|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ** |  |

Милош Гравара

**Анализа имплементације и перформанси изабраних алата за објектно-релационо мапирање**

Дипломски рад

- Основне академске студије -

Нови Сад, 2023.

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Датум: |
|  |
| **ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА** | Лист: |
| 1/1 |

*(Податке уноси предметни наставник - ментор)*

| Врста студија: | **Основне академске студије** |
| --- | --- |
| Студијски програм: | **Рачунарство и аутоматика** |
| Руководилац студијског програма: | **проф. др Милан Рапаић** |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент: | **Милош Гравара** | Број индекса: | **RA 60/2019** |
| Област: | **Електротехничко и рачунарско инжењерство** | | |
| Ментор: | **Др Милан Стојков, доцент** | | |
| НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:   * проблем – тема рада; * начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; * литература | | | |

**НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:**

|  |
| --- |
| **Анализа имплементације и перформанси изабраних алата за објектно-релационо мапирање** |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
| 1. Анализирати стање у области.  2. Израдити спецификацију захтева софтверског решења.  3. Израдити спецификацију дизајна софтверског решења.  4. Имплементирати софтверско решење према израђеној спецификацији.  5. Тестирати имплементирано софтверско решење.  6. Документовати (1), (2), (3), (4) и (5). |

|  |  |
| --- | --- |
| Руководилац студијског програма: | Ментор рада: |
|  |  |

|  |
| --- |
| Примерак за:  - Студента;  - Ментора |

# КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

|  |  |  |  |
| --- | --- | --- | --- |
| Редни број, **РБР**: | |  | |
| Идентификациони број, **ИБР**: | |  | |
| Тип документације, **ТД**: | | монографска публикација | |
| Тип записа, **ТЗ**: | | текстуални штампани документ | |
| Врста рада, **ВР**: | | дипломски рад | |
| Аутор, **АУ**: | | Милош Гравара | |
| Ментор, **МН**: | | др Милан, Стојков | |
| Наслов рада, **НР**: | | Анализа имплементације и перформанси изабраних алата за објектно-релационо мапирање | |
| Језик публикације, **ЈП**: | | српски | |
| Језик извода, **ЈИ**: | | српски / енглески | |
| Земља публиковања, **ЗП**: | | Србија | |
| Уже географско подручје, **УГП**: | | Војводина | |
| Година, **ГО**: | | 2023 | |
| Издавач, **ИЗ**: | | ауторски репринт | |
| Место и адреса, **МА**: | | Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6 | |
| Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога) | | 0 / 0 / 0 / 0 / 0 / 0 / 0 | |
| Научна област, **НО**: | | Софтверско инжењерство и информационе технологије | |
| Научна дисциплина, **НД**: | | Софтверско инжењерство | |
| Предметна одредница/Кључне речи, **ПО**: | | Објектно-релационо мапирање, перформансе, упоређивање | |
| **УДК** | |  | |
| Чува се, **ЧУ**: | | Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад | |
| Важна напомена, **ВН**: | |  | |
| Извод, **ИЗ**: | | апстракт – један пасус који добро описује суштину рада – проблем, мотивацију, назнаку решења и резултат. | |
| Датум прихватања теме, **ДП**: | |  | |
| Датум одбране, **ДО**: | |  | |
| Чланови комисије, **КО**: | Председник: | др Име Презиме, звање |
| Члан: | др Име Презиме, звање  др Име Презиме, звање | |
|  | Члан, ментор: | др Име Презиме, звање | |
|  |  | Потпис ментора | |

