Univerzitet Singidunum

Tehnički fakultet

Online platforma za trgovinu predmetima iz video igara

-Projektni rad-

Predmet:

Internet Softverske Arhitekture

Profesor:
doc. dr. Nebojša Bačanin Džakula

Predmetni asistent:
Petar Biševac

Student:
Stefan Dimitrijević
2018200497

Beograd, 2021. godina

Sadržaj

1. Uvod

Svrha ovog dokumenta je da prikaže, opiše i detaljno dokumentuje pristupne tačke(eng. *endpoints*) koje su izrađene u okviru projekta - aplikacije. Aplikacija je napravljena za završni ispit predmeta ,,Internet softverske arhitekture,, sa svrhom da prikaže rad jednog *backend* i *frontend* frameworka, koji komuniciraju međusobno i sačinjavaju jedan kompletan sistem. Tema ove aplikacije je trgovina digitalnim predmetima(*in-game*) neke fiktivne video igre; korisnik treba da ima mogućnost da postavi svoje predmete u trgovinski centar, kao i da kupuje tuđe predmete, ukoliko prethodno ima dovoljno sredstava na svom nalogu(gde bi se nalog povezivao direktno sa nalogom fiktivne igre, tako da nije moguće dodati sredstva iz same aplikacije).

Tehnologije korišćene na strani servera(***backend***) su:

1. **Spring Boot** – za pružanje osnovnih funkcionalnosti zrna(***Bean***), ožičavanje (***wiring***) istih, automatsko otkrivanje zrna (*autodiscovery*) putem anotacija kao što su @***Service***(mikroservis), @***Component(***Spring komponenta)…
2. **Spring Web** – koji omogućava da se aplikacija izgradi kao ***REST****(REpresentational State Transfer*) API aplikacija.
3. **Spring security** framework – za konfiguraciju bezbednosti, autorizacije i autentifikacije ka pristupnim tačkama (*endpoints*) putem *Java Web Token*-a(JWT), bezbednosnih filtera, korišćenje *B-crypt password encoder*-a...
4. **Lombok**, **MongoDB** i **ModelMapper** – za olakšani rad sa MongoDB(kroz *CRUD* aktivnosti) NoSQL bazom, predstavljanja modela u okviru aplikacije korišćenjem anotacija (Lombok i MongoDB), kao i mapiranje modela u entitete(i obrnuto).

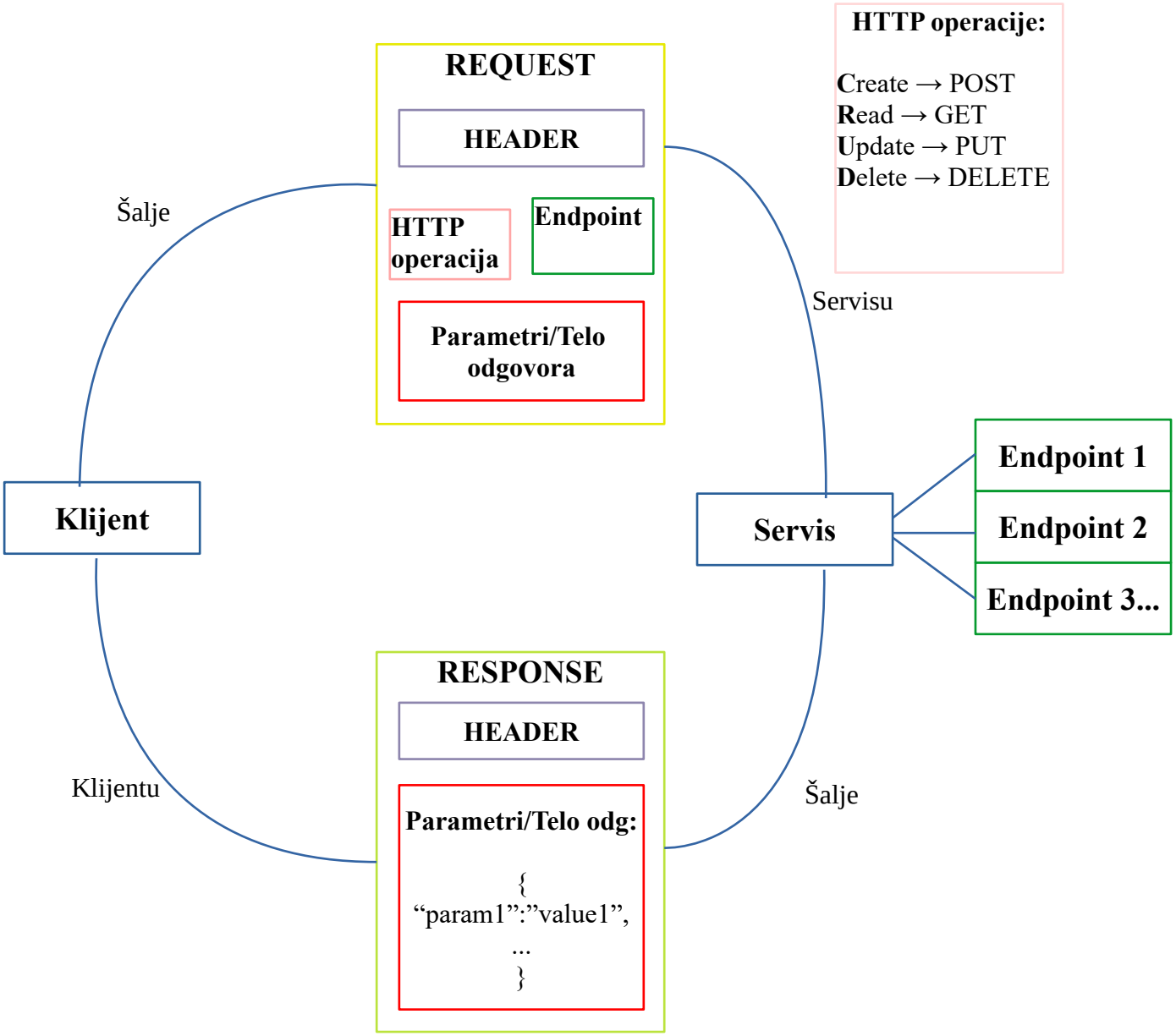
Tehnologije korišćene na strani klijenta(*frontend*):

