# Izrada pozadinskog poslužitelja koristeći programski okvir Springboot na studijskom slučaju TODO aplikacije

**Starčević, Luka**



**VELEUČILIŠTE U ŠIBENIKU**

**ODJEL POSLOVNE INFORMATIKE**

**PREDDIPLOMSKI STRUČNI STUDIJ POSLOVNE INFORMATIKE**

**Luka Starčević**

# IZRADA POZADINSKOG POSLUŽITELJA KORISTEĆI PROGRAMSKI OKVIR SPRING BOOT NA STUDIJSKOM SLUČAJU TODO APLIKACIJE

**Završni rad**

Šibenik, 2023.

**VELEUČILIŠTE U ŠIBENIKU**

**ODJEL POSLOVNE INFORMATIKE**

**PREDDIPLOMSKI STRUČNI STUDIJ POSLOVNE INFROMATIKE**

# IZRADA POZADINSKOG POSLUŽITELJA KORISTEĆI PROGRAMSKI OKVIR SPRING BOOT NA STUDIJSKOM SLUČAJU TODO APLIKACIJE

**Završni rad**

**Kolegij:** Razvoj web aplikacija

**Mentor:** Marko Pavelić

**Student:** Luka Starčević

**Matični broj studenta:** 1219063858

Šibenik, rujan 2023.

**IZJAVA O AKADEMSKOJ ČESTITOSTI**

Ja, Luka Starčević, student/ica Veleučilišta u Šibeniku, JMBAG 1219063858 izjavljujem pod materijalnom i kaznenom odgovornošću i svojim potpisom potvrđujem da je moj završni rad na preddiplomskom/specijalističkom diplomskom stručnom studiju Poslovna informatika pod naslovom: Izrada pozadinskog poslužitelja koristeći programski okvir Spring boot na studijskom slučaju TODO aplikacije

isključivo rezultat mog vlastitog rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu, a što pokazuju korištene bilješke i bibliografija.

Izjavljujem da nijedan dio rada nije napisan na nedozvoljen način, odnosno da je prepisan iz necitiranog rada, te da nijedan dio rada ne krši bilo čija autorska prava.

Izjavljujem, također, da nijedan dio rada nije iskorišten bilo koji drugi rad u bilo kojoj drugoj visokoškolskoj, znanstvenoj ili obrazovnoj ustanovi.

U Šibeniku, 12.9.2023

Student/ica:

Luka Starčević

TEMELJNA DOKUMENTACIJSKA KARTICA

Veleučilište u Šibeniku Završni rad

Odjel Poslovne informatike

Preddiplomski/Specijalistički diplomski stručni studij Poslovna informatika

# IZRADA POZADINSKOG POSLUŽITELJA KORISTEĆI PROGRAMSKI OKVIR SPRING BOOT NA STUDIJSKOM SLUČAJU TODO APLIKACIJE

LUKA STARČEVIĆ

Budačka ulica 210c, 53000 Gospić,lstarcevic789@gmail.com

Sažetak rada (opseg do 300 riječi)

Programski okvir Spring predstavlja modularan i moderna pristup razvijanju poslužiteljskih aplikacija. Pristup podatcima se odvija implementacijom JDBC i JPA specifikacija, te se sigurnosna arhitektura zasniva na implementaciji *FilterChain*-a. PostgreSQL je relacijski sustav upravljanja bazom podataka, kojeg aplikacija studijskog slučaja koristi za pohranu podataka. Promatrana aplikacija se sastoji od značajki stvaranja, ispisivanja, uređivanja i brisanja zadataka, te grupiranja istih u grupe nad kojima se također može izvoditi CRUD operacije.

(71 stranica / 29 slika / 2 tablica / 43 literaturnih navoda / jezik izvornika: hrvatski)

Rad je pohranjen u digitalnom repozitoriju Knjižnice Veleučilišta u Šibeniku

Ključne riječi: Spring, PostgreSQL, JPA, Spring Security, JDBC

Mentor(ica): Marko Pavelić

Rad je prihvaćen za obranu dana:

BASIC DOCUMENTATION CARD

Polytechnic of Šibenik Batchelor/Graduation Thesis

Department of Business informatics

Professional Undergraduate of Business informatics

# CREATING BACKEND APLICATION USING SPRING BOOT FRAMEWORK ON USE CASE OF TODO APPLICATION

LUKA STARČEVIĆ

Budačka ulica 210c, 53000 Gospić, lstarcevic789@gmail.com

Abstract (up to 300 words)

The Spring framework represents a modular and modern approach to developing server applications. Access to data takes place through the implementation of JDBC and JPA specifications, and the security architecture is based on the implementation of *FilterChain*. PostgreSQL is a relational database management system, which the case study application uses to store data. The observed application consists of the features of creating, printing, editing, and deleting tasks, which can also be grouped into groups on which CRUD operations can be performed.

(71 pages / 29 figures / 2 tables / 43 references / original in Croatian language)

Thesis deposited in Polytechnic of Šibenik Library digital repository

Keywords: Spring, PostgreSQL, JPA, Spring Security, JDBC

Supervisor: Marko Pavelić

Paper accepted:

# SADRŽAJ


1. UVOD ... 2
2. PROGRAMSKI OKVIR SPRING ... 3
2.1. Općenito ... 3
2.1.1. Core Container ... 5
2.1.2. Data Access/Integration ... 6
2.1.3. Web ... 6
2.1.4. AOP i Instrumentation ... 6
2.1.5. Test ... 7
2.2. Obrada Web zahtijeva pomoću programskog okvira Spring MVC ... 7
2.3. Pristup podatcima unutar programskog okvira Spring ... 9
2.4. Implementacija sigurnosnog sustava unutar programskog okvira Spring ... 12
3. BAZE PODATAKA ... 16
3.1. Relacijske baze podataka ... 18
3.2. NoSql baze podataka ... 20
3.2.1. Baze podataka Ključ-vrijednost ... 21
3.2.2. Baze podataka temeljene na dokumentima ... 22
3.2.3. Grafne baze podataka ... 22
3.2.4. Baze podataka širokog stupca ... 23
3.3. Baza podataka PostgreSQL ... 23
4. WEB SERVISI ... 27
4.1. Representational state transfer (REST) ... 28
4.1.1. Klijent-Server ... 28
4.1.2. Standardizirano sučelje ... 29
4.1.3. Sistem u slojevima ... 29
4.1.4. Sustav bez stanja ... 29
4.1.5. Kod na zahtjev ... 29
5. ANALIZA POSLUŽITELJSKE APLIKACIJE IZRAĐENE U PROGRAMSKOM OKVIRU SPRING BOOT ... 30
5.1. Pregled arhitekture TODO aplikacije ... 30
5.2. Sigurnosne postavke aplikacije ... 31
5.3. Značajka prijave korisnika ... 36
5.4. Značajka registracije korisnika ... 39
5.5. Značajka dohvaćanja stranice sa zadatcima ... 43
5.5.1. TaskController ... 45
5.5.2. Apstraktna klasa Service ... 46
5.5.3. Implementacija Service klase TaskService ... 48
5.5.4. TaskDataManager ... 49

5.5.5. Serijalizacija i Deserijalizacija Task entiteta ..... 52
5.6. Značajka dodavanja zadatka ..... 56
5.6.1. TaskController ..... 56
5.6.2. ResponseEntity ..... 57
5.6.3. TaskDataManager ..... 58
5.7. Značajka uređivanja zadatka ..... 59
5.7.1. TaskController ..... 60
5.7.2. TaskDataManager ..... 60
5.9. Značajka dohvaćanja pojedinačnog zadatka ..... 64
5.10. Značajka stvaranja grupe zadataka ..... 65
5.11. Značajka uređivanja grupe ..... 67
5.12. Značajka dohvaćanja svih grupa zadataka ..... 68
5.13. Značajka brisanja grupe zadataka ..... 69
5.14. Analiza baze podataka ..... 70
6. ZAKLJUČAK ..... 72
LITERATURA ..... 73
PRILOZI ..... 77

# 1. UVOD

Potreba za napredniji informatičkim sustavima raste iz dana u dan sve učestalijim korištenjem internetskih tehnologija. Možemo zaključiti kako poslužiteljske aplikacije obavljaju ključnu ulogu u svakodnevnom radu informacijskih sustava baziranih na Internetu. Kroz ovaj rad će se istražiti tehnologije i načela koje se koriste prilikom razvoja poslužiteljskih aplikacija. Tehnologije odabrane tehnologije su: programski okvir Spring, unutar kojeg je napisana logika aplikacije, te implementirane sigurnosne značajke; zatim za pohranu podataka je odbaran relacijski pristup, te baza podataka PostgreSQL pošto se savršeno uklapa s tipovima podataka koji se spremaju u bazu. Naposljetku se koristi REST (engl. Representative State Transfer) pristup arhitekturi web servisa. Rad će započeti analizom tehnologija koje se koriste, te pri tome će se osobita pažnja posvetiti programskom okviru Spring, s obzirom da je on ključni element praktičnog djela rada. Nakon opisa tehnologija slijedi analiza izrade Web poslužiteljske aplikacije. Također je važno naglasiti kako će se na brzinu proći kroz određene dijelove funkcionalnosti aplikacije, s obzirom na to da se na više mjesta ponavlja ista logika samo s drugim podatcima. Na kraju svega će se donijeti neki zaključak.

# 2. PROGRAMSKI OKVIR SPRING

## 2.1. Općenito

Programski okvir Spring je alat namijenjen razvijanu poslužiteljskih aplikacija koristeći programski jezik Java. Razvojnim programerima omogućuje korištenje raznih značajki među kojima su: Oblikovni obrazac *Inversion Of Control* (IoC), korištenje POJO (*engl. Plain Old Java Objects*)[1], primjenu *Aspect Oriented Programming* (AOP) pristupa, korištenje JDBC (*engl. Java Database Connectivity*) sloja apstrakcije, te implementaciju raznih ORM (*engl. Object-Relation Mapping*) rješenja (Overview of Spring Framework, n.d.). Njegov razvitak počeo je 2003. godine s ciljem olakšanja razvoja poslovnih aplikacija (*engl. Enterprise level applications*) u okruženju programskog jezika Java koje su se do tada razvijale implementirajući *Java Enterprise Edition* (JEE) programska sučelja (*engl. Aplication programming interface*) i razvojne konvencije. Najveći nedostatci razvijanja aplikacija u JEE okruženu su višak koda uslijed konfiguriranja svih JEE značajki (čak i onih koje se nisu koristile), prekomjerna kompleksnost koda uzrokovanog oslanjanjem na XML (*engl. Extended Markup Language*) datoteke prilikom konfiguracije paketa, te krut pristup baratanja paketima[2] (A Brief History of Spring & Spring Boot, n.d.). Godine 2013. predstavljen je modul programskog okvira Spring, Spring Boot koji je predstavio daljnja značajnija olakšanja razvoja poslužiteljskih Web aplikacija apstrahirajući konfiguracije servera, putem već ugrađenog poslužitelja Apache Tomcat, dodatna poboljšanja i razlike u prikazane u Tablici 1.

[1] Korištenje POJO objekata je značajno iz razloga što apstrahira objekte koda domenske razine od samog programskog okvira

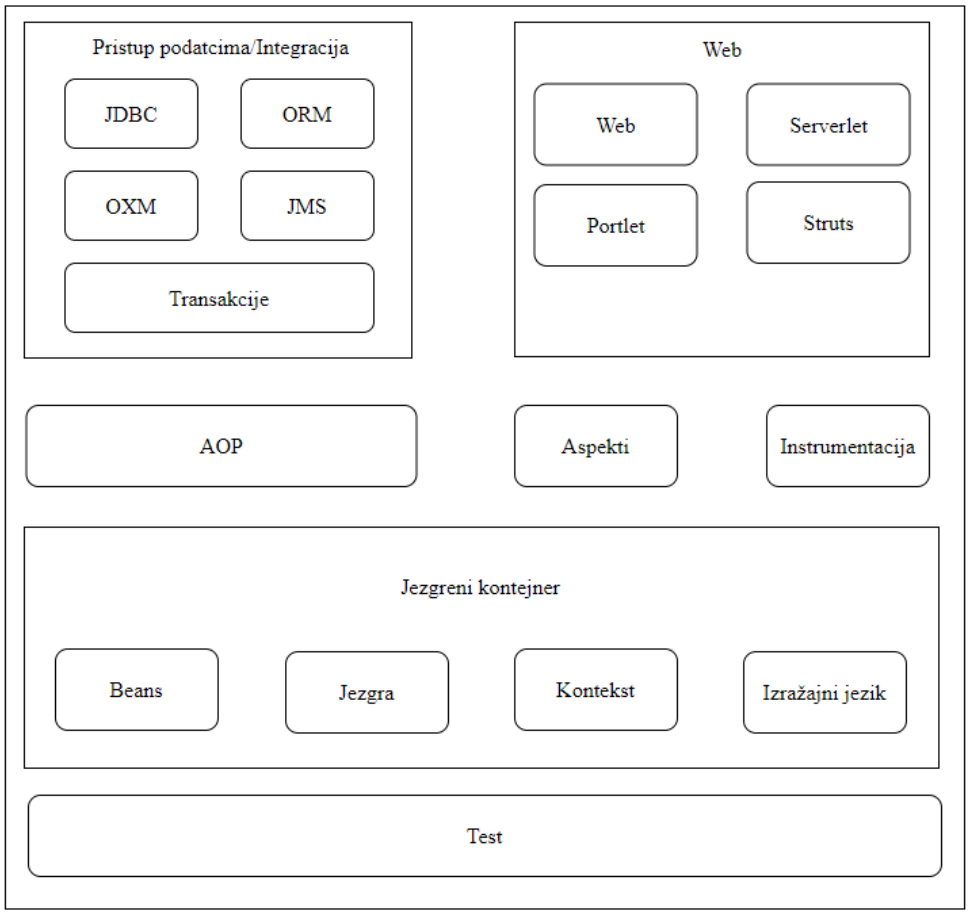
[2] Razvojni programeri su morali tvrdo kodirati sve programske ovisnosti koje je pojedina komponenta koristila

Tablica 1. Usporedba značajki sadržanih u programskom okviru Spring te onih nadodanih u Spring Boot modulu

| **Spring** | **Spring Boot** |
|---|---|
| Cilj mu je pojednostavit razvoj Aplikacija u JEE okruženju | Smanjuje količinu potrebnog koda, te pruža olakšan pristup razvijanju Web aplikacija |
| Glavna značajka programskog okvira Spring je ubacivanje ovisnosti | Glavna značajka Spring Boot-a je Auto Konfiguracija |
| Pojednostavljuje stvari omogućavanjem Razvijanja slabo vezanih aplikacija | Olakšava stvaranje samostojećih aplikacija S manje konfiguracije |
| Razvoji programeri pišu višak koda za Implementaciju minimalnih zadataka | Smanjuje smanjuje repetirajući kod |
| U svrhe pisanja testova potrebno je Postaviti server | Spring Boot pruža ugrađene servere kao što Su *Jetty* i *Tomcat* |
| Ne pruža podršku za *in-memory* baze podataka | Pruža nekolicinu dodataka za rad s *in-memory* Bazama podataka |
| Razvojni programeri ručno definiraju Programske ovisnosti za Spring projekte u Datoteci pom.xml | Spring Boot sadrži implementirani interni Koncept pokretača koji obavlja preuzimanje JAR datoteka programskih ovisnosti deklariranih u datoteci pom.xml |

*Izvor*: https://www.javatpoint.com/spring-vs-spring-boot-vs-spring-mvc

Programski okvir Spring je podijeljen u 20 modula, koji su grupirani u: *Core Container, Data Access/Integration, Web, AOP, Instrumentaton* i *Test,* kao što je prikazano na Slika 1.




Slika 1. Ilustrirani prikaz arhitekture programskog okvira Spring, Izvor: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

### 2.1.1. Core Container

Core Container sadrži slijedeće komponente:

- *Core* – jezgra programskog okvira koja mnoge temeljne značajke kao što su inverzija kontrole i ubacivanje ovisnosti.
- *Beans* – *Bean* predstavlja instancu objekta koji je instanciran, sastavljen, i upravljan od strane Spring IoC kontejnera (Introduction to the Spring IoC Container and Beans:Spring Framework)
- *Context* – dodaje potporu za internacionalizaciju ,proslijeđivanje događaja (*engl. event propagation*) učitavanje resursa i transparentno stvaranje konteksta
- *Expression Language* – omogućava slanje upita i manipulaciju sa dijagramom objekata tokom izvođenja

### 2.1.2. Data Access/Integration

Data Access/Integration sadrži apstrakciju za pristup podatcima baza podataka, te ste sastoji od:

- *JDBC (Java Database Connectivity)* – Apstrahira procese povezivanja i upravljanja transakcijama prema bazama podataka raznih proizvođača.
- *Modul za objektno relacijsko mapiranje(engl. Object Relation Mapping module)* – pruža integracijske slojeve za implementacije ORM-a kao što su: JPA, JDO, Hiberante i iBatis.
- *Modul za mapiranje XML datoteka na objekte (engl. Object XML Mapper)*– omogućuje implementaciju rješenja za mapiranje XML- a na objekte. Neki od spomenutih rješenja uključuju: JAXB, *Castor,* XMLBeans, JiBX i XStream
- *Servis za obavijesti programskog jezika Java (engl. Java Messaging Service)*– pruža modul za stvaranje i obradu poruka unutar Spring platforme
- *Transakcije (engl. Transaction)*– modul koji je zadužen za deklarativno i programatsko upravljanje transakcijama za klase koje implementiraju definirana sučelja ili sve POJO-e domenske razine.

### 2.1.3. Web

Sastavnice *Web* komponente su:

- *Web modul (engl. Web module)* – sadrži osnovnu integraciju web-orijentiranih značajki kao što su učitavanje više datoteka, inicijalizaciju IoC kontejner upotrebom serverlet slušača i web-orijentiranog aplikacijskog konteksta.
- *Web-Serverlet* – modul u kojem je sadržana implementacija *Model-View-Controller (*MVC*)* arhitekture programskog okvira Spring za Web aplikacije. MVC arhitektura jasno odvaja programski kod domenske razine od web formi.
- *Web-Struts* – pruža potporu za integraciju klasičnog *Struts* web sloja sa aplikacijom programskog okvira Spring.

### 2.1.4. AOP i Instrumentation

Modul *AOP* programskog okvira Spring puža programsku implementaciju usklađenosti s aspektom savezništva između pristupa razvijanju web aplikacije, omogućavajući razdvajanje koda putem metodnih presretača (*engl. method interceptors*) i radnih točki (*engl. pointcuts*). *Instrumentation* modul daje podršku implementaciji klase i klasnih učitavača koji će se koristi

na nekim poslužiteljima (Introduction to Spring Framework, n.d.).

### 2.1.5. Test

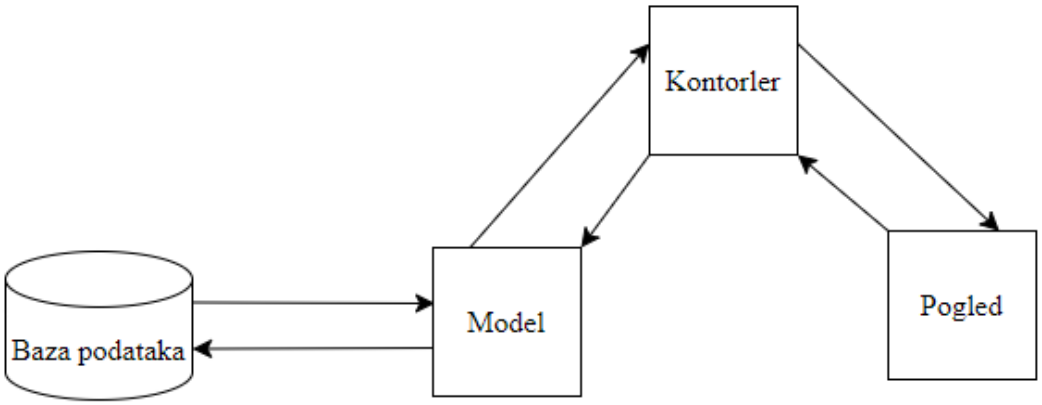
Modul Test daje podršku testiranju komponenti programskog okvira Spring pomoću JUnit ili TestNG paketa, također omogućuje stvaranje komponenti objekta dvojnika (*engl. Mock*) s ciljem postizanja izolacije dijela koda koji se testira (Introduction to Spring Framework, n.d.).

## 2.2. Obrada Web zahtijeva pomoću programskog okvira Spring MVC

Programski okvir Spring Model View Controller (*Spring MVC*) predstavlja rješenje za implementaciju Model,pogled i kontroler (*engl. Model-View-Controller*) razvojne paradigme unutar programskog okvira Spring. Razvojna paradigma MVC nalaže razdvajanje pojedinih funkcionalnosti unutar tri kategorije:

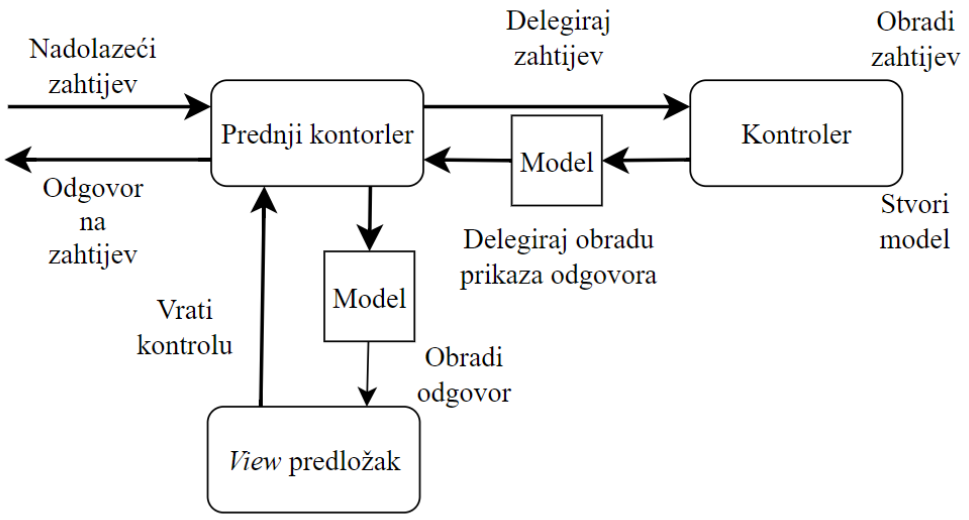
- Model *(engl. Model)* – sadrži logiku baratanja podatcima, uključuje metode za čitanje i pisanje iz baze podataka. Na zahtijev *Controller*-a vraća model s traženim podatcima s kojim se onda dalje manipulira (MVC Framework Introduction, n.d.).
- Pogled *(engl. View)* – stvara korisničko sučelje na temelju modela kojeg je dobio od *Controller*-a. (MVC Framework Introduction, n.d.)
- Kontroler *(engl. Controller)*– obavlja funkciju primanja i obrade zahtjeva na temelju kojih od *Model*-a traži podatke koje zatim prosljeđuje *View*-u na temelju kojih on sastavlja korisničko sučelje, koje je zatim vraćeno korisniku.

Kao što je schematski prikazano na Slika 2.




Slika 2. Grafički prikaz razvojne paradigme MVC

Programski okvir Spring MVC zasniva se na *Front Controller* paradigmi. *DispacherServerlet* predstavlja implementaciju spomenute paradigme unutar programskog okvira Spring MVC. *DispacherServerlet* je zadužen za primanje i slanje HTTP zahtjeva. Deklaracija *DispacherServerlet*-a zajedno s pridruživanjem URL ruta se obavlja unutar *web.xml* datoteke. Jednom kada primi HTTP zahtjev *DispacherServerlet* provjerava svoju konfiguracijsku rutu za URL uzorak pomoću kojega određuje na koji će *Controller* proslijediti zahtjev. Programski okvir Spring omogućuje jednostavno deklariranje klasa *Controller* pomoću bilješke `@Controller` koja ujedino označava klasu za ubacivanje ovisnica. Kako bi *DispacherServerlet* znao za mapiranu *Uniform resource location* (URL) putanju sa deklariranim *Controller*-om, potrebno je označiti klasu i metode unutar nje s anotacijom `@RequestMapping`. Jednom kada je zahtjev delegiran odgovarajućoj *Controller* klasi ona obavlja definiranu logiku i ako je potrebno delegira zadatke *Model* komponenti koja obavlja manipulaciju s podatcima te vraća *Controller*-u model. *Controller* zatim pridodaje ime *View*-a na kojeg bi model trebao biti mapiran, bez datotečnih nastavaka te ga delegira *Front Controller*-u koji ga prosljeđuje sučelju *ViewResolver* (15. Web MVC framework, n.d.). *ViewResolver* sučelje mapira dobiveni model i *View*, dodavajući odgovarajuću putanju i nastavak. *View* sučelje je zaduženo za pripremanje i prosljeđivanje dobivenog modela na obradu pomoću tehnologija kao što su: *Java Server Page (*JSP*)* i *Velocity* predlošci, prikazano na Slika 3 (15. Web MVC framework, n.d.).




