADING = "IS_LOADING";
20  export const LOGIN_REGISTER_CONFIRMATION = "LOGIN_REGISTER_CONFIRMATION";
21
22  // Clanovi
23  export const GET_CLANOVI = "GET_CLANOVI";
24  export const POST_CLANOVI = "POST_CLANOVI";
25  export const PUT_CLANOVI = "PUT_CLANOVI";
26  // Novosti
27  export const GET_NOVOSTI = "GET_NOVOSTI";
28  export const POST_NOVOSTI = "POST_NOVOSTI";
29  // Turniri
30  export const GET_TURNIRI = "GET_TURNIRI";
31  export const POST_TURNIRI = "POST_TURNIRI";
32  // Admini
33  export const GET_ADMINI = "GET_ADMINI";
34  export const PUT_ADMINI = "PUT_ADMINI";
35  // Common
36  export const ADMIN_ERROR = "GET_ADMIN_ERROR";
37  export const CLEAR_SUCCESS = "CLEAR_SUCCESS";
38  export const CLEAR_ERROR = "CLEAR_ERROR";
39
```

Slika 25 - Redux types

Ove tipove koristimo u *reducerima* koji su spomenuti u poglavlju 5.2.2. *Reducer* je funkcija koja prima trenutno stanje i objekt koji je proizvela, ažurira stanje s obzirom kako je definirano unutar funkcije te vraća novo stanje (Redux 2020).

5.2.8. Slanje HTTP zahtjeva

HTTP zahtjevi su korišteni kako bi se prikupili podatci iz baze podataka koji će se zatim prikazati korisniku ili kako bi administrator mogao kreirati novi zapis. Tako se za prikazivanje liste članova kluba, liste turnira i liste zahtjeva za administratora koristi HTTP metoda GET, za kreaciju novog člana metoda POST, a za odobravanje administratora i deaktiviranje člana se koristi metoda PUT.

5.2.8.1. AWS Amplify

Kako bi se poslao zahtjev prema backendu, šalje se zahtjev na API kreiran pomoću servisa API Gateway, to jest, na domenu na kojoj se nalazi. Za to se koristi JavaScript biblioteka AWS Amplify koja omogućuje slanje HTTP zahtjeva REST *endpointu* kreiranog API (AWS Amplify 2020).

Kako bi se mogla koristiti AWS Amplify biblioteka, potrebno je kreirati config.js datoteku koja sadrži objekt s imenima, identifikacijama i nazivima domena AWS Cognito i AWS API Gateway resursa i pri učitavanju web aplikacije pozvati funkciju *Amplify.configure()* koja kao parametar prima taj objekt. Time biblioteka može raditi zahtjeve prema resursima koji su kreirani.

Svaki od rezultata zahtjeva potrebno je povezati s Reduxom, a to se ostvaruje korištenjem funkcije *dispatch()*, koja kao argument prima objekt akcije, koji se sastoji od ključ-vrijednosti

type i *payload*. Za vrijednost tip postavljena je vrijednost koja je definirana u datoteci *types.js*, a za vrijednost *payload* vrijednost *response*.

Varijabla response je vrijednost koju funkcije modula AWS Amplify API primaju kao rezultat HTTP zathjeva na API Gateway. Modul API sadrži funkcije zvane po metodama kao što su GET, POST, PUT, DELETE i PATCH, a kao argument prima *name* – domena API, ruta – *endpoint* API koji treba pozvati i HTTP parametre, koji mogu *biti body, queryStringParameters* i *headers*.

Na slici 26 je prikazan HTTP zahtjev koji administrator poziva kako bi kreirao novog člana kluba, parametar ruta je /admin/clanovi, a HTTP parametar *body* sadrži objekt zapisan u JSON formatu koji sadrži ime, prezime i ostale atribute člana.

Pomoću ključne riječi *await* definirana je asinkrona funkcija koja svoje izvođenje neće završiti dok ne dobije rezultat. Nakon što primi rezultat, s obzirom na to je li zahtjev prošao uspješno, poziva se funkcija *.then()* u kojoj se poziva Redux funkcija *dispatch* kako bi se ažuriralo stanje aplikacije s rezultatom zahtjeva.

Ako je zahtjev rezultirao pogreškom pomoću funkcije *.catch()*, pogreška se ispisuje u terminal aplikacije i, također, se poziva Redux funkcija *dispatch()* kako bi se zapisala pogreška u stanje aplikacije.

Administrator web aplikacije mora poslati JWT Token kako bi uspješno napravio zahtjev na /admin/{proxy+} *endpoint* na API. Taj token je uključen u *headeru* pod ključem *Authorization*. Kako je taj token spremljen u Redux *store* objašnjeno je u nastavku.

```
export function createClan(data) {
  return async function (dispatch, getState) {
    const token = _.get(
      getState(),
      "auth.cognitoUser.signInUserSession.idToken.jwtToken",
      false
    );
    await API.post(name, "/admin/clanovi", {
      body: {
        data,
      },
      headers: { Authorization: token },
    })
      .then((response) => {
        dispatch({
          type: POST_CLANOVI,
          payload: response,
        });
      })
      .catch((error) => {
        console.log(error);
        console.error("createClan() AdminActions.createClan Error: ", error);
        dispatch(adminError(error));
      });
  };
}
```

Slika 26 - akcija createClan

5.2.9. Autentifikacija

Za autentifikaciju korisnika koriste se tri akcije. Jedna za registraciju korisnika u Cognito User Pool, druga za prijavu korisnika i treća koja validira je li korisnik u sesiji još uvijek autentificiran.

Za sve tri akcije koristi se modul unutar AWS Amplify biblioteke zvan *Auth*. Akcija *Signup* i *SignIn* su slične. Obje primaju parametre *username* i *password*, odnosno korisničko ime i lozinku koju korisnik upisuje u formu koja se nalazi na rutama /sign-in i /sign-out, zatim se, kao i u primjeru za HTTP zahtjeve pomoću *dispatch()* funkcije Redux-a, spremaju *response* objekti u Redux *store*.

Obje akcije rade HTTP zahtjev direktno na Cognito User Pool, koji kao *response* vraća *cognitoUser* objekt. Taj objekt sadrži različite ključ-vrijednost elemente; *signInUserSession* je ključ koji se nalazi unutar Redux stanja nakon *SignIn* akcije, čija je pripadajuća vrijednost objekt koji sadrži *idToken*, *refreshToken* i *accessToken*.

Ovi tokeni su korišteni kako bi se provjerilo je li korisnik prijavljen i kako bi autorizirali admin HTTP zahtjeve. Autorizacija je postavljena na /admin/{proxy+} *endpoint* pomoću Cognito User Poola, a unutar *headera* zahtjeva dodan je JWT token atribut *idTokena*.

Pomoću *access* i *refresh* tokena validira se sesija korisnika, to jest radi se provjera treba li korisnik biti prijavljen u web aplikaciju. *Access token* ima vrijeme trajanja od jedan sat, a, ako korisnik ne kreira novi token osvježavanjem stranice, biti će odjavljen.

Refresh token ima zadano trajanje od 30 dana, a s njim korisnik može dobiti novi *access token*. Validnost tokena provjeravamo pomoću *validateUserSession*(Slika 27).