1. Angular *framework* verzije 11 i *TypeScript* jezik – kao osnova za logiku, rutiranje, prikaz komponenti, stilizaciju...
2. Angular Material – prema *Google Material* standardu koji omogućava konzistentan i intuitivan dizajn grafičkog korisničkog interfejsa(*GUI*).

Dodatni alati korišćeni za rad:

1. IntelliJ Community Edition razvojno okruženje sa ubačenom podrškom za Maven build automation alatom.
2. Postman za testiranje ispravnosti *endpointa*
3. NoSQL Booster for MongoDB za upravljanje MongoDB bazom.

2. Arhitektura sistema



Princip rada je sledeći: klijentu je potreban neki resurs sa servisa (uglavnom *Web servisa*), servis nudi(otvara) ovaj resurs svim (po potrebi autentifikovanim i autorizovanim) korisnicima na mreži preko pristupnih tačaka (*endpoints*). Ako klijent želi da pristupi nekom resursu, treba da uputi zahtev(***Request***) servisu, dok servis treba korisniku(klijentu) na ovaj zahtev da odgovori(***Response***) – da li će mu vratiti traženi resurs ili ne, zavisi od mnogih faktora – dva korišćena faktora u ovom projektu su ***autorizacija*** i ***autentifikacija*** korisnika putem *Java Web Tokena*, pošto je ***REST*** stateless protokol(ne pamti stanja, ne održava sesije), ovaj token se šalje pri svakom zahtevu ka serveru, kako bi se izvršila autentifikacija.

Backend i frontend, iako su napisani u različitim tehnologijama(**Spring – Java, Angular – TypeScript**) mogu međusobno da komuniciraju putem poruka – **REST** parova request-response koji su napisani u standardizovanom formatu(najčešće **JSON**), koji omogućava razmenu podataka između 2 heterogena sistema.

3. Opis aplikacije

Pre svega, počnimo sa opisom naših modela i entiteta, jer oni zapravo oslikavaju strukturu poruke koje se prosleđuju između klijenta i servisa, kao i u kom formatu se čuvaju podaci u bazi. Najjednostavniji modeli/entiteti su role(*Roles*), kategorije(*Categories*), kvalitet(*Qualities*) i osobine(*Traits*) predmeta koji se prodaju., sadržina ovih entiteta se uglavnom ogleda u njihovim imenima. Sadržaj samih predmeta(*Items*) predstavlja kategorija, kvalitet i osobina koje sadrže(mogu ih imati samo jednu), datuma kreiranja predmeta, nivoa, vlasnika predmeta(*User*), osnovnu cenu po jedinici proizvoda, količinu, da li je predmet prodat(inicijalno *false*) i kupca predmeta(inicijalno *null*). Entiteti se koriste da preko repozitorijuma(***repository***) komuniciraju sa bazom(***MongoDB***), dok se modeli koriste da prosleđuju podatke sa klijentske strane ka kontrolerima u kojima se nalaze pristupne tačke(*endpoints*), dok se sva poslovna logika izvršava unutar servisa(***services***), od kojih su kontroleri komponovani. Unutar servisa se takođe vrši mapiranje iz modela u entitet(i obrnuto), da bi se podaci poslali u pravilnom formatu ka bazi.