# KEY WORDS DOCUMENTATION

|  |  |  |  |
| --- | --- | --- | --- |
| Accession number, **ANO**: | |  | |
| Identification number, **INO**: | |  | |
| Document type, **DT**: | | monographic publication | |
| Type of record, **TR**: | | textual material | |
| Contents code, **CC**: | | bachelor thesis | |
| Author, **AU**: | | Milos Gravara | |
| Mentor, **MN**: | | Milan Stojkov, associate professor, PhD | |
| Title, **TI**: | | Analysis of implementation and performance of selected object-relational mapping tools | |
| Language of text, **LT**: | | Serbian | |
| Language of abstract, **LA**: | | Serbian / English | |
| Country of publication, **CP**: | | Serbia | |
| Locality of publication, **LP**: | | Vojvodina | |
| Publication year, **PY**: | | 2023 | |
| Publisher, **PB**: | | author’s reprint | |
| Publication place, **PP**: | | Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6 | |
| Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes) | | 0 / 0 / 0 / 0 / 0 / 0 / 0 | |
| Scientific field, **SF**: | | Software Engineering and Information Technologies | |
| Scientific discipline, **SD**: | | Software Engineering | |
| Subject/Key words, **S**/**KW**: | | Ključne reči na engleskom | |
| **UC** | |  | |
| Holding data, **HD**: | | Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad | |
| Note, **N**: | |  | |
| Abstract, **AB**: | | Prevod apstrakta na engleski | |
| Accepted by the Scientific Board on, **ASB**: | |  | |
| Defended on, **DE**: | |  | |
| Defended Board, **DB**: | President: | др Име Презиме, звање |
| Member: | др Име Презиме, звање  др Име Презиме, звање | |
|  | Member, Mentor: | др Име Презиме, звање | |
|  |  | Mentor’s signature | |

**SADRŽAJ**

[КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА 5](#_Toc145493104)

[KEY WORDS DOCUMENTATION 7](#_Toc145493105)

[1. Увод 11](#_Toc145493106)

[2. Објектно-релационо мапирање 13](#_Toc145493107)

[2.1 Наслеђивање 14](#_Toc145493108)

[2.1.1 Једна табела по ланцу наслеђивања 14](#_Toc145493109)

[2.1.2 Једна табела по класи 15](#_Toc145493110)

[2.1.3 Конкретна табела по ланцу наслеђивања 16](#_Toc145493111)

[2.2 Асоцијације 17](#_Toc145493112)

[2.3 Добављање података 18](#_Toc145493113)

[*2.3.1* *Lazy Loading* 18](#_Toc145493114)

[2.3.2 Проблем N+1 упита 18](#_Toc145493115)

[2.3.3 *Eager Loading* 19](#_Toc145493116)

[2.4 Кеширање 20](#_Toc145493117)

[2.5 Обрасци понашања ОРМ алата 20](#_Toc145493118)

[*2.5.1* *Identity Map* 20](#_Toc145493119)

[*2.5.2* *Unit of Work* 21](#_Toc145493120)

[2.6 Употреба ОРМ алата у развоју апликација 21](#_Toc145493121)

[3. Преглед релевантне литературе 23](#_Toc145493122)

[3.1 Методологија претраге литературе 23](#_Toc145493123)

[3.2 Опис претходних истраживања 23](#_Toc145493124)

[3.3 Уочени недостаци у литератури 25](#_Toc145493125)

[3.3.1 Непотпуност 25](#_Toc145493126)

[3.3.2 Одсуство анализе Python ОРМ алата 25](#_Toc145493127)

[4. Преглед изабраних ОРМ алата 27](#_Toc145493128)

[*4.1* *Еntity Framework* 28](#_Toc145493129)

[4.1.1 Aрхитектура алата 28](#_Toc145493130)

[1 asdasd 28](#_Toc145493131)

[4.2 DEMONSTRACIJA 29](#_Toc145493132)

[5. ZAKLJUČAK 31](#_Toc145493133)

[LITERATURA 33](#_Toc145493134)

[BIOGRAFIJA 35](#_Toc145493135)

# Увод

Модерне објектно-оријентисане апликације, поготово веб апликације, користе релационе базе података за перзистенцију података. Због разлика објектне и релационе парадигме, повезивање апликације са релационом базом података и писање упита је често захтеван посао за програмере који може да одузме доста времена у процесу развоја софтвера. Како би се олакшао и убрзао овај процес, заживела је идеја објектно-релационих мапера (ОРМ) који аутоматски премошћавају разлике у наведеним парадигмама, формално познатије као *Object-relational impedance mismatch*. Са друге стране, заступљено је мишљење како коришћење ових механизама може знатно да наруши перформансе апликација приликом комуникације са базом података, због чега се у многим артиклима на интернету наводу како су ОРМ алати анти-шаблон у развоју.