Slika 3.Tok obrade zahtjeva unutar programskog okvira Spring MVC, Izvor:https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

### 2.3. Pristup podatcima unutar programskog okvira Spring

Kada se govori o pristupu podatcima unutar programskog okvira Spring ili općenito bilo kojeg drugog programskog okvira koji komunicira s bilo kojim izvorom podataka, važno je definirati pojam Transakcije. Transakcija predstavlja slijed radnji koje se tretiraju kao jedna cjelina, to znači da u slučaju neizvršavanja jedne radnije, cijela transakcija se nebi trebala izvesti. Koncept transakcije se može predstaviti i kroz pokratu ACID koja predstavlja:

- Atomičnost (*engl. Atomicty*)– transakcija se tretira kao jedna cjelina, odnosno u slučaju ne izvršavanja jedne radnje unutar dane transakcije, cijela transakcija se proglašava neuspjelom.
- Konzistentnost (*engl. Consistency*)– konzistentnost integriteta baze podataka, npr. jedinstveni primarni ključevi u tablicama.
- Izolacija (*engl. Isolation*) – svaka transakcija se tretira zasebno, odvojeno od drugih kako bi se izbjegla korupcija podataka.
- Izdržljivost (*engl. Durability*)– kad se transakcija izvrši, rezultati bi trebali biti trajni odnosno zaštićeni od uklanjanja iz baze podataka uslijed pada sustava (Spring - Transaction Management: tutorialspoint, n.d.).

Implementacija transakcija unutar programskog okvira Spring se zasniva na transakcijskoj strategiji koja definira sučelje `TransactionManager` odnosno pakete

`org.springframework.transaction.PlatformTransactionManager` za imperativno upravljanje transakcijama i `org.springframework.transaction.ReactiveTransactionManager` za reaktivno upravljanje transakcijama. Navedena sučelja se tretiraju kao bilo koji drugi Java *bean*-a te su podložna *Dependency Injection-u.* Unutar programskog okvira Spring postoje dva načina upravljanja transakcijama:

- Deklarativno upravljanje transakcijama – pristup upravljanju transakcijama u kojem se transakcijama upravlja pomoću konfiguracije u vanjskim XML datotekama, što rezultira u odvajanju upravljanja transakcijama od poslovne logike koda.
- Programatsko upravljanje transakcijama – upravljanje se izvršava pomoću programskog koda. Ovaj pristup daje veći stupanj fleksibilnosti, no predstavlja izazov za održavanje.

### 2.3.1. Pristup bazi podataka pomoću JDBC standarda

*Java Database Connectivity* API (JDBC) predstavlja JavaSoft specifikaciju programskog sučelja u kojem su definirana sučelja i klase koje aplikacije pisane u programskom jeziku Java koriste za pristup *Database Managment System*-u (DBMS). Analogno tome JDBC specifikacija omogućuje razvojnim inženjerima spajanje na bazu podataka, pisanje upita na bazu podataka korištenjem programskog jezika *Structured Quey Language* (SQL), te obradu dobivenih rezultata. Uvjet spajanja na bazu podataka je postojanje odgovarajućeg upravljačkog programa baze podataka (Java Persistence API (JPA):IMB, n.d.). Programski okvir Spring sadrži svoju implementaciju JDBC specifikacije te tako dodatno pridonosi pouzdanosti i lakoći održavanja napisanog programskog koda aplikacije. Detaljnija podjela zaduženja između programskog okvira Spring i razvojnog inženjera je prikazana u Tablica 2 (Data Access with JDBC, n.d.).

Tablica 2. Prikaz podjele ovlaštenja između razvojnog inženjera i programskog okvira Spring

| Radnja | Programski okvir Spring | Razvojni inženjer |
|---|---|---|
| Definiranje parametara veze | | X |
| Otvaranje veze s bazom podataka | X | |
| Definiranje SQL upita | | X |
| Deklariranje parametara i prosljeđivanje vrijednosti parametara | | X |
| Pripremanje i pokretanje upita | X | |
| Postavljanje petlje i prolazak kroz rezultate | X | |
| Manipuliranje s pojedinim rezultatom | | X |
| Obrada iznimki | X | |
| Upravljanje s transakcijama | X | |
| Zatvaranje veze, upita i seta rezultata | X | |

Izvor*: https://docs.spring.io/spring-framework/reference/data-access/jdbc.html*

Unutar implementacije specifikacije JDBC programskog okvira Spring ponuđeno je nekoliko pristupa podatcima kroz slijedeća sučelja:

1. `JdbcTemplate` – uz to što je najpopularniji pristup, predstavlja i pristup najniže razine kojeg koriste svi ostali pristupi.
2. `NamedParameterJdbcTemplate` – omotava se oko `JdbcTemplate` -a, te zamjenjuje tradicionalni JDBC „?“ držač rezerviranog mjesa (*engl. Placeholder*) sa imenovanim parametrima. Ovaj pristup pruža bolju dokumentiranost u slučaju kad postoji više parametara SQL upita.
3. `SimpleJdbcInsert` i `SimpleJdbcCall` optimiziraju meta podatke baze podataka kako bi smanjili količinu potrebne konfiguracije.
4. RDBMS objekti – uključujući `MappingSqlQuery`, `SqlUpdate`,

`StoredProcedure` – zahtijevaju stvaranje objekata koji su ponovno upotrebljivi te sigurni po dretve tokom inicijalizacije sloja za pristup podatcima

Uz sve navedeno programski okvir Spring sadrži implementaciju *Java Persistance Api-a* (JPA), kao i za implementaciju drugih *Object-Relation Mapping* tehnologija. JPA predstavlja skup specifikacija kojim se standardizira Object Relationship Mapping (ORM) unutar ekosustava programskog jezika Java. JPA standardizira preslikavanje objekata na jednu ili više tablica korištenjem XML konfiguracijske datoteke ili korištenjem bilješki (Java Persistence API (JPA): IBM, 2023).

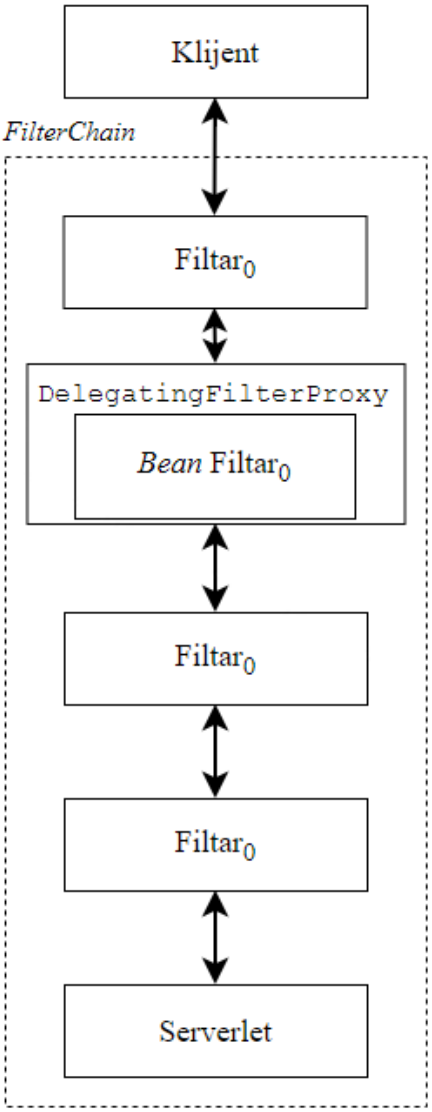
### 2.4. Implementacija sigurnosnog sustava unutar programskog okvira Spring

Implementacija sigurnosnog sustava aplikacija rađenih unutar programskog okvira Spring se zasniva na implementaciji Serverlet filtara. Serverlet filtri su implementirani kroz *FilterChain* koji se zauzvrat sastoji of implementacija `Filter` klase, koje stoje između klijenta i *Serverlet*-a koji u slučaju aplikacija rađenih u programskom okviru Spring MVC predstavlja klasu `DispacherServerlet` koja je spomenuta u prethodnom poglavlju ovoga rada. Prilikom zaprimanja zahtijeva *Container* stvara *FilterChain* te pokreće *Serverlet.* Zaprimljeni zahtjev zatim prolazi kroz stvoreni *FilterChain* te dolazi do *Serverlet*-a na daljnju obradu, pri tome stvorene instance Filter klase obavljaju slijedeće zadatke:

- Sprječavanje pozivanja implementacija klase Filter ili sprječavanje pozivanja *Serverlet*-a koji se nalaze hijerarhijski ispod dane implementacije Filter klase, u kojem slučaju dana implementacija može odgovoriti na HTTP zahtjev.
- Modificiranje objekata koji sadrže HTTP odgovor ili HTTP zahtjev, te prosljeđivanje istih niz *FilterChain*

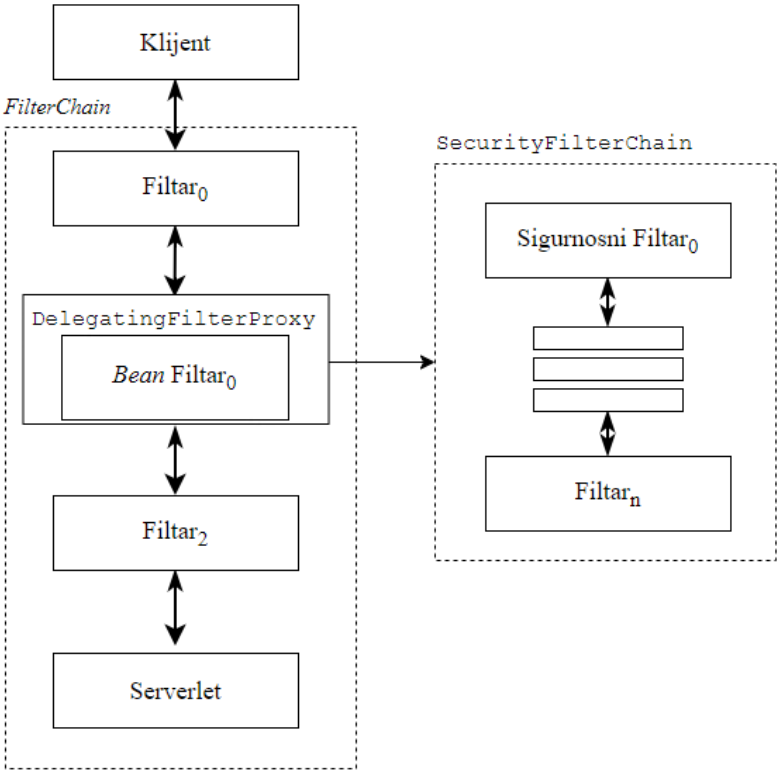
S obzirom da *FilterChain* spada pod životni ciklus i nadležnost odgovarajućeg *Container*-a, potreban je način modificiranja ponašanja stvorenog *FilterChain*-a. Odgovor na to dolazi u obliku implementacije `DelegatingFilterProxy`, ona služi kao svojevrsna premosnica između *Container*-a u kojem se nalazi *FilterChain*, te *AplicationContext*-a programskog okvira Spring. `DelegatingFilterProxy` omogućuje razvojnom programeru definiranje vlastitih implementacija `Filter` klase, odnosno definiranje nove logike filtriranja HTTP zahtijeva primjenom postojećih ili novih implementacija `Filter` klase kao što je prikazano na *Slika 4*, gdje „*Bean Filter"* predstavlja Spring *bean* koji je podložan upravljanju odnosno

stvaranju od samog programskog okvira Spring. Nadalje zbog automatizacije obavljanja ključnih zadataka unutar *Spring Security Context*-a važno je izbjegavati direktno umetanje vlastitih implementacija kase `Filter`, pošto se tako stvarju nepotrebni rizici kao što je npr. rizik od curenja memorije. Taj problem upravljanja *Security Context*-om rješava `FilterChainProxy`, koji predstavlja svojevrsno ishodište za podršku *SecurityServerlet* programskog okvira Spring, te obavlja radnje kao što su upravljanje *Spring Security Context*-om, te podizanja HTTP vatrozida protiv određenih oblika Web napada (Architecture: Spring Security Documentation, n.d.).



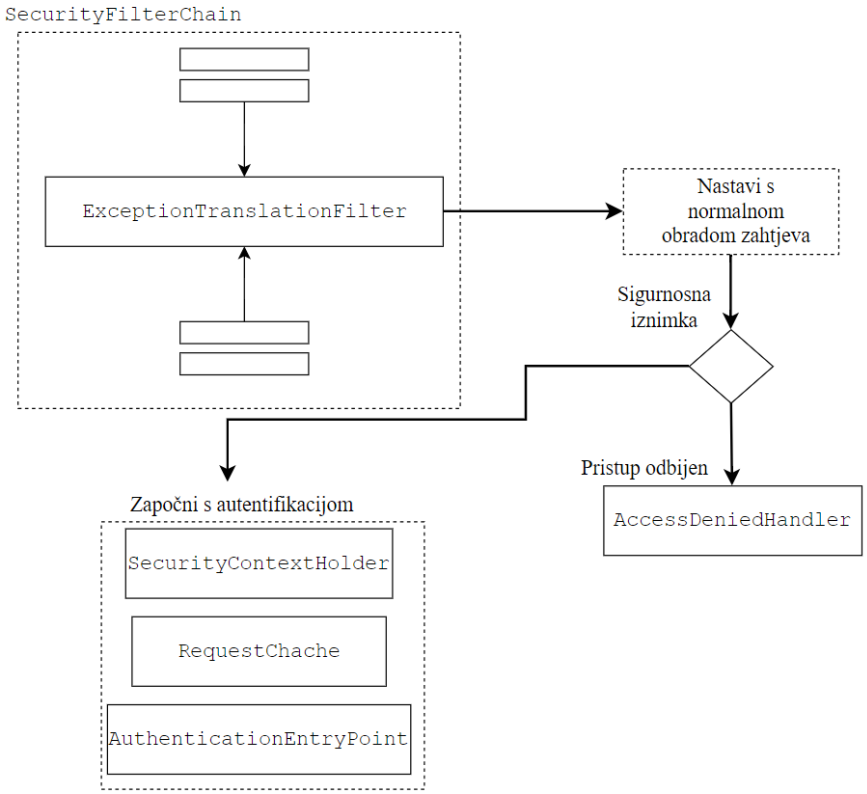

*Slika 4*. Prikaz *FilterChain*-a s umetnutom DelegatingFilterProxy implementacijom klase Filter, Izvor:https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters

Uz navedeno `FilterChainProxy` također pruža mapiranje između različitih instanci `SecurityFilterChain` klase na temelju definiranih URL ruta. Pomoću spomenutog mapiranja, razvojni programer može određivati vlastita sigurnosna pravila za pojedine *endpoint*-e . `SecurityFilterChain` je implementacija `Filter` klase koja u sebi sadrži logiku koju je definirao razvojni programer koristeći nove ili postojeće objekte `SecurityFilter` klase, kao što je prikazano na *Slika 5* (Architecture: Spring Security Documentation, n.d.).




*Slika 5*. Prikaz položaja klase `FilterChainProxy` i `SecurityFilterChain` u kontekstu *FilterChain* stabla, Izvor: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters

*SecurityFilterChain* programskog okvira Spring reagira na iznimke u procesu filtriranja kroz objekt `ExceptionTranslationFilter` koji se umeće u `FilterChainProxy`. Metoda rada je prikazana na Slika 6.




Slika 6. Prikaz interakcije klase ExceptionTranslationFilter sa ostalim komponentama, Izvor: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters

Redoslijed je slijedeći (Architecture: Spring Security Documentation, n.d.):

1. `ExceptionTranslationFilter` poziva „`FilterChain.doFilter(request, response)`", te na taj način učitava ostatak aplikacije
2. Ako korisnik nije autentificiran ili je bačen `AuthenticationException`, tada se započinje proces autentifikacije
    - `SecurityContextHolder` se čisti
    - `HttpServletRequest` se sprema tako da se može koristiti za ponovno reproduciranje izvornog zahtjeva ako provjera autentičnosti bude uspješna.
    - `AuthenticationEntryPoint` koristi se za traženje podataka za prijavu od klijenta.
3. U slučaju odbijanja pristupa, zove se `AccessDeniedHandler`, da obavi logiku koja je definirana u slučaju odbijanja zahtijeva.

# 3. BAZE PODATAKA

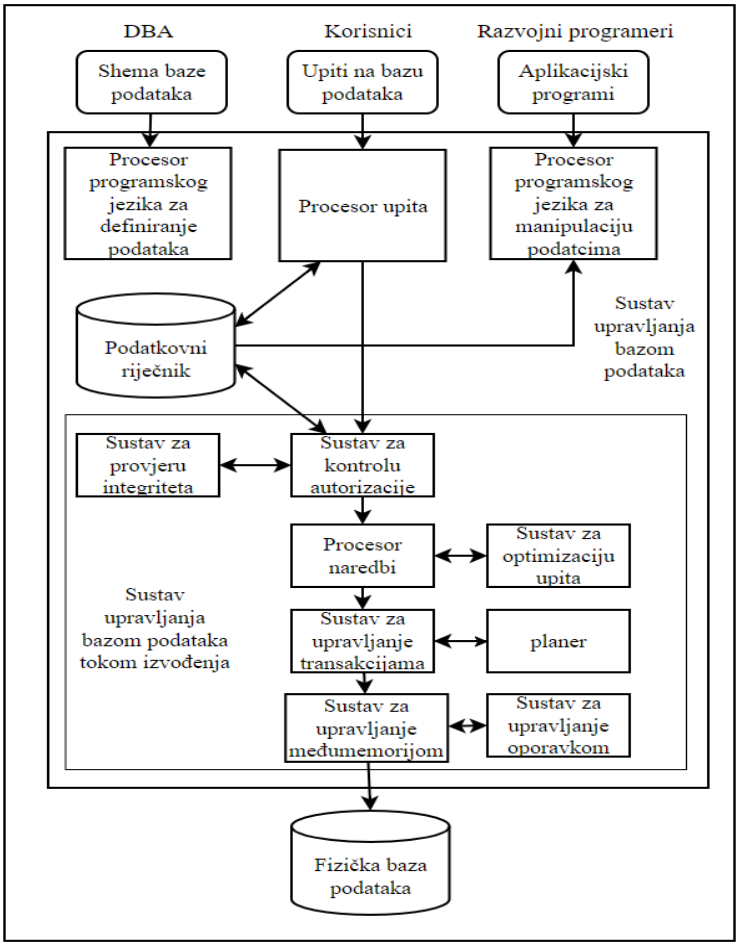
Baze podataka možemo definirati kao strukturiran i organiziran skup podataka, tipično pohranjenih u elektronskom obliku na računalnom sustavu (What Is a Database?: Oracle). Razvoj baza podataka u elektroničkom obliku se može pratiti kroz slijedeće stadije (The Evolution of Database Management System, 2018):

1. Flat files – (Razdoblje od 1960. do 1980), predstavlja strukturu podataka u kojoj se podatci pohranjuju u jednu datoteku ili tablicu. Pohranjeni podatci se spremaju u tekstualnu datoteku gdje zapisi imaju fiksnu duljinu te su odvojeni zarezima, razmacima, tabularima ili drugim sličnim znakovima. Također ne postoje relacije između unesenih podataka.
2. Hijerarhijska baza podataka – (Razdoblje od 1970. do 1990.), Hijerarhijska baza podataka sadrži podatke u hijerarhijskoj strukturi, gdje jedan roditelj može imati više djece, dok jedno dijete može imati samo jednog roditelja. Tip entiteta definira tablica, dok redovi predstavljaju zapise, dok stupci predstavljaju atribute.
3. Mrežna baza podataka – (Razdoblje od 1970. do 1990.), za razliku od hijerarhijske baze podataka, Mrežna baza podataka omogućava više roditelj-dijete veza, te poprima strukturu grafa.
4. Relacijska baza podataka – (Razdoblje od 1980. do danas), relacijski model predstavljen od strane E.F. Codda, pružio je način pridruživanja više tablica uz pomoć relacija koje tvore Primarni i Strani ključevi.
5. Objektno-orijentirana baza podataka – (Razdoblje od 1990 do danas), predstavlja vrstu baze podataka u kojoj se informacije prikazuju kao objekti. Za razliku od relacijskih baza podataka, rade u okviru programskih jezika poput Jave ili C++
6. Objektno-relacijska baza podataka – (Razdoblje od 1990. do danas), prikazuje modificirani objektno orijentirani korisnički prikaz preko već implementiranog sustava upravljanja relacijskom bazom podataka. Sustav upravljanja objektno-relacijskom bazom podataka prevodi podatke u tablice, raspoređene u retke i stupce, a od tada pa nadalje upravlja podacima na isti način kao što se to radi u sustavu relacijskih baza podataka. Na taj način pruža relacijske odnose između entiteta, te dodaje značajke prepisivanja funkcija, te nasljeđivanja tablica.

Sustav upravljanja bazom podataka (*engl. Database Managment System, DBMS*) je softver pomoću kojega krajnji korisnik može kreirati, te upravljati bazama podatka. DBMS pruža

funkcionalnosti čitanja, brisanja, ažuriranja te zaštite podataka sadržanih u bazi podatka kojom upravlja, te se sastoji od slijedećih komponenti (Mullins, n.d.) (Thakur, n.d.):

- Procesor upita*(engl. Quey processor):* obrađuje dobiveni upit u seriju naredbi koje se mogu poslati Sustavu za obradu podataka (*engl. Data manager*) na izvršavanje.
- Upravitelj baze podataka tokom izvođenja *(engl. Runtime Database Manager)* softver kojem je zadaća upravljanje bazom podataka tokom izvođenja programa. Tokom zaprimanja korisničkog upita, pregledava ga te određuje koji su konceptualni zapisi potrebni kako bi se izvršio korisnički upit.
- Sustav upravljanja podatcima *(engl. Data manager)* odgovoran je za upravljane podatcima unutar baze podataka, te pruža sustav za oporavak podataka nakon neke greške. Uključuje Sustav za upravljanje oporavkom (*engl. Recovery manager*) i Sustav za upravljanje međumemorijom (*engl. Buffer manager*), kom je zadaća prijenos podataka između memorije i sekundarne pohrane.
-




*Slika 7*. Prikaz komponenti Sustava za upravljanje bazom podataka, Izvor: https://ecomputernotes.com/fundamental/what-is-a-database/components-of-dbms

Neka od modernih DMBS rješenja su (Mullins, n.d.):

- *Relacijski sustav upravljanja bazom podataka (engl. Relational Database Managment System)* - Predstavlja podatke kao redove u tablici sa fiksnom shemom, te odnosima u tablici definiranim pomoću ključnih stupaca, detaljni opis je naveden u idućem potpoglavlju.
- *NoSql Sustav upravljanja bazom podataka (engl. NoSql DBMS)* – sustav upravljanja bazom podataka namijenjen upravljanju sa labavo povezanim podatcima, kao što su podatci unutar: Dokumentnih baza podataka (*engl. Document database*), Grafnih baza podataka (*engl. Graph database*), Baza podataka ključ-vrijednost (*engl. Key-value database*), Široko stupčana baza podataka (*engl. Wide-column database*)
- *NewSql Sustav upravljanja bazom podataka (engl. NewSql DBMS)* su moderni relacijski sustavi koji koriste SQL, nude isti skalabilni pristup kao i NoSql sustavi, no podložni su ACID principu. Zamišljeni su prema relacijskom modelu podatka, no s distribuiranom arhitekturom.
- *Stupčasti sustav upravljanja bazom podataka (engl. Column-oriented DBMS)* - pohranjuje podatke u tablice s naglaskom na stupcima umjesto na redcima, što pridonosi učinkovitijem pristupu podatcima kad se traži samo podset podataka iz stupca. Pogodan je za primjenu u Skladištima podataka (*engl. Data warehouse*) koji imaju veći broj sličnih podataka.
- *Sustav upravljanja bazom podataka unutar memorije (engl. In-memory DBMS)* oslanja se na glavnu memoriju za pohranu i manipulaciju podatcima. Pruža visoke performanse što se tiče čitanja i pisanja podataka, no zauzvrat troši daleko više resursa. Bilo koji DBMS se može postaviti kao in-memory sustav.
- *Višemodelni sustav upravljanja bazom podataka (Multimodel DBMS)* sustav koji podržava više od jednog modela baze podataka. Korisnici mogu odabrati model koji je najprikladniji za njihove zahtjeve bez potrebe za prebacivanjem na drugi DBMS.
- *Sustav upravljanja bazom podataka u oblaku (engl. Cloud DBMS)* namijenjen upravljanju i pristupanju podatcima u oblaku.