Aplikacija sadrži sledeće kontrolere:

1. ***AuthController*** – sadrži endpoint koji vrši login, autentifikaciju i autorizaciju korisnika
2. ***CategoryController***, ***TraitController***, ***QualityController***, ***RolesController*** – sadrže osnovne CRUD akcije vezane za kategorije, osobine, kvalitet i role.
3. ***UserController*** – zadužen za kreiranje(registraciju) korisničkog naloga i pronalazak korisnika po korisničkom imenu, kao i enkripciju lozinke prilikom registracije.
4. ***ItemController*** – sadrži većinu logike aplikacije, počevši od osnovnih CRUD akcija za kreiranje/dodavanje predmeta, pronalazak svih predmeta(ili po nekom kriterijumu – npr. prema imenu, ili prema korisniku koji je kupio, ili kompleksnijoj pretrazi koja zahteva više kriterijuma), ažuriranje predmeta(npr. Kada korisnik kupi predmet, potrebno je ažurirati da je predmet prodat, kao i kome…). Direktno brisanje predmeta iz baze nije omogućeno, kako bi se mogle analizirati porudžbenice. Detaljniji opis ovih kontrolera biće dat u nastavku.

Takođe, aplikacija sadrži konfiguracione klase da modifikuje postojeće filtere Spring security-ja kako bi umetnula autentifikaciju putem tokena, kao I samu klasu za generisanje i validaciju tokena, autorizaciju na nivou rola ili autentifikacije preko određenih REST API pristupnih tačaka.

3.1. AuthController

AuthController sadrži samo jednu pristupnu tačku koja služi da se korisnik uloguje, koristi se POST HTTP metoda za slanje sadržaja, koji predstavlja model korisnika(User), povratna vrednost ovog API-ja je ResponseEntity, odnosno odgovor servisa, koji unutar sebe enkapsulira podatke(UserModel) koje vraća na front-end, koje klijent koristi da skladišti u lokalu.. Među poslatim podacima nalazi se i HTTP zaglavlje(header) za autorizaciju, čija vrednost sadrži generisani token od strane servera. Ovom API-ju mogu pristupiti svi korisnici, tj. klijenti bez autorizacije ili autentifikacije.

```
@PostMapping("loginuser")
@CrossOrigin(origins="*")
private ResponseEntity<UserModel> loginUser(@RequestBody  UserModel userModel){
```

Slika 1. AuthController

3.2. CategoryController

CategoryController sadrži CRUD operacije sa entitetom kategorije u bazi. Read operaciji može pristupiti svaki korisnik, ali su za Create, Update i Delete neophodni i autentifikacija i autorizacija – potrebno je imati rolu ROLE_ADMIN.

```
@PostMapping("createcategory")
@CrossOrigin(origins="*")
private Category createCategory(@RequestBody CategoryModel categoryModel){
    return categoryService.insert(categoryModel);
}

@PutMapping("updatecategory")
@CrossOrigin(origins="*")
private Category updateCategory(@RequestBody CategoryModel categoryModel){
    return categoryService.update(categoryModel);
}

@GetMapping("findallbyparent")
private List<Category> findAllByParent(@RequestBody CategoryModel categoryModel){
    return categoryService.findAllByParent(categoryModel);
}

@GetMapping("findallcategories")
@CrossOrigin(origins="*")
private List<Category> findAllCategories() { return categoryService.findAll(); }

@GetMapping("findcategorybyname")
@CrossOrigin(origins="*")
private Category findCategoryByName( String name) { return categoryService.findByName(name); }

@DeleteMapping("deletecategorybyid/{id}")
@CrossOrigin(origins="*")
private void deleteCategoryById(@PathVariable("id")  String id) { categoryService.deleteById(id); }
```

Slika 2. CategoryController

Kategorija se kreira POST metodom na API-ju createcategory, koristeći CategoryModel model koji se dohvata iz tela zahteva koji je uputio klijent sa fronta. Metoda vraća Category entitet kao rezultat(ukoliko je korisnik uspešno autentifikovan i ima odgovarajuću autorizaciju).

Ažuriranje se vrši PUT metodom, na endpointu updatecategory, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa(ako važe autentifikacija i autorizacija).

Brisanje se vrši DELETE metodom, na endpointu deletecategorybyid/id_kategorije_koja_se_briše, gde se, ovoga puta, kao podatak, u okviru URI-ja, koristeći @PathVariable anotaciju, ostavlja id kategorije kao String vrednost.

Čitanje se izvršava sa nekoliko endpointa – findallbyparent, findallcategories, findcategorybyname. Svi ovi APIji koriste GET metodu, a kao parametar dobijaju model kategorije, prazno telo zahteva i String podatak koji predstavlja ime kategorije koja se zraži, s tim da se podaci(ukoliko su neophodni) ostavljaju u URL-u, sa key vrednostima tačno onako kako je specificirano u argumentima endpointa. Prva dva endpointa vraćaju listu kategorija(može biti više kategorija), dok poslednja pronalazi tačno jednu metodu.

