Testiranje i implementacija web servisa u Spring frameworku

SEMINARSKI RAD IZ PREDMETA NAPREDNE WEB TEHNOLOGIJE

Amila Borančić 17934/1705 DATUM, Sarajevo

# SADRŽAJ

Uvod

Implementacija i testiranje servisa u Springu

Implementacija jednostavnog web servisa

Unit testovi

Integracijski testovi

Finance Tracker aplikacija

Popis funkcionalnosti i ERD

Setup projekta

Tabele i konekcija na bazu

Kontroler

Demonstracija rada aplikacije

Zaključak

Reference

Popis slika

Isječci koda

# Uvod

Spring Boot je framework koji je uveliko olakšao posao inžinjera 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 nudi i mnogo dependencija koji su namijenjeni za 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. Biće obrazložene različite vrste testova za web servise, zašto su važni i kako se implementiraju u Springu na Finance Tracker aplikaciji.

# 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.

## Implementacija jednostavnog web servisa

Kako bi se web servis što uspješnije i brže implementirao, potrebno je na neki struktuiran 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 na osnovu tih klasa 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.

## 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 nudi mnoge dependencije, od kojih su dva najpopularnija JUnit i Mockito [5].

## 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 koristi 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.

# Finance Tracker

## 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:

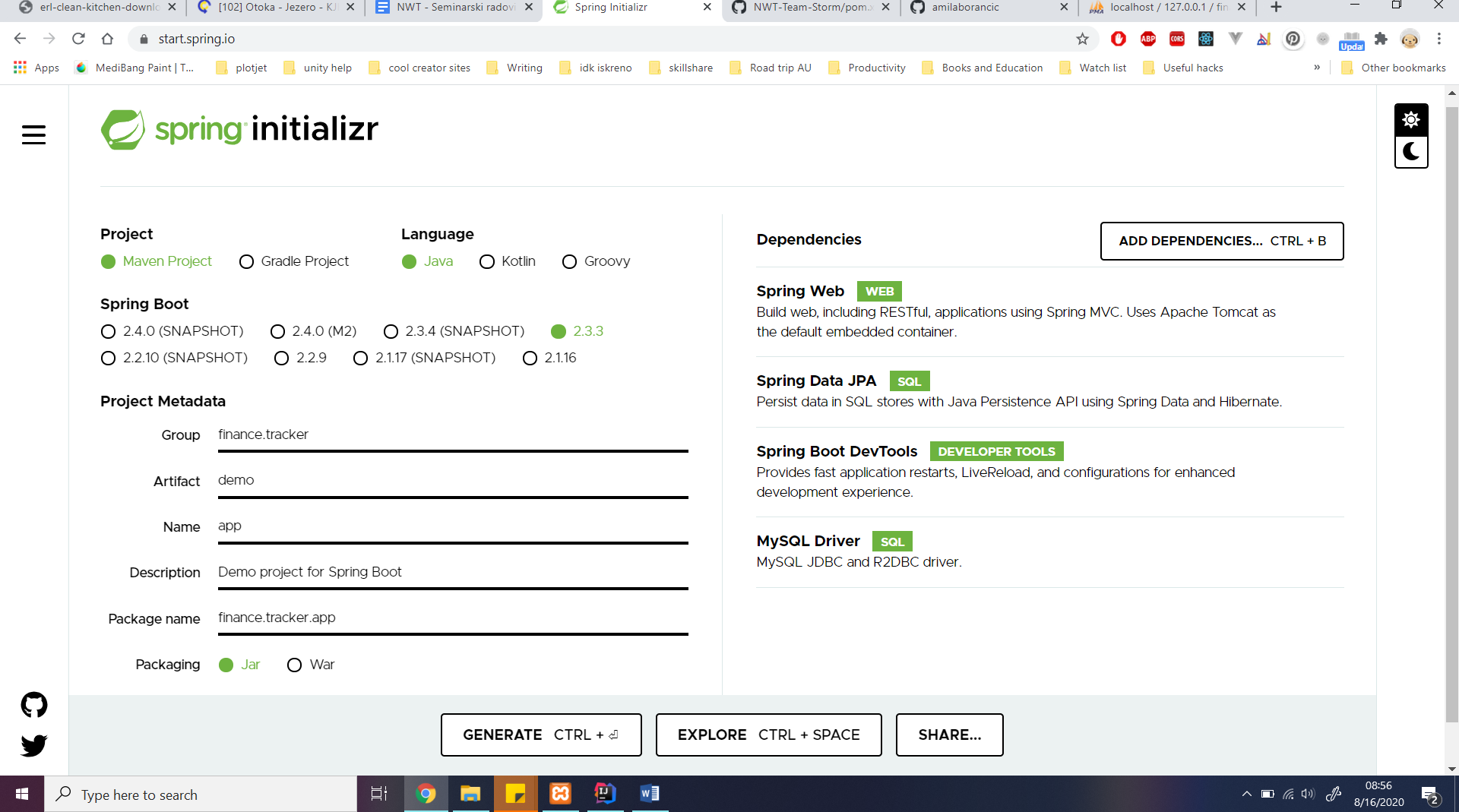
* kreiranje nove transakcije
* sortiranje transakcija po iznosu (rastuće i opadajuće)
* filtriranje transakcija po tipu (INDIVIDUALPAYMENT, REGULARPAYMENT, PURCHASE, INDIVIDUALINCOME, REGULARINCOME)