### 3.1. Relacijske baze podataka

Relacijski model upravljanja bazom podataka predstavlja logički pristup predstavljanju

podataka pohranjenih u bazu podataka u kojem su podatci organizirani u zbirku dvodimenzionalnih stupaca i redaka, gdje stupac predstavlja atribute entiteta, a redovi predstavljaju zapise. Relacijska baza podataka je baza koja implementira relacijski pristup strukturiranja podataka, te se sastoji od sljedećih relacijskih komponenti (Gulati, 2022):

- Relacija: dvodimenzionalna tablica koja se koristi za pohranu zbirke podatkovnih elemenata.
- Tuple – red relacije, odnosno red unutar relacijske tablice koji sadrži informacije entiteta
- Atribut/polje : stupac relacije, koji prikazuje svojstva koja definiraju relaciju.
- Domena atributa : skup unaprijed definiranih atomskih vrijednosti koje atribut može poprimiti, tj. opisuje dopuštene vrijednosti koje atribut može poprimiti.
- Stupanj: to je ukupan broj atributa prisutnih u relaciji.
- Kardinalnost: Određuje broj entiteta uključenih u relaciju, tj. to je ukupan broj redaka prisutnih u relaciji.
- Relacijska shema: to je logičan nacrt relacije, tj. opisuje dizajn i strukturu relacije. Sadrži naziv tablice, njene atribute i njihove vrste:
- Relacijska instanca: to je zbirka zapisa prisutnih u relaciji u određenom trenutku.
- Ključ relacije: To je atribut ili grupa atributa koji se mogu koristiti za jedinstvenu identifikaciju entiteta u tablici ili za određivanje odnosa između dvije tablice. Relacijski ključevi mogu biti 6 različitih vrsta:
    - Ključ kandidata
    - Super ključ
    - Kompozitni ključ
    - Primarni ključ
    - Alternativni ključ
    - Strani ključ

Kako bi se osigurao integritet podataka, relacijski model koristi niz ograničenja, odnosno ograničenja relacijskog integriteta. Ograničenja relacijskog integriteta predstavljaju set pravila koja se provjetravaj prilikom izvođenja svakog uputa na bazu podataka a ona predstavljaju (Gulati, 2022) :

- Ograničenje domene: nalaže da svaki atribut mora imati vrijednost koja se nalazi unutar određenog raspona vrijednosti.
- Ograničenje ključa: svaka relacija mora sadržavati atribut koji će služiti svrsi primarnog

ključa. Primarni ključ je jedinstveni identifikator koji identificira pojedini red tablice odnosno unos, te se preko njega stvara relacija. Kad se nalazi izvan tablice u kojoj je definiran te obavlja funkciju stvaranja relacije, tad se naziva Stranim ključem. Samim time vrijednost primarnog ključa ne smije biti prazna odnosno *NULL*.

- Ograničenje referentnog identiteta – Kada je ostvarena relacija između dvije tablice pomoću Primarnog odnosno Stranog ključa, dana referenca mora postojati na danoj relaciji.

Upiti na relacijsku bazu podataka se obavljaju uz pomoć *Structured Query Language*-a (SQL). Uz slanje upita SQL se još može koristiti za održavanje i optimiziranje performansi baze podataka. Implementacija programskog jezika SQL unutar relacijskih sustava za upravljanje bazama podataka uključuje poslužitelja koji obrađuje SQL upit, te vraća rezultate. Integracija programskog jezika SQL je propisana kroz dvije organizacije za standardizaciju, Američki nacionalni institut za standarde (ANSI), te Međunarodnu organizaciju za standarde (ISO). Do sad niti jedan RDBMS nije usvojio standarde SQL-a u cijelosti. Kako bi upit pisan programskim jezikom SQL bio čitljiv bazi podataka, on mora proći kroz Pretvarač (*engl. Parser*) koji zamjenjuje određene ključne riječi SQL upita s tokenima odnosno simbolima. Tokom ovog procesa provjerava se ispravnost tj. dali SQL upit zadovoljava semantiku odnosno pravila koja osiguravaju ispravnost danog upita. Osim kroz provjeru ispravnosti, upit mora bi autoriziran, tj. korisnik koji je zadao upit mora imati ovlasti za manipuliranje danim podatcima. Nakon što je provedena provjera ispravnosti, te provjera autorizacije, Relacijski mehanizam (*engl. Relational engine*) zatim stvara plan za dohvaćanje, pisanje ili ažuriranje podataka na najučinkovitiji način, te pretvara kod pisan programskim jezikom SQL u programski kod izražen u bajtovima. Nadalje, Mehanizam za pohranu (*engl. Storage engine*) čita dobiveni kod izražen u bajtovima, te na temelju njega čita i pohranjuje u datoteke pohranjene na disku. Nakon obavljene radnje čitanja ili pisanja, Mehanizam za pohranu vraća odgovor korisniku (What Is SQL (Structured Query Language)?, n.d.).

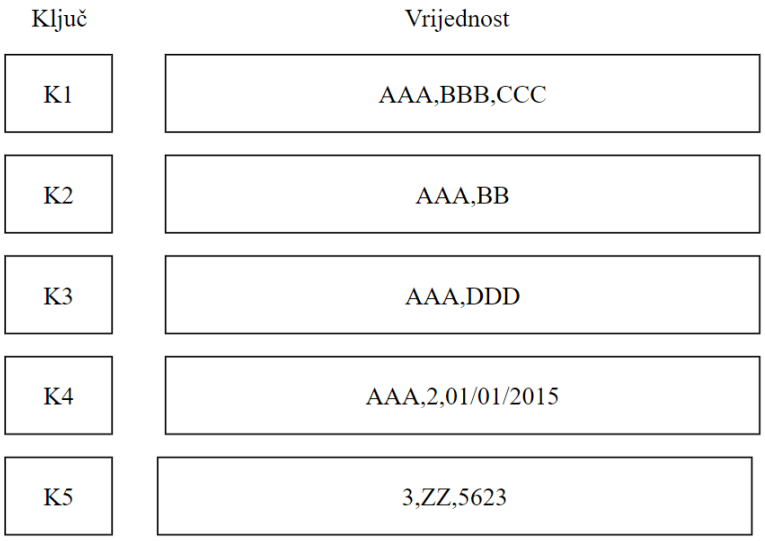
### 3.2. NoSql baze podataka

NoSql pristup dizajnu baze podataka omogućuje pohranu podataka izvan tradicionalne, odnosno relacijske strukture podataka. Umjesto tablica, NoSql baze podataka pohranjuju informacije unutar jedne strukture podatka kao što je dokument tipa *JavaScript Object*

*Notation* (JSON). Ovaj pristup također omogućuje NoSql bazama podataka da budu horizontalno skalabilne[3], umjesto vertikalno kao što su relacijske baze podatka. Pošto NoSql baze podataka nemaju unaprijed definiranu shemu, koriste se za pohranu velikog broja nestrukturiranih podataka preko više različitih servera. NoSql baze podataka možemo podijeliti na: Baze podataka Ključ-vrijednost (*engl. Key-value database*), Baze podataka temeljene na dokumentima (*engl. Document database*), Grafne baze podataka (*engl. Graph database*), Baze podataka širokog stupca (*engl. Wide-column database*).

### 3.2.1. Baze podataka Ključ-vrijednost

Baza podataka Ključ-vrijednost je baza podatka u kojoj se vrijednosti spremaju par ključa i vrijednosti. Ključ predstavlja jedinstveni identifikator kojeg baza podataka koristi kako bi pristupila danoj vrijednosti. Pohranjena vrijednost može poprimiti oblik jednostavnih tipova podatka, kao što su brojevi, pa sve do kompleksnijih brojeva. Baza podataka ključ-vrijednost je slična implementacijama *hash* mapa u programskim jezicima. Svoje podatke spremaju prvo u radu memoriju stroja, te iste zatim preslikavaju u pohranu, na taj način se postiže brže čitanje i pisanje podataka (A Guide to Key-Value Databases: influxData, n.d.).




*Slika 8*. Prikaz sheme Baze podataka Ključ-vrijednost, Izvor: https://commons.wikimedia.org/wiki/File:KeyValue.PNG

[3] Pojam horizontalnog skaliranja u ovom kontekstu podrazumijeva skaliranje baze podataka na više servera, dok vertikalno skaliranje podrazumijeva migriranje baze podataka na server sa više radne memorije, većim prostorom za pohranu i sl.

### 3.2.2. Baze podataka temeljene na dokumentima

Baze podataka temeljene na dokumentima su oblik baza podataka unutar kojih se podatci spremaju unutar dokumenta, obično CML, YAML, JSON ili BSON formata. Svakom od dokumenata se dodjeljuje ključ koji se koristi za pristupanje istom, što ga do neke razine čini sličnom Bazi podataka ključ-vrijednost. Ključevi koji se koriste za pristup podatcima se obično indeksiraju kako bi se ubrzao postupak dohvaćanja vrijednosti. Informacije o pojedinim poljima spremljenim unutar nekog dokumenta se spremaju kao metapodatci koji se zatim mogu koristiti za pisanje upita, odnosno filtriranje sadržaja (Williams, 2021). Implementacije Baza podataka temeljenih na dokumentima kao što je baza podataka MongoDB, sadrže vlastiti jezik kojim se pišu upiti, te koriste spomenute metapodatke za izvršavanje istih (MongoDB vs. MySQL Differences: MongoDB, n.d.). Korištenje Baza podataka temeljenih na dokumentima pruža veću fleksibilnost kod pohrane podatka u odnosu na Relacijske baze podatka, pošto se ne mora pratiti unaprijed definirana shema. Također je olakšan i dohvat podataka pošto se podatci mogu jednostavno iščitavati bez spajanja više tablica.

### 3.2.3. Grafne baze podataka

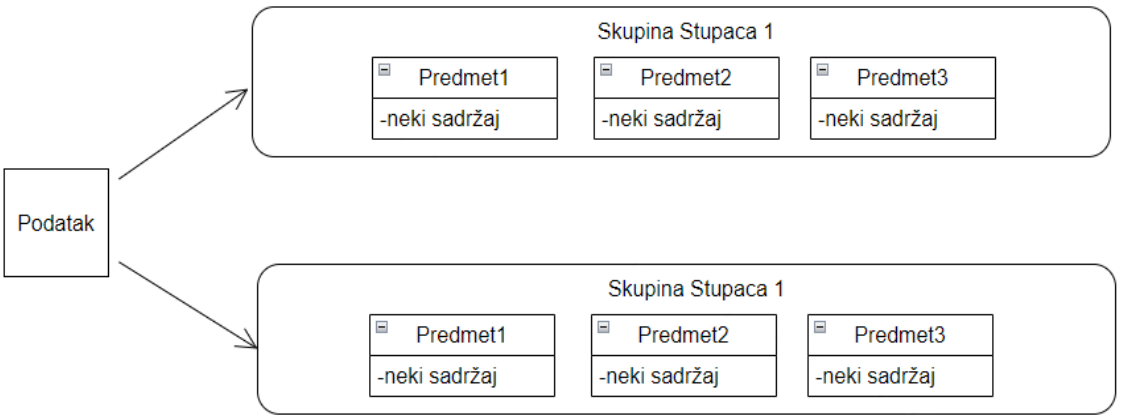
Grafne baze podataka predstavljaju podatke te odnose između podataka u obliku grafa.
Podatke spremaju na način da svaki unos predstavlja čvor grafa dok u isto vrijeme omogućuje spajanje različitih čvorova. Analogno tome Grafne baze podataka omogućuju slanje upita na temelju odnosa između podataka. Sastoje se od čvorova (*engl. nodes*), rubova (*engl. edges*) i svojstava (*engl. properties*). Razlikuju se dvije vrste Grafnih baza podataka (Graph Database Defined: Oracle, n.d.):

- Grafovi svojstava – koriste se za modeliranje odnosa između podataka, analogno tome omogućuju slanje upita temeljenim na odnosima između čvorova. Čvorovi i rubovi mogu imati atribute, odnosno svojstva s kojima su asocirani.
- RDF grafovi - Resource Description Framework (RDF) je vrsta Grafne baze podataka koja pohranjuje podatke kao mrežu objekata te stvara informacije iz postojećih odnosa. RDF podržava stvaranje opcionalnih shema modula, tj. Ontologija (*engl. Ontologies*). Ontologije omogućuju formalni opis podataka. Oni specificiraju klase objekata, svojstva odnosa i njihov hijerarhijski poredak (What is an RDF Triplestore?: Ontotext, 2023).

### 3.2.4. Baze podataka širokog stupca

Baza podatka širokog stupca ili Baza podataka skupine stupaca je vrsta NoSql baze podataka koja pohranjuje podatke u obliku skupine stupaca (*engl*. *Column-family*). Skupina stupaca je skup stupaca i redaka gdje svaki red ima jedinstveni ključ, a svaki stupac ime, vrijednost i vremensku oznaku. Baza širokog stupca organizira podatke tako da može poduprijet bolje performanse prilikom slanja upitna na veliki skup podataka. Svaka Skupina stupaca predstavlja povezanu skupinu podataka, prikazano na Slika 9 (Wide Column Database (Use Cases, Example, Advantages & Disadvantages): DatabaseTown, n.d.).

.




Slika 9. Shema Baze podataka širokog stupca, Izvor: https://databasetown.com/wide-column-database-use-cases/

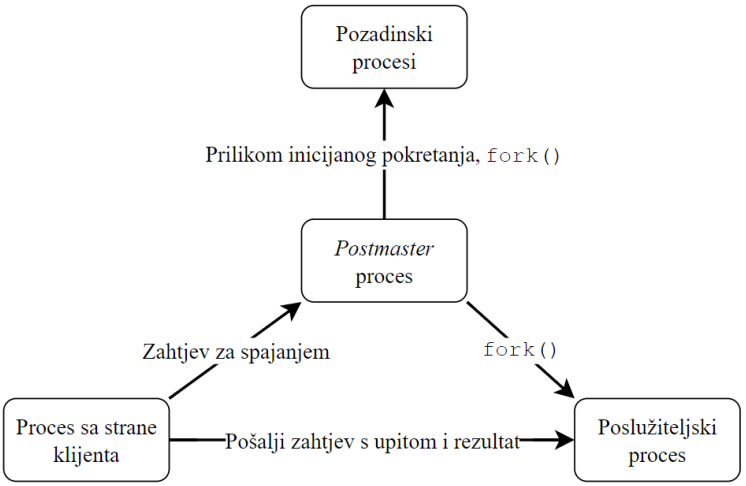
## 3.3. Baza podataka PostgreSQL

PostgreSQL je objektno-relacijski sustav za upravljanje bazom podataka (ORDBMS) temeljen na POSTGRES-u, razvijen je na Sveučilištu Kalifornije u Odsjeku za informatiku Berkeley (What is Postgres: PostgreSQL dokummnetacija, n.d.). Projekt PostgreSQL započeo je 1986. pod vodstvom profesora Michaela Stonebreakera na Kalifornijskom sveučilištu Berkeley. Projekt je izvorno nazvan POSTGRES, u odnosu na stariju bazu podataka Ingres, također razvijenu na Berkeleyju. Godine 1996. projekt je preimenovan u PostgreSQL kako bi se ilustrirala njegova podrška za programski jezik SQL (What is PostgreSQL?: Amazon, n.d.).

Obrada uputa u bazi podataka PostgreSQL počinje uspostavljanje veze između aplikacijskog programa i PostgreSQL poslužitelja. Aplikacijski program prenosi SQL upit poslužitelju te čeka odgovor. Nadgledni proces „*Postmaster*“ na temelju zaprimljenog upita

na zadanom TCP/IP kanalu (*engl. port*) otvara novi poslužiteljski proces, shema je prikazana na *Slika 10*. PostgreSQL implementira „*process per user*“ princip u kojem za svaku novu uspostavljenu vezu, otvara se novi poslužiteljski proces. Kako bi se izbjegla korupcija podataka tokom konkurentskih [4]procesa, koriste se tzv. „semafori“ (*engl. Semaphores)*,odnosno dijeljena memorija (*engl. Shared memory)* (How Connections Are Established: PostgreSQL dokumentacija, n.d.) (The Path of a Query: PostgreSQL dokumentacija, n.d.).

Nadalje dobiveni SQL dolazi do stadija pretvarača (*engl. parser stage*) koji na temelju dobivenog SQL upita stvara stablo upita. Prilikom procesa pretvaranja SQL upit prolazi kroz dvije komponente: *Parser* i *Lexer*. Zadaća *Lexer*-a je zamjena ključnih riječi s tokenima, te ga predaje dalje *Parser*-u na obradu. *Parser* zatim pregledava dobiveni sintaktičku točnost dobivenog SQL upita, te ako zadovoljava set definiranih „*gramatičkih pravila*“ na temelju njega gradi stablo upita (*engl. query tree)* (The Parser Stage: PostgreSQL dokumentacija, n.d.) (The Path of a Query: PostgreSQL dokumentacija, n.d.).




*Slika 10*. Shema *Postmaster* procesa, Izvor: https://severalnines.com/blog/understanding-postgresql-architecture/

Sustav za prepisivanje (*engl. rewrite system*) uzima stablo upita koje je stvorio stadij pretvaranja i traži sva pravila (pohranjena u sistemskim katalozima) (*engl. System catalogs*)

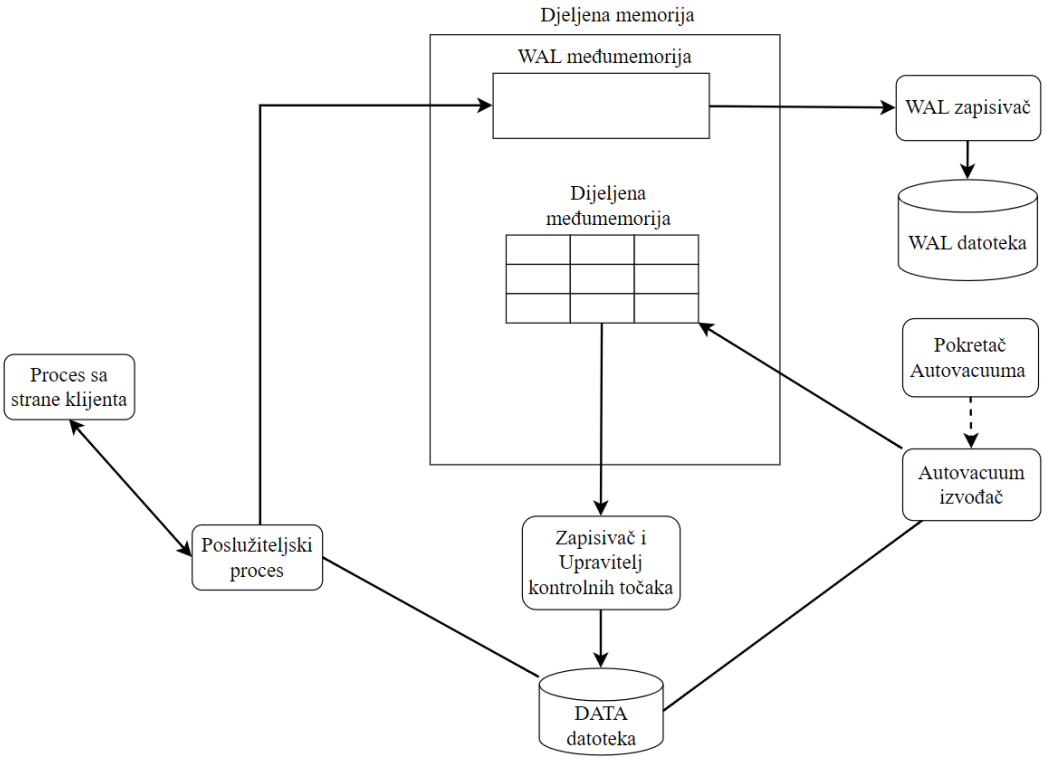
[4] Konkurentski procesi se u ovome kontekstu odnose na više poslužiteljskih procesa koje je podigao „*Postmaster*“

koja će se primijeniti na stablo upita. Izvodi transformacije definirane u tijelima pravila. Jedna primjena sustava prepisivanja je u realizaciji pogleda5 (*engl. view*). Kad god se napravi upit prema pogledu, sustav za prepisivanje prepisuje korisnički upit u upit koji umjesto toga pristupa početnim tablicama danim u definiciji pogleda.

Planer (*engl. Planner*) uzima prepisano stablo upita, te na temelju njega stvara plan upita (*engl. query plan*) koje će služiti kao ulazni podaci za Izvršitelja (*engl. Executor*). To se postiže stvaranjem svih mogućih „*puteva"* koji bi doveli do istog rezultata. Zatim se procjenjuje zahtjevnost predloženih „*puteva"* za izvedbu, te se odabire onaj najmanje zahtijevan. Izvršitelj rekurzivno prolazi kroz „*stablo plana"* (*engl. plan tree*) i dohvaća retke na način kako je predstavljen planom. Izvršitelj koristi sustav za pohranjivanje dok skenira relacije, obavlja sortiranje i spajanje, ocjenjuje kvalifikacije i na kraju vraća izvedene retke (The Path of a Query: PostgreSQL dokumentacija, n.d.).

Dijeljenja memorija (*engl. Shared memory*) predstavlja memoriju rezerviranu za transakcijsku predmemoriju i predmemoriju baze podataka. Najvažniji dijelovi su: Dijeljeni međuspremnik (*engl. Shared buffer*) i WAL međuspremnik (*engl. WAL buffer*). Zadaća Dijeljenog međuspremnika je smanjenje potrebnog čitanja i pisanja sa diska, dok WAL međuspremnik služi kao privremena pohrana podataka koji se pišu u bazu podataka prikazano na *Slika* (Chauhan, 2017).

[5] Odnosi se na virtualnu tablicu koju korisnik definira, najčešće pridruživanjem više različitih tablica




*Slika 11*. Prikaz strukture ORDBMS-a PostgreSQL,
Izvor:https://severalnines.com/blog/understanding-postgresql-architecture/

Procesi prikazani na Slika . su slijedeći (Chauhan, 2017):

- Zapisnik (*engl. Logger*) – Zapisuje nastale greške u zapisničku datoteku(engl. Log file)
- Sustav za praćenje referentnih točaka (engl. Checkpointer) – kada izvođenje programa dođe do referentne točke, sadržaj označenog međuspremnika je zapisan u datoteku
- Zapisivač (*engl. Writer*) – periodički zapisuje sadržaj označenog međuspremnika u datoteku
- WAL zapisivač (*engl. WAL writter*) – zapisuje sadržaj WAL međuspremnika u WAL datoteku
- Pokretač Autovaccuma (*engl. Autovacuum launcher*) – grana autovakuum workera kada je autovakuum omogućen.
- Arhivar (*engl. archiver*) – kada je pokrenut Archive.log mod, kopira sadržaj WAL datoteke u specificirani direktorij
- Prikupljač statistike (*engl. stats collector*) – skuplja informacije o korištenju DBMS-a.

# 4. WEB SERVISI

Razmjena podataka između klijentskih i poslužiteljskih aplikacija u današnjem svijetu se velikim djelom rješava implementacijom Web servisa (*engl. Web services*). Web servis je softverski sustav koji podupire interoperabilnu komunikaciju između dva računala preko mreže, čije je sučelje opisano formatom razumljivim računalu, tj. *Web Service Definition Language*-om (WSDL) (What is a web service?: IBM, n.d.). WSDL pruža klijentu uputstva kako složit zahtjev prema Web servisu, te opisuje sučelje koje je pruženo od strane Web servisa (What is WSDL?: IBM, n.d.). Kako bi se našao Web servis, te otkrile informacije o istom koristi se *Unirvesal Description Discovery, and Integration* (UDDI) specifikacija. UDDI definira na koji se način objavljuju i definiraju informacije o Web servisima (Universal Description, Discovery, and Integration (UDDI): IBM, n.d.). Komponente Web servisa na osnovnoj razini moraju uključivati (Lewis, n.d.):

- Mora biti dostupna preko Interneta
- Mora koristiti XML format za razmjenu informacija
- Interoperabilna je s bilo kojim programskim jezikom ili operativnim sustavom
- Samodostatna je i samoopisna putem standardnih XML semantika
- Može je se otkriti pomoću UDDI standarda

Web servisi se mogu predstaviti u obliku

- *Simple Object Access Protocol* (SOAP) – je specifikacija za prijenos podataka između sustava i aplikacija, koja koristi XML format za prijenos podataka. SOAP ima stroge smjernice implementacije, te se oslanja na HTTP protokol za prijenos podataka, no osim njega još može koristiti SMTP,TCP, i UDP (Lane, 2023). Struktura SOAP paketa je strogo definirana te se sastoji od (Structure of a SOAP message: IBM, n.d.):
    - *SOAP omotnice* (*engl. SOAP envelope*)*:*<Envelope> je izvorni element svakog SOAP paketa, sadrži dva podelementa, izborni <Header> i obavezni <Body>.
    - *SOAP zaglavlje* (*engl. SOAP header*): <Header> bilješka se koristi za prijenos informacija vezanih uz aplikaciju.
    - *SOAP tijelo* (*engl. SOAP body*): <Body> bilješka sadrži informacije namijenjene primatelju paketa.
    - *SOAP greška* (*engl. SOAP fault*):<Fault> se koristi za pohranu informacija o nastalim greškama.
- *GraphQL* – nastao kao odgovor na REST, omogućuje klijentima dohvaćanje traženih

podataka pomoću komunikacije s samo jednim *endpoint-om.* Prilikom slanja upita korisnik može određivati vlastitu strukturu odgovora kojeg će primiti, dodavajući ili oduzimajući tražena polja iz tijela zahtijeva. Jednom kada GraphQL server zaprimi upit, i provjeri njegovu vjerodostojnost, poziva *resolver* metode koje pune podatke na temelju dobivenog upita te ih vraćaju unutar zadanog formata, najčešće JSON-a (What is GraphQL and how does it work?: Postman, 2020) (Big Picture (Architecture): How to GraphQL, n.d.).

- *Google Remote Procedure Call* (*gRPC*) – u ovome pristupu klijentska aplikacija može direktno pozivati metodu koja se nalazi na udaljenom stroju, kao da je prisutna lokalno. Prilikom postavljanja, potrebno je definirati servis, metode koje će se zvati, kao i povratne tipove. S strane servera, server implementira definirane metode, te obrađuje pozive tih metoda. Klijent s druge strane poziva definirane metode (Introduction to gRPC:gRPC dokumnetacija, n.d.).
- *Representational State Transfer API (REST API)* - REST API predstavlja arhitektonski stil za stvaranje Aplikacijskog programskog sučelja koje koristi HTTP protokole za pristupanje podatcima. Detaljniji opis se može pronaći u idućem potpoglavlju.

