|  |  |  |
| --- | --- | --- |
|  | Univerzitet „UNION“  **RAČUNARSKI FAKULTET**  Knez Mihailova 6/VI  11000 Beograd | Broj: |
|  |
| Datum: |
|  |

UNIVERZITET „UNION“

RAČUNARSKI FAKULTET

RAČUNARSKE NAUKE

DIPLOMSKI RAD

Kandidat: Nikola Cvetinović

Broj indeksa: RN-76/19

**Tema rada: Generator Spring Boot aplikacije na osnovu modela podataka**

Mentor rada: prof. dr Bojana Dimić Surla

**Sadržaj**

[1. Uvod 4](#_Toc149154427)

[1.1. Motivacija za razvoj generatora 4](#_Toc149154428)

[1.2. Budućnost razvoja softvera 5](#_Toc149154429)

[1.3. Pregled korišćenih tehnologija 5](#_Toc149154430)

[1.3.1. Spring 5](#_Toc149154431)

[1.3.2. Jakarta XML Binding 6](#_Toc149154432)

[1.4. Slični generatori 6](#_Toc149154433)

[2. Opis strukture modela i konfiguracije 7](#_Toc149154434)

[2.1. Konfiguracija 7](#_Toc149154435)

[2.2. XML model podataka 9](#_Toc149154436)

[2.3. Metamodel u Javi 11](#_Toc149154437)

[2.3.1. AppModel 11](#_Toc149154438)

[2.3.2. Entity model 12](#_Toc149154439)

[2.3.3. Enum model 13](#_Toc149154440)

[2.3.4. Attribute model 14](#_Toc149154441)

[2.3.5. Relation model 15](#_Toc149154442)

[3. Struktura generisane Spring aplikacije 17](#_Toc149154443)

[3.1. Struktura Spring Boot aplikacije 17](#_Toc149154444)

[3.1.1. Sloj Kontrolera 17](#_Toc149154445)

[3.1.2. Servisni sloj: 18](#_Toc149154446)

[3.1.3. Sloj Repositorijuma (Repository Layer) 18](#_Toc149154447)

[3.1.4. Sloj Konfiguracije (Configuration Layer): 18](#_Toc149154448)

[3.2. Unapređivanje strukture: Pristup ponovnom pokretanju generatora 19](#_Toc149154449)

[3.2.1. Problem sa ponovnim pokretanjem generatora 19](#_Toc149154450)

[3.2.2. Rešenje kroz generičke klase 19](#_Toc149154451)

[3.2.3. Prilagođene klase nasleđuju generike 19](#_Toc149154452)

[3.2.4. Prednosti ovog pristupa 19](#_Toc149154453)

[3.2.5. Implementacija pristupa 20](#_Toc149154454)

[3.3. Generisana aplikacija 20](#_Toc149154455)

[3.3.1. Paketi (Slika 3.1) 20](#_Toc149154456)

[3.3.2. Zavisnosti (Dependency Injection): 21](#_Toc149154457)

[4. Opis implementacije generatora 23](#_Toc149154458)

[4.1. Ulazni parametri generatora i pocetna struktura aplikacije 23](#_Toc149154459)

[4.2. Struktura generatora 25](#_Toc149154460)

[4.3. Generik komponente 29](#_Toc149154461)

[4.3.1. GenericEntityWriter 29](#_Toc149154462)

[4.3.2. GenericEnumWriter 31](#_Toc149154463)

[4.3.3. GenericControllerWriter 32](#_Toc149154464)

[4.3.4. GenericRepositoryWriter 34](#_Toc149154465)

[4.3.5. GenericServiceWriter 35](#_Toc149154466)

[4.4. Custom komponente 36](#_Toc149154467)

[5. Zaključak 38](#_Toc149154468)

[6. Literatura 39](#_Toc149154469)

# Uvod

U današnjem digitalnom dobu, razvoj softverskih aplikacija je ključni stub modernog poslovanja i inovacija. U tom kontekstu, kreiranje brzih, efikasnih i skalabilnih aplikacija je izazov sa kojim se suočavaju mnogi programeri i inženjeri širom sveta. U potrazi za rešenjima koja bi ubrzala i pojednostavila proces razvoja softvera, razvijen je alat koja ima potencijal da transformiše način na koji se kreiraju Java (i ostale) aplikacije – aplikacija za generisanje Spring Boot aplikacija na osnovu modela podataka.

Ovaj rad ima za cilj da predstavi razloge za razvoj ove inovativne aplikacije, njene ključne funkcionalnosti i mogućnosti koje se otvaraju za budućnost softverskog inženjeringa. U nastavku će se raspravljati o osnovnim motivima za kreiranje ove aplikacije i njenom potencijalnom doprinosu domenu razvoja softvera.

## Motivacija za razvoj generatora

Razvoj softverskih aplikacija je proces koji zahteva vreme, resurse i stručnost. Tradicionalni pristup zahteva mnogo ručnog kodiranja, testiranja i podešavanja, što može biti dugotrajno i skupo. Da bi se olakšao ovaj proces i ubrzala isporuka softvera, glavna motivacija za razvoj ove aplikacije leži u:

1. Automatizacija razvojnog procesa: Generisanje aplikacija zasnovano na modelu omogućava automatizaciju kreiranja osnovnih komponenti aplikacije, kao što su entiteti, kontroleri, servisi i baze podataka. Ovo dovodi do značajnog smanjenja vremena razvoja.
2. Smanjenje ljudske greške: Ručno kodiranje aplikacija može dovesti do ljudskih grešaka. Ova aplikacija obezbeđuje doslednost i tačnost generisanog koda, smanjujući greške i održavanje.
3. Brža reakcija na promene: U današnjem dinamičnom poslovnom okruženju, brza reakcija na promene je ključna. Aplikacija omogućava programerima da brzo generišu nove funkcionalnosti ili promene u aplikacijama.

## Budućnost razvoja softvera

Generator Spring Boot aplikacija je samo jedan korak u budućnost softverskog inženjeringa. Sveobuhvatna automatizacija, primena veštačke inteligencije i razvoj nove paradigme „koda kao servisa“ otvaraju mogućnosti za potpunu revoluciju načina na koji su softverski projekti vođeni i održavani.

Ovaj rad istražuje korake ka realizaciji ovih mogućnosti i doprinosima razvoju softvera, smanjenju vremena za razvoj, povećanju kvaliteta i tačnosti i podršci inovacijama u različitim industrijama.

U narednim odeljcima ćemo govoriti o tehničkim detaljima ove aplikacije, procesu generisanja Spring Boot aplikacija na osnovu modela, kao i o praktičnim primerima korišćenja i analizi rezultata. Svi ovi aspekti zajedno će pružiti dublji uvid u potencijal i uticaj ove aplikacije na budućnost razvoja softvera.

## Pregled korišćenih tehnologija

### Spring

Spring Boot [1] je moćan alat za razvoj aplikacija u programskom jeziku Java. Ističe se kao kompaktno rešenje koje omogućava razvoj brzih, efikasnih i skalabilnih aplikacija uz minimalan napor. Ovaj alat se često koristi da olakša i ubrza kreiranje veb aplikacija, mikroservisa i raznih Java aplikacija.

Spring Boot zapravo „pakuje“ sve potrebne biblioteke, konfiguracije i koje su potrebne za pokretanje Java aplikacija u jedan pogodan paket. Ovo omogućava programerima da se fokusiraju na razvoj poslovne logike aplikacije umesto da troše vreme na konfigurisanje i podešavanje okruženja.

Ovaj alat pruža ugrađene mehanizme za rukovanje HTTP zahtevima, pristup bazama podataka, upravljanje transakcijama i mnoge druge uobičajene zadatke. Takođe, omogućava jednostavno proširenje funkcionalnosti dodavanjem različitih modula i biblioteka. Sve ovo zajedno čini Spring Boot savršenim alatom za razvoj modernih Java aplikacija koje su brže, bezbednije i lakše za održavanje.

Ukratko, Spring Boot služi kao moćan okvir za razvoj Java aplikacija, omogućavajući programerima brži i efikasniji proces kreiranja softverskih rešenja.

### Jakarta XML Binding

Jakarta XML Binding [2] je softverski okvir koji omogućava Java EE programerima da mapiraju Java klase u XML reprezentacije. JAKSB pruža dve glavne karakteristike: mogućnost maršaliranja Java objekata u XML i obrnuto, tj. demaršaliranja XML-a nazad u Java objekte.

## Slični generatori

Postoji nekoliko alata i platformi koji omogućavaju generisanje aplikacija na osnovu modela ili drugih apstrakcija. Evo nekoliko sličnih generatora aplikacija i njihovih kratkih opisa:

* 1. **Mendix** [3] je platfarma sa malo programskog koda (eng. low-code) koja omogućava kreiranje poslovnih aplikacija putem vizuelnog modeliranja. Korisnici mogu definisati entitete, logiku i korisnički interfejs bez potrebe za pisanjem tradicionalnog koda.
  2. **OutSystems** [4] je još jedna niskokodna platforma za razvoj poslovnih aplikacija. Omogućava brzo kreiranje i razvoj aplikacija putem vizuelnog modeliranja, uz mogućnost prilagođavanja koda ako je potrebno.
  3. **Quick Base** [5] je platforma za izradu poslovnih aplikacija bez programiranja. Omogućava kreiranje prilagođenih baza podataka i aplikacija za različite poslovne svrhe.
  4. **OpenAPI-a** [6] možete generisati server kod za brži razvoj RESTful API-ja. Ovo uključuje generisanje rutiranja, validacije, obrade zahteva i slanja odgovora. Generisani server kod oslobađa razvojni tim od potrebe za ručnim kodiranjem osnovnih funkcionalnosti API-ja.