```
export function validateUserSession(status = "default") {
  return function (dispatch) {
    Auth.currentAuthenticatedUser({ bypassCache: true })
      .then((cognitoUser) => {
        Auth.userSession(cognitoUser)
          .then((session) => {
            if (session.isValid()) {
              dispatch({
                type: AUTH_USER,
                payload: cognitoUser,
              });
            } else {
              dispatch({
                type: UNAUTH_USER,
              });
            }
          })
          .catch((error) => {
            console.error(
              "UserSession() AuthActions.validateUserSession Error:",
              error
            );
            dispatch({
              type: UNAUTH_USER,
            });
          });
      })
      .catch((error) => {
        console.error(
          "CurrentAuthenticatedUser() AuthActions.validateUserSession Error:",
          error
        );
        dispatch({
          type: UNAUTH_USER,
        });
      });
  };
}
```

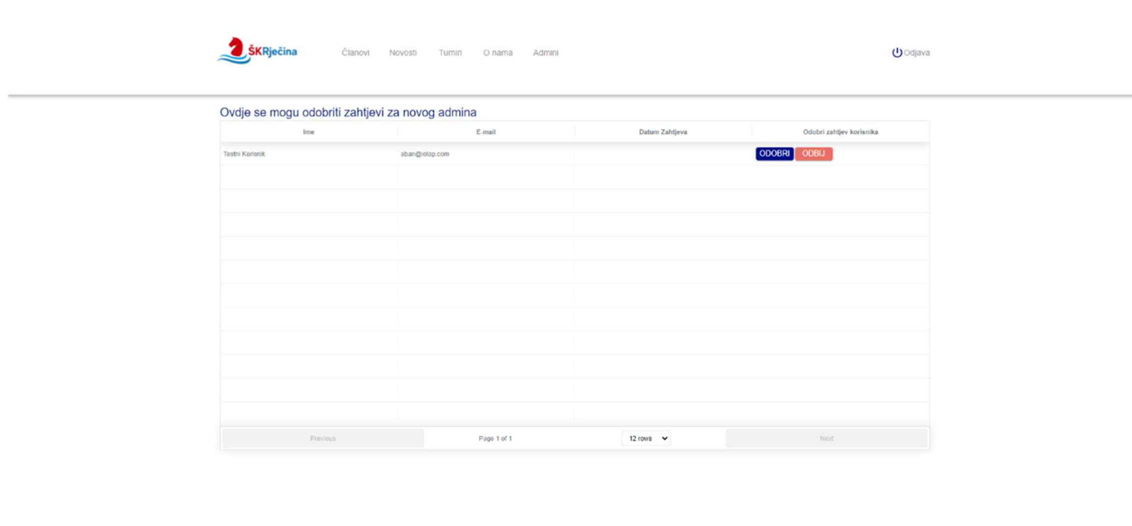
Slika 27 - validateUserSession funkcija

5.2.10. Material UI

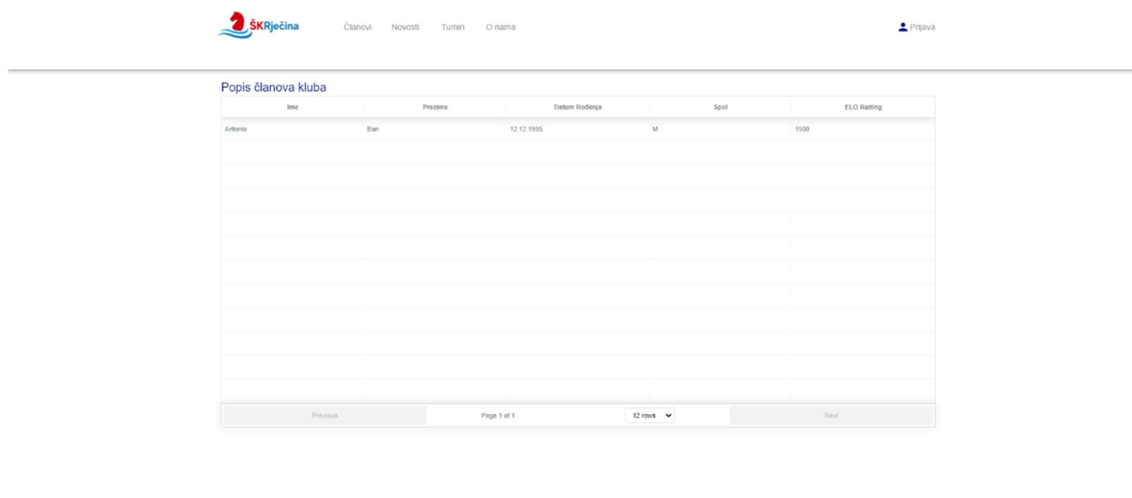
Material UI je biblioteka koja sadrži različite vrste uređenih komponenti koje se mogu koristiti unutar React aplikacije. Komponente koje sadrži Material UI su korištene kako bi se olakšala implementacija responzivnog dizajna s obzirom da se pomoću komponente Grid može definirati koliko će stupaca zauzimati komponenta koja se nalazi unutar komponente grid na skali od jedan do dvanaest. Brojčana vrijednost definira koliko