## 4.1. Representational state transfer (REST)

Representational state transfer predstavlja set ograničenja odnosno vodilja, primjenom kojih se stvaraju REST API web servisi. (Masse, 2011) Dane standarde je predstavio Roy Fielding 2000. godine u svojoj doktorskoj disertaciji koji su podijeljeni u 6 kategorija (Masse, 2011): Klijent-Server (*engl. Cilent-server*), Standardizirano sučelje (*engl. Uniform interface*), Sistem u slojevima (*engl. Layerd system*), Predmemorija (*engl. Cache*), Sustav bez stanja (*engl. Stateless*), Kod na zahtjev (*engl. Code-on-Demand*).

### 4.1.1. Klijent-Server

Ova kategorija smjernica, nalaže odvajanje ovlasti i zadaća između klijenta i poslužitelja. Klijent i server mogu implementirati i koristiti različite tehnologije dok god podliježu smjernicama *Standardiziranog sučelja* Interneta (Masse, 2011).

### 4.1.2. Standardizirano sučelje

Resurs bi trebao biti identificiran kroz jedinstveni URL, te koristiti ishodišne metode mrežnog protokola odnosno DELETE, PUT i GET korištenjem HTTP protokola (Gillis, n.d.), točnije Fielding je naveo slijedeće 4 komponente kao standarde kroz koje sučelja komuniciraju uniformno (Masse, 2011):

1. Prepoznavanje resursa – kao što je navedeno prethodno, svakom internetskom resursu se pristupa putem jedinstvenog URL-a.
2. Manipulacija s resursima putem identifikatora – koncept se zasniva na predstavljanju resursa koji se nalazi na poslužitelju kroz različite reprezentacije, odnosno različite formate ovisno o potrebama klijenta.
3. Samoopisne poruke – željeno stanje resursa se može opisati unutar zahtijeva klijenta.
4. Hypermedia kao osnovica aplikacijskog stanja – unutar (Masse, 2011) se navodi važnost hiplerlinkova kao način pregledavanja i pretraživanja informacija na značajan način.

### 4.1.3. Sistem u slojevima

Omogućuje transparentne implementacije mrežnih posrednika između poslužitelja i klijenta, korištenjem internetskog standardiziranog sučelja. Mrežni posrednici se često koriste u svrhe pojačanja sigurnosti, spremanje odgovora u predmemoriju te *load balancing* (Masse, 2011)

### 4.1.4. Sustav bez stanja

*Sustav bez stanja* (*engl. stateless*) pristup nalaže da su klijentse aplikacije one koje moraju spremati informacije povezne uz stanja iste, što oslobađa resurse poslužitelja za obradu više zahtijeva (Masse, 2011).

### 4.1.5. Kod na zahtjev

U (Masse, 2011) je dana definicija koda na zahtjev, kao situacije u kojoj poslužitelj privremeno prenosi izvršne datoteke klijentu na izvođenje, te je također navedeno kako je ovo jedini izborni standard pošto se stvara tehnološka zavisnost između poslužitelja i klijenta. Tehnološka zavisnost u ovom slučaju se odnosi na programski jezik u kom je pisan program kog izvršna (*engl. executable*) datoteka izvodi (Masse, 2011),

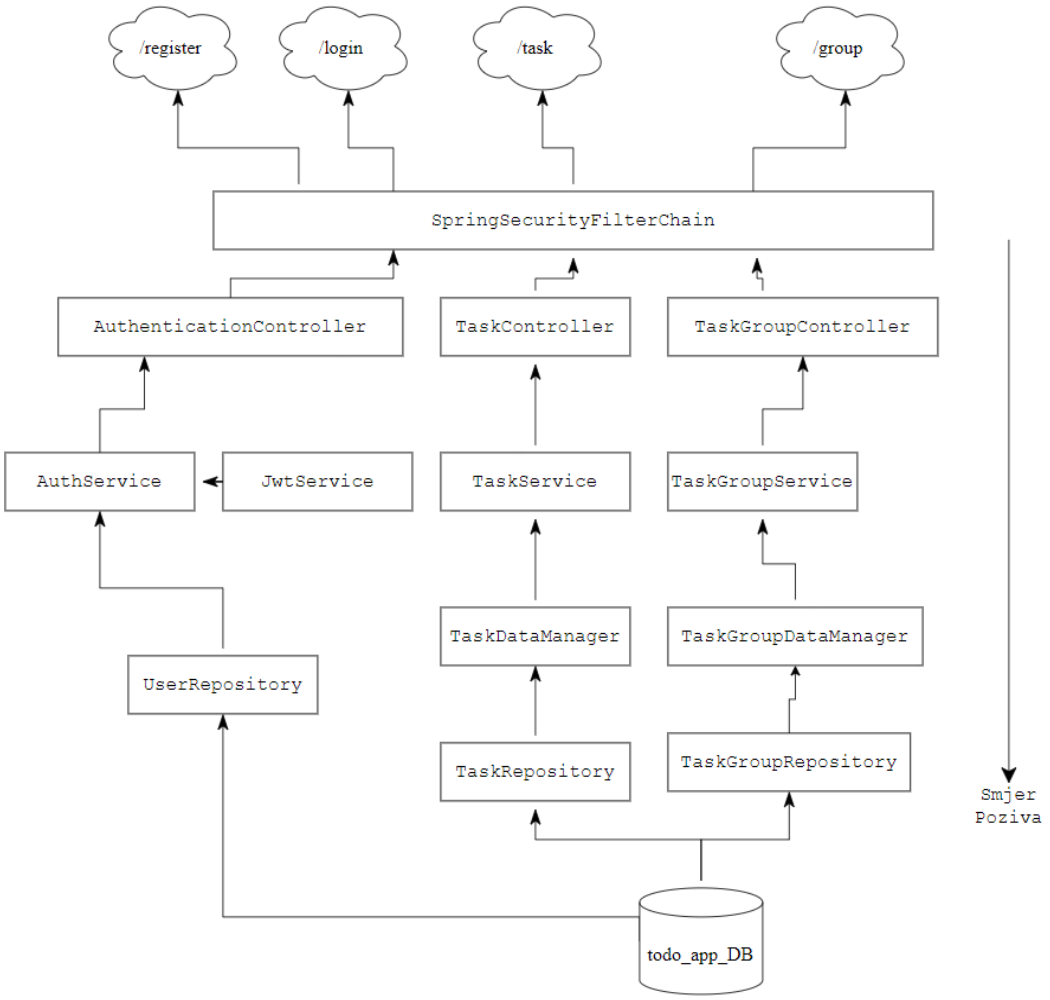
# 5. ANALIZA POSLUŽITELJSKE APLIKACIJE IZRAĐENE U PROGRAMSKOM OKVIRU SPRING BOOT

U ovom poglavlju razmotriti će se primjena prethodno obrađenih tehhnologija na studijskom slučaju aplikacije za praćenje zadataka (*engl. TODO application*). Spomenuta aplikacija obavlja zadaće spremanja, uređivanja, brisanja, dohvaćanja, te dohvaćanja stranice sa zadatcima[6]. Uz to korisnik također ima mogućnost grupiranja zadataka u grupe koje sam definira. Spomenute grupe zadataka je također moguće stvarati, brisati, i pregledavati u listi. U svrhu poboljšanja korisničkog iskustva, te same sigurnosti aplikacije ugrađena je mogućnost *Single Sign On* (SSO), putem implementacije *JSON Web Token*-a (JWT), čiji je programski kod preuzet sa github repozitorija korisnika Alia Bualia. Kao alat za upućivanje zahtjeva na server, te pregeldavanja odgovora se koristi Postman. Analiza će se izvoditi po zadaćama koje aplikacija obavlja, gdje će se na početku analize svake zadaće prikazati pozivi koji se upućuju na server, te odgovor koji se očekuje. Slijedeći korak biti će analiza programskog koda, te na kraju analiza baze podataka.

## 5.1. Pregled arhitekture TODO aplikacije

Arhitektura promatrane aplikacije je podijeljena na 3 logičke cjeline među kojima je logička zavisnost smanjena koliko je to moguće, odnosno primjenjuje se „*Loose coupling*". Tri cjeline od kojih je aplikacija sastavljena su: Sloj kontrolera, Sloj servisa, podatkovni sloj. Ovim pristupom se postiže modularnost aplikacijskog koda te ni jedna komponenta nije direktno ovisna o drugoj. Cijela aplikacija se može podijeliti na 3 značajke, a to su: autentifikacija i autorizacija korisnika, značajka manipuliranja zadatcima, te značajka manipuliranja grupama zadataka, shema arhitekture aplikacije je prikazana *Slika 11* .

[6] Unutar programsko koda koji će biti prikazan u nastavku, zadatak će biti oslovljavan engleskim nazivom „*Task", kao i sve ostale varijable i klase, analogno* tome grupa zadataka će također biti oslovljavana engleskim nazivljem „*TaskGroup".* Razlog tome je pridržavanje konvencija čitljivosti i razumljivosti koda. Također stranica sa zadatcima se odnosi na polje zadataka koje je vraćeno sa servera upotrebom paginacije.




*Slika 11*. Prikaz arhitekture promatrane TODO aplikacije

Detalja analiza pojedinih komponenti arhitekture nadolazi u slijedećim potpoglavljima.

## 5.2. Sigurnosne postavke aplikacije

Implementacija sigurnosne značajke promatrane aplikacije se temelji na primjeni vlastitih sigurnosnih pravila definiranih u implementaciji `SecurityFilterChain`. Implementacija `SecurityFilterChain`, prikazana na *Programski kod 1*, se sastoj od konfiguracije zaštite protiv *Cross-site Request Forgery* (CSRF) napada, te postavki pravila autorizacije HTTP zahtijeva. Može se zamijetiti da su svi zahtjevi na endpoint *„/user/“* dopušteni, odnosno izuzeti od autorizacije, dok svi ostali podliježu autorizaciji, razlog toj iznimci je pristup značajkama prijave i registracije čije rute se nalaze unutar rute *„/user/“*.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {
private final JwtAuthFilter jwtAuthFilter;
  private final AuthenticationProvider authenticationProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception{
        http
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests((authorizeHttpRequests)
->
                        authorizeHttpRequests

.requestMatchers("/user/**").permitAll()
                                .anyRequest()
                                .authenticated()

                )
                .sessionManagement((sessionManagement) ->
                        sessionManagement

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                )
                .authenticationProvider(authenticationProvider)
                .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```

*Programski kod 1.* Prikaz konfiguracije klase `SecurityFilterChain`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

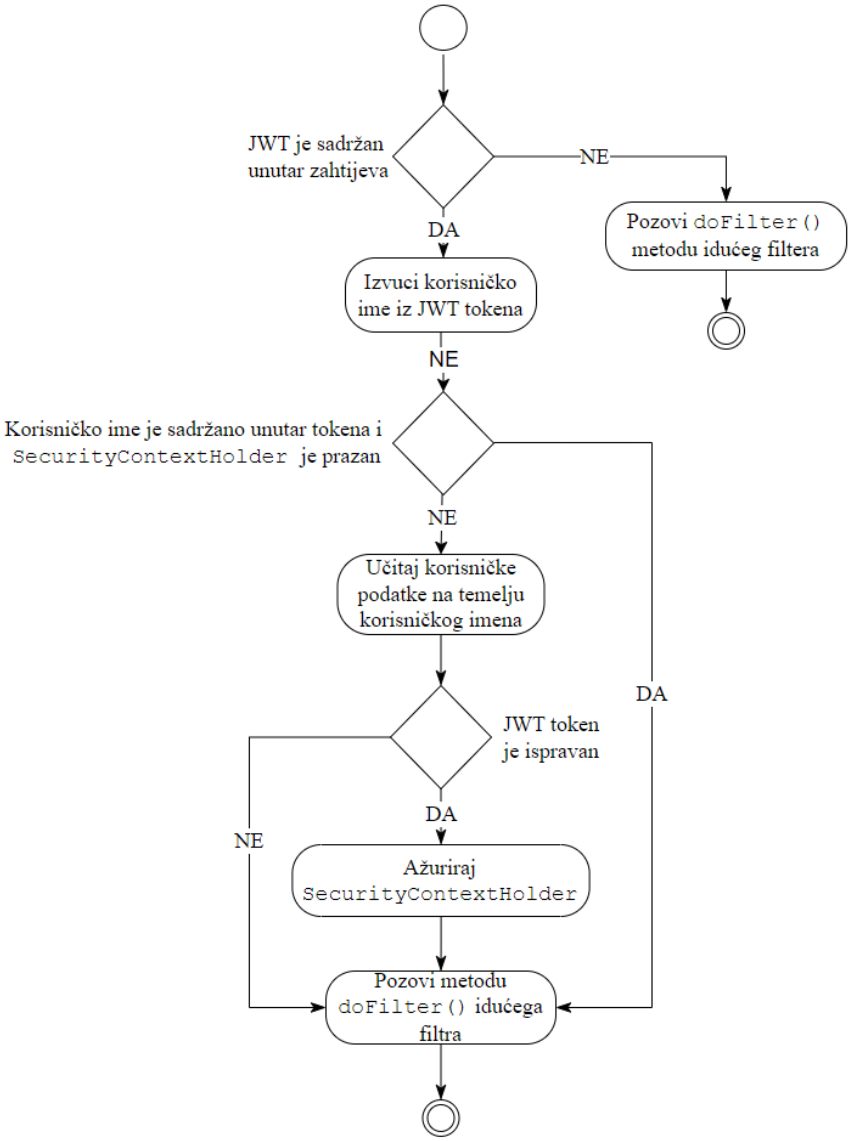
`SessionCreationPolicy` postavka u slučaju promatrane aplikacije označava da je veza između klijenta i poslužitelja bez stanja (*engl. Stateless*) što nalaže da poslužitelj ne mora pamtiti podatke povezane uz vezu sa klijentskom aplikacijom, već su svi podatci koji se odnose na autorizaciju i autentifikaciju korisnika pruženi u samome zahtjevu, ovaj pristup se koristi kod JWT pristupa autentifikaciji i autorizaciji zahtijeva. Nadalje se definira prilagođena logika provjeravanja valjanosti zaprimljenog JWT tokena unutar `jwtAuthFilter` *bean*-a. Klasa `JwtAuthFilter` je implementacija klase `OncePerRequestFilter` te se unutar nje obavlja validacija JWT tokena koji je primljen

u zaglavlju zahtijeva, prikazano na *Programski kod 2* .

```
@Component
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {
    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;
    @Override
    protected void doFilterInternal(
            @NonNull HttpServletRequest request,@NonNull
HttpServletResponse response,@NonNull FilterChain filterChain)
throws ServletException, IOException {
        final  String authHeader =
request.getHeader("Authorization");
        final  String jwt;
        final  String userName;
        if(authHeader == null || !authHeader.startsWith("Bearer
")){
            filterChain.doFilter(request,response);
            return;
        }
        jwt = authHeader.substring(7);
        userName = jwtService.extractUsername(jwt);
        if (userName != null &&
SecurityContextHolder.getContext().getAuthentication() ==
null){
            UserDetails userDetails =
this.userDetailsService.loadUserByUsername(userName);
            if (jwtService.isTokenValid(jwt,userDetails)){
                UsernamePasswordAuthenticationToken authToken =
new UsernamePasswordAuthenticationToken(
                        userDetails,
                        null,
                        userDetails.getAuthorities()
                );
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authToken)
;
            }

        }
        filterChain.doFilter(request,response);
    }
}
```

*Programski kod 2*. Prikaz implementacije klase `JwtAuthFilter`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Tok validacije JWT tokena je prikazan na *Slika 12*, ako je JWT valjan, `SecurityContextHolder` se ažurira s tokenom generiranim na temelju korisničkih detalja sadržanih unutar samog JWT tokena. Jednom kad se `SecurityContextHolder` popuni s podatcima o korisniku, poziva se implementacija metode `doFilter()` iduće `Filter` klase unutar *FilterChain*-a.




*Slika 12*. Prikaz sekvencijsko dijagrama provjeravanja valjanosti JWT tokena unutar klase `JwtAuthFilter`

Važno je također napomenuti kako je objekt tipa `JwtAuthFilter` umetnut prije `AuthenticationProvider` filtra unutar *FilterChaina*, kao što je prikazano *Programski kod 1*, svrha toga je tako da nakon što se završi sa validacijom JWT tokena, `AuthenticationProvider` može obaviti funkciju provjeravanja

`SecurityContextHolder`-a te na temelju zapisa unutar istoga donesti odluku o autorizaciji nadolazećeg zahtijeva. Također kako se može primijetiti na *Programski kod 2*, sama zadaća validacije tokena, te izvlačenja korisničkog imena iz istog unutar klase `JwtAuthFilter` je delegirana klasi `JwtService`, čije su metode za izvlačenje korisničkog imena i validacije tokena prikazane na *Programski kod 3*. Validacija JWT tokena započinje izvlačenjem korisničkog imena, metodom `extractUsername()`, koje se zatim uspoređuje s onim dobivenim u parametrima metode, zatim se provjerava rok trajanja tokena.

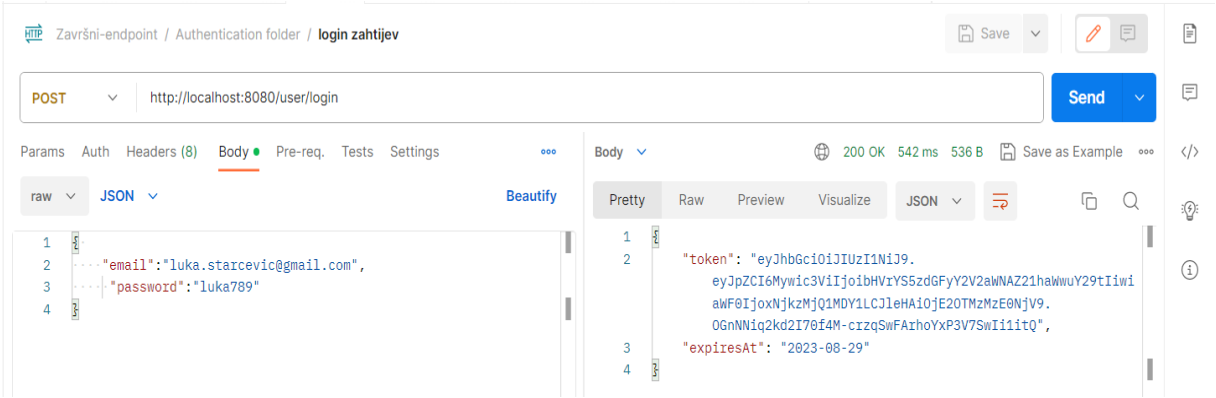
```
@Service
public class JwtService {

    private static final String SECRET_KEY =
"3483a153c257bde245a21622b39775f968c8c849b1f6ad53f6d94f144f4efeb5";

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }


    public <T> T extractClaim(String token, Function<Claims,T>
claimsResolver){
        final  Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    public boolean isTokenValid(String token,UserDetails
userDetails){
        final  String userName = extractUsername(token);
        return (userName.equals(userDetails.getUsername())) &&
!isTokenExpired(token);
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    private Date extractExpiration(String token) {
        return  extractClaim(token,Claims::getExpiration);
    }

    private Claims extractAllClaims(String token){
        return
Jwts.parserBuilder().setSigningKey(getSignInKey()).build().parseCla
imsJws(token).getBody();
    }
```

*Programski kod 3.* Prikaz klase `JwtService` sa metodama korištenim pri validaciji tokena te izvlačenju korisničkog imena, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Rezultat metode se dobiva koristeći *and* operaciju između rezultata usporedbe korisničkog imena te negacije metode `isTokenExpired()`. Kako bi osigurali sigurnu komunikaciju između klijenta i poslužitelja putem JWT tokena koristimo 256-bitni tajni ključ (*SECRET_KEY*) kojeg koristi *hash* metoda sha256 za potpisivanje JWT tokena. Za pristup podatcima kao što su vrijeme isteka, korisničko ime i sl. koristi se metoda `extractAllClaims`() koja prima token te uz pomoć metode `Jwts.parserBuilder`() i *SECRET_KEY* -a izvlači podatke sadržane unutar JWT tokena.

### 5.3. Značajka prijave korisnika

Proces prijave korisnika započinje slanjem korisničkog imena i lozinke na rutu „*user/login*“ sa HTTP Post zahtjevom.




*Slika 13.* Prikaz slanja zahtijeva za prijavom i prikaz dobivenog odgovora korištenjem alata Postman

Tu zadaću autentifikacije i autorizacije preuzima `AuthenticationController` koji mapira zahtjev na metodu `login()`, koja zauzvrat poziva metodu `authenticate()`, klase `AuthService`, prikazano *Programski kod 4.*

```
@RestController
@RequestMapping("user")
@RequiredArgsConstructor
public class AuthenticationController {
    private final AuthService authService;


    @PostMapping("/login")
    public ResponseEntity<AuthenticationResponse> login(
            @RequestBody AuthenticationRequest request
    ){
        return
ResponseEntity.ok(authService.authenticate(request));
    }
}
```

*Programski kod 4*. Prikaz metode `login()` klase `AuthenticationController`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Povratni tip metode `authenticate()` je `AuthenticationResponse`, a kao parametar prima objekt tipa `AuthenticationRequest` koji sadrži polja s e-poštom te lozinkom dobivenih iz tijela primljenog zahtijeva. Samu autentifikaciju delegira `AuthenticationManager`-u koji izvodi provjeru `SecurityContextHolder` -a sa pruženim mu parametrima, te u slučaju neuspjele autentifikacije stvara `AuthenticationException`. U slučaju uspješne autentifikacije, stvara se nova *hash* mapa u kojoj se pohranjuje korisnički identifikator koji se kao parametar uz dohvaćene korisničke podatke prosljeđuje metodi `generateToken()`, kao što je prikazano na *Programski kod 5* . Metoda `generateToken()` klase `JwtService` generira JWT token korištenjem statičke metode `builder()`,`klase Jwts`. U token se uključuju vrijeme isteka, korisničko ime, te dodatni parametri izraženi kroz *extraClaims* polje, koji je u ovome slučaju samo korisnički identifikator primljen kroz parametar metode. Nakon što je generiran, token se vraća na klasu `AuthService` koji ga zajedno s vremenom isteka „pakira“ u `AuthenticationResponse` objekt, te ga vraća kao odgovor na kontroler. Generirani token se nadalje koristi od strane klijenta u „*Authorization“* polju zaglavlja kao autorizacija zahtijeva prema poslužitelju, te se iz njega izvlači korisnički identifikator pomoću kojeg se dohvaćaju korisnički podatci.

```
@Service
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository repository;
    private final PasswordEncoder passwordEncoder;

    private final AuthenticationManager authenticationManager;

    private  final JwtService jwtService;


    public AuthenticationResponse
authenticate(AuthenticationRequest request) {

        authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        request.getEmail(),
                        request.getPassword()
                )
        );
        var user =
repository.findByUserName(request.getEmail()).orElseThrow();
        Map<String,Object> extraClaims = new HashMap<>();
        extraClaims.put("id",user.getId());
        var jwtToken =
jwtService.generateToken(extraClaims,user);
        long expirationTime = jwtService.getExpatriationTime();
        return AuthenticationResponse.builder()
                .token(jwtToken)
                .expirationTime(expirationTime)
                .build();
```

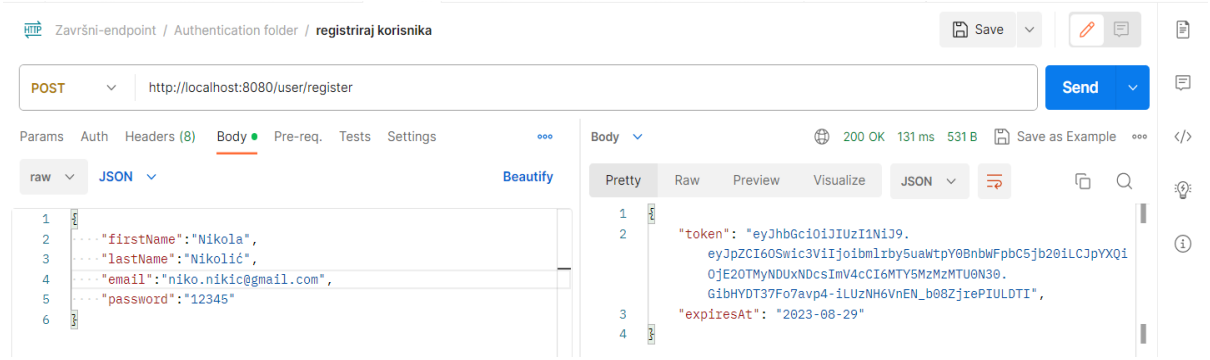
*Programski kod 5*. Prikaz klase `AuthService` sa metodom `authenticate()`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

```
public String generateToken (Map<String,Object> extraClaims,
UserDetails userDetails){
    return
Jwts.builder().setClaims(extraClaims).setSubject(userDetails.ge
tUsername()).setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(getExpatriationTime()
)).signWith(getSignInKey(),
SignatureAlgorithm.HS256).compact();
}
public long getExpatriationTime() {
    return Instant.now().toEpochMilli() + 86400000;
```

*Programski kod 6*. prikaz metode korištene za generiranje JWT tokena unutar klase `JwtService`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

## 5.4.Značajka registracije korisnika

Postupak registracije je sličan postupku prijave, korisnik šalje zahtjev sa imenom, prezimenom, e-poštom, i lozinkom na „*user/register*“ rutu putem HTTP post zahtijeva. Ako je zahtjev uspješno odrađen, kao odgovor je vraćen HTTP odgovor sa statusnim kodom 200, generiranim tokenom i datumom isteka kao što je prikazano na *Slika 14*.