# Opis strukture modela i konfiguracije

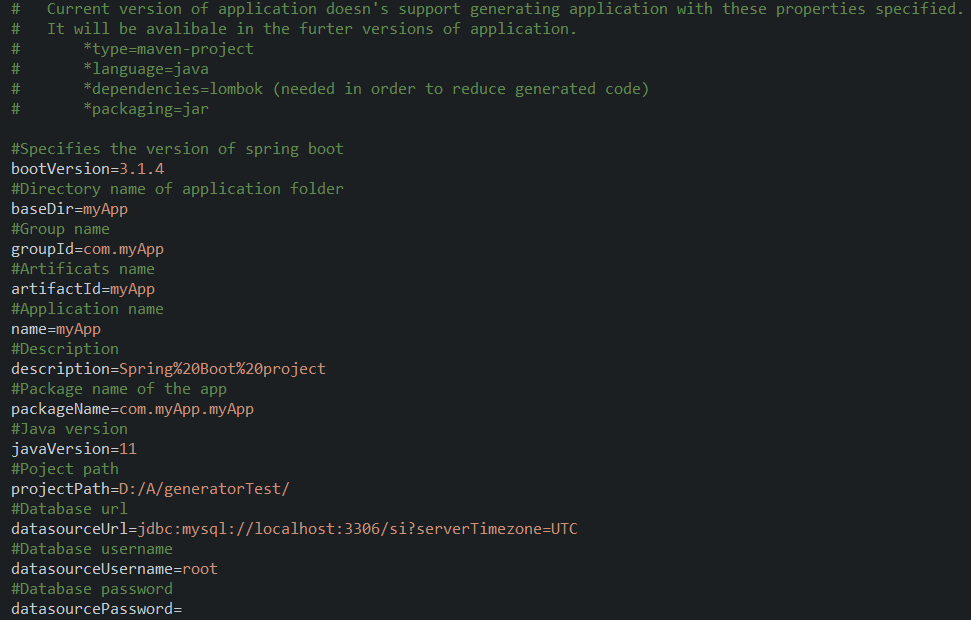
U okviru razmatranja ovog rada, ključni aspekt je detaljan opis strukture i konfiguracije modela na osnovu koga će se vršiti generisanje Spring Boot aplikacije. Razumevanje ovih elemenata je od suštinskog značaja za razvoj aplikacija vođenih modelom. Ovaj deo rada se fokusira na dva glavna aspekta: XML model i metamodel u programskom jeziku Java. Kroz konkretne primere u vidu XML i Java koda analiziramo kako se definišu i implementiraju modeli. Takođe, hajde da pregledamo procese učitavanja ovih modela da bismo bolje razumeli njihovu funkcionalnost. Pored toga, biće objašnjena i konfiguraciji koja je ključna za pravilno funkcionisanje aplikacije.

## Konfiguracija

Konfiguracioni fajl prikazan na slici 2.1 koristi za konfiguraciju generisanja aplikacije i definisanje osnovnih parametara za kreiranje projekta. Struktura ovog fajla je takva da je svaki parametar zadat u posebnom redu, definisan u obliku "naziv parametra” = “vrednost”. Evo opisa pojedinačnih parametara:

1. **bootVersion –** Specifikuje verziju Spring Boot okvira koja će biti korišćena za generisanje aplikacije.
2. **baseDir –** Definiše ime direktorijuma u kojem će biti smeštena generisana aplikacija.
3. **groupId –** Postavlja grupno ime (group ID) za projekat.
4. **artifactsId –** Postavlja ID umetka (artifact ID) za projekat.
5. **name –** Dodeljuje naziv aplikaciji.
6. **description –** Pruža kratki opis aplikacije.
7. **packageName –** Određuje paket u kojem će se generisani kod aplikacije nalaziti.
8. **javaVersion –** Postavlja verziju Java programskog jezika koja će biti korišćena u projektu.
9. **projectPath –** Definiše putanju na kojoj će se generisani projekat smestiti.
10. **datasourceUrl –** Specificira URL adresu baze podataka koju će aplikacija koristiti.
11. **datasourceUsename –** Postavlja korisničko ime za pristup bazi podataka.
12. **datasourcePassword –** Definiše lozinku za pristup bazi podataka.
13. **type –** Otip projekta koji će biti generisan.
14. **language –** Određuje programski jezik koji će se koristiti u generisanom projektu.
15. **dependecies –** Označavaju biblioteke, okvire i dodatke koji će biti uključeni u generisani projekt.
16. **packaging –** Ovaj parametar definiše kako će aplikacija biti zapakovana i izvršena.

Ovaj konfiguracioni fajl omogućava programerima da postave osnovne parametre za generisanje Spring Boot aplikacije, uključujući verziju, naziv, paket, putanju i konfiguraciju baze podataka. Pomoću ovih postavki, aplikacija će biti generisana sa odgovarajućim podešavanjima i strukturom direktorijuma na osnovu specifikacija navedenih u ovom fajlu.



**Slika 2.1** – Primer konfiguracionog fajla

## XML model podataka

XML model koji je osnova za generisanje spring boot aplikacije predstavlja strukturu i definiciju entiteta i enumeracija koji se koriste u softverskom sistemu. Model je organizovan na sledeći način (primer modela prikazan je na slici 2.2):

1. **Entiteti (Entities):** U okviru ovog modela, entiteti predstavljaju ključne koncepte sistema. Svaki entitet ima svoj naziv i definiše svoje atribute (podatke) i relacije (veze) sa drugim entitetima.
2. **Atributi (Attributes):** Svaki entitet sadrži određeni broj atributa koji definišu svojstva tog entiteta. Atributi mogu biti različitih tipova, kao što su "string", "enum", "date" i "double",.. . Takođe, atributi mogu imati dodatne karakteristike, kao što je "nullable" (mogućnost da bude prazan).
3. **Relacije (Relations):** Entiteti su povezani putem relacija koje definišu način kako se oni međusobno povezuju. Relacije imaju karakteristike kao što su "oneToMany", "manyToOne", "manyToMany" i "owningSide". Takođe, moguće je definisati "fetchType" za relacije.
4. **Enumeracije (Enums):** Model takođe uključuje enumeracije koje definišu vrednosti za određene atribute u entitetima. Na primer, enumeracija "tipDokumenta" ima vrednost "faktura", dok enumeracija "tipFakture" ima vrednosti "ulazna\_faktura" i "izlazna\_faktura".

Ovaj XML model služi kao osnova za definisanje strukture podataka i njihovih međusobnih odnosa u okviru softverskog sistema. Korišćenje ovakvog modela omogućava programerima da jasno definišu koncepte koji čine sistem i olakšava razvoj aplikacija koje koriste ove podatke i veze. Razvoj i unapređenje XML modela i njegovog interfejsa za uređivanje može uključivati razne poboljšanja u budućnosti, na primer korišćenjem različitih alata i editora za rad sa XML dokumentima za uređivanje XML modela, korisnicima bi bilo olakšano unošenje podataka i čuvanje promena bez potrebe za ručnim uređivanjem XML koda. Ovo je posebno korisno ako se XML model koristi za kreiranje modela, jer omogućava korisnicima da lako vrše konfiguraciju bez potrebe za dubokim razumevanjem XML sintakse. Ovim XML modelom su podržani svi koncepti za kreiranje modela podataka koji postoje u Javi i JPA (Java persistence API [9]).



**Slika 2.2** – Primer modela napisaong u XML-u

## Metamodel u Javi

Tokom ovog poglavlja, proučavaćemo ključne aspekte implementacije metamodela, uključujući napomene, mapiranje XML elemenata u Java klase i članove, i pristup XML podacima putem metoda i metoda za prikupljanje podataka za čoveka. Takođe ćemo istražiti kako metamodel omogućava čitanje i pisanje XML podataka, uključujući tehnike razvrstavanja i demaršaliranja.

### AppModel

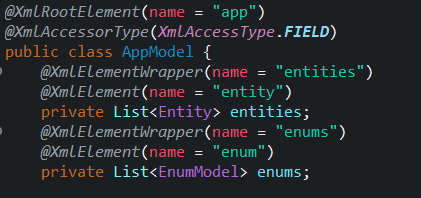
Ovaj Java metamodel (klasa AppModel, slika 2.3) predstavlja XML model koji sadrži strukturu entiteta i enumeracije. Evo ključnih tačaka koje opisuju ovaj metamodel:

* **@XmlRootElеmеnt (name = "app")** označava da se AppModel klasa mapira u XML element pod nazivom "app".
* **@XmlAccеssorTypе (XmlAccessTipe.FIELD)** označava da će članovi klase biti mapirani u XML elemente koristeći pristup polju.

Klasa sadrži dva atributa:

* **entities**: Liste entiteta (entities) označenih kao @XmlElementVrapper(name = "entities"), što ukazuje da su elementi entiteta grupisani unutar XML elementa "entitity".
* **enums**: Liste nabrajanja (EnumModel) označene kao @XmlElementVrapper(name = "enums"), što ukazuje da su elementi enumeracije grupisani unutar XML elementa "enum".

Ovaj Java metamodel (Slika 2.3) služi kao programski model za čitanje i manipulisanje XML podacima koji opisuje strukturu entiteta i enumeracija unutar softverskog sistema. Koristi se kao interfejs između XML podataka i Java aplikacije kako bi se omogućilo programerima da čitaju i pišu XML modele koristeći Java kod.



**Slika 2.3** – AppModel metamodel

### Entity model

Ovaj Java metamodel (klasa Entity, slika 2.4) predstavlja entitet u XML modelu i koristi se za predstavljanje i manipulaciju podacima koji se odnose na entitet. Evo ključnih tačaka koje opisuju ovaj metamodel:

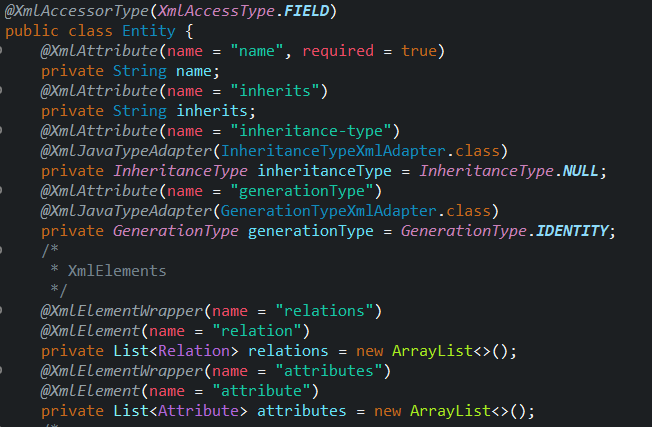
* **@XmlAccessorTipe(XmlAccessTipe.FIELD)** označava da će članovi klase biti mapirani u XML elemente koristeći pristup polju.

Klasa sadrži nekoliko atributa koji se mapiraju u XML atribute entiteta, uključujući ime, nasleđivanje, tip nasleđa i tip generisanja. Ovi atributi opisuju osnovne karakteristike entiteta, kao što su ime, nasledstvo, tip nasleđa i tip generacije. Klasa takođe sadrži dve liste:

* **relations**: Lista relacija (relations) označenih kao @XmlElementVrapper(name = "realtions"). Ovo grupiše elemente odnosa unutar XML elementa "relation". Svaka relacija opisuje povezanost entiteta sa drugim entitetima.
* **attributes**: Lista atributa (attributes) označenih kao @XmlElementVrapper(name = "attributes"). Ovo grupiše elemente atributa u okviru XML elementa "attributes". Svaki atribut predstavlja karakteristiku entiteta.

Ovaj metamodel omogućava programerima da efikasno čitaju i pišu podatke entiteta unutar XML modela koristeći Java kod. Klase InheritanceTipeXmlAdapter i GenerationTipeXmlAdapter se koriste za prilagođavanje kada se mapiraju XML vrednosti u Java enumeracije.

Preko ovog metamodela možete da predstavite strukturu entiteta u XML formatu, što je često korisno u kontekstu definisanja podataka u softverskim sistemima.



