

Testiranje i implementacija web servisa u Spring frameworku

SEMINARSKI RAD IZ PREDMETA NAPREDNE WEB TEHNOLOGIJE

SADRŽAJ

Uvod.....	3
Implementacija i testiranje servisa u Springu.....	4
Implementacija jednostavnog web servisa.....	4
Unit testovi.....	4
Integracijski testovi.....	4
Finance Tracker.....	6
Popis funkcionalnosti i ERD.....	6
Setup projekta pomoću Spring Initializr.....	7
Konekcija na bazu, kreiranje tabela i organizacija projekta.....	7
Kontroler i DTO klase.....	10
Unit testovi.....	13
Integracijski testovi.....	13
Demo.....	14
Zaključak.....	17
Reference	18
Popis slika.....	19

1 UVOD

Spring Boot je framework koji je uveliko olakšao posao inženjera koji se bave izgradnjom i razvojem web servisa za mikroservisnu arhitekturu. Spring je intuitivan framework napisan u programskom jeziku Java koji omogućava razvoj aplikacija uz minimalnu količinu ručne konfiguracije. Pored olakšane implementacije web servisa, Spring olakšava pisanje integracijskih i unit testova kako bi se osiguralo da servisi rade u skladu sa svojim namjenama.

U sklopu ovog rada se implementira *Finance Tracker* aplikacija u Spring Boot frameworku. Za ovu svrhu prvo će se definisati funkcionalnosti koje će ova aplikacija imati, kao i njen ERD. Nakon toga će se korak po korak prolaziti kroz implementaciju ovog servisa, uz prikladna objašnjenja i priložene isječke koda. Također će se razmotriti različite vrste testova, zašto su važni i kako se sprovode na praktičnom primjeru.

2 IMPLEMENTACIJA I TESTIRANJE SERVISA U SPRINGU

U ovom poglavlju se daje kratki uvod u metodologiju implementiranja i testiranja web servisa pomoću Spring Boot-a. Prva podsekcija se dotiče ključnih koraka implementacije, dok naredne dvije služe da se izlože *unit* i *integration* testovi kao primjeri dvije najčešće vrste testiranja koje programer servisa vrši.

2.1 IMPLEMENTACIJA JEDNOSTAVNOG WEB SERVISA

Kako bi se web servis što uspješnije i brže implementirao, potrebno je na neki strukturiran način pristupiti njegovoj implementaciji.

Prvi korak ka tome je definicija osnovnih funkcionalnosti servisa, kao i kreiranje *entity-relationship* dijagrama na kome se jasno vidi kako će izgledati baza servisa. Drugi korak je postavljanje odgovarajuće strukture u projektu. Nešto više o organizaciji projekta biće rečeno prilikom osvrta na praktični primjer *Finance Tracker* aplikacije.

Idući korak je odabir baze koja će se koristiti, kao i inicijalizacija projekta pomoću Spring Initializr platforme, koja uveliko olakšava i skraćuje posao developera [1]. Zatim se prelazi na izgradnju model klasa na osnovu ranije nacrtanog ERD-a koji će se koristiti da se pomoću *code-first* pristupa izgenerišu tabele za bazu podataka [2]. Atribute modela je potrebno validirati pomoću anotacija [3].

Nakon što imamo kreirane model klase, potrebno je kreirati *service*, *repository* i *controller* klase koje sadrže gotovo pa sav funkcionalni dio koda. Detaljan opis ovih klasa i čemu one služe će biti dat u narednom poglavlju.

Što se tiče testiranja implementiranog servisa, Spring omogućava generisanje testnih klasa za svaku *service* i *controller* klasu, što naravno nudi uštedu vremena.

2.2 UNIT TESTOVI

Unit test je test malog opsega koji developer piše kako bi bio siguran da neki određeni dio koda radi ono za šta je namijenjen. *Unit*, odnosno jedinica koda koja se testira, treba biti što je moguće manja i, ukoliko je moguće, izolirana od ostatka koda [4]. U svrhu unit testiranja, Spring Boot omogućava korištenje mnogih pomoćnih frameworka, od kojih su dva najpopularnija JUnit i Mockito [5].

2.3 INTEGRACIJSKI TESTOVI

Integracijski test je test koji služi da se ustanovi da li različiti dijelovi aplikacije koja se gradi ispravno rade zajedno. Integracijski testovi u Springu se najčešće koriste da se ustanovi da li

biznis sloj i *data persistence* sloj ispravno komuniciraju i da li je servis konektovan na odgovarajući bazu [6]. Ovi testovi se obično (mada ne i uvijek) pišu za kontrolere.

3 FINANCE TRACKER

3.1 POPIS FUNKCIONALNOSTI I ERD

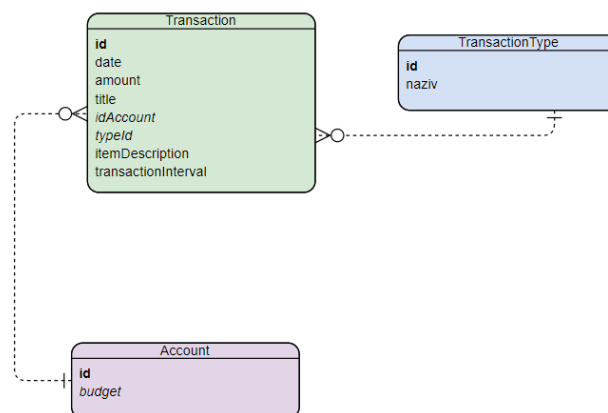
Prvi korak prilikom izgradnje bilo kakve aplikacije je kreiranje popisa osnovnih funkcionalnosti, koji se kasnije može širiti ili modificirati po potrebi. *Finance Tracker* aplikacija treba da u sebi sadrži razne vrste transakcija, prihoda i rashoda korisnika.