dvanaestina zaslona će komponenta zauzimati, a moguće je definirati različite vrijednosti za četiri veličine ekrana: xs (<600px), sm(<960px), md(<1280px), lg(>1280px). Osim komponente grid, korištene su i druge komponente zbog jednostavnosti dodavanja stilova i dobre dokumentacije koja olakšava implementaciju novih komponenti.

5.2.11. Slike zaslona

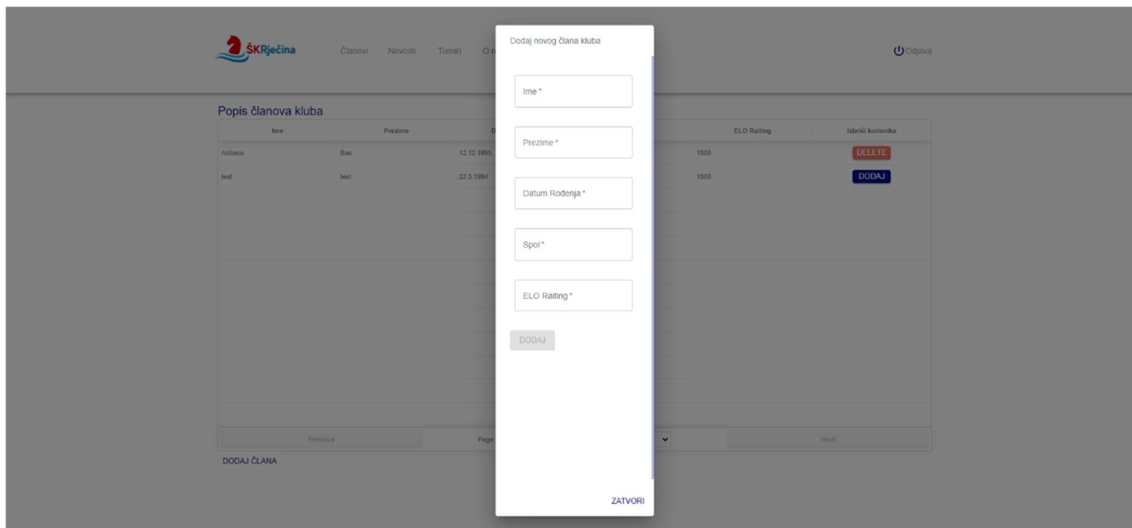
Na slikama 28-39 prikazane su slike zaslona kojima korisnici i administratori mogu pristupiti.



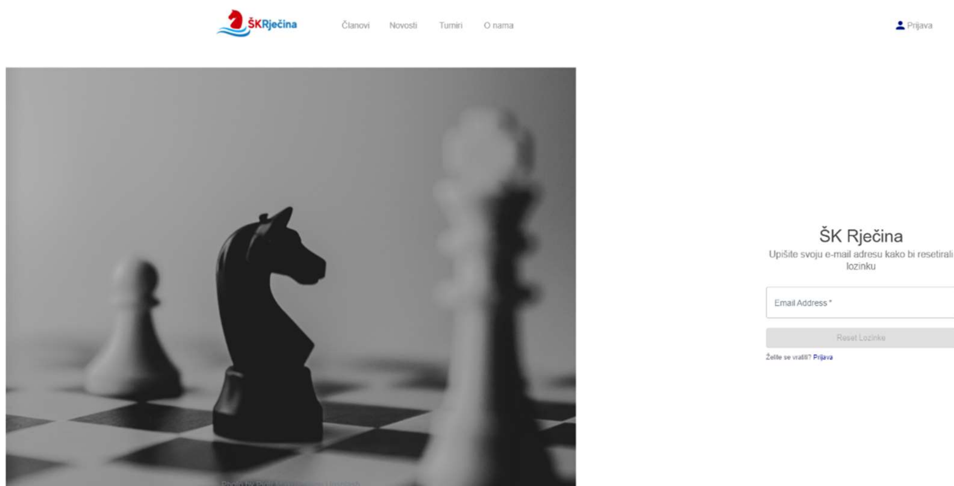
Slika 28 - Admin zaslon: odobravanje zahtjeva



Slika 29 - Javni zaslon: popis članova



Slika 30 - Admin zaslon: dodavanje člana



Slika 31 – Javni zaslon: zaboravljena lozinka

ŠKRječina Članovi Novosti Turniri O nama Prijava

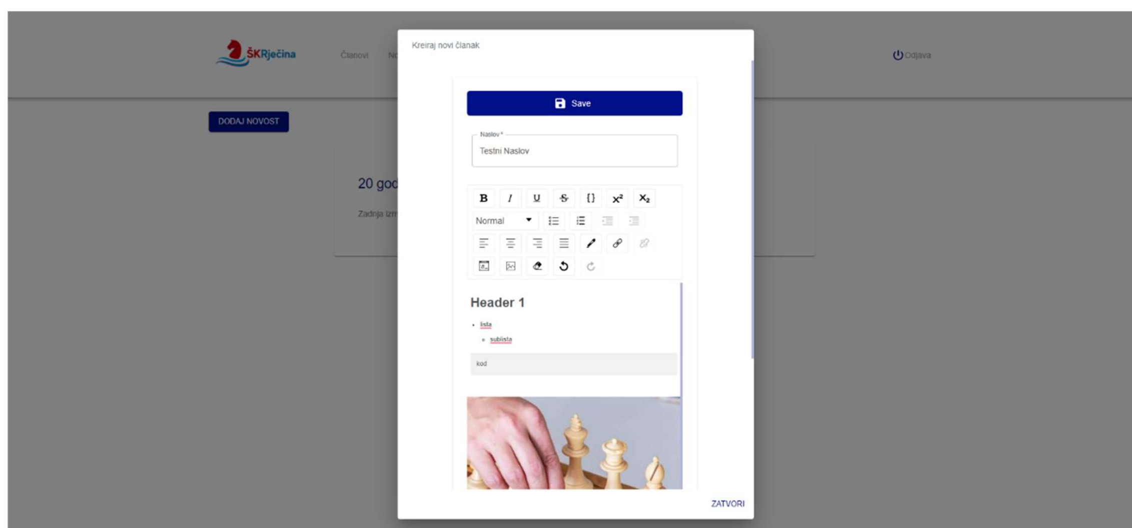
ŠK Rječina

Dobrodošli na stranicu šahovskog kluba Rječina koji se nalazi u općini Jelenje. Ovdje možete saznati više o klubu, njegovim članovima, novosti te informacije o održanim turnirima.

Šahovski klub Rječina - Dražice počeo je djelovati u jesen 1999. na inicijativu Gorana Mufića i Ljube Kukuljana, a službeno je osnovan 21.02.2000. god.

Možete nas pronaći na ovoj adresi

Slika 32 - Javni zaslon: početna stranica



Slika 33 - Admin zaslon: dodavanje novosti

ŠKRječina Članovi Novosti Turniri O nama Prijava

20 godina ŠK Rječine

Zadnja izmjena: 2020-08-28 09:32:23

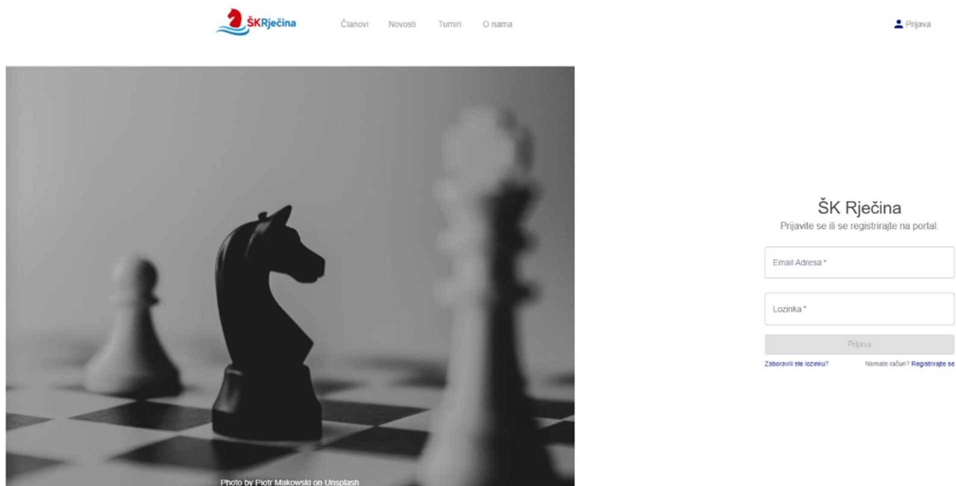
Slika 34 - Javni zaslon: popis novosti

ŠKRječina Članovi Novosti Turniri O nama Prijava

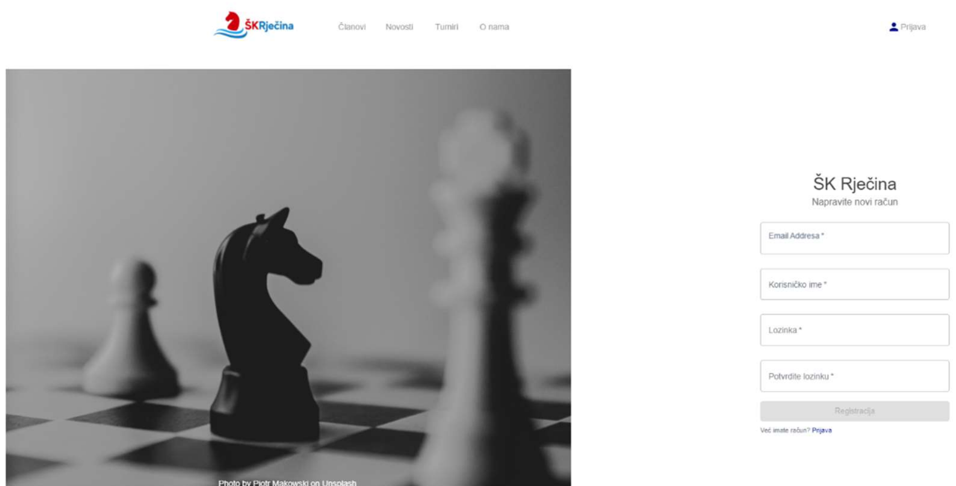
ŠK Rječina