**Slika 2.4** – Entity metamodel

### Enum model

Ovaj Java metamodel (klasa EnumModel, slika 2.5) predstavlja XML prikaz nabrajanja i koristi se za predstavljanje i manipulaciju podacima koji se odnose na enumeracije. Evo ključnih tačaka koje opisuju ovaj metamodel:

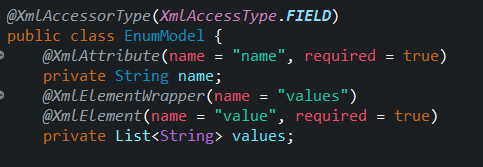
* **@XmlAccessorTipe(XmlAccessTipe.FIELD)** označava da će članovi klase biti mapirani u XML elemente koristeći pristup polju.

Klasa sadrži dva atributa koji se mapiraju u XML atribute nabrajanja:

* **name**: Ovaj atribut predstavlja naziv nabrajanja i označen je kao obavezan = tačno, što znači da je obavezan.
* **values**: Ovaj atribut je lista vrednosti nabrajanja označenih kao @XmlElementVrapper(name = "values") i @XmlElement(name = "value", rekuired = true). Ovo grupiše elemente vrednosti unutar XML elementa "vrednosti". Svaka vrednost predstavlja člana nabrajanja.

Ovaj metamodel omogućava programerima da efikasno čitaju i pišu podatke o nabrajanju unutar XML modela koristeći Java kod. EnumModel i EnumValue klase se koriste za predstavljanje strukture nabrajanja u XML formatu.

Kroz ovaj metamodel, možete predstaviti različite nabrojive vrednosti i koristiti ih za definisanje različitih tipova podataka ili opcija unutar softverskih sistema. Ovo je često korisno u kontekstu konfigurisanja ili definisanja parametara u aplikacijama.



**Slika 2.5** – Entity metamodel

### Attribute model

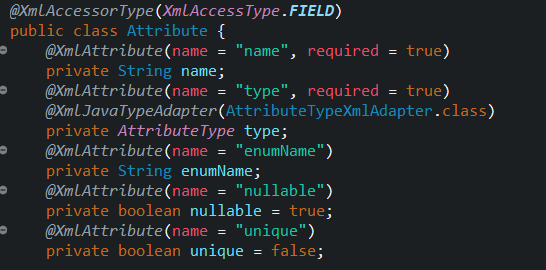
Ovaj Java metamodel (klasa Attribute, slika 2.6) predstavlja XML prikaz atributa entiteta u XML modelu. Ovaj metamodel omogućava predstavljanje i manipulaciju podacima koji se odnose na atribute entiteta. Evo ključnih tačaka koje opisuju ovaj metamodel:

* @XmlAccessorTipe(XmlAccessTipe.FIELD) označava da će članovi klase biti mapirani u XML elemente koristeći pristup polju.

Klasa sadrži nekoliko atributa koji se mapiraju u XML atribute atributa entiteta, uključujući name, type, enumName, nullable i unique. Ovi atributi opisuju karakteristike atributa, kao što su ime, tip podataka, opciono vezivanje nabrajanja, mogućnost nuliranja i jedinstvenost.

* **name** predstavlja naziv atributa i označen je kao required = true, što znači da je obavezan.
* **attributeType** predstavlja tip podataka atributa, a adapter AttributeTipeXmlAdapter se koristi za mapiranje vrednosti atributa u odgovarajuću enumeraciju.
* Atribut **enumName** ukazuje na ime nabrajanja (ako je to atribut tipa enumeracije).
* **nullable** označava da li je atribut opcioni (može biti null).
* **unique** atribut pokazuje da li vrednosti atributa moraju biti jedinstvene unutar entiteta.

Ovaj metamodel omogućava programerima da efikasno čitaju i pišu podatke o atributima entiteta u okviru XML modela koristeći Java kod. Koristi se za predstavljanje i upravljanje strukturom atributa unutar entiteta i često se koristi za definisanje svojstava entiteta u okviru softverskog sistema.



**Slika 2.6** – Attribute metamodel

### Relation model

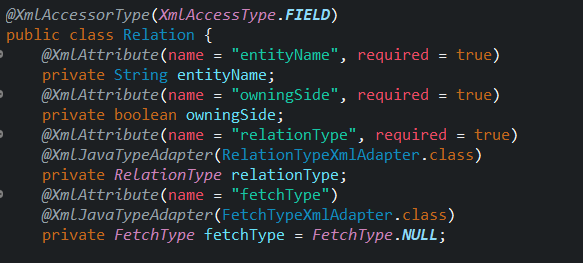
Ovaj Java metamodel (klasa Relation, slika 2.7) predstavlja XML prikaz veze (relacije) između entiteta u XML modelu. Ovaj metamodel omogućava predstavljanje i manipulaciju podacima koji se odnose na odnose između entiteta.

* **@XmlAccessorTipe(XmlAccessTipe.FIELD)** označava da će članovi klase biti mapirani u XML elemente koristeći pristup polju.

Klasa sadrži nekoliko atributa koji mapiraju XML atribute odnosa između entiteta, uključujući entitiName, owningSide, relationType i fetchType. Ovi atributi opisuju karakteristike veze, kao što su naziv povezanog entiteta, vlasnik veze, tip veze i način preuzimanja podataka.

* Atribut **entityName** predstavlja naziv entiteta sa kojim je veza uspostavljena i označen je kao required = true , što znači da je obavezan.
* Atribut **owningSide** ukazuje na to da li je strana entiteta (gde se nalazi veza) vlasnička strana i takođe je označen kao obavezan.
* Atribut **relationType** ukazuje na tip odnosa, a adapter RelationTipeXmlAdapter se koristi za mapiranje vrednosti atributa u odgovarajuću enumeraciju.
* Atribut **fetchType** ukazuje na tip preuzimanja podataka za vezu, a FetchTipeXmlAdapter adapter se koristi za mapiranje vrednosti atributa u odgovarajuću enumeraciju.

Ovaj metamodel omogućava programerima da efikasno čitaju i pišu podatke o odnosima između entiteta unutar XML modela koristeći Java kod. Koristi se za predstavljanje i upravljanje strukturom odnosa između entiteta unutar softverskog sistema. Odnosi između entiteta su važan aspekt u definisanju struktura podataka i odnosa unutar aplikacija.



**Slika 2.7** – Relation metamodel

# Struktura generisane Spring aplikacije

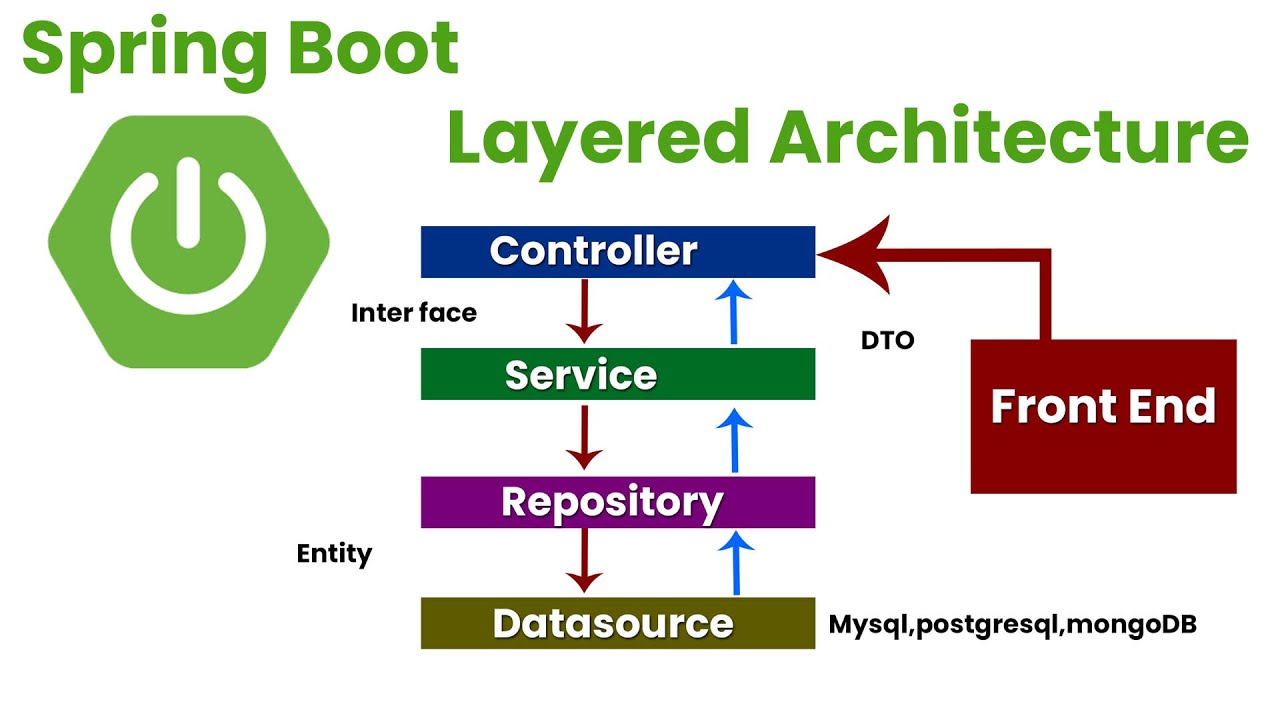
U poglavlju istražićemo osnovnu organizaciju i komponente aplikacije koja je generisana pomoću alata za generisanje Spring Boot aplikacije. Razumevanje strukture generisane aplikacije je ključno za svakog programera koji koristi ove alate, jer omogućava brži i efikasniji razvoj aplikacije.

Ovde ćemo analizirati osnovne direktorijume, konfiguracione datoteke i komponente koje čine generisanu aplikaciju. Takođe ćemo istražiti kako ovi elementi funkcionišu zajedno da bi omogućili da aplikacija funkcioniše kao celina. Kroz ovu analizu, programeri će steći dublje razumevanje o tome kako se njihovi modeli i konfiguracije pretvaraju u stvarne komponente i strukturu aplikacije.

Proučavanje strukture generisane aplikacije omogućava programerima da bolje razumeju kako da implementiraju dodatne funkcionalnosti, prošire aplikaciju i prilagode je specifičnim potrebama projekta. Ovo poglavlje će nam takođe pomoći da se upoznamo sa osnovama upravljanja projektima i organizacije koda u okviru Spring Boot aplikacije vođene modelom.

## Struktura Spring Boot aplikacije

Spring Boot aplikacija ima slojevitu strukturu koja je prikazana na slici 3.1. Ova struktura se često koristi u razvoju softvera za organizovanje aplikacije u različite slojeve ili komponente, od kojih svaka ima svoju specifičnu odgovornost. Slojevita struktura Spring Boot aplikacije sastoji se od sloja kontrolera (Controller layer), servisnog sloja (Service layer), repository sloja (Repository layer) a i u nekim slučajevima i od konfiguracionog sloja (Configuration layer).



**Slika 3.1** – Spring Boot layered arhitektura

### Sloj Kontrolera

Sloj kontrolera je odgovoran za obradu korisničkih zahteva i komunikaciju sa slojem servisnim slojem kako bi se izvršila odgovarajuća radnja. Kontroleri su prvenstveno odgovorni za prihvatanje HTTP zahteva, raščlanjivanje parametara i delegiranje posla na usluge. Ove komponente uključuju različite kontrolere (Spring MVC kontrolere ili REST kontrolere) koji se koriste za definisanje ruta i radnji koje treba izvršiti kao odgovor na zahteve korisnika.