Popis osnovnih funkcionalnosti:

- kreiranje nove transakcije
- sortiranje transakcija po iznosu (rastuće i opadajuće)
- filtriranje transakcija po tipu (INDIVIDUALPAYMENT, REGULARPAYMENT, PURCHASE, INDIVIDUALINCOME, REGULARINCOME)

Svaka transakcija treba da ima svoj naziv, datum, iznos, tip (jedan od navedenih u tekstu iznad), opis kupljenog proizvoda (NULL ukoliko je tip transakcije neki od prihoda) i interval (broj dana nakon kojeg se transakcija REGULAR tipa ponavlja).

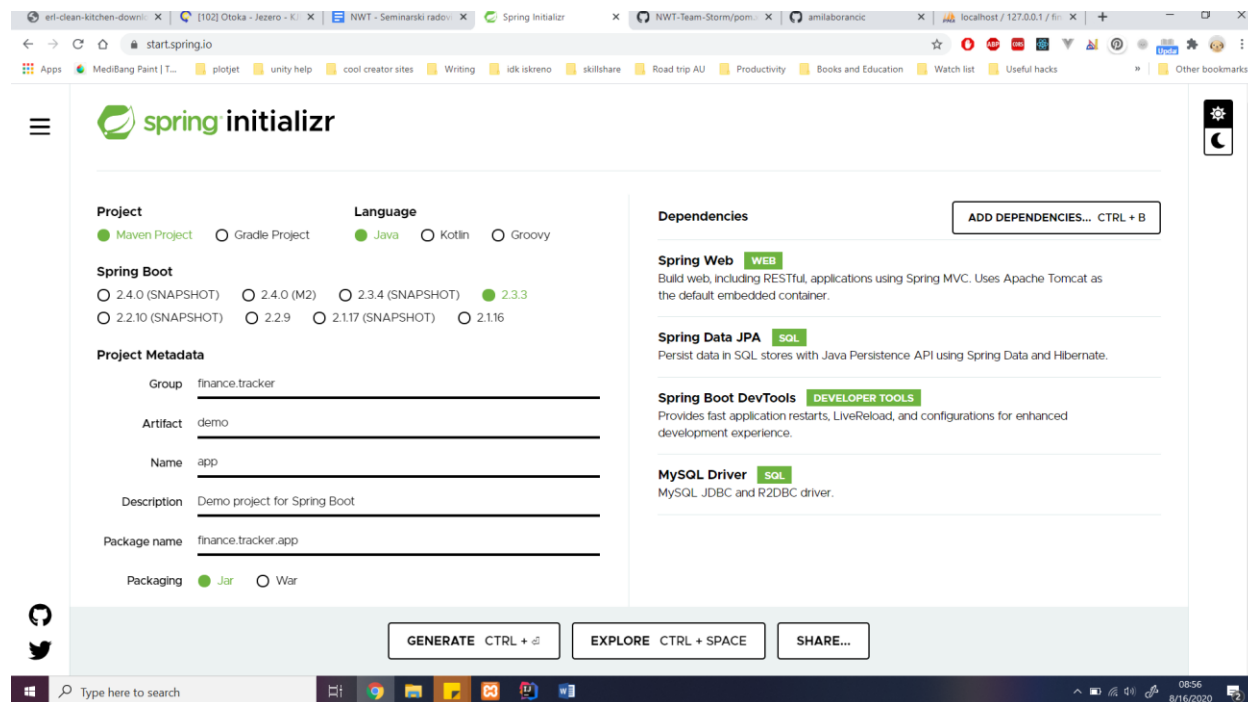
Svaki korisnički račun treba da ima trenutno novčano stanje računa. Treba napomenuti da bi se u mikroservisnoj arhitekturi pravio odvojen servis čija je uloga manipulisanje korisničkim računima, međutim u sklopu ovog rada je odlučeno da se to ne radi zato što je *Finance Tracker* aplikacija veoma mala i korisnički računi se koriste na svega nekoliko mjesta. Slika ispod predstavlja ERD *Finance Tracker* aplikacije:



Slika 1. ERD *Finance Tracker* aplikacije.

3.2 SETUP PROJEKTA POMOĆU SPRING INITIALIZR

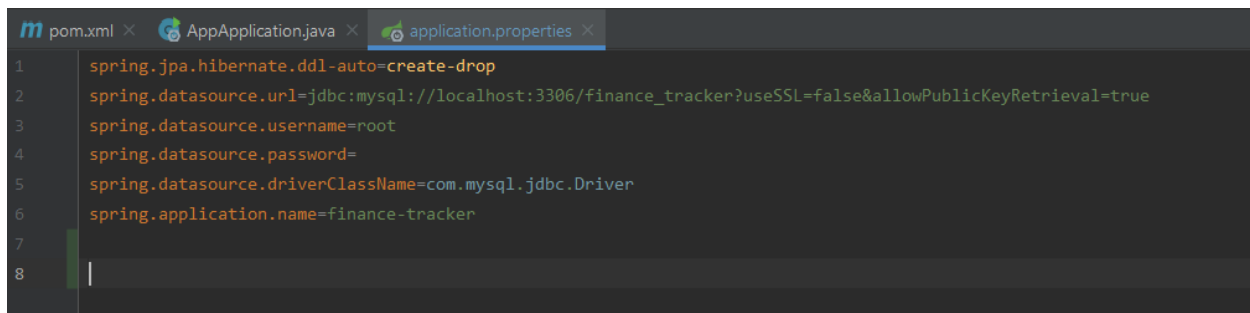
Spring Initializr se koristi kako bi se izgenerisao početni projekat. Odaberu se početni dependencies, verzija Jave koja je podržana na radnoj mašini i daje se naziv projektu kao na slici ispod:



Slika 2. Postavke projekta u Spring Initializr.