Šahovski klub Rječina - Dražice počeo je djelovati u jesen 1999. na inicijativu Gorana Mufića i Ljube Kukuljana, a službeno je osnovan 21.02.2000. god. Zanimljivo je da je reprezentativac Hrvatske i najbolji igrač završnice nedavne Online Šahovske olimpijade - Marin Bosiočić, upravo protiv članova Rječine odigrao prvu simultanku u životu! Jedini remi tada je izvukao Vlatko Kukuljan koji je bio najuspješniji mladi igrač Rječine u prvim godinama djelovanja Kluba. Velike zasluge u početnim godinama djelovanja Kluba je imao i pokojni Ante Fućak, pa Rječina njemu u čast svake godine organizira memorijalni turnir. Od 2006. godine Klub počinje sustavno raditi s mladima, te razvija međunarodnu suradnju s klubovima u Slovačkoj i Austriji. Najveći su uspjesi bili u ženskoj kadetskoj konkurenciji jer su sestre Agata i Viktorija Vujić, te Jelena i Lana Dašić bile ekipne prvakinje Hrvatske! U pojedinačnoj konkurenciji Agata Vujić je bila 2. do 8 godina, a Viktorija Vujić i Jelena Dašić su bile 3. do 15 odnosno 17 godina. Najveći pojedinačni uspjeh Kluba je prvo mjesto u Hrvatskoj Lare Zagorac prošle godine u uzrastu do 9 godina, a velike pohvale zaslužuje i sadašnja kadetska ekipa Rječine - Lara Zagorac, Petar Mikulić, Luka Radić i Ivan Sinjeri za nedavnu pobijedu na otvorenom prvenstvu naše Županije! Uspješni su bili i seniori, koji su se ove godine po treći puta plasirali u 2. ligu Zapad, a najveći uspjeh je bilo 9. mjesto na otvorenom prvenstvu Slovačke u ubrzanom šahu 2009. godine ekipe u sastavu: Marin Relja, Božo Zrilić, Mario Štefan i Ivica Petrić.

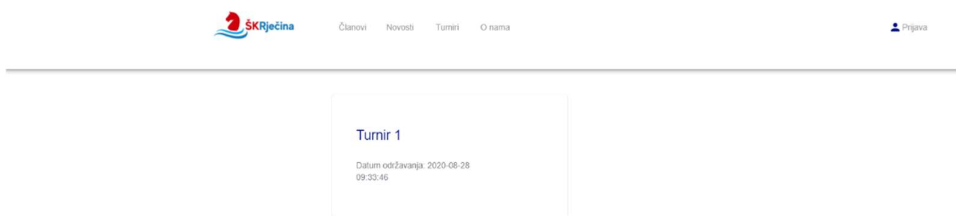