*Slika 14*. Prikaz tijela zahtijeva i rezultata zahtijeva za registracijom korisnika poslanog pomoću alata Postman

Nakon što je primio zahtjev za registracijom korisnika, `AuthenticationController` poziva metodu `register()` klase `AuthService` kojoj kao parametar prosljeđuje objekt tipa `RegisterRequest` , prikazano na *Programski kod 7*. Klasa `RegisterRequest` u sebi sadrži podatke o korisniku dobivene iz tijela primljenog HTTP Post zahtijeva.

```
@PostMapping("/register")
public
ResponseEntity<AuthenticationResponse>
register(
        @RequestBody RegisterRequest
request
){
```

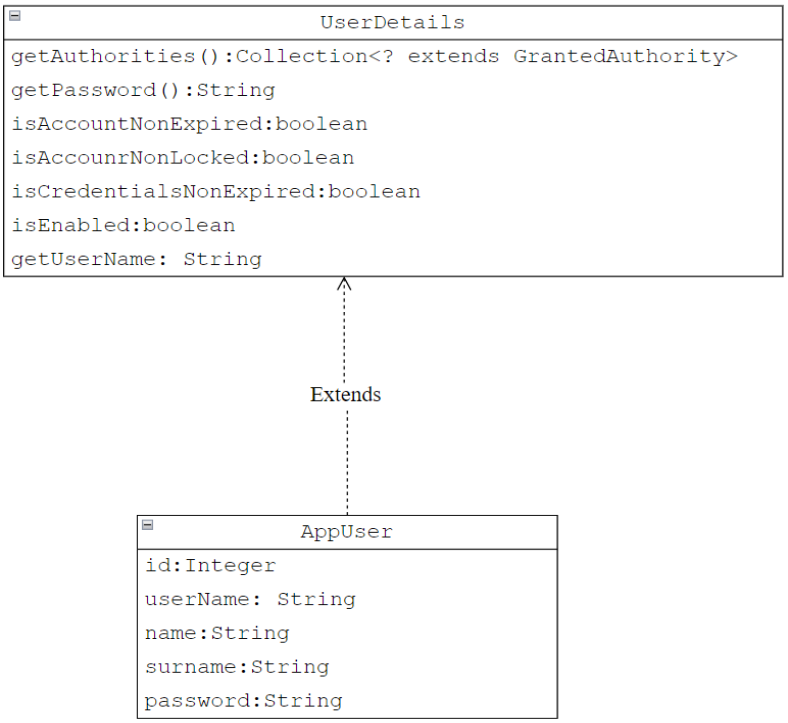
Programski kod 7. Prikaz `register()` metode unutar klase `AuthenticationController`, , Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Nakon što je primila zahtjev, metoda `register()` klase `AuthService`, stvara novu instancu klase `AppUser` na temelju podatka proslijeđenih kroz parametar metode, prikazano na *Programski kod 8*.

```
public AuthenticationResponse register(RegisterRequest
request) {
    var user = AppUser.builder()
            .name(request.getFirstName())
            .surname(request.getLastName())
            .userName(request.getEmail())

.password(passwordEncoder.encode(request.getPassword()))
            .build();
    repository.save(user);
    Map<String,Object> extraClaims = new HashMap<>();
    extraClaims.put("id",user.getId());
    System.out.println(extraClaims);
    var jwtToken = jwtService.generateToken(extraClaims,user);
    long expirationTime = jwtService.getExpatriationTime();
    return AuthenticationResponse.builder()
            .token(jwtToken)
            .expirationTime(expirationTime)
            .build();
```

*Programski kod 8.* Prikaz register() metode klase AuthenticationController, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Klasa `AppUser` je naslijeđena od klase `UserDetails` koja sadrži metode koje pomažu `AuthenticationManager`-u pri autentifikaciji korisnika, polja su prikazana na *Slika 15,* na *Programski kod 9,* prikazana je implementacija metode `getAuthorities()` koja definira koje su sve uloge dostupne objektima klase `AppUser`, u ovom slučaju to je samo jedna uloga: *"USER"* ili korisnik. Nakon što je objekt user instanciran, sprema se u bazu podataka pomoću metode `save()` objekta `UserRepository`. Objekt tipa `UserRepository` se instancira na temelju sučelja `UserRepository` koje naslijeđeno od sučelja `JpaRepository`, te sadrži jednu metodu `findByUserName()`. S obzirom da je implementacija `JpaRepository` sučelja, sučelje `UserRepository` podliježe IoC mehanizmu programskog okvira Spring, što znači da sve metode koje definiramo unutar sučelja, programski okvir Spring implementira kad god otkrije programsku ovisnicu toga tipa. Analogno tome nasljeđuje sve metode definirane unutar sučelja `JpaRepository`, pa samim time i `save()` metodu, koja obavlja funkciju spremanja stvorenog `AppUser` objekta u bazu podataka.




*Slika 15*. Klasni dijagram klase `AppUser`

Kako bi se omogućilo spremanje objekta u bazu podataka putem Hibernate ORM-a, klasa `AppUser` se mapira s entitetom `app_user` unutar baze podataka. Mapiranje se odvija pomoću bilješke `@Entitiy`, te ako potreba postoji, odnosno ako se imena polja klase razlikuju od stupaca u bazi podataka na koje se mapiraju , ta polja se označuju sa bilješkom `@Column` unutar kojeg se navodi ime stupca baze podataka na koju bi se to polje trebalo mapirati, primer je prikazan na *Programski kod 9*. Važno je također napomenuti da radi povećanja zaštite korisničkih podataka, dobivena lozinka se kriptira pomoću 256 bitne enkripcije. Enkripcija se izvodi putem sučelja `PasswordEncoder`, čija instanca, `BCryptPasswordEncoder` se registrira unutar konfiguracijske klase `ApplicationConfig`, metoda koja pokazuje registriranje *bean*-a je prikazana na *Programski kod 10*.Uunutar klase `ApplicationConfig` prikazane na *Programski kod 10*. se registrira implementacija sučelja `PaswordEncoder` kojeg će programski okvir Spring dalje koristiti kao bean.

```
@NoArgsConstructor
@Data
@AllArgsConstructor
@Builder
@Entity(name = "app_user")
public class AppUser implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;
    @Column(name = "user_name")
    private String userName;
    @Column(name = "name")
    private String name;
    @Column(name = "surname")
    private String surname;
    @Column(name = "password")
    private String password;


    @Override
    public Collection<? extends GrantedAuthority>
getAuthorities() {
        return List.of(new SimpleGrantedAuthority("USER"));
    }
//implementacije naslijeđenih metoda
```

*Programski kod 9.* Prikaz klase `AppUser`, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

Nakon obavljene enkripcije lozinke i spremanja korisnika u bazu podataka, generira se token koji se zatim šalje klijentu preko `AuthenticationController`-a. Proces generacije tokena prilikom registracije korisnika je isti kao i prilikom prijave korisnika.

```
@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {
    private final UserRepository repository;
    @Bean
    public UserDetailsService userDetailsService(){
        return username ->
repository.findByUserName(username).orElseThrow(()->new
UsernameNotFoundException("User not found!"));

    }
    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authProvider = new
DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService());
        authProvider.setPasswordEncoder(passwordEncoder());
        return  authProvider;
    }

    @Bean
    public   PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    AuthenticationManager
authenticationManager(AuthenticationConfiguration
configuration) throws Exception {
        return configuration.getAuthenticationManager();
    }
}
```

*Programski kod 10*. Prikaz konfiguracijske klase, Izvor: https://github.com/ali-bouali/spring-boot-3-jwt-security/tree/main

### 5.5. Značajka dohvaćanja stranice sa zadatcima

Dohvaćanje stranice sa zadatcima započinje slanjem HTTP Post zahtijeva na URL *„task/getTaskPage“,* u čijem tijelu korisnik šalje JSON objekt koji ima dva polja, *„pageRequestData“* i *„taskFilterData“.* Objekt *„pageRequestData“* u sebi sadrži slijedeća polja:

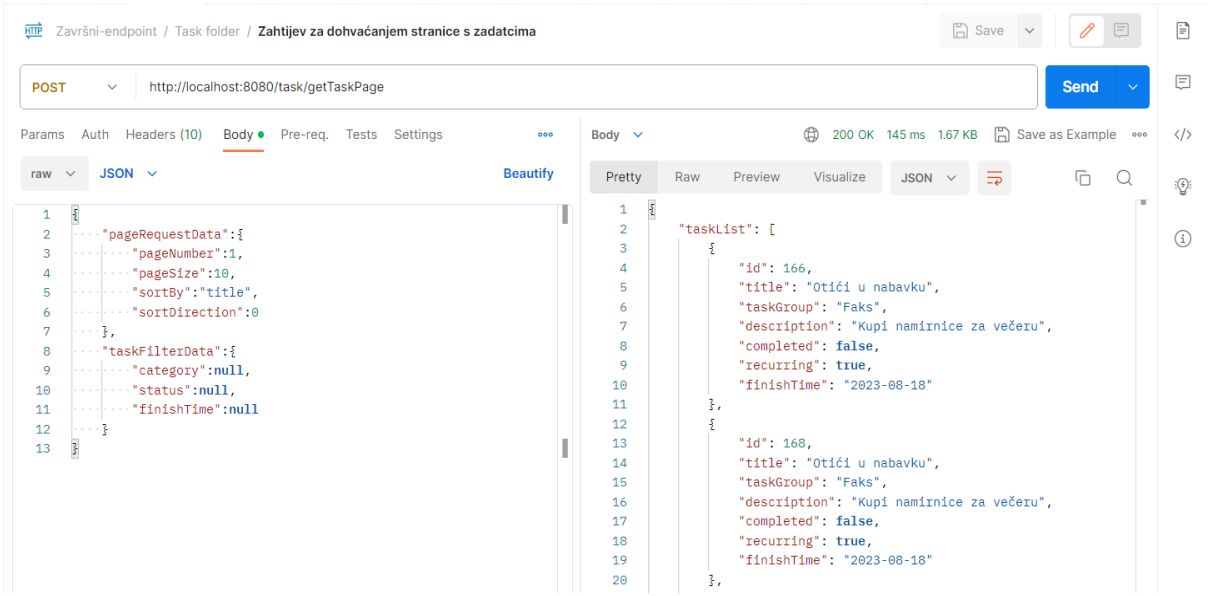
- *„pageNumber“* – s obzirom da je na rezultate ovog zahtijeva primijenjen sustav paginacije, polje *„pageNumber“* označava koju „stranicu“ korisnik želi dohvatiti
- *„pageSize“*- određuje broj elemenata primljene stranice
- *„sortBy“* – unosi se polje `Task` objekta, po kojem korisnik želi sortirati sadržaj

- „*sortDirection*“ – korisnik unosi ili broj 1 ili broj 0, gdje broj 1 označava uzlazno sortiranje, a broj 0 silazno sortiranje

Objekt „*taskFilterData*“ se sastoj od slijedećih polja:

- „*category*“ – korisnik unosi identifikator grupe po kojoj želi filtrirati sadržaj
- „*status*“- korisnik ovo polje popunjava sa *true* ili *false*, te se ono odnosi na status dovršenosti pojedinog zadatka
- „*finishTime*“ – korisnik unosi datum izražen u obliku milisekunda od epohe, te se ispisuju svi datumi prije „*finishTime*“ parametra

Analogno navedenome može se navesti da objekt „*pageRequestData*“ služi za slanje podataka o traženoj „stranici“, te kako će sadržaj na njoj biti sortiran, dok „*taskFilterData*“ sadrži podatke potrebne za sortiranje. Potrebno je također napomenuti da polja objekta „*taskFilterData*“ mogu sadržavati *null* vrijednosni u kom se slučaju neće obavljati nikakvo filtriranje s navedenim poljima. Kao što je spomenuto u prethodnome dijelu, svaki zahtjev prema serveru koji nije na ruti „*/user/*“ mora sadržavati JWT token u polju „*Authorization*“ zaglavlja zahtijeva. Primjer zahtijeva i odgovora servera je prikazan na *Slika 16*.




*Slika 16.* Prikaz zahtijeva za dohvaćanjem stranice zadataka, te dobivenog odgovora. Zahtjev je napravljen pomoću alata Postman

### 5.5.1. TaskController

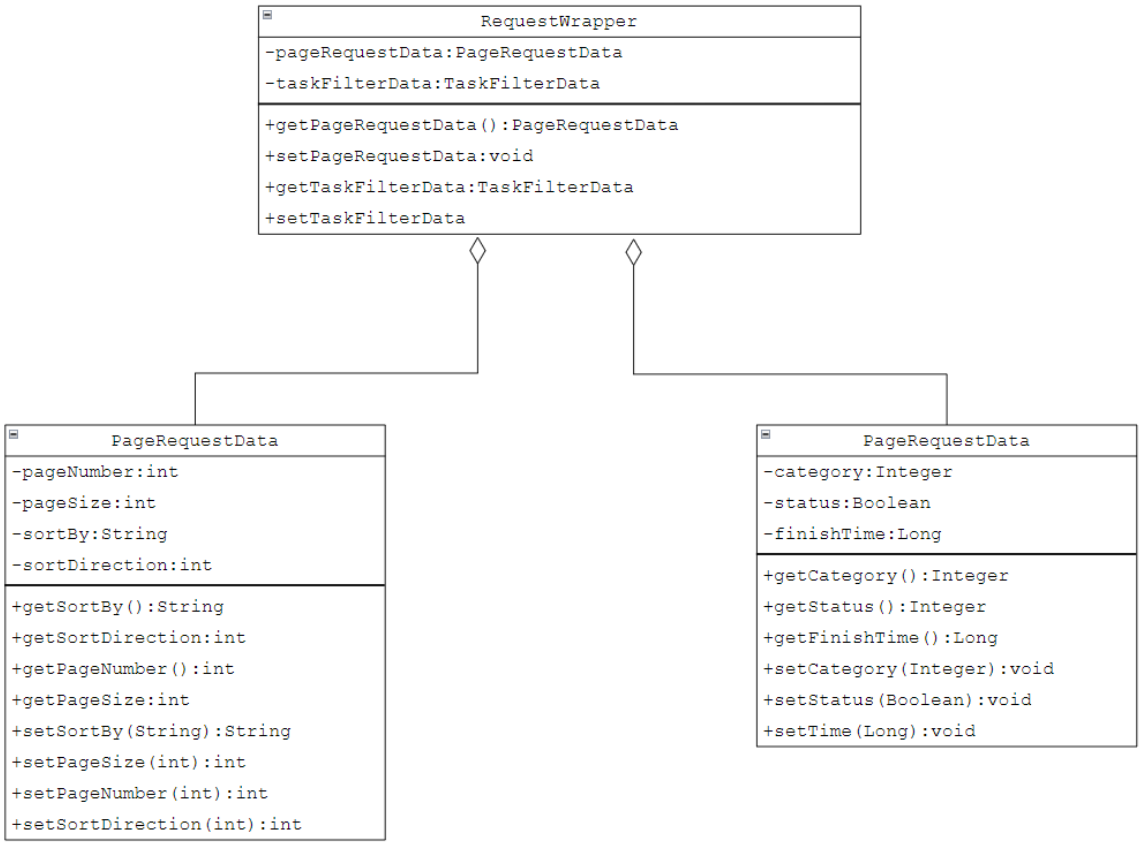
Ulazni točku za dohvaćanje stranice podataka predstavlja klasa `TaskController`. Klasa `TaskController` obrađuje nadolazeći HTTP Post zahtjev na rutu *„/task/getTaskPage“*. Bilješka `@PostMapping` unutar koje se nalazi završna točka rute (*engl. endpoint*), označava koju metodu pozvati kada programski okvir Spring primi zahtjev prema prethodno spomenutoj ruti. U slučaju promatrane aplikacije poziva se metoda `getTaskPage()`, koja kao parametar prima objekt tipa `RequestWrapper`, obilježen sa `@RequestBody`. Bilješka `@RequestBody` označava da se unutar tijela primljenog zahtijeva nalazi objekt tipa `RequestWrapper`, kojeg programski okvir Spring deserijalizira iz primljenog JSON oblika te pruža na korištenje unutar parametra `requestWrapper`. Osim spomenutog parametra na *Programski kod 11* kao parametar metode je još navedeno i polje token tipa *String*, koje je obilježeno sa `@RequestHeader`. Unutar klase `RequestWrapper` su sadržane klase `PageRequestData` i `TaskFilterData`, koje se instanciraju na temelju dobivenih podataka iz deserijaliziranog JSON tijela primljenog zahtijeva, detalji su prikazani na klasnom dijagramu sadržanom na *Slika 17*.

```
@RestController
@RequestMapping("/task")
public class TaskController {
    @Autowired
    TaskService taskService;
    @Autowired
    ResponseResolver responseResolver;


    @PostMapping("/getTaskPage")
    Response getTaskPage(@RequestBody RequestWrapper
requestWrapper, @RequestHeader(HttpHeaders.AUTHORIZATION)
String token){
        return
taskService.getTaskPage(requestWrapper.getPageRequestData(),req
uestWrapper.getTaskFilterData(),token);
    }
}
```

*Programski kod 11.* Prikaz metode `getTaskPage()` unutar klase `TaskController`

Unutar ovog polja se pohranjuje JWT token dobiven iz zaglavlja zahtjeva. Opisana metoda `getTasksPage()` klase `TaskController` vraća rezultate dobivene pozivanjem

metode `getTasksPage()` klase `TaskService`. Klasom `TaskService` upravlja programski okvir Spring, točnije IoC kontejner programskog okvira Spring, što znači da za pristup `TaskService` objektu dovoljno je definirati taj objekt kao programsku ovisnicu klase `TaskController`, te ga označiti sa bilješkom `@Autowired`. Navedena bilješka nalaže programskom okviru Spring, da prilikom stvaranja instance klase `TaskController`, automatski ubaci instancu objekta `TaskService`. Isto vrijedi i za klasu `ResponseResolver` također obilježene sa bilješkom `@Autowired`.




*Slika 17.* Prikaz klasnog dijagrama klase `RequestWrapper`

### 5.5.2. Apstraktna klasa Service

Unutar promatrane aplikacije osnovne funkcionalnosti koje su zajedničke implementacijama `Service` i `DataManager` apstraktnih klasa su čitanje, pisanje, uređivanje i brisanje podataka. Navedene funkcionalnosti su izražene unutar apstraktnih

metoda `DataManger` klase: `read()`, `write()`, `update()` i `delete()`. Spomenute metode implementira svaka implementacija klase `DataManager`. Zadaća `Service` klasa je izvlačenje identifikatora korisnika koji je poslao zahtjev prema serveru, i hvatanje eventualnih iznimki (*engl. exceptions*). Uz to još služi kao i posrednik između `Controller` i `DataManager` arhitektonskog sloja aplikacije. Iz navedenih razloga je stvorena apstraktna klasa `Service` koja unutar svoga konstruktora prima implementaciju klase `DataManager`, koja implementira definirane metode na način da ih poziva i unutar svake proslijedi potrebni parametar kako je prikazano na *Programski kod 12*.

```
@AllArgsConstructor
abstract  class Service<T extends Response> {
    protected final DataManager<T> dataSource;
    private final ResponseResolver responseResolver;
    private  final JwtService jwtService;
    public Response read(int id,String token){
        return catchError(()-> dataSource.read(id,
jwtService.extractId(token)));
    }

    public Response write(T param,String token){
        return catchError(()-> dataSource.write(param,
jwtService.extractId(token)));
    }

    public Response delete(int id,String token){
        return catchError(()-> dataSource.delete(id,
jwtService.extractId(token)) );
    }
    public Response update(T param,String token){
        return catchError(()-> dataSource.update(param,
jwtService.extractId(token)) );
    }

    protected Response catchError(CatchException function) {
        try {
            return function.catchError();
        } catch (Exception e) {
            return responseResolver.mapExceptionToFailure(e);
        }
    }

    protected int extractId(String token){
        return jwtService.extractId(token);
    }
```

*Programski kod 12*. Prikaz apstraktne klase `Service`

Funkcionalnost hvatanja grešaka se odvija kroz metodu `catchError()` koja kao parametar prima lambda funkciju. Spomenuta lambda funkcija je definirana kroz programsko sučelje `CatchException` prikazanog na *Programski kod 13*, te služi kao način prosljeđivanja metode koja bi se trebala izvesti i koja bi trebala vratiti vrijednost ili baciti `Exception` u slučaju greške metodi `catchError()` kao parametar funkcije. Izvlačenje korisničkog identifikatora iz JWT tokena se odvija uz pomoć metode `extractId()` klase `JwtService`, programski kod iste je prikazan na Programski kod 14. Spomenuta metoda je izložena za korištenje implementacijama klase `Service` kroz metodu `extractId()`.

```
@FunctionalInterface
interface CatchException{
    Response catchError() throws Exception;
}
```

*Programski kod 13*. Prikaz funkcijskog sučelja `CatchException`

```
public Integer extractId(String token) {
    String jwt = token.substring(7);
    final  Claims claims = extractAllClaims(jwt);

  return (Integer) claims.get("id");
}
```

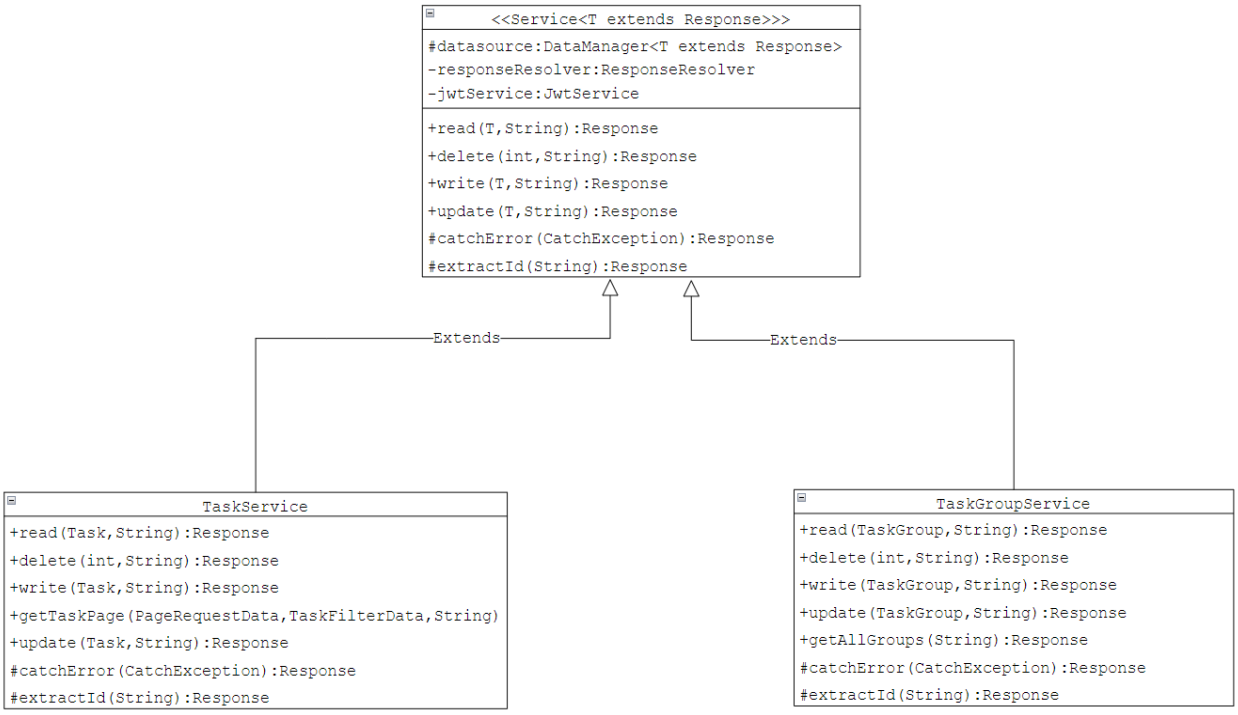
Programski kod 14. Prikaz metode za izvlačenje identifikatora korisnika iz JWT tokena, koja se nalazi unutar klase `JwtService`

### 5.5.3. Implementacija Service klase TaskService

Klasa `TaskService` je implementacija apstraktne klase `Service`, klasni dijagram je prikazan na *Slika 18*. Klasa `TaskService` koristi `TaskDataManager` implementaciju klase `DataManager`, što je izraženo kroz anotaciju `@TaskDataManagerQualifier`.

Spomenuta bilješka diktira mehanizmu IoC programskog okvira Spring, koju implementaciju klase `DataManager` ubaciti prilikom instanciranja `TaskService` klase. Za vraćanje stranice sa zadatcima korisni se `getTaskPage()` metoda klase `TaskService` koja zauzvrat poziva istoimenu metodu klase `TaskDataManager`. Na Programski kod 15. možemo primijetiti da se za izvlačenje identifikatora korisnika poziva prethodno spomenuta metoda `extractId()` klase `Service`. Metoda `getTaskPage()`

klase `TaskDataManager` ne stvara iznimke, te stoga nije potreban mehanizam hvatanja istih.