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.

## Setup projekta pomoću Spring Initializr

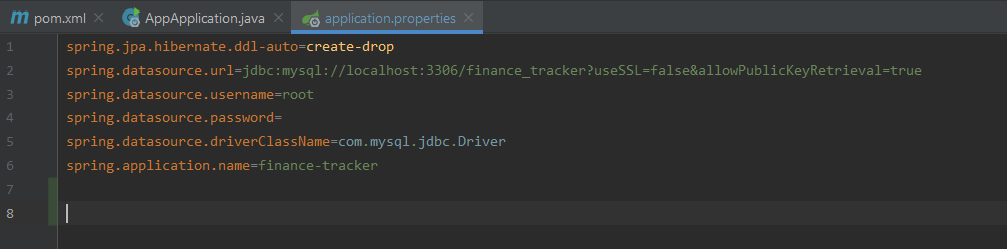
Spring Initializr se koristi kako bi se izgenerisao početni projekat. Odaberu se početni dependencies, verzija Jave koja je podržana na mašini i daje se naziv projektu kao na slici ispod:



Slika 1. Postavke projekta u Spring Initializr.

## Konekcija na bazu, kreiranje tabela i organizacija projekta

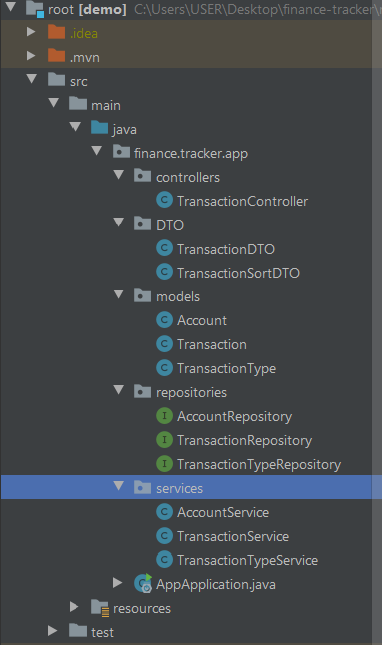
Kako aplikacija ne bi radila sa hardkodiranim podacima, potrebno je odabrati način spajanja na bazu i bazu popuniti nekim testnim podacima. Spring nam nudi ugrađenu bazu podataka koja se može koristiti pomoću H2 ili HSQL dependencija [7] kao i driver za MySql bazu podataka, koji je odabran za ovaj projekat. 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.



Slika 2. Neophodni podaci za konekciju na bazu.

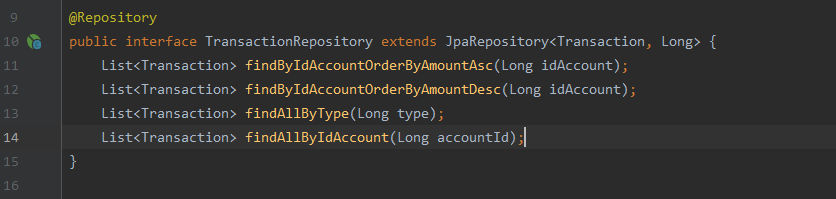
Nakon što su završene postavke za pristup bazi, iduće što je potrebno uraditi je kreirati modele koji će predstavljati tabele za bazu. Modeli za *Finance Tracker* aplikaciju su *Transaction*, *TransactionType* i *Account*, kao što je prikazano na ERD-u ranije. Snimak ekrana za modele neće biti priložen 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 3. Organizacija projekta.

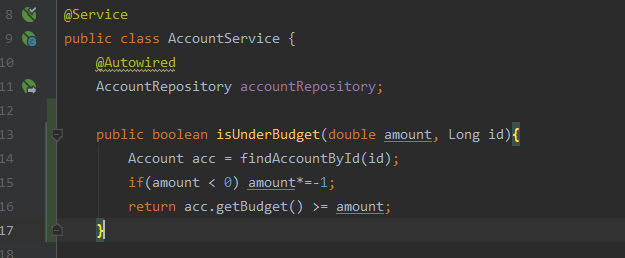
Na slici iznad dat je prikaz organizacije samog projekta. *model* klase sadrže klase, *Account*, *Transaction* i *TransactionType. TransactionType* klasa je u principu mogla biti neki *enum* ali znatno je jednostavnije čuvati *TransactionType* kao zasebnu klasu nego kao pobrojani tip radi kasnijeg manipulisanja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



Slika 4. 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.