3.3 KONEKCIJA NA BAZU, KREIRANJE TABELA I ORGANIZACIJA PROJEKTA

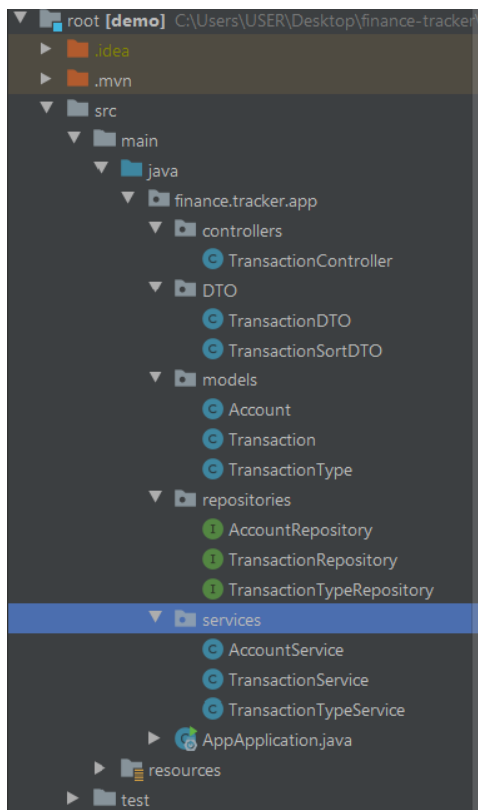
Kako aplikacija ne bi radila sa hardkodiranim podacima, potrebno je odabrati način spajanja na bazu i bazu popuniti nekim testnim podacima. Spring nam nudi ugrađenu bazu podataka koja se može koristiti pomoću H2 ili HSQL dependencija [7] kao i driver za MySQL bazu podataka, koji je odabran za ovaj projekat. U *application.properties* fajlu definišemo *connection string*, metod kojim će se puniti tabele u bazi prilikom svakog pokretanja (*create-drop* znači da će se prilikom svakog pokretanja servisa tabele nanovo kreirati a prilikom svakom gašenja brisati), kredencijale za pristup bazi i još par detalja.



```
1 spring.jpa.hibernate.ddl-auto=create-drop
2 spring.datasource.url=jdbc:mysql://localhost:3306/finance_tracker?useSSL=false&allowPublicKeyRetrieval=true
3 spring.datasource.username=root
4 spring.datasource.password=
5 spring.datasource.driverClassName=com.mysql.jdbc.Driver
6 spring.application.name=finance-tracker
7
8
```

Slika 3. Neophodni podaci za konekciju na bazu.

Nakon što su završene postavke za pristup bazi, iduće što je potrebno uraditi je kreirati modele koji će predstavljati tabele za bazu. Modeli za *Finance Tracker* aplikaciju su *Transaction*, *TransactionType* i *Account*, kao što je prikazano na ERD-u ranije. Snimak ekrana za modele neće biti priložen u ovom radu radi uštede prostora. Kako bi projekat bio dobro organizovan, preporučljivo je razdvojiti svu logiku aplikacije u zasebne foldere, odnosno *packages* u Spring Boot terminologiji, pa je to urađeno i u ovom projektu.



Slika 4. Organizacija projekta.

Na slici iznad dat je prikaz organizacije samog projekta. *Modeli* sadrže klase *Account*, *Transaction* i *TransactionType*. *TransactionType* klasa je u principu mogla biti *enum* ali znatno je jednostavnije čuvati *TransactionType* kao zasebnu klasu nego kao pobrojani tip radi kasnijeg manipulisanja istom. Sloj iznad *model* klasa su *repository* klase. Uloga *repository* klasa jeste

dobavljanje i spašavanje podataka u bazu, bilo pomoću direktnih upita ili pomoću Spring Data JPA [8], što je korišteno u ovom projektu.

```
9      @Repository
10     public interface TransactionRepository extends JpaRepository<Transaction, Long> {
11         List<Transaction> findByIdAccountOrderByAmountAsc(Long idAccount);
12         List<Transaction> findByIdAccountOrderByAmountDesc(Long idAccount);
13         List<Transaction> findAllByType(Long type);
14         List<Transaction> findAllByIdAccount(Long accountId);
15     }
16
```

Slika 5. TransactionRepository klasa.

Jedna *repository* klasa izgleda kao na slici iznad. Kao što se može primijetiti, nigdje nema definicije metoda klase *TransactionRepository*. Spring Data JPA omogućava da se na osnovu naziva metoda u *repository* klasi zna tačno šta koja od metoda radi, odnosno, u pozadini se naziv ovih metoda pretvara u upit koji se izvodi nad bazom. Tako recimo metoda *findByIdAccountOrderByAmountAsc* prvo pronalazi transakcije na osnovu *id* računa koji joj se proslijedi, nakon čega sortira dobijene transakcije po iznosu transakcije u rastućem poretku.

Sloj iznad *repository* klase predstavljaju *service* klase. Njihova uloga je da se izdvoji biznis logika od *data-access* logike koji sprovedu *repository* klase. Dakle, one direktno komuniciraju sa *repository* klasama, dok *controller* klase komuniciraju sa *repository* klasama samo preko *service* klase. Metode neke *service* klase mogu varirati u svojoj kompleksnosti i dužini - od par linija koda do mnogo kompleksnije logike.

```
8      @Service
9      public class AccountService {
10          @Autowired
11          AccountRepository accountRepository;
12
13          public boolean isUnderBudget(double amount, Long id){
14              Account acc = findAccountById(id);
15              if(amount < 0) amount*=-1;
16              return acc.getBudget() >= amount;
17          }
18
```

Slika 6. Primjer jednostavne service klase.