Идеја овог рада је, да се за најпопуларније програмске језике у развоју апликација, истраже начини имплементације одговарајућих алата за објектно-релационо мапирање, као и да се одреди њихов утицај на деградацију перформанси приликом њихове употребе. За потребе истраживања, одабрани алати су: *Entity Framework, Hibernate* и *SQLAlchemy* због велике заступљености језика С#, Јава и Пајтон у развоју модерних апликација.

Први део рада, поглавље 2, односи се на генералан поглед на ОРМ алате и основне концепте који су заступљени у њиховој имплементацији. У наредном поглављу, пружен је кратак увид у релевантну литературу која се бавила овом тематиком и идентификоване су празнине које овај рад тежи да попуни. Четврто поглавље, представља детаљан опис сваког од одабраних ОРМ алата и објашњена је имплементација најбитнијих механизама којима се користе наведени алати. У петом поглављу, представљена је мeтодологија којом се анализирао утицај изабраних ОРМ алата на перформансе апликације, као и резултати спроведеног истраживања. Последње поглавље представља свеукупан поглед на вршено истраживање и закључке изведене из добијених резултата. Додатно, наведене су могућности за проширивање рада у будућим истраживањима.

# Објектно-релационо мапирање

Као што је већ наведено у претходном поглављу, сврха ОРМ алата је да премосте разлику између објектне и релационе парадигме, познатију као oбјектно-релациона неусклађеност импеданси (*Object-relational impedance mismatch*). Овај термин је преузет из електроинжењерства и представља проблем који се јавља приликом повезивања два струјна кола са различитим импедансама, услед којег долази до слабљења преноса снаге. Генерално, термин неусклађеност импеданси, користи се у инжењерству и рачунарским наукама да се укаже на немогућност ефикасне интеракције између два система.

С тим на уму, може се повући паралела како би се описале некомпатибилности између објектно-оријентисане и релационе парадигме које изазивају проблеме у дизајну и имплементацији апликација заснованих на њиховој комбинацији.

Неусклађеност две парадигме настаје због тога што оне, у основи, имају различит поглед на концепт података. Програми написани у објектно-оријентисаним језицима се фокусирају на објекте, инстанце класе која садржи атрибуте, који представљају структуру, али и методе којима се дефинише понашање инстанце. Објекти се не гледају само кроз призму података који у себи енкапсулирају, већ и кроз способност трансформације тих података и интеракције са другим објектима са којима могу податке да размењују. Са друге стране, релациона парадигма заснована је на математичком појму релације и основни концепт инстанце представљен је редом у табели. Овим се акценат ставља на структуру података, док је могућност трансформације података одвојена од структуре.

Разлике објектно оријентисане и релационе парадигме које изазивају проблеме у развоју могу да се манифестују кроз неколико различитих концепата који ће бити наведени у наредним секцијама.

## Наслеђивање

У објектно-оријентисаном програмирању, наслеђивање омогућава класама да усвајају атрибуте и методе својих наткласа, изграђујући тако хијерархију објеката. Meђутим, наилази се на изазове када се покуша применити ова хијерархија на релациону парадигму. Релациони модели су фокусирани на структуралне односе између података, без обраћања посебне пажње на хијерархију између табела.

Објектно-релациони мапери пружају неколико начина за решавање овог проблема:

### Једна табела по ланцу наслеђивања

Све класе из хијерархије наслеђивања се представљају једном табелом. Колоне табеле представљају својства свих класа. Увођењем посебног дискриминатор обележја, за сваки ред у табели се означава која је његова одговарајућа класа у дефинисаној хијерархији, чиме се омогућава мапирање.