3.3. ItemController

ItemController sadrži CRUD operacije sa entitetom predmeta(items) u bazi, kao i operacije za malo kompleksniju pretragu, kupovinu proizvoda…

```
@PostMapping("additem")
@CrossOrigin(origins="*")
private Item addItem(@RequestBody ItemModel itemModel) { return itemsService.insert(itemModel); }

@GetMapping("findallitems")
@CrossOrigin(origins="*")
private List<Item> findAllItems() { return itemsService.findAllItems(); }

@GetMapping("findall")
@CrossOrigin(origins="*")
private List<Item> findAll() { return itemsService.findAll(); }

@GetMapping("finditemsbyslug")
@CrossOrigin(origins = "*")
private List<Item> findAllBySlug(String slug) { return itemsService.findAllBySlug(slug); }

@GetMapping("finditemsbyname/{ItemName}")
@CrossOrigin(origins = "*")
private List<Item> findAllByItemName(@PathVariable("ItemName") String ItemName) {
    return itemsService.findAllByItemName(ItemName);
}
```

Slika 3. ItemController deo 1

Predmet(item) se kreira od strane običnog ulogovanog korisnika POST metodom na API-ju aditem, koristeći ItemModel model koji se dohvata iz tela zahteva koji je uputio klijent sa fronta. Metoda vraća Item entitet kao rezultat..

Čitanje se izvršava sa nekoliko endpointa – findallitems (traži predmete koji nisu prodati), findall (traži sve predmete), finditemsbyslug i finditemsbyname, svim endpointima se pristupa GET metodom, koji takođe vraćaju listu predmeta, dok samo poslednje dve uzimaju argumente – finditemsbyslug dohvata parametar slug iz URL-a i traži predmete po slug-u, dok metoda nakon nje dohvata ime predmeta kao @PathVariable i traži predmete po imenu. O složenijim pretragama biće reči kasnije.

```
@PutMapping("updateitem")
@CrossOrigin(origins = "*")
private Item update(@RequestBody ItemModel itemModel) { return itemsService.update(itemModel); }

@PostMapping("buyitem")
@CrossOrigin(origins = "*")
private Item buyItem(@RequestBody Map< String, String> json){
    var username :String  = json.get("username");
    var item_id :String  = json.get("item_id");
    return itemsService.buyItem(username,item_id);
}

@GetMapping("fetchuserpurchases/{username}")
@CrossOrigin(origins = "*")
private List<ItemModel> findAllByBuyer(@PathVariable("username") String username){
    return itemsService.findAllByBuyer(username);
}

@PostMapping("complexsearch")
@CrossOrigin(origins = "*")
private List<ItemModel> complexSearch(@RequestBody Map<String, String> json){
    return itemsService.findAllByComplex(json);
}
```

Slika 4. ItemController deo 2

Ažuriranje se vrši PUT metodom, na endpointu updateitem, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa(ako važe autentifikacija i autorizacija).

Brisanje se vrši DELETE metodom, na endpointu. Još jedan vid ažuriranja je endpoint buyitem koji kao svoj parametar dohvata custom objekat sa fronta, u vidu mape, gde su ključevi tipa String, a vrednosti takođe. Mapa sadrži dve vrednosti – jedna je username korisnika koji kupuje(buyer), dok druga predstavlja id predmeta koji se kupuje od strane datog korisnika deletecategorybyid/id_kategorije_koja_se_briše, gde se, ovoga puta, kao podatak, u okviru URI-ja, koristeći @PathVariable anotaciju, ostavlja id kategorije kao String vrednost.

Poslednje dve metode su kompleksnija pretraga, prva koja pretražuje na osnovu @PathVariable username koja predstavlja korisničko ime onog korisnika čije porudžbenice tražimo, dok se druga odnosi na kompleksnu pretragu(opet custom objekat) prema više različitih kriterijuma; kategorija, osobina, kvalitet, cenovni rang...

3.4. QualityController

QualityController sadrži CRUD operacije sa entitetom kvaliteta u bazi. Read operaciji može pristupiti svaki korisnik, ali su za Create, Update i Delete neophodni i autentifikacija i autorizacija – potrebno je imati rolu ROLE_ADMIN.

```
@GetMapping("getallqualities")
@CrossOrigin(origins = "*")
public List<Quality> findAllQualities() { return qualityService.findAll(); }

@PostMapping("createquality")
@CrossOrigin(origins="*")
private Quality createCategory(@RequestBody QualityModel qualityModel){
    return qualityService.insert(qualityModel);
}

@PutMapping("updatequality")
@CrossOrigin(origins = "*")
private Quality updateCategory(@RequestBody QualityModel qualityModel) { return qualityService.save(qualityModel); }

@DeleteMapping("deletequality/{id}")
@CrossOrigin(origins = "*")
private void deleteCategory(@PathVariable("id") String id) { qualityService.deleteById(id); }
```

Slika 5. QualityController

Čitanje se izvršava sa jednim endpointom – getallqualities, koja, koristeći GET metodu dohvata sve kvalitete iz baze i vraća ih kao listu Qualityobjekata(entiteta).