Na slici je prikazana jedna od metoda *AccountService* klase. Metoda je veoma jednostavna - ona provjerava da li neka osoba ima dovoljno sredstava na svom računu. U projektu koji je malog opsega kao *Finance Tracker* koji je implementiran za ovaj rad, korist *service* klasa ne dolazi do potpunog izražaja, međutim one su nezaobilazne u većim projektima.

Prije nego što se uđe u opis *controller* klase, ukratko će biti objašnjeno kako se popunjavaju tabele u bazi prilikom pokretanja servisa. Kontrolerima će biti posvećena čitava naredna sekcija ovog rada. U svrhu popunjavanja tabela prilikom pokretanja se koristi *DemoCommandLineRunner* klasa koja vrši *override* metode *run*. U toj metodi se piše kod koji će popuniti tabele u bazi.

```

30 @Component
31 class DemoCommandLineRunner implements CommandLineRunner {
32     @Autowired
33     private TransactionTypeService transactionTypeService;
34     @Autowired
35     private AccountService accountService;
36     @Autowired
37     private TransactionService transactionService;
38
39     @Override
40     public void run(String... args) throws Exception {
41         //dodamo sve tipove transakcija koji su zadani postavkom zadatka
42         TransactionType INDIVIDUALPAYMENT = new TransactionType( naziv: "INDIVIDUALPAYMENT");
43         TransactionType REGULARPAYMENT = new TransactionType( naziv: "REGULARPAYMENT");
44         TransactionType PURCHASE = new TransactionType( naziv: "PURCHASE");
45         TransactionType INDIVIDUALINCOME = new TransactionType( naziv: "INDIVIDUALINCOME");
46         TransactionType REGULARINCOME = new TransactionType( naziv: "REGULARINCOME");
47
48         transactionTypeService.save(INDIVIDUALPAYMENT);
49         transactionTypeService.save(REGULARPAYMENT);
50         transactionTypeService.save(PURCHASE);
51         transactionTypeService.save(INDIVIDUALINCOME);
52         transactionTypeService.save(REGULARINCOME);

```

Slika 7. Popunjavanje tabele tipova transakcija.

Na slici iznad je prikazan primjer punjenja tabele *transaction_types* sa pet tipova koji su definisani ranije. Prvo što se radi je *autowiring* svih servisa koji su kreirani. *Autowire anotacija* služi da Spring izvrši *dependency injection* svih *beans* koji su mu potrebni za rad [9]. Nakon toga, sve što je potrebno uraditi je ručno kreirati par instanci i pozvati metodu *save* da se one spase u bazu. Rezultat pokretanja servisa je prikazan na slici ispod.

+ Options

↔

↕

id

amount

date

id_account

item_description

title

transaction_interval

type_id

Edit

Copy

Delete

1

-5.55

2020-08-18 14:59:55

1

A set of 100 Naruto stickers

100 Sticker Pack

NULL

3

Edit

Copy

Delete

2

-40.35

2020-08-18 14:59:55

2

Racun za struju za tekuci mjesec

Racun za struju

30

2

Edit

Copy

Delete

3

-100

2020-08-18 14:59:55

2

Slanje novca

Slanje novca ka inostranstvu

NULL

1

Edit

Copy

Delete

4

200

2020-08-18 14:59:55

1

Neko je kupio Vas artikal.

Prihod od narudzbe iz online storea

NULL

4

Edit

Copy

Delete

5

1500

2020-08-18 14:59:55

2

Plata za tekuci mjesec

Plata

30

5

↑

Check all

With selected:

Edit

Copy

Delete

Export

+ Options

↔

↕

id

naziv

Edit

Copy

Delete

1

INDIVIDUALPAYMENT

Edit

Copy

Delete

2

REGULARPAYMENT

Edit

Copy

Delete

3

PURCHASE

Edit

Copy

Delete

4

INDIVIDUALINCOME

Edit

Copy

Delete

5

REGULARINCOME

↑

Check all

With selected:

Edit

Copy

Delete

+ Options

↔

↕

id

budget

Edit

Copy

Delete

1

500

Edit

Copy

Delete

2

1000

↑

Check all

With selected:

Edit

Slika 8. Popunjene tabele nakon pokretanja servisa.

3.4 KONTROLER I DTO KLASA

Nakon što su objašnjene uloge *model*, *service* i *repository* klasa, prelazi se na idući korak u implementaciji servisa - *kontroler*.