*Slika 18.* Prikaz klasnog dijagrama implemmentacija klase `Service`

```
@Service
public class TaskService extends
com.daily_todo.service.Service<Task> {


    @Autowired
    public TaskService(@TaskDataManagerQualifier
DataManager<Task> dataSource, ResponseResolver
responseResolver, JwtService jwtService) {
        super(dataSource, responseResolver, jwtService);
    }


    public Response getTaskPage(PageRequestData
pageRequestData, TaskFilterData taskFilterData, String token){
       TaskDataManager taskDataManager = (TaskDataManager)
dataSource;
       return new TaskPage(((TaskDataManager)
dataSource).getTasksPage(pageRequestData,taskFilterData,extrac
```

*Programski kod 15*. Prikaz programskog koda klase TaskService

Zadaća klase `TaskDataManager` je manipulacija `Task` entitetom, odnosno implementacija funkcionalnosti čitanja, pisanja, uređivanja i brisanja. To se obavlja kroz implementaciju apstraktne klase `DataManager`, koja kao što je spomenuto u prethodnom odlomku sadrži definicije metoda koje `TaskDataManager` implementira. Pristup podatcima u slučaju dohvaćanja stranice sa zadatcima se izvode kroz metodu `getTasksPage()`, koja pristupa podatcima putem sučelja `EntityManager`. `EntityManager` je dio implementacije standarda JPA unutar programskog okvira Spring. Kako bi se implementirala mogućnost filtriranja sadržaja baze podataka koristi se metoda `criteriaQuery.where()`, unutar metode `getTasksPage()`, kao što je prikazano na *Programski kod 16*.

```
public List<Task> getTasksPage(PageRequestData pageRequestData,
TaskFilterData taskFilterData,Integer userId) {

    CriteriaQuery<Task> criteriaQuery =
criteriaBuilder.createQuery(Task.class);
    Root<Task> taskRoot = criteriaQuery.from(Task.class);
    Predicate predicate =
getPredicate(taskFilterData,taskRoot,userId);
    criteriaQuery.where(predicate);
    setOrder(pageRequestData,criteriaQuery,taskRoot);
    TypedQuery<Task> typedQuery =
entityManager.createQuery(criteriaQuery);
    typedQuery.setFirstResult(pageRequestData.getPageNumber() *
pageRequestData.getPageSize());
    typedQuery.setMaxResults(pageRequestData.getPageSize());
```

*Programski kod 16.* Prikaz metode `getTasksPage()` klase `TaskDataManager`, Izvor: https://github.com/codeforgeyt/jpa-paging-sorting-filtering/blob/main/src/main/java/com/codeforgeyt/jpapagingsortingfiltering/repository/EmployeeCriteriaRepository.java

Postupak definiranja filtra započinje pozivom `getPredicate()` metode, koja na temelju podataka dobivenih unutar objekta *taskFilterData* stvara objekt `Predicate`. Proces stvaranja objekta `Predicate` unutar metode `getPredicate()` započinje definiranjem objekta *predicates* tipa `List<Predicate>`, te provjeravanjem pojedinačnih parametara objekta `TaskFilterData`. Provjerava se za *null* vrijednosti parametara, parametri koji su prazni se ne dodavaju u *predicates* polje. Ukoliko neki parametar nije prazan poziva se metoda `predicates.add()`, koja dodaje novi predikat pozivanjem neke od metoda za usporedbu *criteriaBulder* objekta, kao što je prikazano na *Programski kod 17*. Nakon što su sva polja

provjerena, polje *predicate* se sjedinjuje u jedan objekt pozivanjem metode *and* objekta *criteriaBuilder* nad *predicates* objektom tipa `List<Predicate>`.

```
    private Predicate getPredicate(TaskFilterData taskFilterData,
    Root<Task> taskRoot,Integer userId) {

        List<Predicate> predicates = new ArrayList<>();

    predicates.add(criteriaBuilder.equal(taskRoot.get("userId"),use
    rId));
        if (taskFilterData.getStatus() != null){


    predicates.add(criteriaBuilder.equal(taskRoot.get("isCompleted"
    ),taskFilterData.getStatus()));
        }
        if (taskFilterData.getFinishTime() != null ){

    predicates.add(criteriaBuilder.lessThanOrEqualTo(taskRoot.get("
    finishTime"),taskFilterData.getFinishTime()));
        }
        if (taskFilterData.getCategory() != null){


    predicates.add(criteriaBuilder.equal(taskRoot.get("taskGroup").
    get("id"),taskFilterData.getCategory()));
        }
        return  criteriaBuilder.and(predicates.toArray(new
```

*Programski kod 17*. Prikaz metode `getPredicate()` klase `TaskDataManager`, Izvor: https://github.com/codeforgeyt/jpa-paging-sorting-filtering/blob/main/src/main/java/com/codeforgeyt/jpapagingsortingfiltering/repository/EmployeeCriteriaRepository.java

Nadalje se poziva metoda `setOrder()` koja na temelju podataka iz *pageRequestData* objekta, tj. polja *sortDirection* i *sortBy* izvodi sortiranje dobivenih podataka, kao što je prikazano na *Programski kod 18*. Postavljanje podataka o paginaciji se izvodi pozivanjem metode `typedQuery.setFirstResult()` koja na temelju broja stranice i veličine stranice određenih od strane `getPageNumber()` i `getPageSize()` gettera objekta *pageRequestData* određuje koji će podatci biti sadržani u trenutnoj stranici. Metoda `typedQuery.setMaxResults()` zatim određuje broj podataka koji će biti sadržani na „stranici“ zadataka koja će se vratiti kao odgovor na poziv metode. Na kraju `typedQuery.getResultList()` vraća polje na temelju upita stvorenog pomoću

objekata *entityManager* i *typedQuery*.

```
private void setOrder(PageRequestData pageRequestData,
CriteriaQuery<Task> criteriaQuery, Root<Task> taskRoot) {
    if (pageRequestData.getSortDirection() == 1){

criteriaQuery.orderBy(criteriaBuilder.asc(taskRoot.get(pag
eRequestData.getSortBy())));
    }
   else {

criteriaQuery.orderBy(criteriaBuilder.desc(taskRoot.get(pa
geRequestData.getSortBy())));
    }
}
```

Programski kod 18. Prikaz metode `setOrder()` klase `TaskDataManager`, Izvor: https://github.com/codeforgeyt/jpa-paging-sorting-filtering/blob/main/src/main/java/com/codeforgeyt/jpapagingsortingfiltering/repository/EmployeeCriteriaRepository.java

### 5.5.5. Serijalizacija i Deserijalizacija `Task` entiteta

Serijalizacija i deserijalizacija entiteta odnosno objekata unutar programskog okvira Spring se obično odvaja automatski upotrebom Jackson *View* serijalizacijske tehnologije. Ovaj pristup stvara problem unutar promatrane aplikacije, pošto se oblici podataka klase `Task` razlikuju od oblika podatka koji bi trebali biti vraćeni u JSON formatu. Točnije polje „*finishedTime“* je tipa `Long`, dok ga klijent zahtijeva u obliku datuma izraženog kao formatirani `string`, uz to polja *isCompleted* i *recurring* su deklarirana u tipu `integer`-a, dok ih klijent zahtijeva u `bool` formatu, za referencu vidi *Programski kod 19* i *Slika 16*. Ovaj problem može biti riješen definiranjem vlastitih serializatora, pogledaj *Programski kod 20*. Stvaranje serializatora počinje stvaranjem klase koja se nasljeđuje od klase `StdSerializer`. Unutar naslijeđene klase, implementira se metoda `serialize()`. Spomenuta metoda pruža pristup varijabli `jsonGenetrator` tipa `JsonGenerator` koju koristimo kako bi sastavili JSON odgovor. Proces se sastoji od definiranja naziva polja, pridruživanja odgovarajućih polja `Task` objekta, te pozivanja odgovarajućih metoda, kao što je prikazano na *Programski kod 20*. Kad je serializator definiran, pridružuje se odgovarajućem entitetu, odnosno klasi upotrebom bilješke `@JsonSerialize` unutar koje navodimo tip sterilizatora kojeg koristimo kao što je prikazano na *Programski kod 19*.

```
@Entity(name = "task")
@Table(name = "task")
@JsonDeserialize(using = TaskDeserializer.class)
@JsonSerialize(using = TaskSerializer.class)
@NoArgsConstructor

public class Task  extends  Response{

    @Id
    @GeneratedValue(
            strategy = GenerationType.IDENTITY
    )

    @Column(name = "id")
    private Integer id;

    @JsonProperty("isCompleted")
    @Column(name = "completed")
    @Nullable
    private  Integer isCompleted;
    @Column(name = "created_time")

     private Long createdTime;
    @Column(name = "finish_date")
    @Nullable
    private Long finishTime;
    @Nullable
    @JoinColumn(name = "category_id",referencedColumnName =
"id")
    @ManyToOne(cascade = CascadeType.DETACH)
    private TaskGroup taskGroup;
    @Column(name = "description")
    @Nullable
    private String description;
    @Column(name = "task_title")
    @Nullable
    private String title;
    @Column(name = "recurring")
    @Nullable
    private Integer recurring;
    @JsonIgnore
    @Column(name = "user_id")
    private Integer userId;

    // getteri setteri i konstruktori
```

*Programski kod 19.* Prikaz klase `Task`

Analogno tomu, prilikom postupka deserializacije javlja se isti problem radi kog je serializator i rađen, pojedina polja zahtjeva imaju različiti tip podatka od onog koji je definiran

unutar `Task` klase, što znači da se mora stvoriti nova implementacija deserializatora. Prednost vlastitog deserializatora nad onim kojeg stvori alat Jackson, je taj da možemo definirati vlastitu logiku pretvorbe određenih polja. Proces definicije deserializatora je relativno sličan procesu definiranja serializatora. Postupak započinje nasljeđivanjem klase `StdDeserializer`, te se implementira metoda `deserialize()`, koja daje pristup instanci klase `JsonNode` pomoću koje se pristupa pojedinim poljima primljenog JSON objekta. Vrijednosti polja JSON objekta se spremaju u varijable kako je i prikazano na *Programski kod 21.* Jednom kad su sve varijable definirane, na temelju istih se stvara objekt tipa `Task` koji se zatim vraća kao rezultat poziva metode. Daljnji proces pridruženja deserializera klasi `Task` je sličan procesu pridruženja serializera. Željena klasa se označi bilješkom `@JsonDeserialize`, te se u njega unese tip deserializera kojeg se koristi, kao što je prikazano i na *Programski kod 19.*

```
public class TaskSerializer extends StdSerializer<Task> {
    public TaskSerializer(Class<Task> t) {
        super(t);
    }

    public TaskSerializer(){
        this(null);
    }


    @Override
    public void serialize(Task task, JsonGenerator
jsonGenerator, SerializerProvider serializerProvider) throws
IOException {
        jsonGenerator.writeStartObject();
        jsonGenerator.writeNumberField("id", task.getId());

jsonGenerator.writeStringField("title",task.getTitle());

jsonGenerator.writeStringField("taskGroup",task.getTaskGroupI
nfo().getTitle());

jsonGenerator.writeStringField("description",task.getDescript
ion());
        jsonGenerator.writeBooleanField("completed",
task.getCompleted());
```

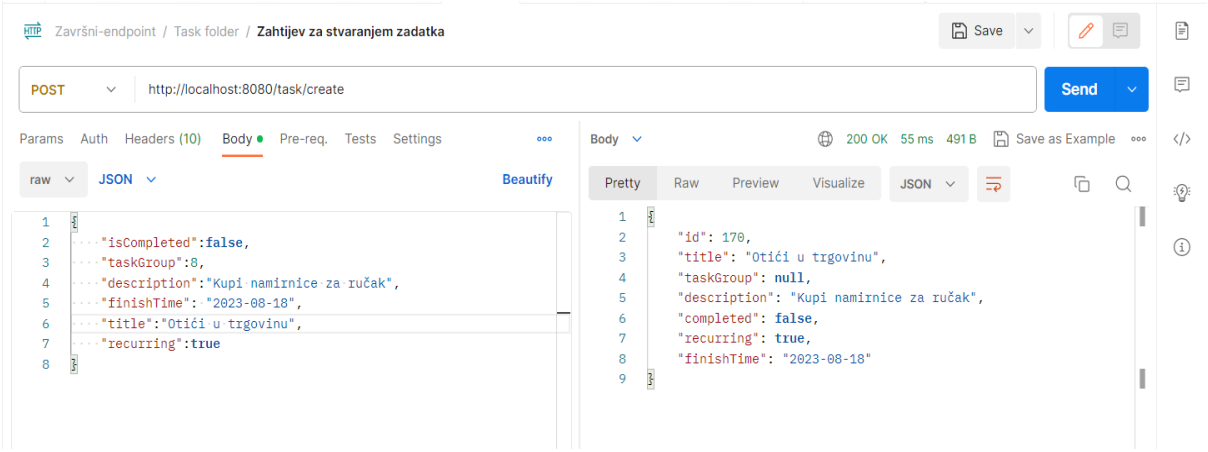
*Programski kod 20.* Prikaz implementacije klase za serijalizaciju objekta `Task`

```
public class TaskDeserializer extends StdDeserializer<Task> {
    public TaskDeserializer(Class<?> vc) {
        super(vc);
    }

    public TaskDeserializer(){
        this(null);
    }

    @Override
    public Task deserialize(JsonParser jsonParser,
DeserializationContext deserializationContext) throws
IOException, JacksonException {
        JsonNode node =
jsonParser.getCodec().readTree(jsonParser);


        String title = node.get("title").isNull() ? null :
node.get("title").asText();
        String description = node.get("description").isNull() ?
null : node.get("description").asText();
        Integer taskGroupId = node.get("taskGroup").isNull() ?
null : (Integer) ((IntNode)
node.get("taskGroup")).numberValue();
        String finishTime = node.get("finishTime").isNull() ?
null : node.get("finishTime").asText();
        boolean isCompleted = !node.get("isCompleted").isNull()
&& (Boolean) ((BooleanNode)
node.get("isCompleted")).booleanValue();
        boolean recurring = !node.get("recurring").isNull() &&
(Boolean) ((BooleanNode) node.get("recurring")).booleanValue();
        TaskGroup taskGroup = new TaskGroup(taskGroupId);
        int recurringId = recurring ? 1 : 0;
        int isCompletedId = isCompleted ? 1 : 0;
        if (!node.has("id")){
            return new Task(isCompletedId,
DateConverter.fromStringToMillis(finishTime),taskGroup,descript
ion,title,recurringId,Instant.now().toEpochMilli());
        }
        else {
           Integer id = node.get("id").isNull() ? null :
(Integer) ((IntNode) node.get("id")).numberValue();
            return new Task(id,isCompletedId,
DateConverter.fromStringToMillis(finishTime),taskGroup,descript
ion,title,recurringId,Instant.now().toEpochMilli());
        }
    }
```

Programski kod 21. Prikaz implementacije klase za deserijalizaciju JSON objekta

## 5.6. Značajka dodavanja zadatka

Postupak dodavanja novog zadatka započinje slanjem HTTP Post zahtijeva na rutu „*task/create*“. Unutar tijela zahtijeva sadržane su informacije o zadatku koji se pokušava stvoriti kao što je prikazano na *Slika 19*. Ako je zadatak uspješno stvoren, vraća se HTTP odgovor sa zadatkom koji se poslao unutar tijela zahtijeva za stvaranjem zadatka.




*Slika 19*. Primjer zahtijeva za stvaranjem zadatka korištenjem alata Postman

### 5.6.1. TaskController

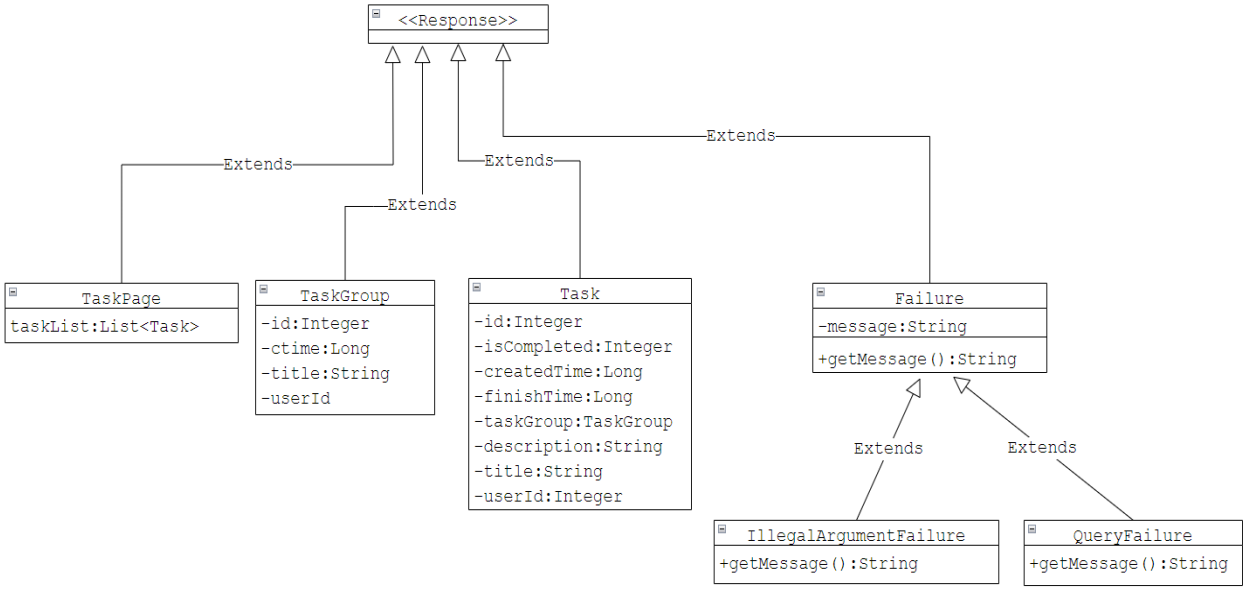
Unutar klase `TaskController`, kao i na prethodno opisanoj funkcionalnosti dohvaćanja stranice sa zadatcima, primljeni HTTP zahtjev se mapira na metodu `saveTask()`, prikazanoj na *Programski kod 22*.

```
@PostMapping("/create")
ResponseEntity<Response> saveTask(@RequestBody Task task,
@RequestHeader(HttpHeaders.AUTHORIZATION) String token) throws
Exception {
    Response response = taskService.write(task,token);
    return responseResolver.evaluate(response);
}
```

Programski kod 22. Prikaz metode `saveTask()` klase `TaskController`

Nakon poziva metode `saveTask()`, poziva se metoda `write()` klase `TaskService`. Te se odgovor te metode preko varijable *response* prenosi metodi `evaluate()` klase `ResponseResolver`. Koja na temelju povratnog tipa metode

`write()`, vraća objekt tipa `ResponseEntity` sa pripadajućim statusnim kodom. Važno je istaknuti da je unutar promatrane aplikacije, klasa `Response` ishodišna klasa od koje su naslijeđene sve klase koje se mogu vratiti kao odgovor, kao što je prikazano na *Slika 20.*




Slika 20. Klasni dijagram povratnih tipova promatrane aplikacije

### 5.6.2. ResponseEntity

Klasa `ResponseEntity` se sastoj od dvije metode: `evaluate()` i `mapExceptionToFailure()`. Metoda `evaluate` služi mapiranju funkcijskog parametra *response* na odgovarajući objekt `ResponseEntity` sa prikladnim HTTP statusnim kodom kako je prikazano na *Programski kod 23*, te se koristi unutar `Controller` klasa za mapiranje dobivenih rezultata od strane poziva metoda klase `Service`. Dok metoda `mapExceptionToFailure` mapira objekt tipa `Exception`, dobiven kroz poziv funkcije na odgovarajući objekt tipa `Failure`. Zadaća ove metode je obrada iznimki koje se dogode tokom izvođenja aplikacije, te na taj način sprječavanje nepredviđenog ponašanja iste. Koristi se isključivo u klasi `Service` kao način odgovora na uhvaćene iznimke. Programski kod je prikazan na *Programski kod 24*.

```
public ResponseEntity<Response> evaluate(Response response){
    if (response instanceof Failure){
        if (response instanceof QueryFailure) {
            return status(HttpStatus.NOT_FOUND).body(new
QueryFailure(((QueryFailure) response).getMessage()));
        }
        else if (response instanceof IllegalArgumentFailure){
            return status(HttpStatus.BAD_REQUEST).body(new
IllegalArgumentFailure(((IllegalArgumentFailure)
response).getMessage()));
        }
        else {
            return status(HttpStatus.BAD_REQUEST).body(new
QueryFailure(((Failure) response).getMessage()));
        }


    }
    return status(HttpStatus.OK).body(response);
 }
```

*Programski kod 23*. Prikaz metode `evaluate()` sadržane unutar klase `ResponseEntity`

```
public Failure mapExceptionToFailure(Exception exception){
   if (exception instanceof QueryException){
       return new QueryFailure(exception.getMessage());
   }
   else if(exception instanceof IllegalArgumentException){
       return new
IllegalArgumentFailure(exception.getMessage());
   } else if (exception instanceof SQLException) {
       return new Failure(exception.getMessage());
   }


    return new Failure(exception.getMessage());
}
```

*Programski kod 24*. Prikaz metode `mapExceptionToFailure()` klase `ResponseResolver`

### 5.6.3. TaskDataManager

Funkciju spremanja zadatka obavlja metoda `write()` unutar klase `TaskDataManager`. Prvi korak kod spremanja zadatka je dodjeljivanje korisnikovog id-a zadatku. Zatim se obavlja provjera dostupnosti identifikatora `TaskGroup` entiteta, ukoliko je identifikator nije dostupan, kao identifikator grupe se dodjeljuje brij 14 koji označava zadatke koji nisu dodijeljeni ni jednoj grupi, kao što je prikazano na *Programski kod 25*.

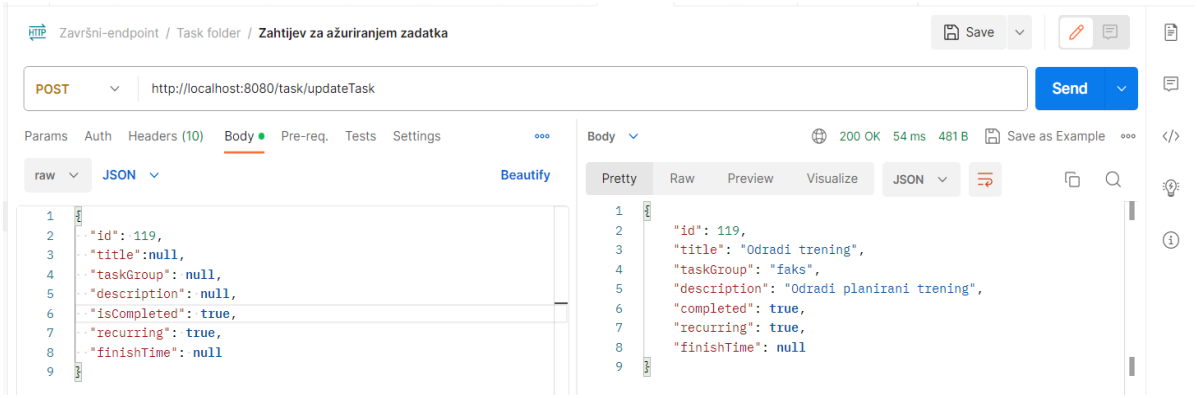
Ukoliko je identifikator grupe sadržan, provjerava se dali ta grupa postoji i dali korisnik koji je pokušava dodijeliti ima ovlast pristupa grupi, to se obavlja pozivanjem metode `findByUserIdAndId()` implementacije `TaskGroupRepository` sučelja. Ukoliko je grupa dostupna, zadatak se sprema u bazu podataka pozivanjem metode `save()` implementacije sučelja TaskRepository. Ukoliko grupa nije dostupna stvara se nova instanca iznimke `IllegalArgumnetException()`, koju zatim `TaskService` hvata i dalje obrađuje na način pojašnjen u prijašnjem potpoglavlju. Programski kod metode `write` je prikazan na *Programski kod 25*.

```
public Task write(Task task,int userId) throws Exception{
    task.setUserId(userId);
     if(task.getTaskGroupInfo().getId() == null){
         task.setTaskGroup(14);
        return taskRepository.save(task);
      }
     else {
         Optional<TaskGroup> tempTaskGroup =
taskGroupRepository.findByUserIdAndId(userId,task.getTaskGr
oupInfo().getId());
         if (tempTaskGroup.isPresent()){
             return taskRepository.save(task);
         }
         throw new IllegalArgumentException("TaskGroup with
given Id doesn't exist!");
     }
}
```

*Programski kod 25*. Prikaz metode write() klase TaskDataManager

### 5.7. Značajka uređivanja zadatka

Zahtjev za uređivanjem zadatka počinje slanjem objekta zadatka kojeg se želi urediti u JSON obliku putem HTTP Post zahtijeva na „*task/updateTask“* rutu. Tijelo zahtijeva je slično onome za stvaranje, no razlikuje se u tome što se unose vrijednosti samo onih polja čije vrijednosti korisnik kani promijeniti, te naravno s uključenim poljem identifikatora, koje ne smije biti prazno. Ako je ažuriranje zadatka prošlo uspješno, vraća se objekt zadatka s ažuriranim vrijednostima, te HTTP statusnim kodom 200 . Primjer slanja zahtijeva sa odgovorom pomoću alata Postman se nalazi na *Slika 21*.