Slika 35 - Javni zaslon: o nama



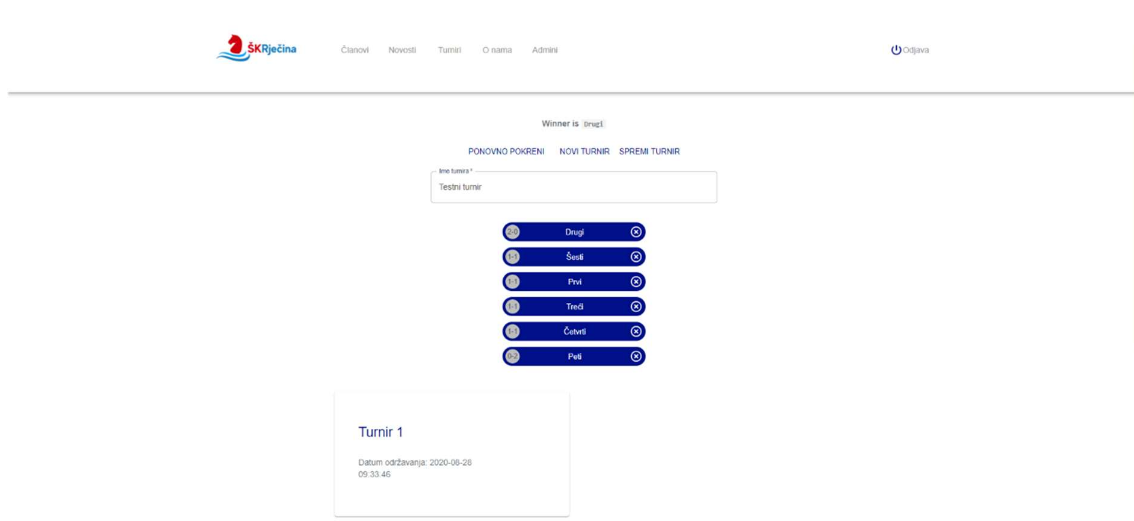
Slika 36 - Javni zaslon: prijava



Slika 37 - Javni zaslon: registracija



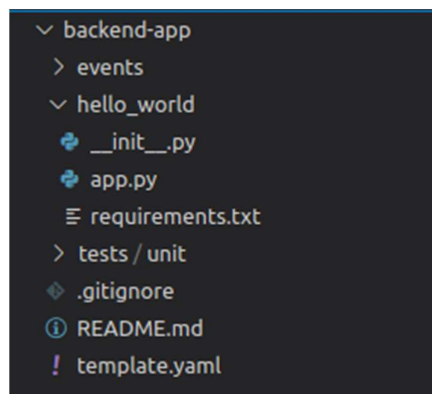
Slika 38 - Javni zaslon: popis turnira



Slika 39 - Admin zaslon: kreacija turnira

5.3. AWS Serverless Application Model (SAM)

Za izradu backend dijela web aplikacije korišten je servis AWS Serverless Application Model (SAM) (AWS SAM 2020), koji je open-source framework za razvoj *serverless* aplikacija. Pruža mogućnost kreiranja resursa pomoću AWS Cloudformation template-a i jednostavne komande za kreiranje i učitavanje aplikacije u oblak. Nakon instalacije SAM paketa na operativni sustav, projekt se izrađuje pomoću naredbe *sam init*, koja izradi projekt (Slika 40).



Slika 40 - projekt kreiran pomoću SAM alata

Projekt u ovom trenutku sadrži mapu *hello_world* u kojoj se nalaze datoteke *app.py* i *requirements.txt*. Datoteka *app.py* sadrži logiku Lambda funkcije, a *requirements.txt* sadrži popis *python* modula o kojima ovisi projekt. Datoteka *template.yaml* je AWS Cloudformation template koji sadrži definiciju *hello_world* Lambda funkcije i definiciju API s resursom /hello kojem je dodana metoda GET (Slika 41).

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function # More info about
    Properties:
      CodeUri: hello_world/
      Handler: app.lambda_handler
      Runtime: python3.8
      Events:
        HelloWorld:
          Type: Api # More info about API Event Source
          Properties:
            Path: /hello
            Method: get
```