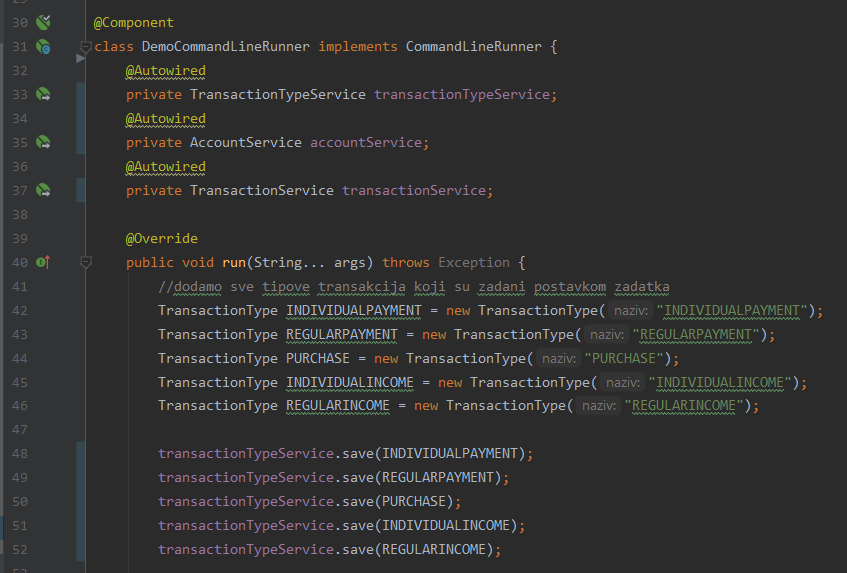
Sloj iznad *repository* klasa predstavljaju *service* klase. Njihova uloga je da se izdvoji biznis logika od *data-access* logike koji sprovode *repository* klase. Dakle, one direktno komuniciraju sa *repository* klasama, dok *controller* klase komuniciraju sa *repository* klasama samo preko *service*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



Slika 5. 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.

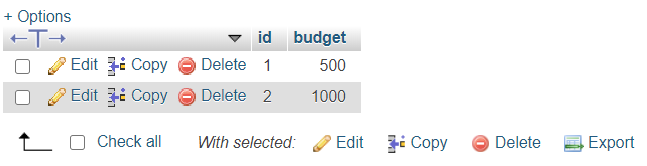
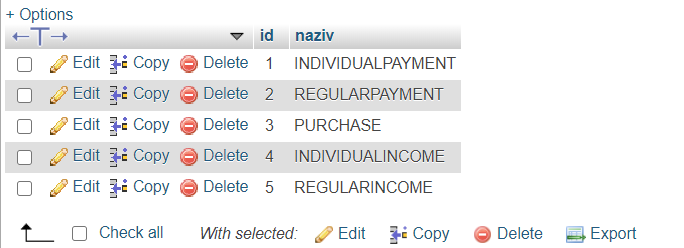
Prije nego što se uđe u opis *controller* klasa, ukratko će biti objašnjeno kako se popunjavaju tabele u bazi prilikom pokretanja servisa. Kontrolerima će biti posvećena čitava naredna sekcija ovog rada. U svrhu popunjavanja tabela prilikom pokretanja se koristi *DemoCommandLineRunner* klasa koja vrši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



Slika 6. 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.