Предности овог приступа су што добро прати еволуцију објектног модела, јер су модификације у шеми знатно олакшане, као и што нема потребе за операцијама спајања (*join*) које су скупе из аспекта извршавања упита.

Са друге стране, јавља се проблем неискоришћеног простора. Пошто сваки ред у табели има колоне сваког подтипа, овакво мапирање ће резултовати у мноштву колона са непостојећим (*null*) вредностима. Такође, величина овакве табеле може да представља уско грло система приликом операција читања и писања.

A screenshot of a computer

Description automatically generatedСлика 2.1 Једна табела по ланцу наслеђивања

### Једна табела по класи

Основна (родитељска) класа има своју табелу која садржи ссвојства која су заједничка за све изведене класе. Свака изведена (дечја) класа такође има своју табелу која садржи само она својства која су јој специфична. Табеле изведених класа обично имају страни кључ који се повезује са примарним кључем родитељске табеле.

A diagram with text and arrows

Description automatically generated with medium confidence Предности овог приступа су што је база података нормализована, што значи да неће постојати додатне колоне које носе непостојеће вредности као што је то случај са претходним начином.

Ипак, проблем код овог приступа је што читање целокупног објекта често захтева операције спајања између основне и изведене табеле, што може да деградира перформансе.

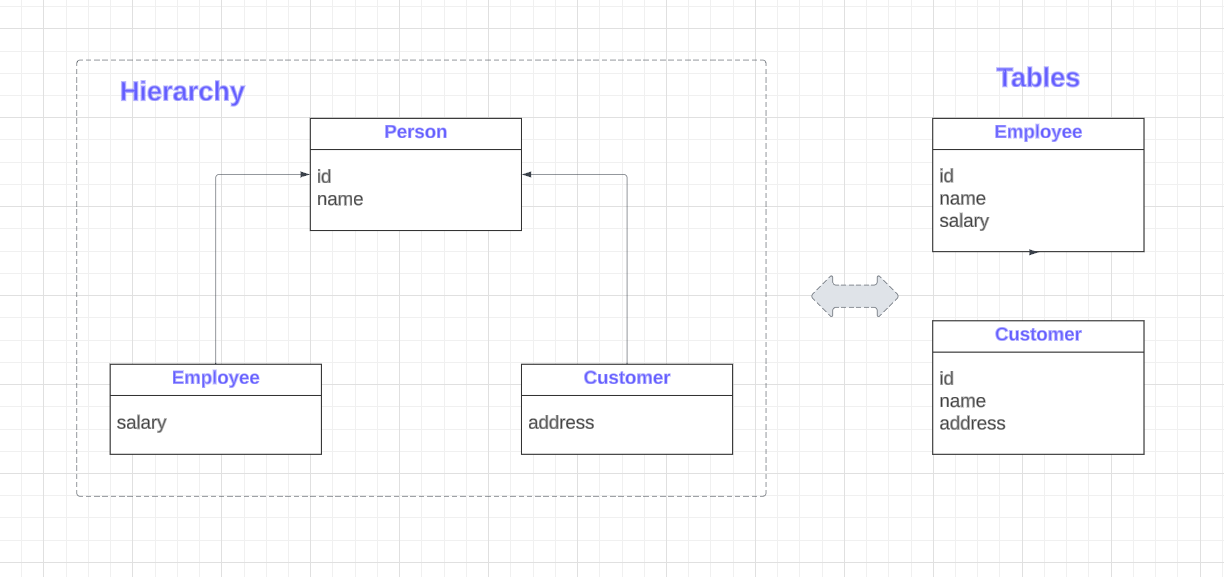
Слика 2.2 Једна табела по класи

### Конкретна табела по ланцу наслеђивања

У овој стратегији, свака конкретна класа, односно, класа која може бити инстанцирана у објектно-оријентисаном моделу, има своју табелу у релационој бази података. Ова табела садржи колоне за сва својства класе, укључујући својства која наслеђује из својих родитељских класа.