Kontroleri predstavljaju klase u kojima se definišu *endpoints* koje će okidati neki *web client*. Neke od uloga kontrolera su komunikacija među servisima (ukoliko ih je više, što je čest slučaj), odgovaranje na zahtjeve koje klijent šalje, *error handling* sa odgovarajućim porukama itd. U slučaju *Finance Tracker* aplikacije, kontroler *TransactionController* sadrži većinu funkcionalnosti ove aplikacije, pa je od interesa da se on detaljnije obradi.

```
19 @RestController
20 @RequestMapping("/transaction")
21 public class TransactionController {
22     @Autowired
23     TransactionService transactionService;
24     @Autowired
25     TransactionTypeService transactionTypeService;
26     @Autowired
27     AccountService accountService;
28
29     @PostMapping("/new")
30     public Long createNewTransaction(@RequestBody TransactionDTO newTransaction) throws Exception {
31         Long typeId = newTransaction.getType();
32         String typeName = transactionTypeService.findTypeById(typeId).getNaziv();
33         //validacija
34         if((typeName.equals("REGULARINCOME") || typeName.equals("INDIVIDUALINCOME")) && newTransaction.getItemDescription() != null)
35             throw new Exception("Transakcije tipa INCOME nemaju opis kupljenog proizvoda.");
36         if(!typeName.equals("REGULARINCOME") && !typeName.equals("REGULARPAYMENT") && newTransaction.getTransactionInterval() != null)
37             throw new Exception("Transakcije koje nisu REGULAR tipa se ne ponavljaju periodično.");
38
39         //provjera da li ima dovoljno sredstava na racunu ukoliko se radi o rashodu
40         if((typeName.equals("REGULARPAYMENT") || typeName.equals("INDIVIDUALPAYMENT") || typeName.equals("PURCHASE")) &&
41             !accountService.isUnderBudget(newTransaction.getAmount(), newTransaction.getIdAccount()))
42             throw new Exception("Nemate dovoljno sredstava za ovu transakciju!");
43
44         Transaction transaction = new Transaction(new Date(), newTransaction.getAmount(), newTransaction.getTitle(),
45             typeId, newTransaction.getItemDescription(), newTransaction.getTransactionInterval(), newTransaction.getIdAccount());
46         transactionService.save(transaction);
47
48         //update stanja na racunu
49         Account acc = accountService.findAccountById(newTransaction.getIdAccount());
50         accountService.updateAccountBudget(newTransaction.getIdAccount(), newBudget: acc.getBudget() + newTransaction.getAmount());
51
52         return transaction.getId();
53     }
54 }
```

Slika 9. TransactionController - metoda createNewTransaction

Kontroler označavamo anotacijom *@RestController* kako bi Spring Boot upisao podatke direktno u *response body*. Zatim anotacijom *@RequestMapping* označavamo dio *endpointa* koji je zajednički za sve metode ove klase.

Iduća stvar koja je od interesa na slici iznad je implementacija metode *createNewTransaction*. Ova metoda se nalazi na *endpointu* */transaction/new* i koristi se za prvu funkcionalnost - dodavanje nove transakcije, tipa je POST i prima tijelo zahtjeva koje je tipa *TransactionDTO*. *TransactionDTO* je *data transfer object* [10] koji se koristi da se Springu kaže kako treba da izgleda tijelo zahtjeva. *TransactionDTO* objekat ne sadrži sve attribute koje sadrži *Transaction* model.

Prvo što se radi u metodi je da se izdvoji tip transakcije. Ovo je neophodno radi validacije koja je izdefinisana ranije. Iz *transactionTypeService* dobijemo naziv tipa pomoću id-ja (ovo je urađeno radi bolje čitljivosti) i počinjemo sa validacijom. U tri slučaja u kojima može doći do bacanja izuzetka korišten je ugrađeni *Exception* tip izuzetka, što bi se moglo mnogo unaprijediti pisanjem vlastitih tipova izuzetaka.

Ukoliko su sve validacije prošle, iz *TransactionDTO* objekta koji je proslijeđen se kreira nova transakcija koja se spašava u bazu. Nakon toga, potrebno je ažurirati stanje na računu skidanjem ili dodavanjem novca na račun (u ovisnosti koji je tip transakcije u pitanju). Tu se koristi *accountService* metoda *updateAccountBudget* koja je prikazana u prethodnom poglavlju u opisu *service* klasa. Konačno, ukoliko je sve prošlo uredno, kao rezultat se vraća id uspješno dodane transakcije.

Iduće dvije funkcionalnosti su sortiranje po iznosu transakcije i filtriranje po tipu transakcije. Iako mogu zvučati komplikovano, Spring omogućava da *endpointi* za ove dvije funkcionalnosti izgledaju posve jednostavno:

```
55      @PostMapping("/sort/amount")
56      public List<Transaction> sortTransactionsByAmount(@RequestBody TransactionSortDTO dto){
57          return transactionService.getTransactionsSortedByAmount(dto);
58      }
59
60      @GetMapping("/filter/type")
61      public List<Transaction> filterTransactionsByType(@RequestParam Long typeId){
62          return transactionService.getTransactionsByType(typeId);
63      }
64  }
65
```

Slika 10. *TransactionController* - metode *sortTransactionByAmount* i *filterTransactionsByType*

Kao što se vidi na slici iznad, *endpointi* za ove dvije funkcionalnosti se sastoje od samo jedne linije koda. Sva dodatna logika je pomjerena u *TransactionService* klasu:

```
20 @ public List<Transaction> getTransactionsSortedByAmount(TransactionSortDTO dto){
21     if(dto.getOrderAsc()) return transactionRepository.findByIdAccountOrderByAmountAsc(dto.getAccountId());
22     return transactionRepository.findByIdAccountOrderByAmountDesc(dto.getAccountId());
23 }
24 public List<Transaction> getTransactionsByType(Long typeId) { return transactionRepository.findAllByType(typeId); }
27
```

Slika 11. Dio *TransactionService* klase.