### Servisni sloj:

Sloj servisa sadrži poslovnu logiku aplikacije. Servisi obavljaju različite operacije i poslovne funkcije, uključujući validaciju podataka, obradu poslovnih pravila i komunikaciju sa slojem repositorijuma za pristupanje podacima.

Ove komponente obuhvataju različite servise koji sadrže metode za izvršavanje specifičnih poslovnih zadataka. Servisi su obično označeni kao @Service.

### Sloj Repositorijuma (Repository Layer)

Sloj repositorijuma koristi se za pristupanje podacima, često putem baze podataka. Ovde se definišu operacije za čitanje, upisivanje, brisanje i ažuriranje podataka. Ovaj sloj izdvaja komunikaciju sa bazom podataka od servisa.Ove komponente obuhvataju različite repositorijume označene kao @Repository. Repositorijumi često koriste Spring Data JPA za olakšavanje komunikacije sa bazom podataka.

### Sloj Konfiguracije (Configuration Layer):

Sloj konfiguracije obuhvata konfiguracione klase koje definišu postavke aplikacije. Ovde možete definisati različite beane, filtere, interceptore i druge konfiguracije koje će se primenjivati u celoj aplikaciji. Ove komponente obuhvataju klase označene kao @Configuration, @Bean i druge anotacije za definisanje konfiguracionih elemenata.

## Unapređivanje strukture: Pristup ponovnom pokretanju generatora

U poglavlju „Poboljšanje strukture: pristup ponovnom pokretanju generatora“ istražićemo izazove koji nastaju prilikom ponovnog pokretanja generatora na istoj putanji aplikacije i kako da rešimo probleme koji se traju. Takođe ćemo razmotriti kako da implementiramo generičke klase i prilagođene klase da bismo efikasno upravljali ovim problemima.

### Problem sa ponovnim pokretanjem generatora

Ponovno pokretanje generatora aplikacija može izazvati problem gaženja postojećih generisanih datoteka. Ovaj problem postaje posebno izazovan kada su korisnici već dodali prilagođenu logiku u generisane datoteke, a ponovno pokretanje generatora može izbrisati te promene.

### Rešenje kroz generičke klase

Da bismo rešili ovaj problem, razmotrićemo koncept generičkih klasa. Generičke klase dozvoljavaju da se postojeće generisane komponente, kao što su servisi i kontroleri, smeste u posebne pakete. Ove generičke klase će uvek biti zamenjene prilikom ponovnog pokretanja generatora.

### Prilagođene klase nasleđuju generike

Prilagođene klase predstavljaju klase koje nasleđuju generičke klase. Za razliku od Generic, prilagođene klase se kreiraju samo ako već ne postoje. Ovaj pristup omogućava da se sačuvaju prilagođene metode i logika koja se odnosi na poslovnu logiku aplikacije, jer se već postojeće prilagođene klase neće koristiti.

### Prednosti ovog pristupa

Korisnicima je dozvoljeno da dodaju sopstvene metode i logiku generisanim komponentama, a te promene se čuvaju kada se generator ponovo pokrene.Ovaj pristup čuva strukturu aplikacije i omogućava razdvajanje generisanih i prilagođenih komponenti. Generičke klase mogu se ponovo koristiti u različitim projektima.

### Implementacija pristupa

Poglavlje će takođe pokriti detalje o tome kako se ovaj pristup implementira, uključujući organizaciju paketa, generičko nasleđivanje klasa i pravila za dodavanje prilagođene logike.

## Generisana aplikacija

Generisana aplikacija predstavlja svestranu backend aplikaciju koju možete koristiti za različite potrebe. Može služiti kao backend za web aplikaciju, servisna aplikacija, aplikacija za upravljanje bazom podataka i više. Pored osnovnih komponenata kao što su repozitorijumi, servisi i kontroleri, omogućava vam i prilagođavanje putem custom komponenata. To znači da možete dodati svoju poslovnu logiku i funkcionalnost. Takođe, aplikacija je idealna za testiranje i podržava razvoj softvera prema najboljim praksama. S obzirom na svoju fleksibilnost, generisana aplikacija ima širok spektar primena i obezbeđuje brz početak razvoja.

Struktura generisane aplikacije je osmišljena kao slojevita arhitektura kako bi omogućila jasno razdvajanje odgovornosti i organizaciju koda. Ovaj koncept olakšava održavanje, proširivost i razumevanje generisanog projekta. Evo glavnih komponenti i strukture generisane aplikacije:

### Paketi (Slika 3.2)

**.model:** Ovde se nalaze generisane klase koje predstavljaju modele podataka, odnosno entitete baze podataka. Svaki entitet predstavlja jednu tabelu u bazi. Takođe, ovde se nalaze i enumeracije koje su definisane u XML modelu.

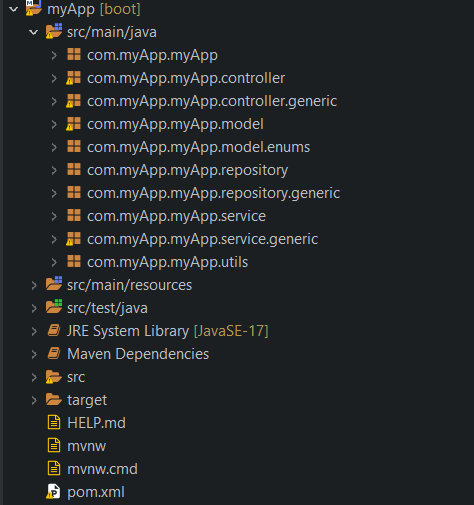
**.model.enums:** Ovaj paket sadrži generisane enumeracije koje se koriste u modelima.

**.repository:** Ovde se nalaze generički (generic) i prilagođeni (custom) repozitorijumi. Generic repozitorijumi sadrže osnovne metode za pristup bazi podataka, dok se custom repozitorijumi koriste za dodatne funkcionalnosti koje korisnici žele da dodaju.

**.service:** Servisni sloj takođe je podeljen na generički i prilagođeni deo. Generic servis sadrži osnovne operacije (npr. CRUD) nad modelima, dok custom servis omogućava korisnicima da dodaju svoje posebne funkcionalnosti.

**.controller:** Kontroler sloj takođe sadrži generičke i custom komponente. Generic kontroleri pružaju osnovne REST API endpointe za generičke servise, dok custom kontroleri omogućavaju dodatne, korisnički definisane operacije.

**.utils:** Ovde se nalazi pomoćni kod koji podržava različite aspekte generisane aplikacije. Na primer, tu se nalazi i ApiUtil klasa koja omogućava upravljanje paginacijom i sortiranjem rezultata.



**Slika 3.2** – Struktura generisane aplikacije – paketi

### Zavisnosti (Dependency Injection):

Da bi omogućila lako zamenu i testiranje komponenti, aplikacija koristi dependency injection. Na primer, servisima se ubrizgavaju odgovarajući repozitorijumi, što omogućava lako testiranje i zamenu implementacije.

**Generičke i Custom Komponente:** Struktura podržava generičke komponente za osnovne operacije, ali takođe omogućava i dodavanje korisničkih komponenti sa prilagođenim funkcionalnostima. Ovo omogućava korisnicima da dodaju svoj specifičan kod bez gubitka prilikom ponovnog generisanja.

**Slojevita Arhitektura:** Generisana aplikacija sledi slojevitu arhitekturu sa jasno definisanim slojevima za modele (entitete), repozitorijume, servise i kontrolere. Ova struktura omogućava odvajanje odgovornosti i olakšava razumevanje kako aplikacija funkcioniše.

**Paginacija i Sortiranje:** Generisana aplikacija takođe podržava paginaciju i sortiranje rezultata, što je posebno korisno za REST API-je koji vraćaju velike količine podataka.

Ova slojevita struktura i fleksibilnost omogućavaju generisanoj aplikaciji da bude prilagodljiva različitim zahtevima projekata i lako održiva. Kroz dodavanje generičkih i custom komponenti, korisnici mogu kreirati aplikaciju koja tačno odgovara njihovim potrebama, dok struktura slojeva čini razvoj i podršku za aplikaciju jasnim i organizovanim.

# Opis implementacije generatora

U ovom poglavlju, dublje ćemo istražiti proces implementacije generatora aplikacija i razmotriti ključne komponente, alate i pristupe koji omogućavaju automatsko generisanje Spring Boot aplikacija na osnovu modela. Implementacija generatora je ključna faza u razvoju alata koji omogućava korisnicima da brzo i efikasno stvaraju aplikacije, čime se znatno ubrzava proces razvoja softvera.

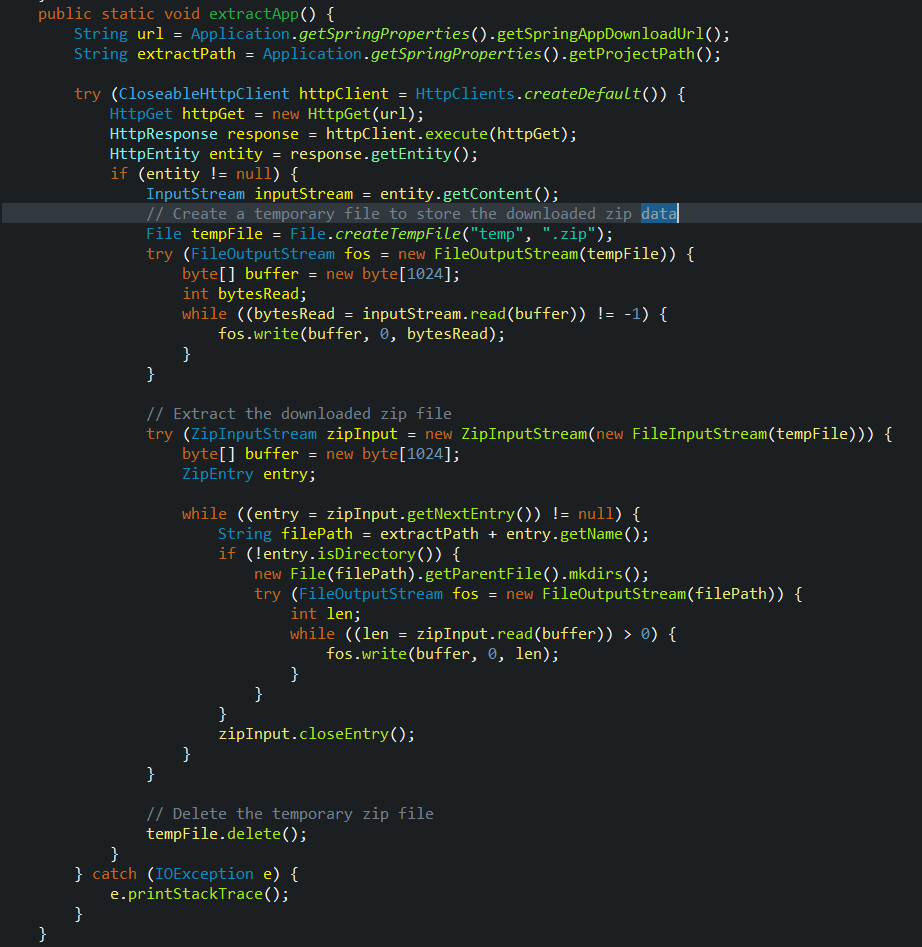
## Ulazni parametri generatora i pocetna struktura aplikacije

U poglavlju "Učitavanje Meta Modela iz XML-a i Konfiguracionih Parametara" istražićemo ključne korake u procesu učitavanja meta modela iz XML datoteke i konfiguracionih parametara iz properties fajla. Ovi ulazni parametri igraju ključnu ulogu u procesu generisanja aplikacije, jer definišu strukturu modela i podešavanja za generisanje.