Предности овог приступа су што су упити знатно поједностављени у односу на претходну стратегију пошто не захтевају операције спајања. Такође, уколико се промени неко својство изведене класе, промене ће бити независне од других табела.

Са друге стране, јавља се проблем редундансе, с обзиром на чињеницу да свака табела садржи сва својства једне конкретне класе. Осим што се, очигледно, повећава величина простора у бази података, овакав приступ резултује проблемима са ажурирањем основне класе, јер свака измена у основној класи мора да се пропагира на све изведене табеле.

Слика 2.3 Конкретна табела по ланцу наслеђивања

## Асоцијације

Услед разлика у начинима представљања односа између ентитета две парадигме, може настати неколико проблема. Да би се разумели проблеми који настају, неопходно је разумети кључне разлике. Релационе базе података успостављају односе између табела помоћу страних кључева ка другим табелама, док објекти енкапсулирају референце ка другим објектима које се чувају у меморији током извршавања програма. Поред тога, објекти, кроз употребу колекција, могу да чувају референце ка мноштву других објеката помоћу само једног поља. Са друге стране, релациона база података, због нормализације, захтева да све везе ка другим табелама буду изражене кроз атомичне вредности. Ово доводи до инверзије у структурама података између објеката и табела. Ова појава се лако може објаснити кроз пример:

Посматра се апликација која опслужује веб продавницу и нуди могућност да се различити производи из понуде сместе у корпу за куповину. Посматрано из апсекта објектне парадигме, објекат „корпа“ ће садржати у себи листу ставки које представљају производе који су тренутно смештени у њој. Ако се ово представило у релационој бази података, табела ставка требало би, кроз колону која би представљала идентификациону ознаку корпе, да има ограничење страног кључа ка табели корпа, јер табела корпа, због прве нормалне форме, не сме да има колоне које садрже неатомичне вредности.

Решење које је могуће у овом случају је, да се за сваки објекат чува поље које репрезентује примарни кључ у одговарајућој табели, како би се могло извршити мапирање између референци објеката и страних кључева. Другим речима, реализује се кроз ограничење страног кључа.

## Добављање података

При раду са релационим базама података, поготово у модерним апликацијама, начин на који систем добавља податке игра кључну улогу у његовој перформантности. Две основне стратегије за добављање података у објектно-релационим маперима су *Lazy Loading* и *Eager Loading*. Иако обе стратегије нуде различите приступе у решавању истог проблема, свака има своје специфичне предности и изазове у коришћењу. Неадекватно коришћење датих механизама може непосредно довести до деградације у перформансама система, које могу резултовати непотребним упитима ка бази података, изазивајући додатно опетерећење у систему и продужити време одзива апликације. Стога је од велике важности разумети како ове стратегије функционишу, како би се, правилном применом, постигле жељене перформансе система.

### *Lazy Loading*

*Lazy Loading* је механизам који омогућава учитавање повезаних података само када су они заиста потребни. Другим речима, подаци се не довлаче из базе података све док апликација експлицитно не захтева приступ тим подацима, што може побољшати ефикасност система. Међутим, непромињељено коришћење *Lazy Loading-a* може довести до такозваног ***N+1 query*** проблема, који ствара значајно оптерећење базе података.

### Проблем N+1 упита

Већина модерних ОРМ алата користе *Lazy Loading* као основно понашање за добављање података. С обзиром на чињеницу да се не добављају подаци које апликација експлицитно не затражи, у тренутку када они постану неопходни за обраду, систем ће слати упите ка бази за сваки неопходан податак. Овакво понашање неретко резултује проблемом N+1 упита ка бази. Ова појава се лако може објаснити на примеру:

Претпоставка је да постоји база података која опслужује друштвену мрежу у којој су моделоване табеле Објава (Post) и Ознака (Tag). Однос између ових табела је један према више, односно једна објава може да садржи више ознака. Уколико би се у систему користио *Lazy Loading*, добављање података би се свело на:

* Основни упит којим би се добавиле све објаве.
* За сваку објаву која се добави, прави се додатни упит како би се добавиле све ознаке везане за конкретну објаву.