Kvalitet se kreira POST metodom na API-ju createquality, koristeći QualityModel model koji se dohvata iz tela zahteva koji je uputio klijent sa fronta. Metoda vraća Quality entitet kao rezultat(ukoliko je korisnik uspešno autentifikovan i ima odgovarajuću autorizaciju).

Ažuriranje se vrši PUT metodom, na endpointu updatequality, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa(ako važe autentifikacija i autorizacija).

Brisanje se vrši DELETE metodom, na endpointu deletequality/id_kvaliteta_koji_se_briše, gde se, ovoga puta, kao podatak, u okviru URI-ja, koristeći @PathVariable anotaciju, ostavlja id kvaliteta kao String vrednost.

3.5. RolesController

RolesController sadrži CRUD operacije sa entitetom rola u bazi. Read operaciji može pristupiti svaki korisnik, ali su za Create, Update i Delete neophodni i autentifikacija i autorizacija – potrebno je imati rolu ROLE_ADMIN.

```
@GetMapping("getallroles")
@CrossOrigin(origins = "*")
public List<Roles> findAllTraits() { return rolesService.findAllRoles(); }

@PostMapping("createrole")
@CrossOrigin(origins="*")
private Roles createRole(@RequestBody RolesModel rolesModel) { return rolesService.createRole(rolesModel); }

@PutMapping("updaterole")
@CrossOrigin(origins = "*")
private Roles updateRole(@RequestBody RolesModel rolesModel) { return rolesService.updateRole(rolesModel); }

@DeleteMapping("deleterole/{id}")
@CrossOrigin(origins = "*")
private void deleteRoleById(@PathVariable("id") String id) { rolesService.deleteRoleById(id); }
```

Slika 6. RolesController

Čitanje se izvršava sa jednim endpointom – getallroles, koja, koristeći GET metodu dohvata sve role iz baze i vraća ih kao listu Roles objekata(entiteta).

Rola se kreira POST metodom na API-ju createrole, koristeći RolesModel model koji se dohvata iz tela zahteva koji je uputio klijent sa fronta. Metoda vraća Roles entitet kao rezultat(ukoliko je korisnik uspešno autentifikovan i ima odgovarajuću autorizaciju).

Ažuriranje se vrši PUT metodom, na endpointu updaterole, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa(ako važe autentifikacija i autorizacija).

Brisanje se vrši DELETE metodom, na endpointu deleterole/id_role_koji_se_briše, gde se, ovoga puta, kao podatak, u okviru URI-ja, koristeći @PathVariable anotaciju, ostavlja id role kao String vrednost.

3.6. TraitController

TraitController sadrži CRUD operacije sa entitetom osobina u bazi. Read operaciji može pristupiti svaki korisnik, ali su za Create, Update i Delete neophodni i autentifikacija i autorizacija – potrebno je imati rolu ROLE_ADMIN.

```
@GetMapping("getalltraits")
@CrossOrigin(origins = "*")
public List<Trait> findAllTraits() { return traitService.findAll(); }

@PostMapping("createtrait")
@CrossOrigin(origins="*")
private Trait createCategory(@RequestBody TraitModel traitModel) { return traitService.insert(traitModel); }

@PutMapping("updatetrait")
@CrossOrigin(origins = "*")
private Trait updateCategory(@RequestBody TraitModel traitModel) { return traitService.save(traitModel); }

@DeleteMapping("deletetrait/{id}")
@CrossOrigin(origins = "*")
private void deleteTraitById(@PathVariable("id") String id) { traitService.deleteById(id); }
```

Slika 7. TraitController

Čitanje se izvršava sa jednim endpointom – getalltraits, koja, koristeći GET metodu dohvata sve osobine iz baze i vraća ih kao listu Trait objekata(entiteta).

Rola se kreira POST metodom na API-ju createtrait, koristeći TraitModel model koji se dohvata iz tela zahteva koji je uputio klijent sa fronta. Metoda vraća Trait entitet kao rezultat(ukoliko je korisnik uspešno autentifikovan i ima odgovarajuću autorizaciju).

Ažuriranje se vrši PUT metodom, na endpointu updatetrait, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa(ako važe autentifikacija i autorizacija).

Brisanje se vrši DELETE metodom, na endpointu deletetrait/id_role_koji_se_briše, gde se, ovoga puta, kao podatak, u okviru URI-ja, koristeći @PathVariable anotaciju, ostavlja id role kao String vrednost.