Ulazni parametri aplikacije predstavljaju osnovnu osnovu za generisanje aplikacije. XML model opisuje strukturu aplikacije, uključujući entitete, atribute i odnose između njih, dok properties fajl definiše konfiguracije kao što su verzija Spring Boot-a, putanja gde će se generisati aplikacija, JDBC URL za bazu podataka i druge ključne postavke. Pravilno definisani i učitani ulazni parametri su ključni za uspešno generisanje aplikacije.

Važno je napomenuti da aplikacija može da se pokrene kao izvršni .jar fajl, što omogućava korisnicima da je izvršavaju bez potrebe za eksternim razvojnim okruženjima. Ovaj pristup čini generisanje i pokretanje aplikacije jednostavnim i pristupačnim, omogućavajući korisnicima da brzo dobiju funkcionalnu aplikaciju bez komplikacija. Uslovi za uspešno generisanje uključuju da se XML model i properties fajl nalaze u istom folderu kao i izvršni .jar fajl generatora. Ovo obezbeđuje jednostavno pronalaženje i učitavanje potrebnih datoteka, što je ključno za inicijalizaciju procesa generisanja. U narednim sekcijama ovog poglavlja, detaljno ćemo razmotriti kako se izvršava učitavanje XML modela i konfiguracionih parametara, kao i kako se ovi podaci dalje koriste u procesu generisanja Spring Boot aplikacije.

Podaci iz konfiguracionog fajla igraju ključnu ulogu prilikom prvog pokretanja generatora za određenu putanju, omogućavajući optimizaciju samog procesa. Konfiguracioni fajl sadrži važne parametre koji su prethodno konfigurisani prema potrebama korisnika. Ovi parametri uključuju URL projekta sa start.spring.io [8], putanju na lokalnom sistemu gde će se ekstrahovati projekat, i druge postavke. Prilikom prvog pokretanja generatora za određenu putanju, on čita konfiguracioni fajl kako bi dobio vrednosti ovih parametara. Ovo eliminiše potrebu da korisnik ručno unosi ove podatke svaki put kada želi da generiše novi projekat. Na osnovu ovih parametara, generator može automatski da preuzme projekt sa start.spring.io, bez potrebe za ručnim unošenjem URL adrese svaki put. Takođe, generator može automatski postaviti putanju na lokalnom sistemu gde će projekat biti ekstrahovan (Slika 4.1). Ovaj proces olakšava i ubrzava pokretanje generatora, štedi vreme korisnika i smanjuje mogućnost grešaka pri unosu ovih osnovnih informacija. Nakon prvog pokretanja generatora, parametri se mogu lako ažurirati u konfiguracionom fajlu ukoliko se promene zahtevi ili potrebno je generisati projekte na različitim lokacijama. Ovo omogućava prilagodljivost i lakoću upravljanja parametrima generisanja. Korišćenje konfiguracionog fajla na ovaj način doprinosi efikasnosti i praktičnosti celokupnog procesa generisanja aplikacije. Korisnicima omogućava brži početak i olakšava održavanje parametara konfiguracije.



**Slika 4.1** – Primer modela napisaong u XML-u

## Struktura generatora

Struktura generatora (Slika 4.2) je osmišljena kako bi omogućila fleksibilnost i proširivost. Ključni koncepti i komponente u ovoj strukturi uključuju: Writer Interfejs: Ovaj interfejs definiše osnovne metode koje svaka komponenta za generisanje mora implementirati. Na primer, JavaEntityWriter, JavaServiceWriter, i slične komponente implementiraju ovaj interfejs. Ovaj pristup omogućava jednostavno dodavanje novih pisaca (writer) za generisanje koda za različite delove aplikacije ili za podršku za različite jezike.

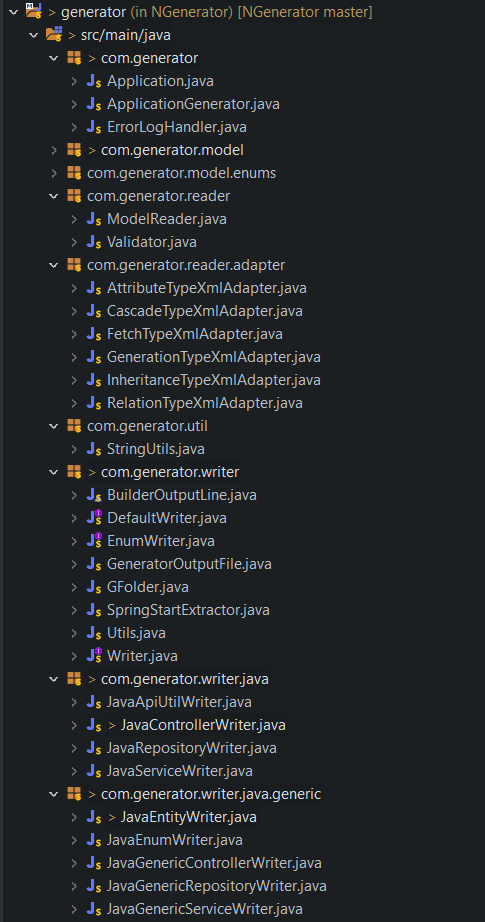
**Paket writer.java:** Ovde se nalaze komponente za generisanje Java aplikacije. Na primer, JavaEntityWriter je odgovoran za generisanje Java koda za entitete, dok JavaServiceWriter generiše kod za servise. Kroz ovakvu organizaciju, dodavanje ili ažuriranje komponenti za generisanje postaje efikasan proces.

**Proširivost za Različite Jezike:** Struktura generatora omogućava proširivost za generisanje koda u različitim programskim jezicima. Na primer, ako se odluči za podršku za Kotlin, može se jednostavno dodati novi paket (npr. writer.kotlin) sa odgovarajućim komponentama za generisanje koda za Kotlin.

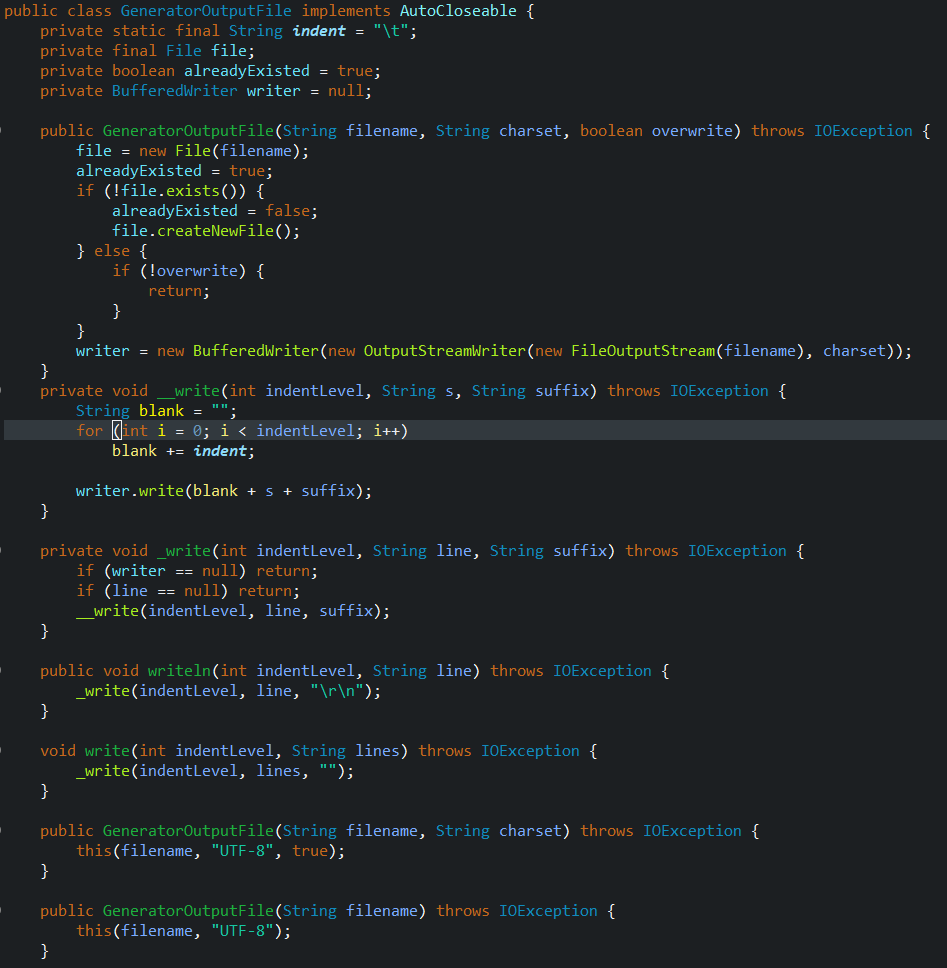
**Konfiguracioni Fajl (Properties):** Generator koristi konfiguracioni fajl (properties) za podešavanje različitih parametara, kao što su verzija Spring Boot-a, putanja za generisane fajlove, i druge relevantne opcije. Ovo omogućava laku konfiguraciju generisanja aplikacije bez potrebe za ručnim izmenama izvornog koda generatora.

**Referenca na XML Meta Model:** Generator koristi XML meta model koji je deo istog projekta (nalazi se u paketu model) kao osnova za generisanje aplikacije. Ovaj model sadrži informacije o entitetima, atributima, relacijama i drugim elementima aplikacije.

**Fleksibilnost za Budućnost:** Struktura generatora je osmišljena tako da omogući laku proširivost i unapređenje u budućnosti. Kroz definisanje različitih writer komponenti i paketa za različite jezike, projekat je spreman da podrži različite tehnologije i jezike bez potrebe za velikim izmenama u kodu generatora. Ova struktura omogućava generisanje aplikacija temeljenih na XML meta modelu uz fleksibilnost za različite konfiguracije i proširenja, čineći generator moćnim alatom za ubrzanje razvoja aplikacija.