*Slika 21.* Primjer slanja zahtijeva za ažuriranje zadatka korištenjem alata Postman

### 5.7.1. TaskController

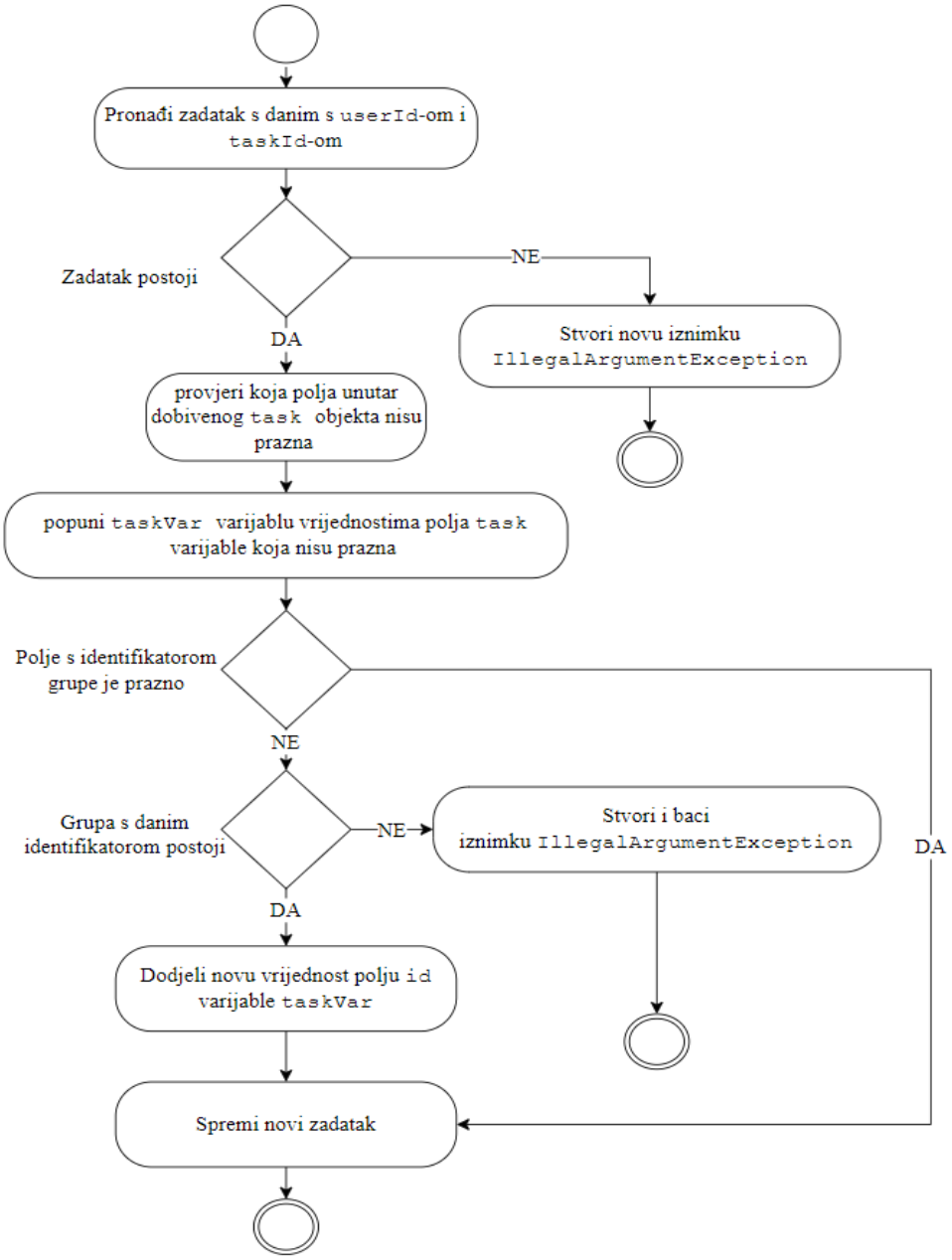
Ažuriranje zadatka počinje pozivanjem `updateTask()` metode klase `TaskController`. Metoda `updateTask()` zauzvrat poziva metodu `update()` klase `TaskService`, te rezultat prosljeđuje metodi `evaluate()` klase `ResponseResolver` na daljnju obradu. Metoda obrade zahtijeva unutar klase `ResponseResolver` je raspravljena u prethodnom potpoglavlju. Programski kod metode `updateTask()` je prikazan na *Programski kod 27* .

```
@PostMapping("/updateTask")
ResponseEntity<Response> updateTask(@RequestBody Task task ,
@RequestHeader(HttpHeaders.AUTHORIZATION) String token){
    Response response = taskService.update(task,token);
    return responseResolver.evaluate(response);
}
```

Programski kod 26. Prikaz `updateTask()` metode klase `TaskController`

### 5.7.2. TaskDataManager

Daljnja logika obrade zahtijeva za ažuriranjem zadatka odvija se unutar metode `update()` klase `TaskDataManager` koja je prikazana na *Programski kod 27*. Samo zapisivanje i ažuriranje vrijednosti zadatka obavlja metoda `save()` implementacije sučelja `TaskRepository`, detaljan prikaz logike ažuriranja zadatka je prikazan na *Slika 22*



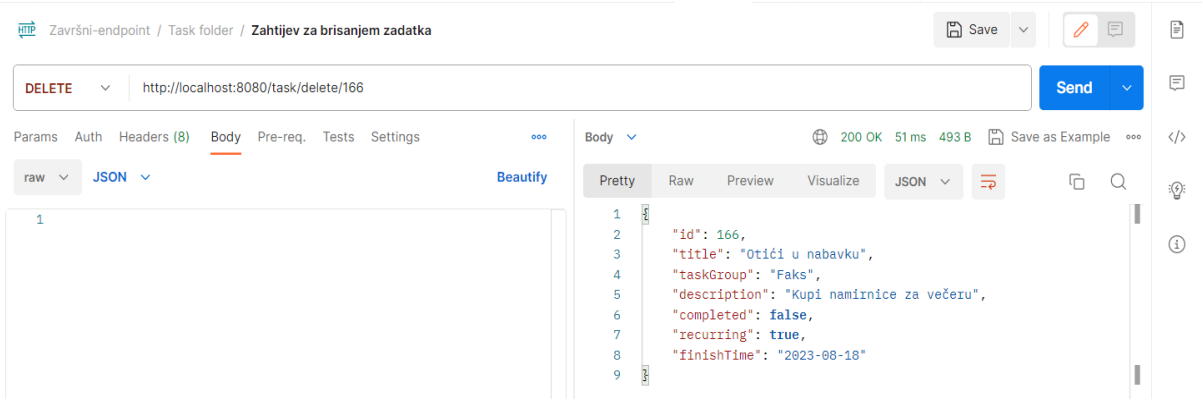

*Slika 22.* Dijagram toka ažuriranja zadatka pozivanjem metode `update()`

```
@Override
public Task update(Task task,int userId) throws Exception{
   Optional<Task> tempTask =
taskRepository.findByIdAndUserId(task.getId(),userId);
   if (tempTask.isPresent()){
       Task taskVar = tempTask.get();
      if (task.getCompleted() != null){
           taskVar.setIsCompleted(task.getCompleted());
       }
      if(task.getRecurring() != null){
           taskVar.setRecurring(task.getRecurring());
       }
      if (task.getDescription() != null){
           taskVar.setDescription(task.getDescription());
       }
       if(task.getTitle() != null){
           taskVar.setTitle(task.getTitle());

       }
       if (task.getTaskGroupInfo().getId() != null){
           Optional<TaskGroup> tempTaskGroup =
taskGroupRepository.findByUserIdAndId(userId,task.getTaskGroupI
nfo().getId());
           if (tempTaskGroup.isPresent()) {
               taskVar.setTaskGroupObject(new
TaskGroup(task.getTaskGroupInfo().getId()));
           }
           else {
               throw new IllegalArgumentException("That group
doesn't exist!");
           }
       }
       if (task.getFinishTime() != null){

taskVar.setFinishTime(DateConverter.fromMilisToStringFormat(tas
k.getFinishTimeMilis()));
       }

       taskVar.setUserId(userId);

    return taskRepository.save(taskVar);

   }
    else {
        throw new IllegalArgumentException("Incorrect task
id!");
   }
```

*Programski kod 27.* Prikaz metode `update()` klase `TaskDataManager`

### 5.8. Značajka brisanja zadatka

Zahtjev za brisanjem zadatka se obavlja slanjem HTTP Delete zahtijeva na rutu „*task/delete/{id}*“ gdje „id“ predstavlja identifikator zadatka kojeg se nastoji obrisati. Ukoliko je zahtjev uspješno proveden, poslužitelj će vratiti odgovor sa izbrisanim zadatkom i HTTP statusnim kodom 200, vidi primjer *Slika 23*.




Slika 23. Zahtjev za brisanjem zadatka sa identifikatorom 166, te prikaz dobivenog odgovora

Kao i u prethodnim primjerima, postupak brisanja zadatka počinje u `TaskController`-u mapiranjem zahtijeva na metodu `deleteTask()`. Za mapiranje se u ovome slučaju koristi bilješka `@DeleteMapping` kako bi se naglasilo kako se očekuje HTTP Delete zahtjev. Umjesto iz tijela zahtijeva kao u prijašnjim primjerima, id zadatka se dobiva iz varijable putanje (*engl. Path variable*), te kako bi označili da se parametar funkcije id dobiva iz varijable putanje, označavamo ga sa bilješkom `@PathVariable`, vidi *Programski kod 28*.

```
@DeleteMapping("/delete/{id}")
ResponseEntity<Response> deleteTask(@PathVariable("id") int
id, @RequestHeader(HttpHeaders.AUTHORIZATION) String token){
    Response response = taskService.delete(id,token);
    return responseResolver.evaluate(response);
}
```

*Programski kod 28*. Prikaz programskog koda metode deleteTask() klase TaskController

Proces brisanja zadatka je poprilično jednostavan, te se svodi na pozivanje metode `delete()` klase `TaskDataManager`. Prvi korak je provjera dali postoji zadatak koji se pokušava izbrisati, odnosno dali korisnik ima ovlasti brisanja istog. Ukoliko ima, poziva se

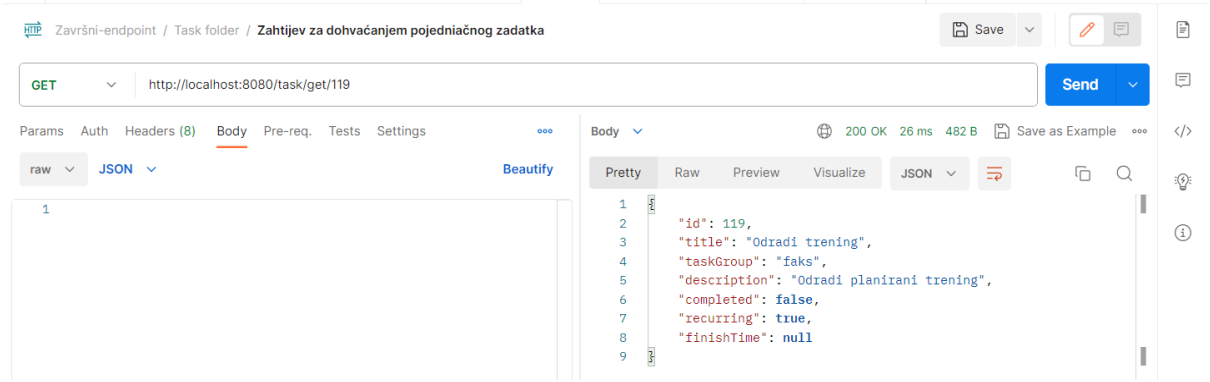
metoda `deleteById()` implementacije sučelja `TaskRepository`. No ukoliko se dogodila pogreška tokom pokušaja brisanja stvara se nova `QueryException` iznimka, koju dalje obrađuje nadležna `Service` klasa, vidi *Programski kod 29*.

```
@Override
public Task delete(int id,int userId) throws Exception{
    Optional<Task> task
=taskRepository.findByIdAndUserId(id,userId);
    if (task.isPresent())
    {
        taskRepository.deleteById(id);
        return task.get();
    }
    throw new QueryException("There is no Task with given
id!");
```

*Programski kod 29*. Prikaz metode `delete()` klase `TaskDataManager`

### 5.9. Značajka dohvaćanja pojedinačnog zadatka

Za dohvaćanje pojedinačnog zadatka, klijent šalje poslužitelju HTTP Get zahtjev na rutu „*task/get/{id}*, gdje je id, predstavlja identifikator zadatka kojem se pristupa. Ukoliko je zahtjev prema poslužitelju uspješan, klijet prima `Task` objekt u JSON formatu, te statusni kod 200, kao na *Slika 24*.




*Slika 24*. Primjer zahtjeva za dohvaćanjem zadatka sa odgovorom poslužitelja korištenjem alata Postman

Ulaznu točku obrade zahtjeva za dohvaćanjem pojedine aktivnosti predstavlja `getTask()` metoda klase `TaskController`, koja je na nadolazeći zahtjev mapirana pomoću bilješke `@GetMapping`, prikazano na Programski *kod 30* .

```
@GetMapping("/get/{id}")
ResponseEntity<Response> getTask(@PathVariable("id") Integer
id, @RequestHeader(HttpHeaders.AUTHORIZATION) String token){
    Response response = taskService.read(id,token);
    return responseResolver.evaluate(response);
}
```

Programski kod 30. Prikaz metode `getTask()` unutar klase `TaskController`

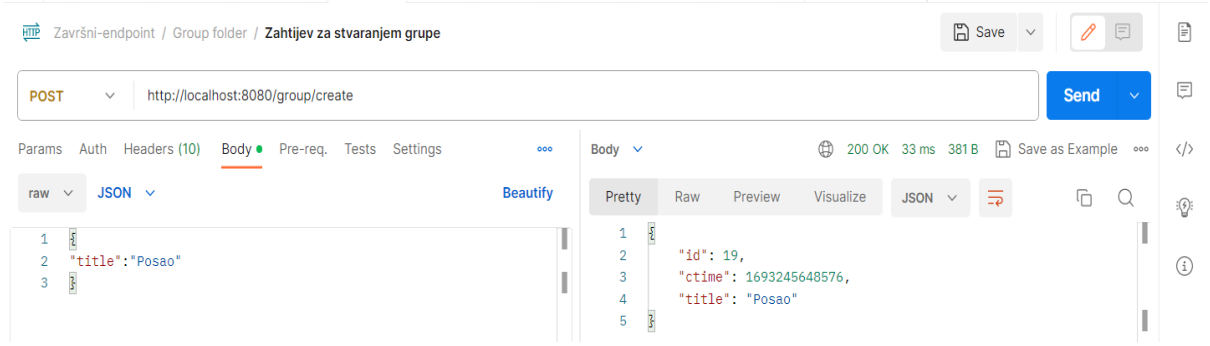
Nadalje, zahtjev se delegira preko metode `read()` klase `TaskService` na `TaskDataManager` klasu koja dohvaća traženi zadatak preko metode `read()`. Metoda *read* poziva metodu `findByIdAndUserId()` sučelja `TaskRepository`, koja traži zapis zadatka unutar baze podataka koji odgovara pruženom identifikatoru korisnika i identifikatoru zadatka. Ukoliko nađe zadatak koji odgovara danim parametrima vraća ga na klasu `TaskController` preko klase `TaskService`. Rezultat neuspješnog zahtjeva proizvodi `QueryException` prikazan na *Programski kod 31*.

```
@Override
public Task read(int id,int userId) throws  Exception{
    Optional<Task> taskTemp =
taskRepository.findByIdAndUserId(id,userId);
    if (taskTemp.isPresent()){
        return taskTemp.get();
    }
    else {
        throw new QueryException("Desired Task not found!");
    }
}
```

*Programski kod 31.* Prikaz implementacije `read()` metode klase `TaskDataManager`

## 5.10. Značajka stvaranja grupe zadataka

Osim stvaranja zadataka, korisnik ima i opciju stvaranja grupe zadataka. Ovo se postiže slanjem HTTP Post zahtjeva na rutu *„/group/create“,* gdje u tijelu zahtjeva navodi ime grupe koje želi kreirati. Ukoliko je zahtjev uspješno obrađen, poslužitelj vraća JSON format `TaskGroup` objekta, unutar kojeg se nalazi identifikator grupe, naziv grupe, te vrijeme stvaranja iste izražene u milisekundama od epohe, vidi *Slika 25*. Obrada zahtijeva započinje pozivanjem metode `createTaskGroup()` klase `TaskGroupController`, koja je mapirana na „*/create“* rutu korištenjem bilješke `@PostMapping`.




*Slika 25*. Prikaz zahtijeva za stvaranjem grupe poslanog prema poslužitelju korištenjem alata Postman

```
@PostMapping("/create")
ResponseEntity<Response> createTaskGroup(@RequestBody TaskGroup
taskGroup, @RequestHeader(HttpHeaders.AUTHORIZATION) String
token){
    Response writeResponse =
taskGroupService.write(taskGroup,token);
    return responseResolver.evaluate(writeResponse);
}
```

*Programski kod 32*. Prikaz metode createTaskGroup() klase TaskGroupController

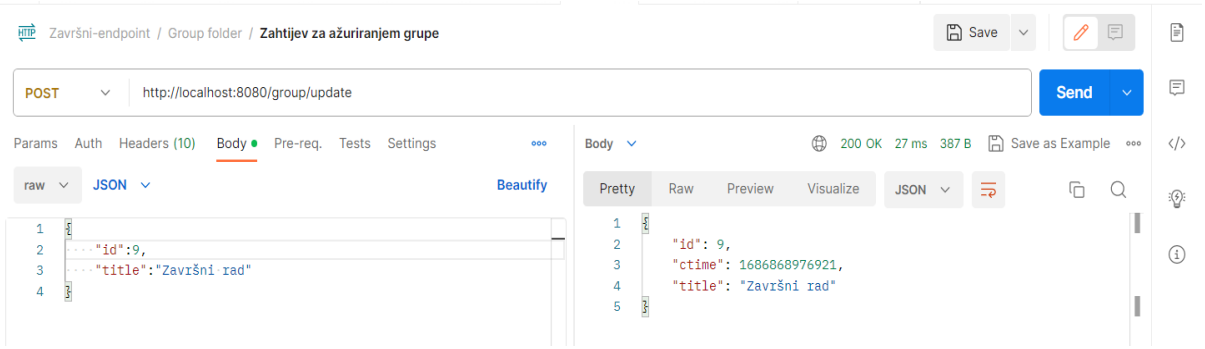
Metoda `write()` klase `TaskGroupDataManager`, pridružuje korisnički identifikator dobivenom objektu tipa `TaskGroup`, te poziva metodu `save()` sučelja `TaskGroupRepository`. U slučaju neuspjelog spremanja try-catch klauza hvata eventualnu nastalu grešku, te stvara novu iznimku tipa `QueryException`. Prilikom uspješnog spremanja `TaskGroup` objekta u bazu podataka, isti je vraćen kao rezultat metode, vidi *Programski kod 33*.

```
@Override
public TaskGroup write(TaskGroup param,int userId) throws
Exception{
    try {
        param.setUserId(userId);
        return taskGroupRepository.save(param);
    } catch (Exception e){
        throw new QueryException("Saving object failed!");
    }
}
```

*Programski kod 33*. Prikaz implementacije metode `write()` klase `TaskGroupDataManager`

## 5.11. Značajka uređivanja grupe

Značajka uređivanja grupe se sastoji isključivo od promjene naslova stvorene grupe zadataka. Proces započinje slanjem HTTP Post zahtijeva na rutu „*group/update*“. Unutar tijela zahtijeva se stavlja identifikator grupe zadataka kojoj se namjerava promijeniti naslov, te sam naslov. Ukoliko je zahtjev uspješno obrađen, klijentu se vraća novi objekt tipa `TaskGroup` u JSON formatu, s novim naslovom, vidi *Slika 26*.




*Slika 26*. Primjer zahtijeva za ažuriranjem grupe zadataka poslanog putem alata Postman

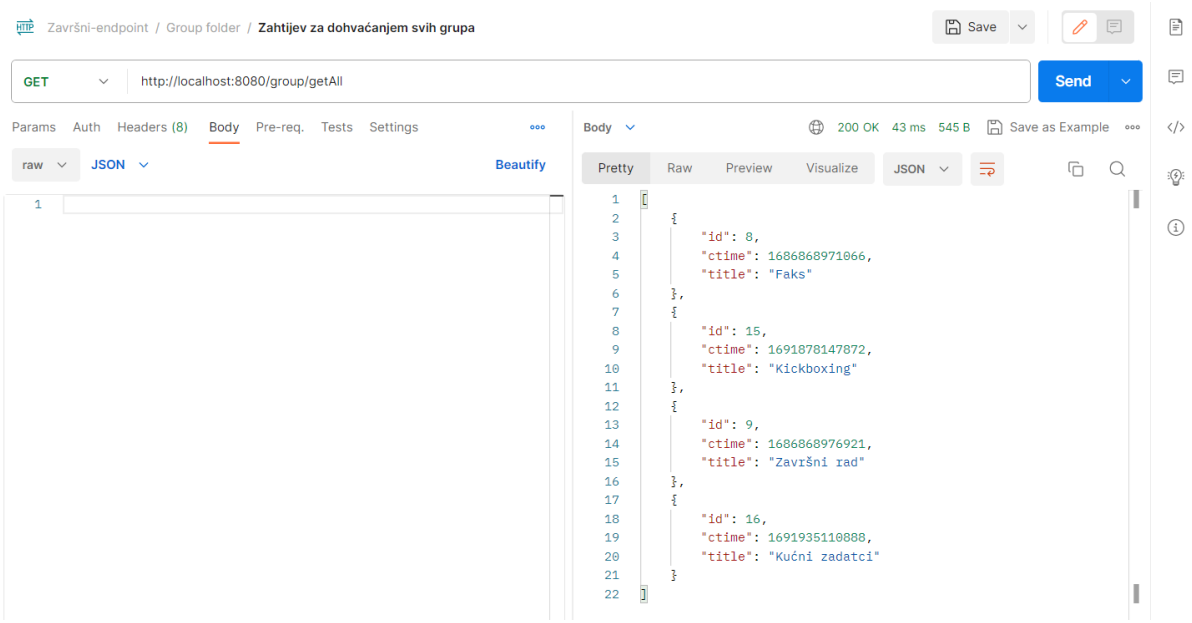
Jedina bitna razlika koju je važno napomenuti u odnosnu na druge funkcionalnosi manipuliranja grupama zadataka, jest implementacija metoda `update()` unutar klase `TaskGroupDataManager`. Proces ažuriranja vrijednosti započinje provjerom dostupnosti tražene grupe, ako je grupa dostupna provjerava se jeli naslov prazan. Ako naslov nije prazan i tražena grupa je pronađena unutar baze podataka, ista se popunjava novim vrijednostima i sprema u bazu podataka, kao što je prikazano na *Programski kod 34*.

```
@Override
public TaskGroup update(TaskGroup param, int userId) throws
Exception{
    Optional<TaskGroup> taskGroup =
taskGroupRepository.findByUserIdAndId(userId,param.getId());
    if (taskGroup.isPresent()){
        TaskGroup tskGroup = taskGroup.get();
        if(param.getTitle() != null &&
!param.getTitle().isEmpty())
        {
            tskGroup.setTitle(param.getTitle());
            tskGroup.setUserId(userId);
            return taskGroupRepository.save(tskGroup);
        }
        else {
            throw new IllegalArgumentException("title field
cannot be empty!");
        }
    }
    else {
        throw new
IllegalArgumentException(String.format("Group with id:%s
```

*Programski kod 34.* Prikaz metode `update()` klase `TaskGroupDataManager`

## 5.12. Značajka dohvaćanja svih grupa zadataka

Za razliku od značajke dohvaćanja stranice sa zadatcima koja je imala značajku paginacije, filtracije i sortiranja, značajka dohvaćanja svih grupa zadataka nema navedene. Sastoji se isključivo od slanja HTTP Get metode na rutu „*group/getAll“,* te ukoliko poslužitelj uspješno obradi zadatak, vratit će polje objekta `TaskGroup` u JSON formatu kao što je prikazano na *Slika 27.* Metoda `getAllGroups()` klase `TaskGroupController` delegira poziv metodi `getAllGroups()` klase `TaskGroupService`, koja zatim izvlači korisnički identifikator iz tokena te prosljeđuje poziv metodi `getAllGroups()` klase `TaskGroupDataManager` vidi *Programski kod 35*.




*Slika 27.* Primjer zahtijeva i odgovora za izlistavanje svih dostupnih grupa koristeći alat Postman

Unutar klase `TaskGroupDataManager` metoda *getAllGroups* poziva metodu `findAllByUserId()` sučelja `TaskGrouprRepository`, koje kao rezultat vraća sve zapise `TaskGroup` entiteta unutar baze podataka koji sadrže specificirani korisnički identifikator. Rezultati metode su vraćeni u obliku polja `TaskGroup` objekata, vidi *Programski kod 35*.