3.7. UserController

UserController sadrži svega dve operacije – kreiranje korisnika putem createaccount endpointa i pretragu korisnika po korisničkom imenu putem findbyusername endpointa.

```
@PostMapping("createaccount")
@CrossOrigin(origins = "*")
private User createAccount(@RequestBody  UserModel entity){
    entity.setPassword(passwordEncoder.encode(entity.getPassword()));

    return userService.insert(entity);
}

@GetMapping("finduserbyusername")
private User findUserByUsername(String username) { return userService.findByUsername(username); }
```

Slika 8. UserController

Endpoint za kreiranje korisnika koristi POST metodu i samim tim dohvata telo zahteva koje korisnik upućuje, tipa UserModel objekta, dok kao rezultat vraća User entitet. I na kraju metoda za pronalazak korisnika koristi GET metodu, gde se parametar username dohvata iz URL-a, i na osnovu kojeg se pretražuje korisnik u bazi koji ima odgovarajuće korisničko ime. Kao rezultat vraća objekat(entitet) tipa User.

4. Opis komunikacije

Nakon što je opisana komunikacija sa servisom, potrebno je opisati kako se komunikacija uspostavlja sa strane korisnika, ka kom API-ju i u kom formatu. Za komunikaciju koristi se standardno HTTP protokol, sa odgovarajućim domenskim imenom/IP adresom, po potrebi brojem socketa i identifikatorom resursa kojem želimo da pristupimo.

Pošto je aplikacija simulirana na lokalnoj mašini, koristi se localhost domensko ime sa standardnim portom 8080.

4.1. Komunikacija sa Category API-jem

Kao što je opisano, CategoryController se sastoji od osnovnih *CRUD* akcija - *POST* metodom na API-ju *createcategory se,* koristeći *CategoryModel* model, kreira kategorija u bazi. Metoda vraća Category entitet kao rezultat, ažuriranje se vrši *PUT* metodom, na endpointu *updatecategory*, koristi isti argument kao i funkcija za kreiranje, takođe vraća rezultat istog tipa, brisanje se vrši *DELETE* metodom, na endpointu *deletecategorybyid,* Čitanje se izvršava sa nekoliko endpointa, ali na ovom frontu je iskorišćen samo *findallcategories* endpoint Pošto je reč o CategoryControlleru, njemu se pristupa preko /category mapiranja.

```
findAllCategories():Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/category/findallcategories";

  return this.http.get<any>(url,{observe: 'response'});
}

createCategory(category: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/category/createcategory";



  return this.http.post<any>(url,category);
}

updateCategory(category: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/category/updatecategory";

  return this.http.put<any>(url,category);
}

deleteCategory(Id: string):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/category/deletecategorybyid/" +Id;

  return this.http.delete<any>(url);
}
```

Slika 9. Servis za komunikaciju sa CategoryController

U prethodnom poglavlju napomenuto je koji resurs koristi koju putanju, identično je i na frontendu, što se može zaključiti sa slike 9 – read metoda ne uzima nijedan argument, s obzirom da traži sve kategorije zapisane u bazi.

Metoda za kreiranje kategorije uzima argument koji predstavlja kategoriju sledeće strukture:

```
const category = {
  "name":form.value.categoryName,
  "parent": form.value.parentCategory
};
```

Slika 10. format kategorije za kreairanje

Sa slike se vidi da kategorija uzima prost parametar, tipa String za ime, i *parent* parametar, koji predstavlja roditelj-kategoriju, date kategorije(ukoliko postoji). *Parent* je istog formata i strukture kao i osnovna kategorija.

Update struktura je identična strukturi za kreiranje, s tim dodatkom da se dodaje i parametar koji treba da identifikuje kategoriju koja se ažurira.

Delete uzima samo identifikator kategorije koju želimo da uklonimo iz baze.

4.2. Komunikacija sa Quality API-jem

Komunikacija sa Quality API-jem je identična kao i sa kategorijom: imamo po jednu metodu za svaku od 4 akcije(*CRUD*).

```
findAllQualities(){

  const url = "http://localhost:8080/quality/getallqualities";

  return this.http.get<any>(url,{observe: 'response'});
}

createQuality(quality: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/quality/createquality";

  console.log(quality);


  return this.http.post<any>(url,quality);
}

updateQuality(quality: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/quality/updatequality";

  return this.http.put<any>(url,quality);
}

deleteQuality(Id: string):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/quality/deletequality/" +Id;

  return this.http.delete<any>(url);
}
```

Slika 11. Servis za komunikaciju sa QualityController

Standardno, metoda za kreaciju uzima objekat kvaliteta, takođe i metoda za ažuriranje, delete metoda uzima samo ID kvaliteta koji se briše iz baze, struktura za ažuriranje je predstavljena na sledećoj slici:

```
const quality = {
  "id": form.value.updateQuality,
  "qualityName": form.value.updateQualityName,
  "color": form.value.updateQualityColor
};
```

Slika 12. format kvaliteta za ažuriranje

Format za kreiranje je identičan, s razlikom što ne sadrži ID kategorije(pošto je prvi put kreiramo).