**Slika 4.2** – Primer modela napisaong u XML-u



**Slika 4.3** – EntityWriter kreiranje entiteta

Klasa GeneratorOutputFile (Slika 4.3) je odgovorna za upisivanje sadržaja u izlazni fajl. Ova klasa se koristi za generisanje različitih delova izlaznog koda u fajlu tokom izrade generatora aplikacije. Evo kratkog opisa klase i njene metode writeln: GeneratorOutputFile omogućava kreiranje i upisivanje sadržaja u fajl. Konstruktor ove klase omogućava kreiranje novog fajla ili otvaranje postojećeg, u zavisnosti od toga da li fajl već postoji. Takođe, postoji opcija da li želimo prepisati postojeći fajl ili dodavati sadržaj na postojeći. Metoda writeln se koristi za upisivanje linije teksta u fajl. Ova metoda prima dva parametra: indentLevel koji označava nivo uvlačenja teksta, i line koji predstavlja sam tekst koji treba upisati. Tekst se upisuje sa odgovarajućim brojem tabulatora (indentacija) u zavisnosti od vrednosti indentLevel. Opciono, može se koristiti dodatni sufiks (u ovom slučaju, prelazak u novi red \r\n). Ova klasa je korisna za organizaciju i upisivanje generisanog koda u izlazne fajlove, čime se omogućava kreiranje aplikacije ili komponenata generisanih pomoću generatora. Metoda writeln je posebno korisna za formatiranje generisanog koda tako da bude čitljiv i organizovan.

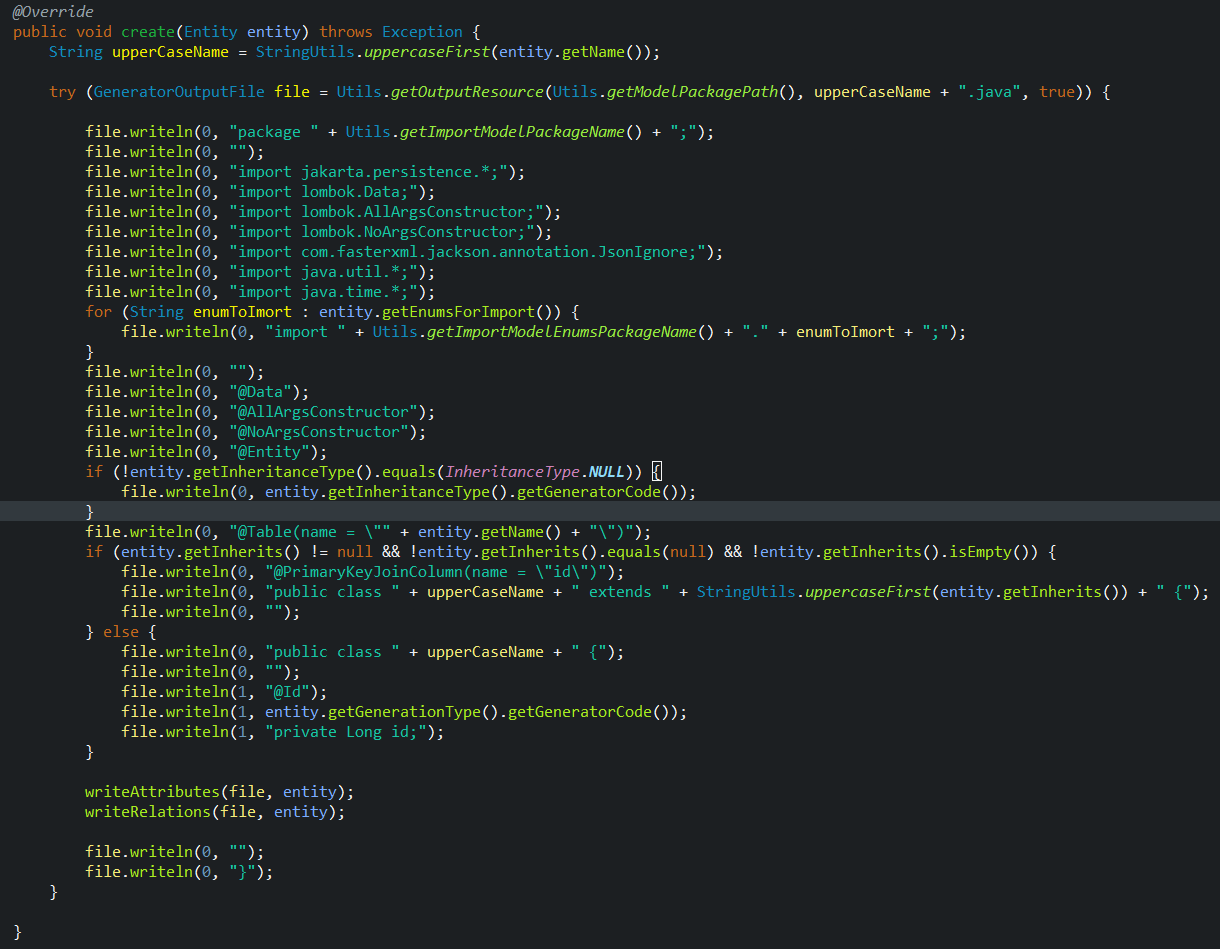
## Generik komponente

Generičke komponente predstavljaju ključnu osnovu generisane aplikacije, obuhvatajući klasične operacije koje većina aplikacija zahteva, uključujući CRUD operacije sa generisanim endpoint-ima. Da bi omogućila generisanje kako generičkih, tako i prilagođenih komponenata, aplikacija koristi layered arhitekturu.

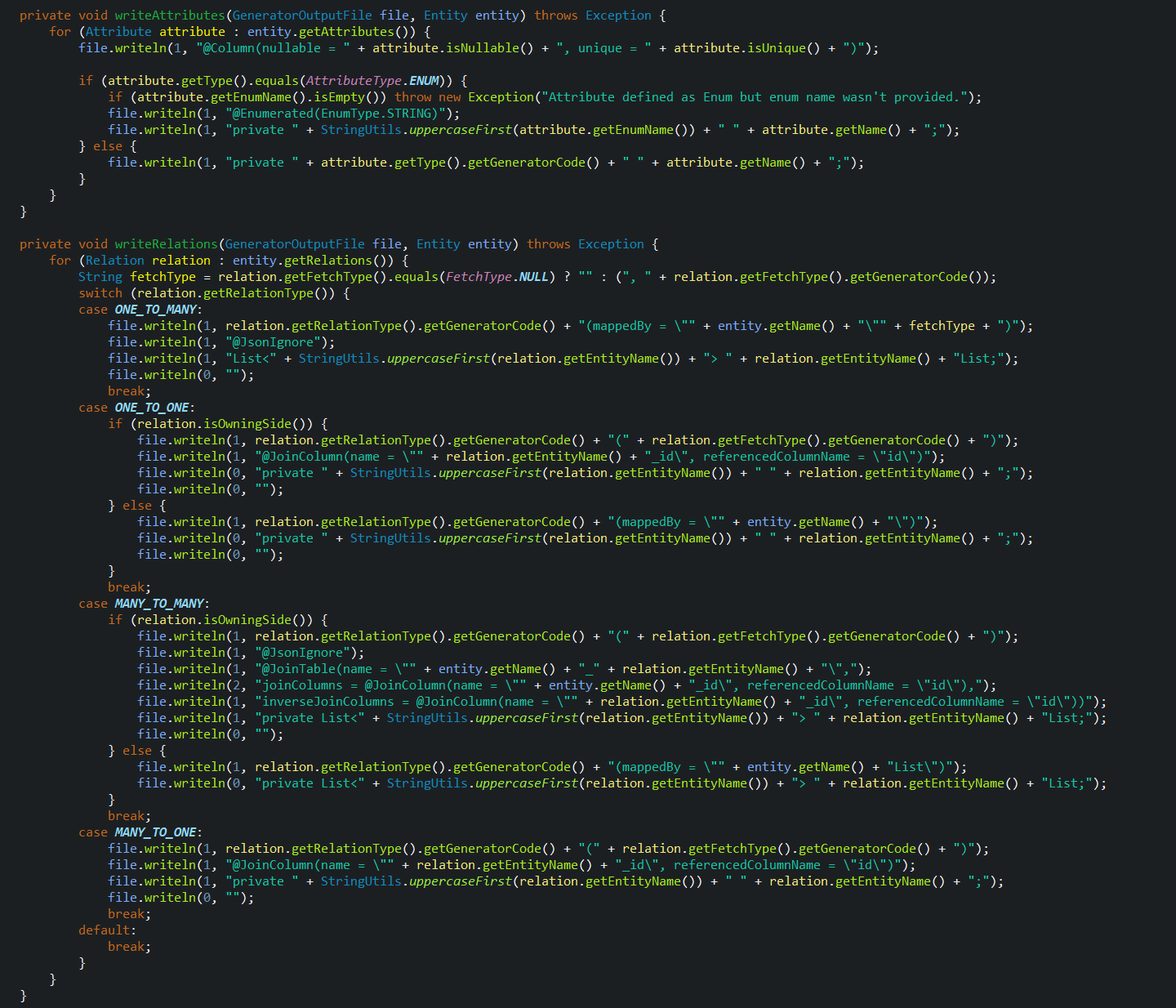
### GenericEntityWriter

EntityWriter (slika 4.4) je ključna komponenta generatora odgovorna za generisanje veza i relacija između objekata na osnovu podataka ucitanih iz XML modela. Ova komponenta se ističe po tome što je uvek generička, što znači da nema custom varijante koja bi je nasledila.

Metoda create ove komponente generiše Java klase koje predstavljaju entitete (objekte) u aplikaciji. Generiše se Java kod koji uključuje anotacije, atribute, veze i relacije između entiteta na osnovu podataka iz XML modela. Ova metoda takođe proverava i pravilno nasleđivanje ukoliko postoji. Metoda writeAttributes se koristi za generisanje atributa za entitete, uključujući tipove podataka i odgovarajuće anotacije za bazu podataka. Ovde se takođe tretira i poseban slučaj kada je atribut tipa ENUM. Metoda writeRelations generiše kod za veze između entiteta, uključujući "one-to-many," "many-to-one," "one-to-one," i "many-to-many" relacije. Kod se generiše u zavisnosti od vrste relacije i toga da li je trenutni entitet vlasnik relacije. Ova komponenta je od suštinskog značaja za generisanje koda koji odražava strukturu baze podataka i kako entiteti međusobno komuniciraju. Obezbeđuje da entiteti budu pravilno mapirani na bazu podataka i da se očuva konzistentnost i integritet podataka u generisanoj aplikaciji.



**Slika 4.4** – GenericEntityWriter



**Slika 4.5** – GenericEntityWriter metode

### GenericEnumWriter

GenericEnumWriter (Slika 4.6) je odgovoran za generisanje Java enumeracija (enum) na osnovu informacija iz EnumModel objekta. Metoda create je relativno jednostavna i izgleda kako sledi U generisanom enumu, postavlja se paket (package) na osnovu konfiguracije i naziva enuma. Generiše se definicija enuma u formatu public enum { ... }. Unutar enuma, za svaku vrednost (vrednosti su navedene u EnumModel) generiše se red sa vrednošću, koristeći enumType.toUpperCase() + "," kako bi se osiguralo da su vrednosti napisane velikim slovima i da su odvojene zarezom. Kako se može primetiti, ova metoda je relativno jednostavna, i njen zadatak je da generiše Java enum klase na osnovu podataka iz EnumModel.