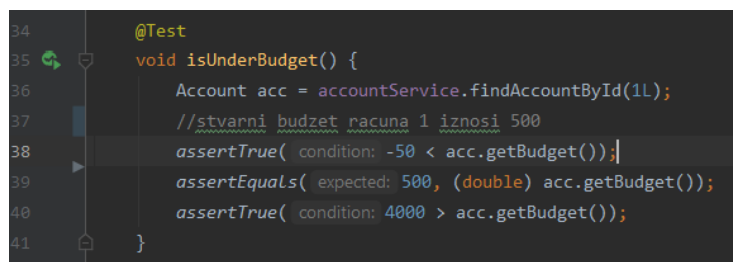
Što se tiče metode za sortiranje po iznosu, ona prima *TransactionSortDTO* objekat kao parametar. Ovaj objekat se sastoji od dva atributa - id računa na koji ide transakcija i *boolean* atributa koji označava da li se sortira u rastućem ili opadajućem poretку. Prvo se provjerava da li je atribut *asc* true, i ako jeste znači da se sortira u rastućem (eng. *ascending*), a u suprotnom u opadajućem poretку. Nakon toga se poziva jedna od metoda *findByIdAccountOrderByAmountAsc* ili *findByIdAccountOrderByAmountDesc* iz *TransactionRepository*, u ovisnosti od toga koja vrsta sortiranja je odabrana i vraća se rezultat upita.

Metoda za filtriranje je nešto kraća i sastoji se samo od poziva metode *findAllByType* iz *TransactionRepository*, te vraćanja rezultata upita.

S obzirom da se radi o malom projektu, može se primijetiti da na nekoliko mjesta nema pretjerane koristi od korištenja *service* klase, s obzirom da se ista funkcionalnost može postići i kada bi se direktno iz kontrolera komuniciralo sa *repository* klasama, međutim to nije dobra praksa i iz tog razloga je odlučeno da se ona ne koristi ni u ovom radu.

3.5 UNIT TESTOVI

Unit testovi za *Finance Tracker* aplikaciju su pisani za sve *service* klase. S obzirom da *service* klase sadrže biznis logiku i direktno komuniciraju sa *repository* klasama, one su pogodan kandidat za unit testove.



```
34      @Test
35      void isUnderBudget() {
36          Account acc = accountService.findAccountById(1L);
37          //stvarni budzet racuna 1 iznosi 500
38          assertTrue( condition: -50 < acc.getBudget());
39          assertEquals( expected: 500, (double) acc.getBudget());
40          assertTrue( condition: 4000 > acc.getBudget());
41      }
```

Slika 12. Unit test za metodu *isUnderBudget* service klase *AccountService*.

Na slici iznad je prikazan jedan unit test koji pripada klasi *AccountServiceTest*. Testira se metoda *isUnderBudget* u tri različita slučaja. Prvo se testira da li se na računu nalazi dovoljno sredstava za neki vid kupovine od 50 novčanih jedinica i očekuje se da ima (pošto se u stvarnosti na računu nalazi 500 novčanih jedinica). Zatim se budžet testira na jednakost pomoću *assertEquals* metode, a zatim da li je budžet manji od 4000 novčanih jedinica (očekuje se da jeste).

Ovim je utvrđeno da metoda *isUnderBudget* radi onako kako treba da radi, s obzirom na to da su testirani svi slučajevi koji mogu nastupiti. Preostali unit testovi neće biti navedeni u sklopu rada radi uštede prostora, s obzirom da se svakako nalaze u projektu koji će biti priložen uz rad.

3.6 INTEGRACIJSKI TESTOVI

Integracijski testovi, kao što je već ranije rečeno, služe da se testira interakcija različitih dijelova aplikacije. Iz ovog razloga su *controller* klase pogodne za integracijske testove, s obzirom da one komuniciraju sa *service* klasama, koje onda komuniciraju sa *repository* klasama. Ispravnim testiranjem kontrolera se može istestirati većina koda aplikacije.

```
AccountServiceTest.java × TransactionServiceTest.java × TransactionControllerTest.java ×
36 @Test
37 void createNewTransaction() throws Exception {
38
39     //sve uredi
40     TransactionDTO t1 = new TransactionDTO((double) -200, title: "Desk", idAccount: 1L, type: 3L, itemDescription: "Wooden work desk.", transactionInterval: null);
41     mockMvc.perform(post( uriTemplate: "/transaction/new")
42         .contentType(MediaType.APPLICATION_JSON)
43         .content(asJsonString(t1)))
44         .andExpect(status().isOk());
45
46     //INCOME transakcija sa opisom proizvoda - baca izuzetak
47     TransactionDTO t2 = new TransactionDTO((double) 10, title: "Nesto", idAccount: 1L, type: 4L, itemDescription: "nesto nesto", transactionInterval: null);
48     Assertions.assertThrows(Exception.class,
49         ()->mockMvc.perform(post( uriTemplate: "/transaction/new").contentType(MediaType.APPLICATION_JSON).content(asJsonString(t2))));
50
51
52     //PURCHASE transakcija se periodično ponavlja - baca izuzetak
53     TransactionDTO t3 = new TransactionDTO((double) -10, title: "blabla", idAccount: 1L, type: 3L, itemDescription: "smlsm", transactionInterval: 20);
54     Assertions.assertThrows(Exception.class,
55         ()->mockMvc.perform(post( uriTemplate: "/transaction/new").contentType(MediaType.APPLICATION_JSON).content(asJsonString(t3))));
56
57     //prekoracen budzet - baca izuzetak
58     TransactionDTO t4 = new TransactionDTO((double) -1000000, title: "blabla", idAccount: 1L, type: 3L, itemDescription: "smlsm", transactionInterval: null);
59     Assertions.assertThrows(Exception.class,
60         ()->mockMvc.perform(post( uriTemplate: "/transaction/new").contentType(MediaType.APPLICATION_JSON).content(asJsonString(t4))));
61 }
62
```

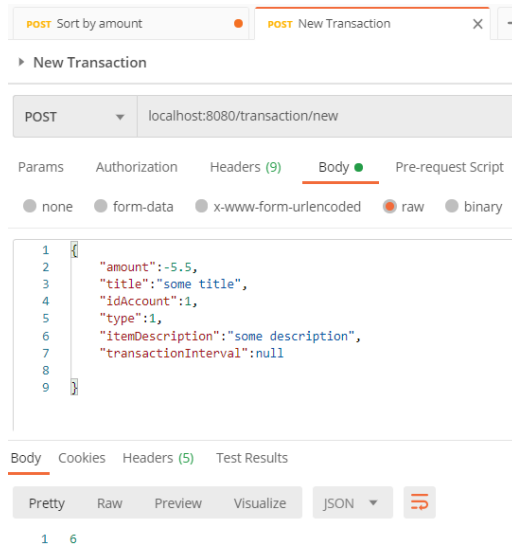
Slika 13. Integracijski test za metodu `createNewTransaction` kontrolera `TransactionController`.