4.3. Komunikacija sa Roles API-jem

Komunikacija sa Roles API-jem je identična kao i sa kategorijom: imamo po jednu metodu za svaku od 4 akcije(*CRUD*).

```
findAllRoles(){

  const url = "http://localhost:8080/roles/getallroles";

  return this.http.get<any>(url,{observe: 'response'});
}

createRole(role: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/roles/createrole";

  console.log(role);

  return this.http.post<any>(url,role);
}

updateRole(quality: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/roles/updaterole";

  return this.http.put<any>(url,quality);
}

deleteRole(Id: string):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/roles/deleterole/" +Id;

  return this.http.delete<any>(url);
}
```

Slika 13. Servis za komunikaciju sa RolesController

Pronalazimo sve role u bazi, bez potrebnih parametara, kreiramo i ažuriramo rolu sa odgovarajućom strukturom i formatom(skoro identičnim) i brišemo rolu sa datim ID-em u bazi:

```
const role = {
  "id":form.value.updateRole,
  "roleName":form.value.updateRoleName
};
```

Slika 14. format role za ažuriranje

Kao i za prethodne primere, insert je identičan ažuriranju, bez ID-a role.

4.4. Komunikacija sa Traits API-jem

Komunikacija sa Traits API-jem je identična kao i sa kategorijom: imamo po jednu metodu za svaku od 4 akcije(*CRUD*).

```
findAllTraits():Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/traits/getalltraits";

  return this.http.get<any>(url, {observe: "response"});
}

createTrait(trait: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/traits/createtrait";

  return this.http.post<any>(url,trait);
}

updateTrait(trait: any):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/traits/updatetrait";

  // console.log(trait);

  return this.http.put<any>(url,trait);
}

deleteTrait(Id: string):Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/traits/deletetrait/" +Id;
  // console.log(Id);

  return this.http.delete<any>(url);
}
```

Slika 15. Servis za komunikaciju sa TraitsController

Standardno, metoda za kreaciju uzima objekat *osobine*, takođe i metoda za ažuriranje, delete metoda uzima samo ID osobine koja se briše iz baze, struktura za ažuriranje je predstavljena na sledećoj slici:

```
const trait = {
  "id":form.value.updateTrait,
  "traitName":form.value.updateTraitName
};
```

Slika 16. format osobine za ažuriranje

Kao i u prethodnim slučajevima, za kreaciju se koristi identična struktura kao i za ažuriranje objekata, izuzev ID-a osobine koji se šalje ka serveru.

4.5. Komunikacija sa User API-jem

Komunikacija sa *UserController* je nešto drugačija, sadrži svega dve metode, jednu za registraciju, drugu za logovanje korisnika.

```
registerUser(username: String, password: String, email: String, country: String) : Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/user/createaccount"
  const user = {
    "username":username,
    "password":password,
    "email":email,
    "country":country
  };

  return this.client.post<any>(url, user);


}

logOnUser(username: String, password:String) : Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/auth/loginuser";
  const user ={
    "username":username,
    "password":password
  };

  return this.client.post<any>(url, user, {observe: 'response'});
}
```

Slika 17. Servis za komunikaciju sa UserController

Registracija korisnika komunicira sa *createaccount* endpointom, a kao parametre uzima korisničko ime, lozinku, email i državu porekla, pa sve to kombinuje u jedan User objekat(koji se *backendu* isporučuje kao *UserModel* model) i *POST* metodom prosleđuje na backend za obradu, a kao odgovor dobija *User* entitet, kao što je specificirano ranije. Nakon ovoga korisnik se može ulogovati.

Metoda za logovanje tradicionalno koristi samo korisničko ime i lozinku, i te parametre, takođe, prosleđuje *POST* metodom, na *loginuser* endpoint. Ukoliko korisnik postoji u bazi, korisnik je uspešno ulogovan, a servis će korisniku vratiti u odgovoru, u okviru *AUTHORIZATION* headera, *Java Web Token* koje klijent čuva u lokalu i šalje u svakom pratećem zahtevu radi potrebe autentifikacije i autorizacije od strane servera.

4.6. Komunikacija sa Item API-jem

Poslednji na listi je *ItemController*, koji sadrži nekoliko akcija čitanja/pretraga i po jednu akciju za kreiranje i ažuriranje predmeta u bazi(bez akcije brisanja).