Slika 41 - Definicija lambda funkcije i API resursa

Za projekt šahovske web aplikacije na isti način su definirane funkcije *AdminFunkcija* i *PublicFunkcija*, sa resursima /{proxy+} i /admin/{proxy+}, a metoda koja je dodana tim resursima je ANY – koja zatim podržava sve HTTP zahtjeve kao što su GET, POST, DELETE, PUT, PATCH.

5.3.1. REST API

REST je akronim za *Representational State Trasfer* (CodeAcademy 2020). To je arhitektonski stil za distribuirane hipermedijske sustave, a predstavljen je 2000. godina u disertaciji Roya Fieldinga (RESTful API 2020). API koji je kreiran na API Gatewayu je kolekcija resursa i metoda koji su integrirani sa HTTP *endpointovima*, Lambda funkcijama i ostalim AWS servisima. Ovaj API koristi zahtjev-odgovor model gdje klijent šalje zahtjev servisu, a servis odgovara nazad sinkrono (AWS API Gateway REST API 2020).

5.3.2. Flask i FlaskRESTful

Flask je web framework koji pruža različite alate, biblioteke i tehnologije koji dozvoljavaju razvoj web aplikacije (Flask 2020). Pomoću Flask klase, koja je dio ovog framework-a, kreira se Flask aplikacija instanciranjem iste unutar spomenute datoteke *app.py*. Pomoću biblioteke flask_cors aplikaciji se omogućuje CORS za sve definirane resurse i metode. FlaskRESTful je ekstenzija Flask koji daje podršku razvoju REST API u Pythonu.

U FlaskRESTful koristi se klasom *Resource*. Kreiraju se klase koje nasljeđuju klasu *Resource*. Svaka klasa može imati definirane metode kao što su GET, POST, PUT, DELETE, PATCH. Nakon kreacije klase i definiranja metode, ta se klasa dodaje u Flask API objekt i definira se za koji će se resurs API pozvati ta klasa.

Flask API objekt je kreiran pomoću Api klase koja je dio FlaskRESTful biblioteke. Inicijalizira se pomoću objekta koji predstavlja Flask aplikaciju.

Pomoću funkcije *add_resource()* dodaju se kreirane klase kao resurs aplikacije. Funkcija kao argumente prima: kreiranu klasu koja nasljeđuje *Resource* klasu i putanju kao što je naprimjer - /admin/clanovi.

Kreirani resursi detaljnije su objašnjeni u sljedećim poglavljima, a na slici 42 je programski kod implementacija Flask i FlaskRESTful, te definicija ruta.

```
from flask import Flask
from flask_restful import Api
from flask_cors import CORS
import awsgi
import route_handler
import api_public
import api_admin

app = Flask(__name__)
CORS(app)
api = Api(app)
api.add_resource(route_handler.Proxy, '/<path:content>')
api.add_resource(api_public.Turniri, '/turniri')
api.add_resource(api_public.Novosti, '/novosti')
api.add_resource(api_public.Clanovi, '/clanovi')
api.add_resource(api_admin.AdminTurniri, '/admin/turniri')
api.add_resource(api_admin.AdminNovosti, '/admin/novosti')
api.add_resource(api_admin.AdminClanovi, '/admin/clanovi')
api.add_resource(api_admin.AdminAdmini, '/admin/admini')

def lambda_handler(event, context):
    return awsgi.response(app, event, context, base64_content_types={"application/zip"})

if __name__ == '__main__':
    app.run()
```

Slika 42 - Implementacija Flask i FlaskRESTful

5.3.2.1. Pomoćna klasa DB

Klasa DB kreirana je zbog pojednostavljenja slanja zahtjeva prema DynamoDB servisu i manipulaciju podataka unutar kreiranih tablica. Klasa koristi boto3 biblioteku koja je AWS SDK za Python programski jezik (Boto3 2020).

5.3.2.2. Javne klase

Unutar web aplikacije nalaze se tri klase koje se pozivaju u slučaju poziva na rute: /turniri, /novosti i /clanovi. Svaka od ovih klasa sadrži metodu *get()*, primjer programskog koda metode za rutu /novosti može se vidjeti na slici 43.

```python
class Novosti(Resource):
    def __init__(self):
        self.dynamodb = DB()

    def get(self):
        """
        /novosti: GET
            Ova metoda vraća popis novosti.
            Args:
            Returns:
                [
                    {
                        id: string,
                        active: boolean,
                        date: string,
                        description: base64 string,
                        title: string
                    },

                ]

            Raises:
                InternalServerError: Ako dođe do pogreške u radu AWS servisa
        """
        try:
            params = {
                'TableName': 'PortalNovosti',
                'ScanFilter': {
                    'active': {'AttributeValueList': [True], 'ComparisonOperator': 'EQ'},

                }
            }
            response = self.dynamodb.scan(**params)
            response = response.get("Items", [])

            return response, 200
        except Exception as e:
            traceback.print_exc()
            print(str(e))
            return 'Internal server error', 500
```

Slika 43 - Klasa Novosti

U inicijalizaciji klase Novosti, inicijalizira se pomoćna klasa DB, kako bi se moglo pristupiti funkciji *scan()*. Pomoću ključne riječi *try* kreira se blok koji će pokrenuti kod i vidjeti je li se dogodila greška u izvođenju. Ključna riječ *except* definira što će se dogoditi u slučaju pogreške.

U bloku *try* definiraju se parametri koji sadrže ime tablice i uvjet koji definira koje podatke će upit na DynamoDB tablicu vratiti kao odgovor. U *response* varijablu sprema se odgovor DynamoDB servisa, a to je objekt koji sadrži informacije o tablici kao što je broj skeniranih vrijednosti, broj ukupnih vrijednosti, a pod ključem *Items* nalazi se lista objekata. Svaki objekt te liste predstavlja jedan zapis unutar tablice. Ako je zahtjev za podatcima prema DynamoDB tablici prošao bez greške, klijentu se vraća lista i status kod 200 koji označava da je zahtjev uspješno odrađen.

U slučaju pogreške python klasa *Exception* i python biblioteka *traceback* se koriste kako bi se ispisala poruka greške. Zatim se klijentu vraća *string* Internal server error i status kod 500 koji označava da je došlo do pogreške. Klase *Turniri* i *Korisnici* i njihove *get()* metode su gotovo identične ovom programskom kodu.

5.3.2.3. Administratorske klase

Administratorske klase i njihove metode se pozivaju kod zahtjeva na /admin/{proxy+} resurs API. U klasi *AdminAdmini*, definirana je PUT metoda. Slanjem zahtjeva PUT na rutu /admin/admini, administrator web aplikacije može odobriti ili odbiti registraciju novog administratora na web aplikaciju.