Metoda `createNewTransaction` je interesantna za integracijsko testiranje s obzirom da može nastupiti mnogo slučajeva kada kreiranje transakcije ne uspije. Prvo kreiramo `TransactionDTO` objekat sa podacima za koje očekujemo da test uspije. Zatim se pomoću `mockMvc` [11] pošalje `mock` HTTP zahtjev kontroleru da se vidi da li kontroler ispravno radi. Nakon toga se slučaj po slučaj testiraju sve situacije koje mogu nastupiti a u kojima će kontroler baciti izuzetak. Prvi takav slučaj je da INCOME tip transakcije ima opis proizvoda (što nije dozvoljeno u specifikaciji). Kreira se transakcija `t2` koja je INCOME tipa a ima `itemDescription` koji nije null i pokuša se poslati `mock` POST zahtjev da se ta transakcija spasi u bazu. S obzirom da se očekuje bacanje izuzetka, koristi se metoda `assertThrows` koja kao argumente prima tip izuzetka (`Exception.class`) i lambda funkciju kojom se šalje zahtjev.

Drugi slučaj je da se PURCHASE transakcija periodično ponavlja, što također nije dozvoljeno po specifikaciji, i treći je da je budžet prekoračen, odnosno da na računu nema dovoljno sredstava da se izvrši transakcija. Tu se također koristi metoda `assertThrows` da se utvrdi da li kontroler zaista baca izuzetak.

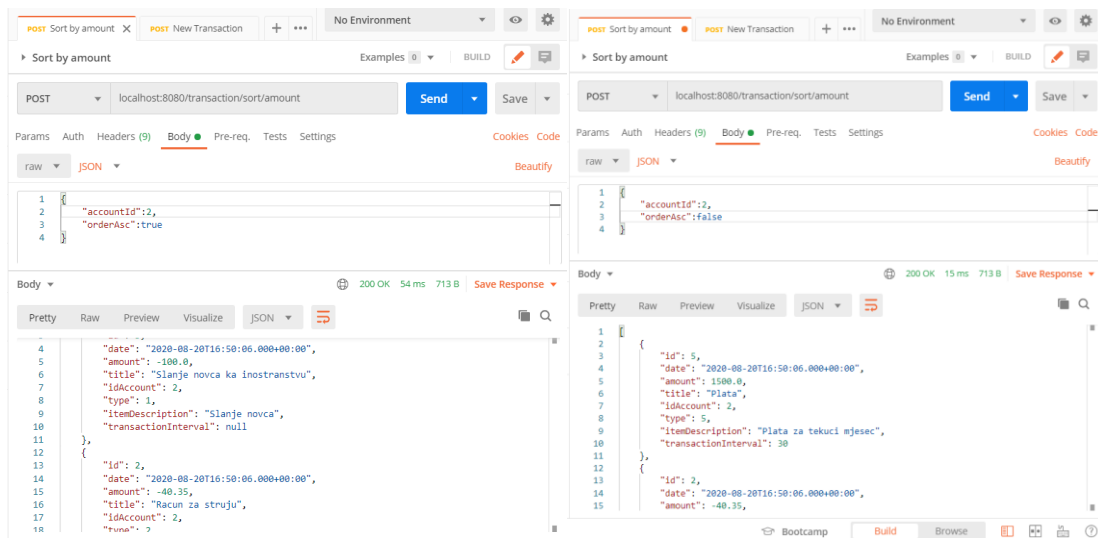
3.7 DEMO

U ovoj sekciji će biti dati kratki prikaz rada aplikacije. Svi zahtjevi su poslani kroz alat POSTMAN. Prva funkcionalnost koja je definisana u sekciji 3.1 je kreiranje nove transakcije, pa je to prvo što će se pokazati.



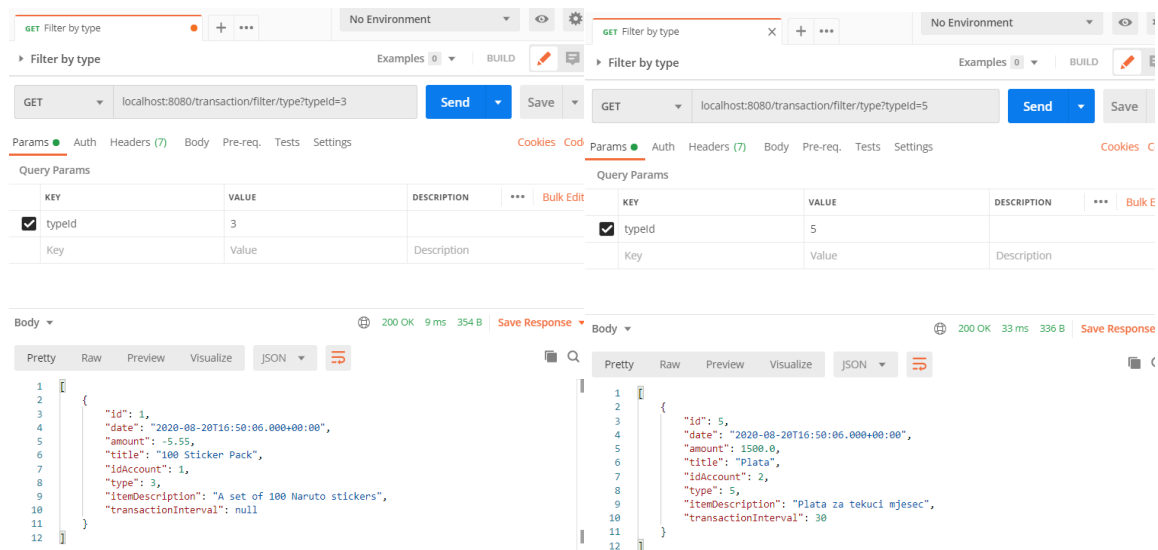
Slika 14. Kreiranje nove transakcije.