За N објава у бази, уместо да се шаље само један упит, ово би резултовало са N+1 упитом ка бази података, одакле и потиче назив проблема. N+1 проблем може да изазове спор одзив система, као и да лимитира скалабилност система, стога је битно узети у обзир неколико могућих солуција. Једна од њих је *Eager Loading*.

using (var context = new BlogContext())

{

var posts = context.Posts.ToList();

foreach (var post in posts)

{

var tags = context

.Tags

.Where(t=>t.post\_id == post.post\_id)

.ToList();

}

}

Листинг 2.1 N+1 проблем

### *Eager Loading*

*Eager Loading* подразумева учитавање свих повезаних података у исто време када и основни објекат. Ово значи да апликација одмах добавља све неопходне релације и податке, минимизујући број упита ка бази података. Иако овај приступ решава проблем N+1 упита и користан је када су сви подаци одмах потребни, може представљати оптерећење у случајевима када нису неопходни сви подаци.

Из претходно наведеног може се закључити да избор између две наведене стратегије треба да буде базиран на конкретним потребама апликације и перформансама система.

## Кеширање

Кеширање у системима објектно-релационог мапирања је једна од основних техника за повећање ефикасности апликација, редукујући потребу за учесталим упитима ка бази података. Чувањем често коришћених података у меморији за брже преузимање, ова функционалност се позиционира између апликационог слоја и слоја базе података. Међутим, како база података може бити измењена од стране других корисничких процеса или апликација, кеширани подаци могу постати застарели и неосвежени у кешу, што може довести до проблема са актуелношћу података. Због тога је инвалидација кеша од суштинског значаја, јер осигурава да кеширани подаци остану релевантни и синхронизовани са актуелним стањем у бази. ORM алати обично нуде опције за конфигурисање политика кеширања на нивоу појединачних класа, омогућавајући различите стратегије за различите типове података.

## Обрасци понашања ОРМ алата

У контексту потребе за перзистенцијом података у апликацијама, често се примењују специфични дизајн обрасци како би се обезбедила ефикасност, конзистентност и одрживост кода. Ови обрасци пружају систематизоване начине за решавање уобичајених проблема који настају при интеракцији између апликације и базе података. Концепти као што су *Identity Map* и *Unit of Work* [20] представљају дубоко укорењене обрасце који играју кључну улогу у ефективном раду ОРМ алата. У наредним секцијама пружен је дубљи увид у ова два обрасца.

### *Identity Map*

*Identity Map* је дизајн образац који има циљ да одржава конзистентност учитаних објеката у рам меморији одређене сесије или трансакције [20]. Главна предност овог образца је у спречавању многобројних учитавања истог објекта из базе података током једне сесије чиме може доћи до аномалија модификације. Основна идеја овог обрасца је да систем, у специјализованим мапама, чува објекте који су претходно добављени из базе података. За избор кључа мапе, најчешће се узима кључ табеле у бази података. Када систем треба да учита објекат, прво се врши провера да ли се тражени објекат налази у мапи, и уколико се налази у меморији, систем не мора да шаље упит ка бази. Са друге стране, уколико се тражени објекат не налази у мапи, систем шаље упит ка бази, добавља га, и смешта у мапу за будуће коришћење.

### *Unit of Work*

*Unit of Work* [20] обезбеђује механизам за праћење промена над објектима. Његова сврха је да се обезбеди атомичност у примени промена: или се све промене успешно примене, или ништа од њих. Oвим се осигурава конзистентност података и ефикасна администрација ресурса. Синхронизација са базом података се може вршити са сваком променом објекта, али то може да доведе до великог броја позива ка бази података, што потенцијално доводи до деградације перформанси. Идеја *Unit of Work* обрасца је да постоји централна компонента која води рачуна о свим променама над објектима модела и управља трансакцијама и синхронизацијом са базом података.