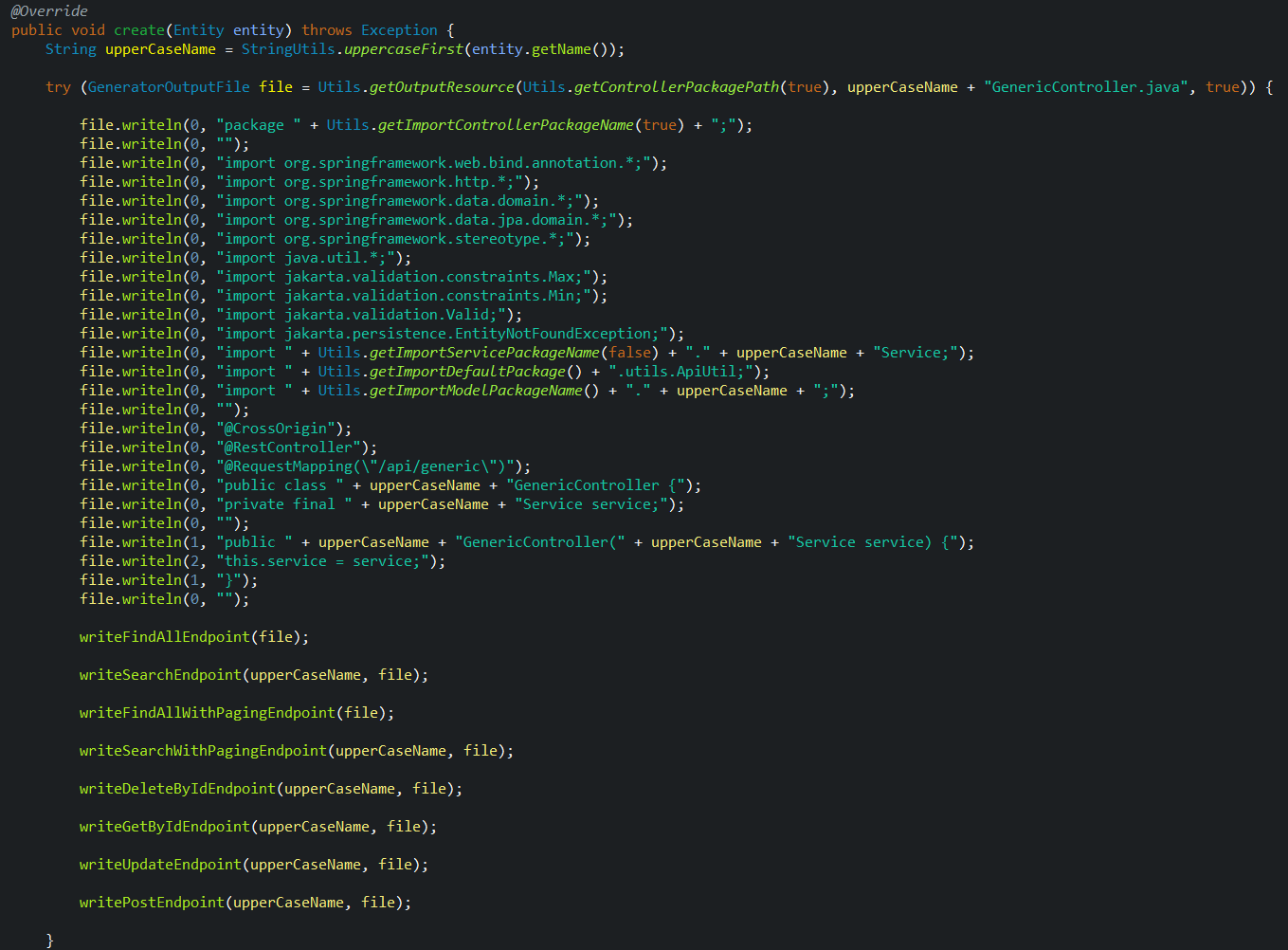


**Slika 4.6** – GenericEnumWriter

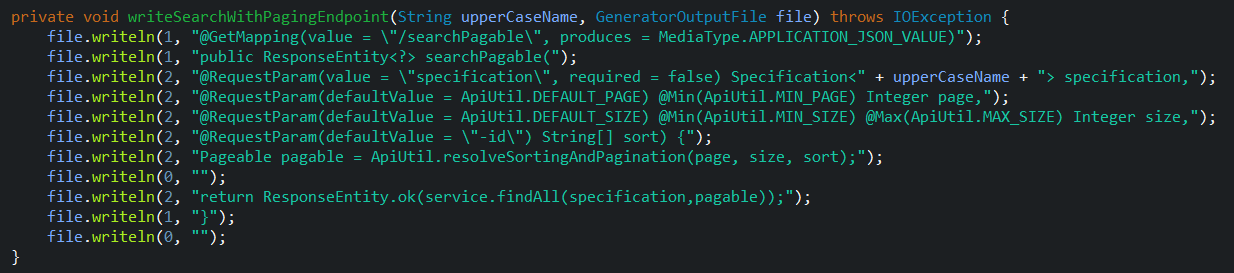
### GenericControllerWriter

GenericControllerWriter (Slika 4.7) ima zadatak generisanja kontrolera za entitete unutar generisane aplikacije. Evo kako ova metoda radi: Prvo, otvara GeneratorOutputFile resurs za generisanje Java klase kontrolera, a ime klase određuje se na osnovu imena entiteta. Postavlja se odgovarajući paket (package) za kontroler na osnovu konfiguracije i naziva paketa. Importuju se potrebne biblioteke i klase za kontroler, uključujući Spring Framework anotacije i druge klasa za funkcionalnosti kontrolera. Generiše se Java klasa za kontroler sa odgovarajućim anotacijama kao što su @RestController i @RequestMapping za definisanje RESTful endpointa. Konstruktor klase kontrolera se generiše sa injektovanjem servisa za odgovarajući entitet. Zatim, metode za različite endpointe se generišu. Na primer, metode za pretragu, brisanje, ažuriranje i kreiranje entiteta su implementirane.

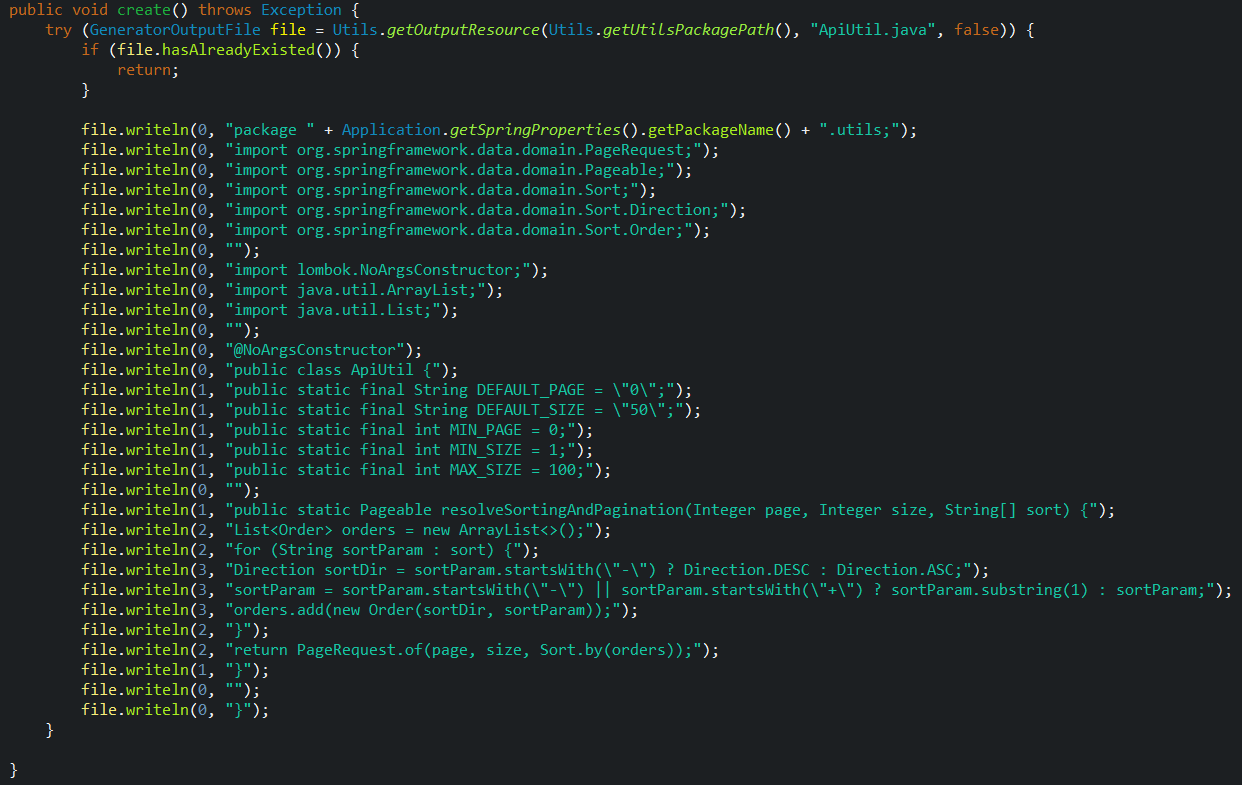
Na primer writeSearchWithPagingEndpoint(Slika 4.8) metoda se koristi za generisanje endpointa za pretragu sa paginacijom. Ova metoda dodaje odgovarajuće anotacije i parametre za pretragu sa mogućnošću filtriranja, straničenja i sortiranja rezultata. Primenjuju se parametri kao što su specifikacija, broj stranice, veličina stranice i sortiranje, a zatim se poziva servis za pretragu sa ovim parametrima. Ova funkcionalnost je omogućena zahvaljujući ApiUtilWriter (samo se jednom generiše, Slika 4.9) klasi i njenim metodama, pre svega metodi resolveSortingAndPagination. Ova metoda se koristi za rešavanje i pravilno definisanje parametara vezanih za straničenje i sortiranje rezultata pretrage. Pretraga sa paginacijom postaje jednostavna i prilagodljiva parametrima koji se mogu lako podesiti putem RESTful endpointa, čime se omogućava korisnicima da pretragu prilagode svojim potrebama i preferencijama.