Dakle, kao tijelo POST zahtjeva šaljemo atribut *amount*, *title*, *idAccount*, *type*, *itemDescription* i *transactionInterval*, kao što je pokazano u sekciji 3.4. *Endpoint* na koji se šalje zahtjev glasi `/transaction/new`, na port 8080. Iduće što se testira je sortiranje:



Slika 15. Sortiranje u rastućem i opadajućem poretku, respektivno.

Vidimo da sortiranje radi ispravno jer se u prvom slučaju transakcije zaista sortiraju rastuće a u drugom opadajuće, kao što i treba. Slijedi filtriranje:



Slika 16. Filtriranje po tipu.

Za filtriranje ne šaljem tijelo zahtjeva s obzirom da se radi o GET zahtjevu, već kao URL parametar šaljem *id* tipa po kojem filtriramo i kao rezultat dobijamo niz transakcija tog tipa.

ZAKLJUČAK

Spring Boot je framework koji uveliko olakšava razvoj web servisa. Određene postavke, validacija podataka i još mnoge stvari se mogu na jednostavan način riješiti anotacijama.

U svrhu ovog seminarskog rada razvijen je jednostavni web servis koji ima nekoliko jednostavnih funkcionalnosti. Ove funkcionalnosti je moguće proširiti i nadograditi, a sam servis je moguće poboljšati dodavanjem vlastitih tipova izuzetaka, strožijom validacijom podataka, te dodavanjem autorizacije i autentikacije.

REFERENCE

- [1] Spring Initializr: <https://freecontent.manning.com/wp-content/uploads/initializing-a-spring-boot-project-with-spring-initializr.pdf> [Posljednji pristup: 17.08.2020.]
- [2] Code-first pristup: https://www.tutorialspoint.com/entity_framework/entity_framework_code_first_approach.htm#:~:text=Code%20First%20modeling%20workflow%20targets,C%23%20or%20VB.Net%20classes. [Posljednji pristup: 17.08.2020.]
- [3] Validacija pomoću anotacija: <https://www.baeldung.com/spring-boot-bean-validation> [Posljednji pristup: 17.08.2020.]
- [4] Unit testovi: <https://stackoverflow.com/questions/5357601/whats-the-difference-between-unit-tests-and-integration-tests> [Posljednji pristup: 15.08.2020.]
- [5] JUnit i Mockito: <https://www.springboottutorial.com/spring-boot-unit-testing-and-mocking-with-mockito-and-junit> [Posljednji pristup: 15.08.2020.]
- [6] Integracijski testovi: <https://reflectoring.io/spring-boot-test/> [Posljednji pristup: 15.08.2020.]
- [7] Embedded baze podataka za Spring: <https://www.baeldung.com/spring-boot-h2-database> [Posljednji pristup: 16.08.2020.]
- [8] Spring Data JPA: <https://spring.io/projects/spring-data-jpa> [Posljednji pristup: 18.08.2020.]
- [9] Autowire anotacija: <https://www.javatpoint.com/autowiring-in-spring> [Posljednji pristup: 18.08.2020.]
- [10] Data transfer object: <https://stackoverflow.com/questions/1051182/what-is-data-transfer-object#:~:text=A%20Data%20Transfer%20Object%20is,itself%20and%20the%20UI%20layer.> [Posljednji pristup: 18.08.2020.]
- [11] MockMvc: <https://dzone.com/articles/testing-spring-mvc-with-spring-boot-14-part-1> [Posljednji pristup: 20.08.2020.]

POPIS SLIKA

Slika 1 ERD <i>Finance Tracker</i> aplikacije.....	6
Slika 2 Postavke projekta u Spring Initializr.....	7
Slika 3 Neophodni podaci za konekciju na bazu.....	8
Slika 4 Organizacija projekta.....	8
Slika 5 <i>TransactionRepository</i> klasa.....	9
Slika 6 Primjer jednostavne <i>service</i> klase.....	9
Slika 7 Popunjavanje tabele tipova transakcija.....	10
Slika 8 Popunjene tabele nakon pokretanja servisa.....	10
Slika 9 <i>TransactionController</i> - metoda <i>createNewTransaction</i>	11
Slika 10 <i>TransactionController</i> - metode <i>sortTransactionByAmount</i> i <i>filterTransactionsByType</i>	12
Slika 11 Dio <i>TransactionService</i> klase.....	12
Slika 12 Unit test za metodu <i>isUnderBudget</i> service klase <i>AccountService</i>	13
Slika 13 Integracijski test za metodu <i>createNewTransaction</i> kontrolera <i>TransactionController</i>	14
Slika 14 Kreiranje nove transakcije.....	15
Slika 15 Sortiranje u rastućem i opadajućem poretku, respektivno.....	15
Slika 16 Filtriranje po tipu.....	16