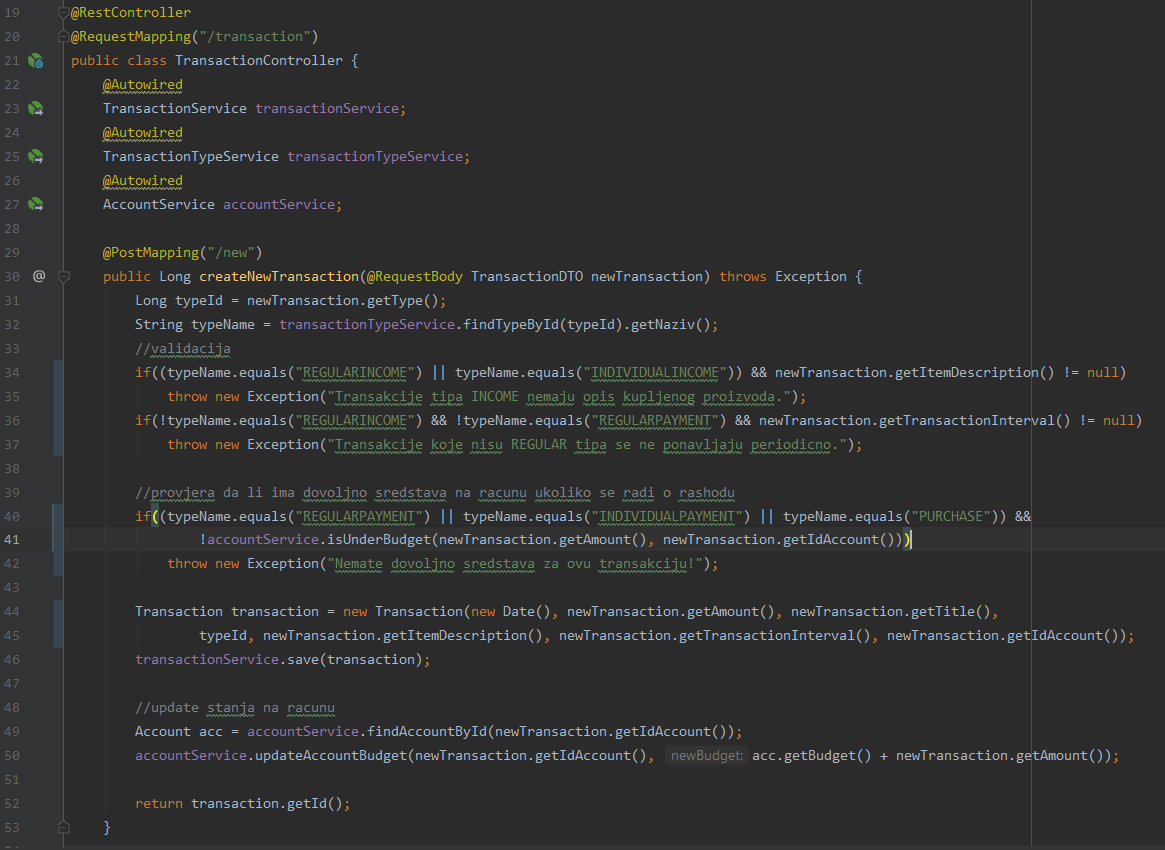
Slika 7. Popunjene tabele nakon pokretanja servisa.

## Kontroler i DTO klase

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.



Slika 8. TransactionController - metoda createNewTransaction

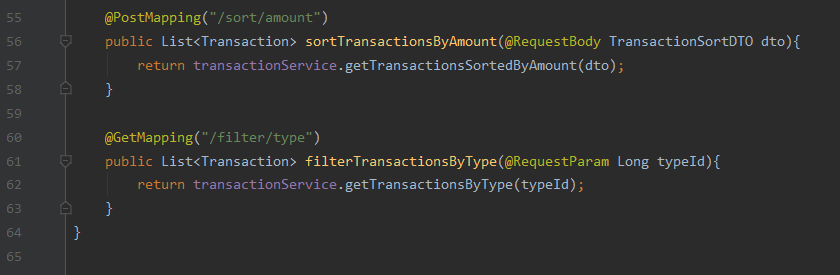
Kontroler označavamo anotacijom @RestController kako bi Spring Boot upisao podatke direktno u *response body*. Zatim anotacijom @RequestMapping označavamo dio *endpointa* koji je zajednički za sve metode ove klase.

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 atribute 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.

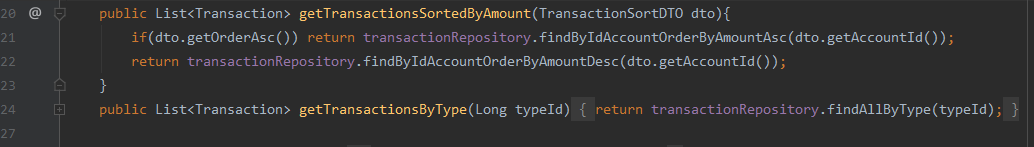
Ukoliko su sve validacije prošle, iz *TransactionDTO* objekta koji je proslijeđen se kreira nova transakcija koja se spašava u bazu. Nakon toga, potrebno je ažurirati stanje na računu skidanjem ili dodavanjem novca na račun (u ovisnosti koji je tip transakcije u pitanju). Tu se koristi *accountService*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



Slika 9. 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:



Slika 10. 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 poretku. 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 poretku. Nakon toga se poziva jedna od metoda *findByAccountOrderByAmountAsc* ili *findByAccountOrderByAmountDesc* 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* klasa, 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.

## Unit testovi

dkmckdlmd

## Integracijski testovi

sklnclks

# 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.]

# Popis slika

scscs