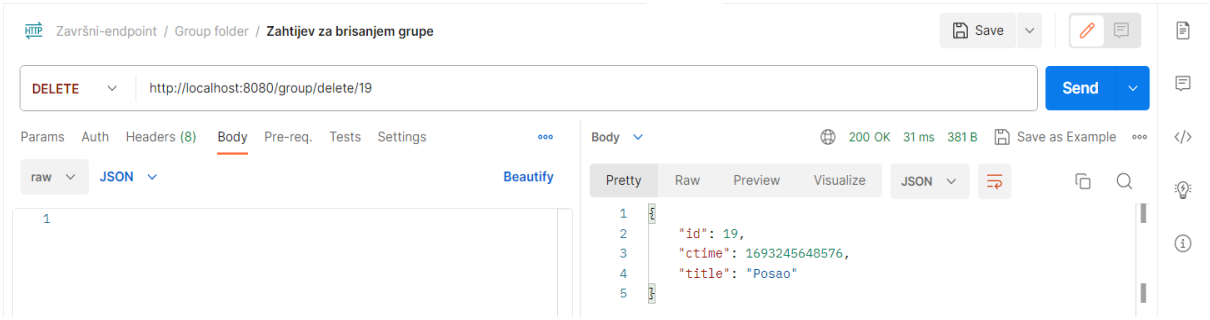
```
public List<TaskGroup> getAllGroups(Integer userId){
    return taskGroupRepository.findAllByUserId(userId);
}
```

*Programski kod 35.* Prikaz metode `getAllGroups()` klase `TaskGroupDataManager`

## 5.13. Značajka brisanja grupe zadataka

Implementacija značajke brisanja grupe zadataka je identična značajci brisanja zadatka, te iz tog razloga se neće zalaziti u detalje iste. Identično značajci brisanja zadataka, da bi se obrisala grupa zadataka, šalje se HTTP Delete zahtjev sa identifikatorom grupe zadataka koju se kani izbrisati, kao varijablom rute, na rutu *„group/delete/{id}“*. Ako je zahtjev za brisanjem uspješno proveden, poslužitelj će vratiti izbrisani objekt kao odgovor zajedno sa HTTP Statusnim kodom 200, kao što je prikazano na *Slika 28*. Važno je napomenuti kako između entiteta `Task` i entiteta `TaskGroup` vrijedi odnos više na jedan, kao što je izraženo

bilješkom `@ManyToOne` iznad `taskGroup` polja entiteta `Task`, vidi *Programski kod 19*. Navedeno ograničenje nalaže da se ne može izbrisati entitet `TaskGroup`-a dok god postoji relacija sa nekim entitetom tipa `Task`.




*Slika 28.* Primjer upita za brisanje grupe aktivnosti, izveden uporabom alata Postman

## 5.14. Analiza baze podataka

Baza podataka PostgreSQL predstavlja okosnicu pohrane podataka unutar promatrane TODO aplikacije. Kako bi Spring boot aplikacija mogla komunicirati s bazom podataka potrebno je postaviti postavke veze unutar *application.properties* datoteke, čiji je sadržaj prikazan na *Programski kod 36.*

```
spring.datasource.url=jdbc:postgresql://localhost:5432/todo_app
spring.datasource.username=postgres
spring.datasource.password=postgresql
spring.jpa.hibernate.ddl.-auto=create-drop,update

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.P
ostgreSQLDialect
```

*Programski kod 36.* Prikaz postavki veze između Spring boot aplikacije i baze podataka PostgreSQL

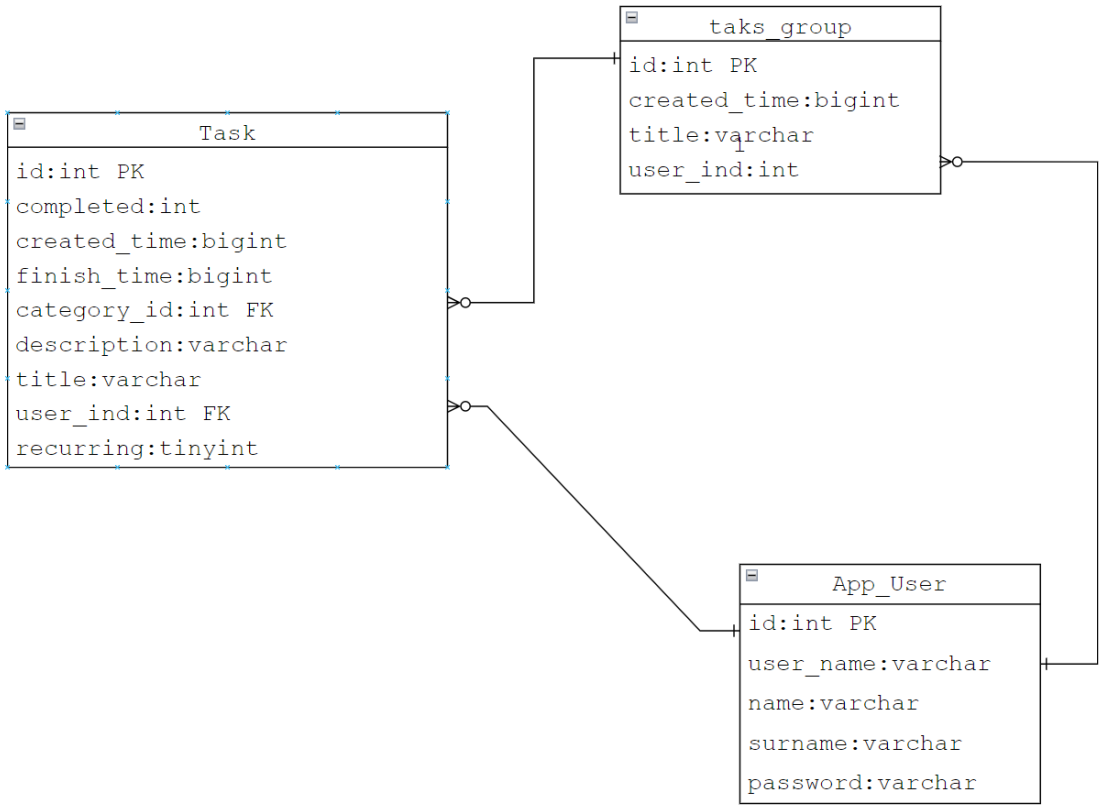
Konfiguracija na *Programski kod 36* se sastoji od:

- `spring.datasource.url` – ovaj parametar specificira URL za povezivanje sa bazom podataka korištenjem JDBC upravljačkog programa.
- `spring.datasource.username` – konfiguracija korisničkog imena za pristup bazi podataka
- `spring.datasource.password` – konfiguracijski zapis lozinke za pristup bazi

podataka

- `spring.jpa.hibernate.ddl.-auto` –konfiguracijski zapis za automatsko generiranje sheme Hibernate ORM-a
- `spring.jpa.show-sql` – pruža postavke omogućavanja ili onemogućavanja ispisa SQL upita Hibernate ORM-a
- `spring.jpa.properties.hibernate.dialect` – definira koji će se SQL dijalekt koristiti
- `spring.jpa.properties.hibernate.format_sql` – omogući ili onemogući konfiguriranje SQL upita u konzoli.

Uz konfiguracijsku datoteku također je potrebno uključiti potrebne upravljačke programe, koji se navode unutar *pom.xml* datoteke. Shema korištene baze podataka je izražena kroz *Slika 29.*




*Slika 29.* ER dijagram korištene baze podataka

# 6. ZAKLJUČAK

Kroz teoretski dio rada istražena je arhitektura programskog okvira Spring, analogno njemu i ona programskog okvira Spring boot, arhitektura pristupa podatcima korištenjem JDBC i JPA standarda, te arhitektura sigurnosnog sustava filtriranja zahtijeva primljenih na poslužitelj. Spomenute su i različite vrste baza podatka, te analizirana svojstva istih. Zatim su analizirani i definirani različiti oblici Web servisa u upotrebi. Na samom kraju je analizirana aplikacija rađena pomoću programskog okvira Spring boot, te je na konkretnom primjeru prikazana upotreba *FilterChain* sustava filtriranja zahtijeva spomenutog u teoretskom djelu, uz to je također prikazana upotreba Hibernate ORM-a sustava i implementaciju standarda JPA prilikom mapiranja entiteta na POJO `Task`, `TaskGroup` i `AppUser`. Na kraju je unutar jednog potpoglavlja u kratkim crtama prikazan način povezivanja aplikacije rađene unutar programskog okvira Spring na bazu podataka PostgreSQL, te ERD dijagram korištene baze podataka. Također su prikazani i primjeri slanja upita i primanja rezultata za svaku od funkcionalnosti aplikacije korištenjem alata Postman. Iz analize promatrane aplikacije može zaključiti kako je za razvoj modernih poslužiteljskih aplikacija potrebna integracija i interoperatibilnosti između mnoštva različitih tehnologija i znanja.

# LITERATURA


*15. Web MVC framework*. (n.d.). Preuzeto 4. Kolovoz 2023 iz https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-introduction

*A Brief History of Spring & Spring Boot*. (n.d.). Preuzeto 16. Srpanj 2023 iz Open Source Agenda: https://www.opensourceagenda.com/blog/a-brief-history-of-spring-spring-boot

*A Guide to Key-Value Databases: influxData*. (n.d.). (influxdata) Preuzeto 7. Kolovoz 2023 iz https://www.influxdata.com/key-value-database/

*Architecture: Spring Security Documentation*. (n.d.). Preuzeto 5. Kolovoz 2023 iz Spring Security Documentation: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-exceptiontranslationfilter

*Big Picture (Architecture): How to GraphQL*. (n.d.). Preuzeto 11. Kolovoz 2023 iz GraphQL: https://www.howtographql.com/basics/3-big-picture/

Chauhan, C. (2. Listopad 2017). *Understanding the PostgreSQL Architecture: Severalnines*. Dohvaćeno iz Severalnines: https://severalnines.com/blog/understanding-postgresql-architecture/

*Data Access with JDBC*. (n.d.). Dohvaćeno iz Spring Framework Documentation: https://docs.spring.io/spring-framework/reference/data-access/jdbc.html

Gillis, A. S. (n.d.). *REST API (RESTful API): TechTarget*. Preuzeto 12. Kolovoz 2023 iz TechTarget: https://www.techtarget.com/searchapparchitecture/definition/RESTful-API

*Graph Database Defined: Oracle*. (n.d.). (Oracle) Preuzeto 8. Kolovoz 2023 iz https://www.oracle.com/autonomous-database/what-is-graph-database/

Gulati, V. (24. Siječanj 2022). *Relational Model in DBMS*. (Scaler) Preuzeto 7. Kolovoz 2023 iz https://www.scaler.com/topics/dbms/relational-model-in-dbms/

*How Connections Are Established: PostgreSQL dokumentacija*. (n.d.). Preuzeto 9. Kolovoz 2023 iz PostgreSQL dokumentacija: https://www.postgresql.org/docs/current/connect-estab.html

*Introduction to gRPC:gRPC dokumnetacija*. (n.d.). Dohvaćeno iz gRPC dokumnetacija: https://grpc.io/docs/what-is-grpc/introduction/

*Introduction to Spring Framework*. (n.d.). Preuzeto 16. Srpanj 2023 iz Web dokumentacija programskog okvira Spring: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

*Introduction to the Spring IoC Container and Beans:Spring Framework.* (n.d.). Preuzeto 17. Srpnja 2023 iz https://docs.spring.io/spring-framework/reference/core/beans/introduction.html

*Java Persistence API (JPA): IBM*. (2. Svibanj 2023). Preuzeto 4. Kolovoz 2023 iz Java Persistence API (JPA): https://www.ibm.com/docs/en/was-liberty/zos?topic=overview-java-persistence-api-jpa

*Java Persistence API (JPA):IMB.* (n.d.). (IBM) Preuzeto 4. Kolovoz 2023 iz https://www.ibm.com/docs/en/was-liberty/zos?topic=overview-java-persistence-api-jpa

Lane, K. (28. Lipanj 2023). *What Is a SOAP API?: Postman*. Dohvaćeno iz Postman: https://blog.postman.com/soap-api-definition/

Lewis, S. (n.d.). *web services: TechTarget*. Preuzeto 11. Kolovoz 2023 iz TechTarget: https://www.techtarget.com/searchapparchitecture/definition/Web-services

Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces* (1. izd.). (S. S. Laurent, Ur.) "O'Reilly Media, Inc.", 2011. Preuzeto 12. Kolovoz 2023 iz https://books.google.hr/books?hl=hr&lr=&id=eABpzyTcJNIC&oi=fnd&pg=PR3&dq=rest+api&ots=vAXw60gfNG&sig=Q4R2AqfqXZM45Nl4Xh5Z07P2EiQ&redir_esc=y#v=onepage&q=rest%20api&f=false

*MongoDB vs. MySQL Differences: MongoDB.* (n.d.). Preuzeto 9. Kolovoz 2023 iz MongoDB: https://www.mongodb.com/compare/mongodb-mysql

Mullins, C. S. (n.d.). *database managment system (DBMS)*. (TechTarget) Preuzeto 7. Kolovoz 2023 iz https://www.techtarget.com/searchdatamanagement/definition/database-management-system

*MVC Framework Introduction*. (n.d.). (GeeksForGeeks) Preuzeto 2. Kolovoz 2023 iz https://www.geeksforgeeks.org/mvc-framework-introduction/

*Overview of Spring Framework*. (n.d.). Preuzeto 16. Srpanj 2023 iz Web dokumentacija programskog okvira Spring: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/spring-introduction.html

*Spring - Transaction Management: tutorialspoint*. (n.d.). Preuzeto 4. Kolovoz 2023 iz https://www.tutorialspoint.com/spring/spring_transaction_management.htm

*Spring vs Spring boot vs Spring MVC*. (n.d.). Preuzeto 16. Srpanj 2023 iz JavaTpoint: https://www.javatpoint.com/spring-vs-spring-boot-vs-spring-mvc

*Structure of a SOAP message: IBM*. (n.d.). Preuzeto 11. Kolovoz 2023 iz IBM: https://www.ibm.com/docs/en/cics-ts/5.4?topic=format-structure-soap-message

Thakur, D. (n.d.). *What are the Components of DBMS?: Computer Notes*. Preuzeto 8. Kolovoz 2023 iz Computer Notes: https://ecomputernotes.com/fundamental/what-is-a-database/components-of-dbms

*The Evolution of Database Management System*. (Studeni 2018). Preuzeto 7. Kolovoz 2023 iz UKEssays: https://www.ukessays.com/essays/information-technology/the-evolution-of-database-management-system-information-technology-essay.php

*The Parser Stage: PostgreSQL dokumentacija*. (n.d.). Preuzeto 9. Kolovoz 2023 iz PostgreSQL dokumentacija: https://www.postgresql.org/docs/current/parser-stage.html

*The Path of a Query: PostgreSQL dokumentacija*. (n.d.). Preuzeto 9. Kolovoz 2023 iz PostgreSQL dokumentacija: https://www.postgresql.org/docs/current/connect-estab.html

*Universal Description, Discovery, and Integration (UDDI): IBM*. (n.d.). Preuzeto 11. Kolovoz 2023 iz IBM: https://www.ibm.com/docs/en/radfws/9.6.1?topic=standards-universal-description-discovery-integration-uddi

*What Is a Database?: Oracle*. (n.d.). Preuzeto 7. Kolovoz 2023 iz Oracle: https://www.oracle.com/database/what-is-database/#WhatIsDBMS

*What is a Relational Database (RDBMS)?* (n.d.). (Oracle) Preuzeto 7. Kolovoz 2023 iz https://www.oracle.com/database/what-is-a-relational-database/

*What is a web service?: IBM*. (n.d.). Preuzeto 11. Kolovoz 2023 iz IBM: https://www.ibm.com/docs/en/cics-ts/5.1?topic=services-what-is-web-service

*What is an RDF Triplestore?: Ontotext*. (9. Kolovoz 2023). Dohvaćeno iz Ontotext: https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/

*What is GraphQL and how does it work?: Postman*. (16. Rujan 2020). Preuzeto 11. Kolovoz 2023 iz Postman: https://blog.postman.com/what-is-a-graphql-api-how-does-it-work/

*What is Postgres: PostgreSQL dokummnetacija*. (n.d.). Preuzeto 9. Kolovoz 2023 iz PostgreSQL Dokumentacija: https://www.postgresql.org/docs/current/intro-whatis.html

*What is PostgreSQL?: Amazon*. (n.d.). Preuzeto 9. Kolovoz 2023 iz Amazon: https://aws.amazon.com/rds/postgresql/what-is-postgresql/

*What Is SQL (Structured Query Language)?* (n.d.). (Amazon) Preuzeto 7. Kolovoz 2023 iz https://aws.amazon.com/what-is/sql/

*What is WSDL?: IBM*. (n.d.). Preuzeto 11. Kolovoz 2023 iz IBM: https://www.ibm.com/docs/en/app-connect/11.0.0?topic=services-what-is-wsdl

*Wide Column Database (Use Cases, Example, Advantages & Disadvantages): DatabaseTown*. (n.d.). Preuzeto 9. Kolovoz 2023 iz DatabaseTown: https://databasetown.com/wide-column-database-use-cases/

*Wide Column Store: Scylla*. (n.d.). (Scylla) Preuzeto 8. Kolovoz 2023 iz https://www.scylladb.com/glossary/wide-column-store/

Williams, A. (3. Studeni 2021). *NoSQL database types explained: Document-based databases: TechTarget*. (TechTarget) Preuzeto 8. Kolovoz 2023 iz https://www.techtarget.com/searchdatamanagement/tip/NoSQL-database-types-explained-Document-based-databases

# PRILOZI

## Popis slika

*Slika 1. Ilustrirani prikaz arhitekture programskog okvira Spring, Izvor: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html* ..... 5
*Slika 2. Grafički prikaz razvojne paradigme MVC* ..... 7
*Slika 3.Tok obrade zahtjeva unutar programskog okvira Spring MVC, Izvor:https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html* ..... 9
*Slika 4. Prikaz FilterChain-a s umetnutom DelegatingFilterProxy implementacijom klase Filter, Izvor:https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters* ..... 13
*Slika 5. Prikaz položaja klase FilterChainProxy i SecurityFilterChain u kontekstu FilterChain stabla, Izvor: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters* ..... 14
*Slika 6. Prikaz interakcije klase ExceptionTranslationFilter sa ostalim komponentama, Izvor: https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters* ..... 15
*Slika 7. Prikaz komponenti Sustava za upravljanje bazom podataka, Izvor: https://ecomputernotes.com/fundamental/what-is-a-database/components-of-dbms* ..... 17
*Slika 8. Prikaz sheme Baze podataka Ključ-vrijednost, Izvor: https://commons.wikimedia.org/wiki/File:KeyValue.PNG* ..... 21
*Slika 9. Shema Baze podataka širokog stupca* ..... 23
*Slika 10. Shema Postmaster procesa, Izvor: https://severalnines.com/blog/understanding-postgresql-architecture/* ..... 24
*Slika 11. Prikaz arhitekture promatrane TODO aplikacije* ..... 31
*Slika 12. Prikaz sekvencijsko dijagrama provjeravanja valjanosti JWT tokena unutar klase JwtAuthFilter* ..... 34
*Slika 13. Prikaz slanja zahtijeva za prijavom i prikaz dobivenog odgovora korištenjem alata Postman* ..... 36

*Slika 14. Prikaz tijela zahtijeva i rezultata zahtijeva za registracijom korisnika poslanog pomoću alata Postman* ..... *39*
*Slika 15. Klasni dijagram AppUser klase* ..... *41*
*Slika 16. Prikaz zahtijeva za dohvaćanjem stranice zadataka, te dobivenog odgovora. Zahtjev je napravljen pomoću alata Postman* ..... *44*
*Slika 17. Prikaz klasnog dijagrama RequestWrapper klase* ..... *46*
*Slika 18. Prikaz klasnog dijagrama Service klase* ..... *49*
*Slika 19. Primjer zahtijeva za stvaranjem zadatka korištenjem alata Postman* ..... *56*
*Slika 20. Klasni dijagram povratnih tipova promatrane aplikacije* ..... *57*
*Slika 21. Primjer slanja zahtijeva za ažuriranje zadatka korištenjem alata Postman* ..... *60*
*Slika 22. Dijagram toka metode update()* ..... *61*
*Slika 23. Zahtjev za brisanjem zadatka sa id-om 167, te prikaz dobivenog odgovora* ..... *63*
*Slika 24. Primjer zahtjeva za dohvaćanjem zadatka sa odgovorom poslužitelja korištenjem alata Postman* ..... *64*
*Slika 25. Prikaz zahtijeva za stvaranjem grupe poslanog prema poslužitelju korištenjem alata Postman* ..... *66*
*Slika 26. Primjer zahtijeva za ažuriranjem grupe zadataka poslanog putem alata Postman* . *67*
*Slika 27. Primjer zahtijeva i odgovora za izlistavanje svih dostupnih grupa koristeći alat Postman,* ..... *69*
*Slika 28. Primjer upita za brisanje grupe aktivnosti, izveden uporabom alata Postman* ..... *70*
*Slika 29. ER dijagram korištene baze podataka* ..... *71*

## Popis tablica

*Tablica 1. Usporedba značajki sadržanih u programskom okviru Spring te onih nadodanih u Spring Boot modulu* ..... *4*
*Tablica 2. Prikaz podjele ovlaštenja između razvojnog inženjera i programskog okvira Spring* ..... *11*

## Popis programskih kodova

*Programski kod 1. Prikaz konfiguracije SecurityFilterChain-a* ..... *32*
*Programski kod 2. Prikaz implementacije JwtAuthFilter klase, Izvor: obrada autora* ..... *33*
*Programski kod 3. Prikaz JwtService klase sa metodama korištenim pri validaciji tokena te*

*izvlačenju korisničkog imena* ..... *35*
*Programski kod 4. Prikaz login metode klase AuthenticationController* ..... *37*
*Programski kod 5. Prikaz klase AuthService sa metodom authenticate* ..... *38*
*Programski kod 6. prikaz metode korištene za generiranje JWT tokena unutar klase JwtService* ..... *38*
*Programski kod 7. Prikaz register() metode unutar klase AuthenticationController* ..... *39*
*Programski kod 8. Prikaz metode register() unutar klase AuthService, Izvor: obrada autora* ..... *39*
*Programski kod 10. Prikaz klase AppUser* ..... *42*
*Programski kod 11. Prikaz konfiguracijske klase* ..... *43*
*Programski kod 12. Prikaz metode getTaskPage() unutar klase TaskController* ..... *45*
*Programski kod 13. Prikaz apstraktne klase Service* ..... *47*
*Programski kod 14. Prikaz CatchException funkcijskog sučelja* ..... *48*
*Programski kod 15. Prikaz metode za izvlačenje identifikatora korisnika iz JWT tokena, koja se nalazi unutar klase JwtService* ..... *48*
*Programski kod 16. Prikaz programskog koda klase TaskService* ..... *49*
*Programski kod 17. Prikaz metode getTasksPage() klase TaskDataManager klase* ..... *50*
*Programski kod 18. Prikaz metode getPredicate() klase TaskDataManager* ..... *51*
*Programski kod 19. Prikaz metode setOrder() klase TaskDataManager* ..... *52*
*Programski kod 20. Prikaz Task klase* ..... *53*
*Programski kod 21. Prikaz implementacije klase za serijalizaciju klase Task* ..... *54*
*Programski kod 22. Prikaz implementacije klase za deserijalizaciju JSON objekta* ..... *55*
*Programski kod 23. Prikaz metode saveTask() klase TaskController* ..... *56*
*Programski kod 24. Prikaz metode evaluate() sadržane unutar klase ResponseEntity* ..... *58*
*Programski kod 25. Prikaz metode mapExceptionToFailure() klase ResponseResolver* ..... *58*
*Programski kod 26. Prikaz metode write() klase TaskDataManager* ..... *59*
*Programski kod 27. Prikaz updateTask() metode klase TaskController* ..... *60*
*Programski kod 28. Prikaz update() metode klase TaskDataManager, Izvor: obrada autora* *62*
*Programski kod 29. Prikaz programskog koda metode deleteTask() klase TaskController* ... *63*
*Programski kod 30. Prikaz metode delete() klase TaskDataManager* ..... *64*
*Programski kod 31. Prikaz metode getTask() unutar klase TaskController* ..... *65*
*Programski kod 32. Prikaz implementacije read() metode klase TaskDataManager, Izvor: obrada autora* ..... *65*

*Programski kod 33. Prikaz metode createTaskGroup() klase TaskGroupController* ..... *66*
*Programski kod 34. Prikaz implementacije metode write() klase TaskGroupDataManager* .. *66*
*Programski kod 35. Prikaz metode update() klase TaskGroupDataManager* ..... *68*
*Programski kod 36. Prikaz metode getAllGroups() klase TaskGroupService*..... ***Pogreška! Knjižna oznaka nije definirana.***
*Programski kod 37. Prikaz metode getAllGroups() klase TaskGroupDataManager*. ***Pogreška! Knjižna oznaka nije definirana.***
*Programski kod 38. Prikaz postavki veze između Spring boot aplikacije i baze podataka PostgreSQL* ..... *70*