**Slika 4.7** – GenericControllerWriter



**Slika 4.8** – writeSearchWithPagingEndpoint metoda



**Slika 4.9** – ApiUtilWriter

### GenericRepositoryWriter

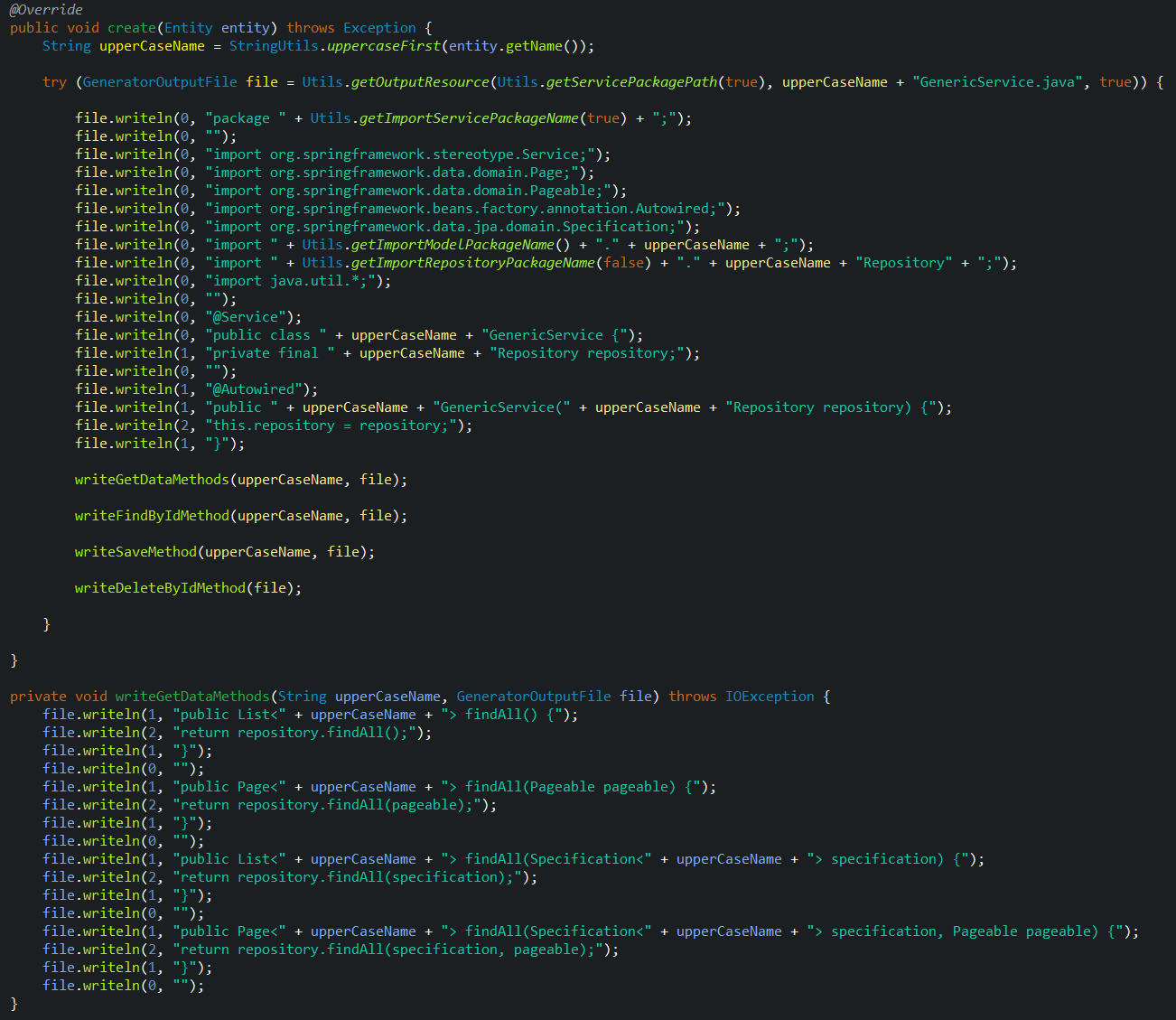
GenericRepositoryWriter (Slika 4.10) klasa se koristi za generisanje repozitorijumske komponente za JPA (Java Persistence API). Ova komponenta je odgovorna za interakciju sa bazom podataka i pruža osnovne CRUD (Create, Read, Update, Delete) operacije za rad sa entitetima GenericRepositoryWriter generiše repozitorijum za određeni entitet. Na osnovu metapodataka o entitetu koji se prenose kao argument (parametar entity), metoda create generiše Java interfejs koji nasleđuje JpaRepository i definiše metode za pretragu i dohvat entiteta. Prvo se određuje naziv entiteta, a zatim se koristi GeneratorOutputFile za kreiranje izlaznog fajla koji će sadržati generisani kod. U izlaznom fajlu se definiše paket za repozitorijum, importuju se potrebne biblioteke i klase, kao što su Specification i Pageable za pretragu i straničenje. Takođe, importuje se i odgovarajući entitet (entity) kako bi repozitorijum bio vezan za taj entitet. Generiše se Java interfejs za repozitorijum (Repository) koji je anotiran sa @Repository. Interfejs nasleđuje JpaRepository, a definisane su metode za pretragu sa specifikacijama (specifikacije se koriste za definisanje uslova pretrage) i sa straničenjem. Ova metoda omogućava generisanje repozitorijumske komponente koja će omogućiti rad sa entitetima u aplikaciji. Repozitorijumi su ključni deo Spring Data JPA i omogućavaju jednostavan pristup i manipulaciju podacima u bazi.



**Slika 4.10** – Primer modela napisaong u XML-u

### GenericServiceWriter

GenericServiceWriter (Slika 4.11) klasa je odgovorna za generisanje servisne komponente u Spring Boot aplikacijama. Servisne komponente obično sadrže poslovnu logiku aplikacije i služe za obradu podataka, komunikaciju sa repozitorijumima (baza podataka), i pružaju servise koji se koriste u kontrolerima. Evo kratkog opisa metoda i klasa iz ServiceWriter: create(Entity entity) metoda: Ova metoda generiše servis za određeni entitet. Na osnovu metapodataka o entitetu (parametar entity) i paketa u kojem se servis treba nalaziti, generiše se Java servis koji je anotiran sa @Service. Servis ima zavisnost prema odgovarajućem repozitorijumu (u ovom slučaju prema Repository za dati entitet) koji se obezbeđuje kroz konstruktor servisa. Takođe, generiše se set metoda za dohvatanje i manipulaciju podacima u skladu sa definisanim operacijama. writeGetDataMethods metoda: Ova metoda generiše metode za dohvatanje podataka iz baze za dati entitet. To uključuje metode za dohvatanje svih entiteta, entiteta sa straničenjem, kao i metode za filtriranje podataka na osnovu specifikacija. Ove metode koriste odgovarajući repozitorijum za dohvatanje podataka. Obe klase (kontroleri i servisi) su ključne komponente u arhitekturi Spring aplikacije. Kontroleri služe za mapiranje HTTP zahteva na odgovarajuće servisne metode, dok servisi obavljaju poslovnu logiku i interakciju sa podacima. Ovakva arhitektura omogućava odvajanje poslovne logike od prezentacionog sloja i pruža jasnu organizaciju koda.



**Slika 4.11** – Primer modela napisaong u XML-u

## Custom komponente

(Custom) ServiceWriter (Slika 4.12) klasa je odgovorna za generisanje praznih prilagođenih (custom) servisa u Spring Boot aplikacijama. Ova klasa pruža korisnicima mogućnost dodavanja prilagođenih funkcionalnosti u generisane aplikacije. Evo kratkog opisa metode i dodatnih informacija: create(Entity entity) metoda: Ova metoda generiše custom servis za određeni entitet, ako već ne postoji na datoj putanji. Prilagođeni servisi su prazne klase koje su označene sa @Service anotacijom. Svrha ovih prilagođenih servisa je omogućiti korisnicima da implementiraju svoje custom funkcionalnosti i poslovnu logiku koje se odnose na određeni entitet. Generiše se kao prazna klasa kako bi se osiguralo da korisnički kod nikada ne bude zamenjen ili izgubljen prilikom ponovnog pokretanja generatora. Dodatno, svi custom servisi implementirani u generisanoj aplikaciji su dostupni putem dependency injection mehanizma, zahvaljujući upotrebi @Service anotacije. Ako korisnik napiše svoju implementaciju custom servisa, ta implementacija će biti korišćena umesto generisane. Ovo se postiže označavanjem korisničke implementacije sa @Primary anotacijom, što je standardni način za označavanje primarne komponente koja se koristi u slučaju višestrukih implementacija za isti tip servisa. Ova funkcionalnost omogućava korisnicima da prilagode aplikaciju svojim potrebama i da dodaju custom logiku na bezbedan način bez rizika da će je izgubiti prilikom ažuriranja ili ponovnog generisanja aplikacije.



**Slika 4.12** – Primer modela napisaong u XML-u

# Zaključak

Ovaj generator aplikacija ima značajnu budućnost i donosi niz prednosti koje omogućavaju laku prilagodbu, proširivost i održavanje. Evo zaključka koji sumira ključne tačke i buduće perspektive ovog sistema: Fleksibilnost i Prilagodljivost: Generator aplikacija pruža korisnicima fleksibilnost da definišu modele i konfiguracije aplikacije putem XML-a i properties fajlova. Ovo omogućava laku prilagodbu za različite projekte i poslovne potrebe. Višejezičnost i Višeprojektna Podrška: Implementacija struktura generatora omogućava podršku za više programskih jezika i različite projekte. Trenutna implementacija se fokusira na Java, ali se može proširiti na druge jezike i tehnologije. Laka Proširivost: Klasna struktura i konceptualni okvir generatora omogućavaju laku proširivost. Novi pisaci (Writers) i servisi se lako dodaju kako bi podržali dodatne funkcionalnosti i tehnologije. Očuvanje Korisničkih Prilagodbi: Prilagođeni kod koji korisnici dodaju u generisane aplikacije se očuvava kroz prazne custom komponente. Ovo osigurava da korisnički kod nikada ne bude zamijenjen ili izgubljen prilikom ponovnog generisanja aplikacije. Održivost i Skalabilnost: Struktura generatora olakšava održavanje i skaliranje generisanih aplikacija. Izvorni kod koji generiše je dobro organizovan i pridržava se najboljih praksi razvoja softvera. Kontinuirani Razvoj: Ovaj generator aplikacija ima potencijal za kontinuirani razvoj i poboljšanja. Dodavanje novih pisaca, proširenje funkcionalnosti i poboljšanja su mogući u skladu sa potrebama i povratnim informacijama korisnika. U zaključku, ovaj generator aplikacija otvara put za jednostavno i brzo kreiranje različitih tipova aplikacija, uz očuvanje prilagođenih korisničkih implementacija. Sa njegovim struktuiranim pristupom i mogućnošću prilagođavanja, očekuje se da će postati važan alat za razvoj i održavanje softverskih sistema u budućnosti.

# 6. Literatura

1. Spring   
   <https://spring.io/projects/spring-boot>
2. Jakarta XML Binding

<https://en.wikipedia.org/wiki/Jakarta_XML_Binding>

1. Medix

https://www.mendix.com/

1. OutSystems

https://www.outsystems.com/

1. Quick Base   
   <https://microservices.io/book>
2. OpenAPI-a

<https://www.openapis.org/>

1. Spring initializr

<https://start.spring.io/>

1. Spring initializr

<https://start.spring.io/>

1. Spring Data JPA

<https://spring.io/projects/spring-data-jpa>