Unutar parametra *body* nalaze se dvije vrijednosti: *approved* čija je vrijednost true (ako administrator želi odobriti zahtjev) i email (email korisnika čiji se zahtjev odobrava). Unutar bloka *try* vrši se provjera postoje li unutar parametra *body* potrebne vrijednosti, a ako neka od vrijednosti ne postoji, na klijent se vraća odgovor koja vrijednost nije uključena u zahtjev i status kod 400 (Slika 44).

```python
def put(self):
    """
        /admin/admini: PUT
            U ovoj metodi korisnik koji se pokušao registrirati na portal se ili approva ili rejecta, u oba slučaja,
            user će u AWS Cognito biti ažuriran te će se moći prijaviti na portal ako je approvan.
            Args:
                body:
                    approved(boolean), required: True ako se korisnika approva.
                    email(string), required: e-mail korisnika
            Returns:
                    string: "Success"
            Raises:
                InternalServerError: If there is an error with AWS services
        """
    try:
        body = request.get_json()

        if not body:
            return 'Missing body parameters', 400
        if not body.get('email', ''):
            return 'Missing email', 400

        email = body["email"]
        data = self.getUserData(email)
        if not data:
            return "No user for given email", 400

        content = self.getCognitoUser(email)
        if not content:
            return "No user with this email", 400
        if body["approved"]:
            self.approveUser(email, data)
        else:
            self.declineUser(email, data)

        return "Success", 200
    except Exception as e:
        print(traceback.format_exc())
        print(str(e))
        self.CredentialMethods.rollBack()
        return str(e), 500
```

Slika 44 - PUT metoda

Zatim se pomoću funkcije *getUserData* koja prima kao argument email korisnika, pretražuje DynamoDB tablica *PortalAdmin* u koju se zapisuju podatci o korisniku pokretanjem spomenute pre-signup-lambde(Slika 45).

```python
def getUserData(self, email):
    params = {
        'TableName': 'PortalAdmin',
        'ScanFilter': {'email': {'AttributeValueList' : [email], 'ComparisonOperator': 'EQ'}}
    }
    result = self.dynamodb.scan(**params)
    result = result.get('Items', [])
    if result:
        return result[0]
    return None
```

Slika 45 - getUserData funkcija

Nakon što su uspješno pronađeni podatci o korisniku iz baze podataka, pomoću funkcije *getCognitoUser*, koja također prima argument email, vrši se dodatna provjera postoji li zahtjev korisnika u Cognito User Pool-u (Slika 46).

```python
def getCognitoUser(self, email):
    content = self.cognito.admin_get_user(
        UserPoolId="eu-central-1_PbagcL7Sc",
        Username=email,
    )
    return content
```

Slika 46 - getCognitoUser funkcija

Pomoću varijable *approved* poziva se funkcija *approveUser* - ako *approved* ima vrijednost true, a funkcija *declineUser* - ako je vrijednost false. Kada se korisnik odobrava, ažurira se atribut *active* iz false u true i ažurira se zahtjev korisnika u CognitoUserPool-u. Ako se korisnikov zahtjev za registraciju odbija, zahtjev se briše iz CognitoUserPool-a i brišu se podatci o korisniku iz tablice PortalAdmini (Slika 47).

```python
def declineUser(self, email, data):
    self.cognito.admin_delete_user(
        UserPoolId="eu-central-1_PbagcL7Sc",
        Username=email,
    )
    self.dynamodb.delete_item(TableName='PortalAdmin', Key={'id': data['id']})

def approveUser(self, email, data):
    self.dynamodb.update_item(
        TableName='PortalAdmin',
        Key={
            'id': data['id']
        },
        AttributeUpdates={
            'active': {
                'Value': True
            },
        },
        ReturnValues='ALL_NEW'
        )
    self.cognito.admin_confirm_sign_up(
        UserPoolId="eu-central-1_PbagcL7Sc",
        Username=email,
    )
    self.cognito.admin_update_user_attributes(
        UserPoolId="eu-central-1_PbagcL7Sc",
        Username=email,
        UserAttributes=[{"Name": "email_verified", "Value": "true"}]
    )
```

Slika 47 - approveUser i declineUser funkcije

5.3.3. Popis ruta i metoda

U tablici se nalazi popis svih ruta i metoda koje web aplikacija trenutno podržava te su opisani parametri koje svaka od metoda prima i koji odgovor vraća klijentu.

Ruta	Metoda	Prima	Vraća
/<path:content>	Any	{}	'Ovaj resurs ne postoji'
/turniri	GET	{}	Lista objekata koji sadrže atribute turnira
/novosti	GET	{}	Lista objekata koji sadrže atribute novosti
/clanovi	GET	{queryStringParameters:{limit, lastEvaluatedKey, authenticated}}	Listu objekata koji sadrže atribute članova. Ako je authenticated False, lista ne sadrži članove kojima je atribut active – False.
/admin/turniri	POST	{body:{title, data}}	'Turnir je spremljen'
/admin/novosti	POST	{body:{title, description}}	'Novi članak je dodan!'
/admin/clanovi	POST	{body:{data}}	'Novi član je dodan!'
/admin/clanovi	PUT	{body:{id}}	'Član je pobrisan!'
/admin/admini	GET	{queryStringParameters:{limit, lastEvaluatedKey}}	Lista objekata koji sadrže atribute korisnika koji je podnio zahtjev za registracijom
/admin/admini	PUT	{body:{approved, email}}	'Uspjeh!'

Tablica 1 - Popis ruta i metoda

6. Budući rad na projektu

Izrađena web aplikacija će se nastaviti ažurirati s novim funkcionalnostima, po zahtjevu klijenta. Neke od mogućnosti koje su predložene klijentu su:

Stranica galerije koja će administratorima nuditi mogućnost kreiranja mapa i mogućnost učitavanja i brisanja datoteka. Ova funkcionalnost bi zahtijevala kreaciju dodatne S3 kantice u koju bi se: a) direktno s klijenta učitavale fotografije , b) fotografije bi se učitavale preko API i Lambda funkcije.

Integracija postojeće web domene na kojoj se nalazi statična stranica pomoću AWS servisa Route53.

Dizajn i user experience nije bio fokus tijekom izrade web aplikacije, već se razvoj usredotočio na implementaciju funkcionalnosti svih dijelova aplikacije.

7. Zaključak

Kroz izradu web aplikacije za šahovski klub htio sam se detaljnije upoznati s AWS servisima i istražiti mogućnosti koje imaju. Iako već imam iskustva u radu s AWS servisima tijekom izrade web aplikacije upoznao sam se s različitim aspektima koje do sada nisam znao, a većina se tiče same kreacije resursa.