Počevši od samih pretraga, imamo ih 4: *findall, findBySlug, findAllByItemName, findMyItems i* koje pronalaze sve predmete koji nisu prodati(upućuje ka *findallitems* endpointu), pronalazi predmete prema slug-u (Stringu), predmete prema istom imenu i na kraju predmete koje je dati korisnik kupio. Na kraju imamo i kompleksnu pretragu *complexSearch* koja vrši pretragu predmeta na osnovu više različitih kriterijuma

```
public findAll(): Observable<HttpResponse<Item[]>>{
  var headers_object = new HttpHeaders().set("Content-Type",'application/json');//.set("Authorization",localStorage.getItem("token") ? "Bearer "

  return this.client.get<Item[]>("http://localhost:8080/items/findallitems", {observe: "response", headers: headers_object});
}

public findAllBySlug(slug: string) : Observable<HttpResponse<Item[]>>{
  let data = new HttpParams().set("slug", slug);
  return this.client.get<Item[]>("http://localhost:8080/items/finditemsbyslug", {observe: "response", params: data });
}

public findAllByItemName(ItemName: string) : Observable<HttpResponse<Item[]>>{

  var headers_object = new HttpHeaders();
  headers_object.append('Content-Type', 'application/json');
  headers_object.append("Authorization", "Basic " + btoa("petar:admin"));

  let data = new HttpParams().set("ItemName", ItemName);
  return this.client.get<Item[]>("http://localhost:8080/items/finditemsbyname", {observe: "response", params: data, headers: headers_object });

}
```

Slika 18. Servis za komunikaciju sa ItemControllera pretrage

Kao i *complexSearch* koji uzima više parametara za pretragu i ima različitu strukturu od ostalih upita:

```
public complexSearch(complexParams:Object): Observable<HttpResponse<any>>{

  const url = "http://localhost:8080/items/complexsearch";

  return this.client.post<any>(url, complexParams, { observe: 'response' });

}
```

Slika 19. Servis za komunikaciju sa ItemControllera – kompleksna pretraga

```
const complexSearch = {
  "search": search,
  "category": typeof(category) === 'string' ? category:category.name,
  "trait":typeof(trait) === 'string' ? trait:trait.traitName,
  "quality":typeof(quality) === 'string' ? quality:quality.qualityName,
  "levelLow":form.value.level[0],
  "levelHigh":form.value.level[1],
  "amountLow":form.value.amount[0],
  "amountHigh":form.value.amount[1],
  "priceLow":form.value.price[0],
  "priceHigh":form.value.price[1]
};
```

Slika 20. Struktura kompleksne pretrage

Kao što se može videti sa slike, kompleksna pretraga sadrži više kriterijuma:

- Search parametar – navodi ime predmeta koji pretražujemo
- Category parametar – kategoriju predmeta
- Trait parametar – osobinu predmeta
- Quality parametar – kvalitet predmeta
- LevelLow i LevelHigh – opseg nivoa predmeta koji treba tražiti
- AmountLow i AmountHigh – opseg količine za dati predmet koji se traži
- PriceLow i PriceHigh – opseg cene za dati predmet koji se traži

Na kraju ostaju dve metode CREATE i UPDATE:

```
public buyItem(username: String, item_id: String){

  const url = "http://localhost:8080/items/buyitem";

  const body = {
    "username": username,
    "item_id": item_id
  };

  return this.client.post<any>(url, body, { observe: 'response' });
}

public createItem(item: any){
  const url = "http://localhost:8080/items/additem";


  return this.client.post<any>(url, item, { observe: 'response' });
}
```

Slika 21. Servis za komunikaciju sa ItemControllera – kupovina i dodavanje predmeta

Metoda kupovine predmeta, koja predstavlja varijantu Update akcije, uzima korisničko ime korisnika koji je kupio predmet, kao i ID predmeta koji je kupljen, pa se u skladu s time ažurira predmet koristeći buyitem endpoint.

CreateItem metoda uzima strukturu predmeta kao Item model i šalje na backend, te se taj predmet insertuje u bazu i koristi ulogovanog korisnika kao prodavača. Struktura predmeta koji se šalje data je na sledećoj slici:

```
this.itemService.createItem({
  "itemName":form.value.item_name,
  "basePrice":form.value.basePrice,
  "amount":form.value.amount,
  "level":form.value.level,
  "category":form.value.category,
  "trait":form.value.trait,
  "quality":form.value.quality,
  "owner":{"username": localStorage.getItem('username') }
}).subscribe();
```

Slika 22. Struktura predmeta koji se ubacuje u bazu

Model predmeta(Item) se sastoji od imena predmeta, osnovne cene predmeta, količine date jedinice predmeta koja se prodaje, nivoa predmete, modela kategorije, svojstva i kvaliteta predmeta, kao i modela korisnika(User) koji se šalje pod ključen owner šalje u bazu, korisničko ime i upisuje kao vlasnik datog predmeta.