Osim detaljnijeg upoznavanja s AWS servisima, upoznao sam se s postavljanjem Flask aplikacije i definiranjem ruta, s čim do sada nisam imao puno iskustva. Smatram da sam tijekom izrade ove web aplikacije dopunio svoje znanje o AWS servisima i da sam njihovim opisom i opisom implementacije ukazao na prednosti *serverless* pristupa razvoju web aplikacija.

Verzija web aplikacije objašnjena u ovom diplomskom radu je trenutno u fazi testiranja, a nadam se da će konačan projekt biti gotov krajem 2020. godine.

Cijelom programskom kodu moguće je pristupiti na GitHub repozitorijima:

Frontend kod:

https://github.com/aban-uniri/Master-Thesis-Web-Application-Frontend

Backend kod:

https://github.com/aban-uniri/Master-Thesis-Web-Application-Backend

8. Popis literature

2020. *AWS.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://aws.amazon.com/what-is-aws/.

2020. *AWS Amplify.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://docs.amplify.aws/lib/restapi/getting-started/q/platform/js.

2020. *AWS API Gateway.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html.

2020. *AWS API Gateway REST API.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-rest-api.html.

2020. *AWS Cloudformation.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html.

2020. *AWS Cloudfront.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html.

2020. *AWS Cloudfront.* 7. Rujan. Pokušaj pristupa 7. Rujan 2020. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/HowCloudFrontWorks.html.

2020. *AWS Cognito.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html.

2020. *AWS Cognito.* 7. Rujan. Pokušaj pristupa 7. Rujan 2020. https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-integrating-user-pools-with-identity-pools.html.

2020. *AWS DynamoDB.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

2020. *AWS IAM.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.

2020. *AWS Lambda.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

2020. *AWS S3.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://aws.amazon.com/s3/.

2020. *AWS SAM.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://aws.amazon.com/serverless/sam/.

2020. *BabelJS.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://babeljs.io/docs/en/.

2020. *Boto3.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

2020. *CodeAcademy.* 3. Ruja. Pokušaj pristupa 3. Rujan 2020. https://www.codecademy.com/articles/what-is-rest.

2020. *Flask.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://pymbook.readthedocs.io/en/latest/flask.html.

2020. *JavaScript.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

2020. *ReactJS.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://reactjs.org/docs/create-a-new-react-app.html.

2020. *ReactJS Tutorial.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://reactjs.org/tutorial/tutorial.html.

2020. *Redux.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://redux.js.org/tutorials/essentials/part-1-overview-concepts.

2020. *RESTful API.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://restfulapi.net/.

2020. *Same-origin policy.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

2020. *TechTerm.* 31. Kolovoz. https://techterms.com/definition/nosql.

2020. *The Next Web.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://thenextweb.com/dd/2017/09/19/should-developers-be-afraid-of-zuckerbergs-bearing-gifts/.

2020. *Visual Studio Code.* 3. Rujan. Pokušaj pristupa 3. Rujan 2020. https://code.visualstudio.com/docs.

2020. *W3Schools.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://www.w3schools.com/REACT/react_state.asp.

2020. *Webpack.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://webpack.js.org/concepts/.

2020. *Wikipedia API.* 31. Kolovoz. https://en.wikipedia.org/wiki/API.

2020. *Wikipedia SPA.* 31. Kolovoz. Pokušaj pristupa 31. Kolovoz 2020. https://en.wikipedia.org/wiki/Single-page_application.

Popis ilustracija

Slika 1 - Cognito JWT Token 8
Slika 2 - Izmjena tokena za dozvolu pristupa resursima 9
Slika 3 - Prikaz HTTP zahtjeva Cloudfront distribuciji 14
Slika 4 - Arhitektura AWS servisa i resursa 16
Slika 5 - Bucket policy 17
Slika 6 - Konfiguracija CORS 18
Slika 7 - Izvor web aplikacije 18
Slika 8 - Struktura API 20
Slika 9 - Autorizacija pomoću Cognita 20
Slika 10 - Rute 21
Slika 11 - Definicija DynamoDB tablice 22
Slika 12 - SPA arhitektura web aplikacije 24
Slika 13 - Početna struktura 25
Slika 14 - Konačna struktura 26
Slika 15 - Funkcijska komponenta 27
Slika 16 - Modalni prozor 28
Slika 17 - Klasa NoviClan 29
Slika 18 - render funkcija 30
Slika 19 - inputHandler funkcija 30
Slika 20 - componentDidUpdate funkcija 31
Slika 21 - definiranje ruta 32
Slika 22 - Definicija privatne rute 33
Slika 23 - Prosljeđivanje odgovora dijete komponenti 34
Slika 24 - funkcija mapStateToProps 34
Slika 25 - Redux types 35
Slika 26 - akcija createClan 37
Slika 27 - validateUserSession funkcija 39
Slika 28 - Admin zaslon: odobravanje zahtjeva 40
Slika 29 - Javni zaslon: popis članova 40
Slika 30 - Admin zaslon: dodavanje člana 41
Slika 31 – Javni zaslon: zaboravljena lozinka 41

Slika 32 - Javni zaslon: početna stranica 42
Slika 33 - Admin zaslon: dodavanje novosti 42
Slika 34 - Javni zaslon: popis novosti 43
Slika 35 - Javni zaslon: o nama 43
Slika 36 - Javni zaslon: prijava 44
Slika 37 - Javni zaslon: registracija 44
Slika 38 - Javni zaslon: popis turnira 44
Slika 39 - Admin zaslon: kreacija turnira 45
Slika 40 - projekt kreiran pomoću SAM alata 45
Slika 41 - Definicija lambda funkcije i API resursa 46
Slika 42 - Implementacija Flask i FlaskRESTful 47
Slika 43 - Klasa Novosti 48
Slika 44 - PUT metoda 50
Slika 45 - getUserData funkcija 50
Slika 46 - getCognitoUser funkcija 51
Slika 47 - approveUser i declineUser funkcije 51

Tablica 1 - Popis ruta i metoda 52