## Употреба ОРМ алата у развоју апликација

Објектно-релациони мапери представљају моћан инструмент у развоју савремених софтверских апликација и омогућавају рад са базама података на вишем нивоу апстракције. Стварна моћ ORM-ова лежи у њиховој способности да преведу објектно-оријентисане моделе у релационе упите и обрнуто, без потребе за ручним писањемем *SQL* упита. Ово не само што значајно поједностављује развој и одржавање кода, већ, такође, смањује могућност грешака у комуникацији са базом података. Међутим, употреба ORM-а такође захтева дубоко разумевање њихових механизама и образаца, како би се избегли потенцијални проблеми у перформансама или неслагања у репрезентацији података. Управо из тог разлога, неопходно је ставити акценат на комбиновању предности ORM алата са добрим познавањем основа релационих база података.

У овом поглављу разматрани су теоријски аспекти и изазови који се јављају при мапирању између објектно-оријентисане и релационе парадигме. Приказани концепти и изазови служе као основа за разумевање значаја и сложености која се крије иза употребе објектно релационих мапера.

У наредним поглављима, ставиће се акценат на конкретне ОРМ алате, где ће бити пружен увид у то како су различити алати приступили решавању наведених изазова.

# Преглед релевантне литературе

Проучавање релевантне литературе је кључни корак у истраживачком процесу који помаже у разумевању постојећих методологија. У контексту тематике објектно-релационих мапера, критички осврт на литературу обезбеђује основу за сагледавање предности и недостатака различитих решења, као и мотивацију за писање овог рада, како би се попуниле одређене празнине у литератури.

## Методологија претраге литературе

Претрага је вршена коришћењем различитих научних база података, укључујући *Scopus*, *IEEE Xplore*, *Research Gate*. Неке од кључних речи које су коришћене за претрагу су: "објектно-релационо мапирање", "анализа перформанси ОРМ алата" и "Object-relational impendace mismatch". Такође, с обзиром на чињеницу да је подељењо мишљење о употреби ОРМ алата у развоју софтвера, један део пажње је усмерен на артикле написане од стране признатих научника и програмера у области објектно-релационих мапера и софтвер архитектуре. Поред научних чланака и радова, у обзир су узете и књиге посвећене овој тематици, где се посебно истичу дела Мартина Фаулера [16], Мајкла Бајера [19] и Влада Михалцеа [17], који се сматрају за референтне личности у области објектно-релационог мапирања.

## Опис претходних истраживања

У овој подсекцији представљена је синтеза најзначајнијих закључака из претходних истраживања о објектно-релационим маперима. Истраживања су се фокусирала на различите аспекте: од теоријске основе ORM-а, преко различитих техничких имплементација, до перформанси и реалних примена у различитим сценаријима.

У свом познатом чланку, *The Vietnam of Computer Science* [18], Тед Њуард прави аналогију објектно-релационог мапирања са америчком војном кампањом у Вијетнамском рату, при чему наводи да употреба алата за објектно-релационо мапирање у раним фазама развоја софтвера пружа ефикасност, док у каснијим фазама, са све већом употребом и улагањем времена, деградације перформанси нису занемарљиве због превелике разлике између две парадигме. Његова главна теза је да не постоји добро решење за премошћавање наведених разлика и да сва постојећа решења укључују болне компромисе.

У раду Цветковића и Јанковића [8] излагана је детаљна анализа и поређење два популарна ORM алата за .NET: *Entity Framework* (EF) и *NHibernate*. Противећи се распрострањеном мишљењу да ORM алати уводе значајан *overhead* у односу на традиционалне приступе приступу подацима, аутори показују да је, за већину уобичајених упита, перформанса EF и *NHibernate-а* слична директним SQL упитима. Иако су примећени изузеци у неким случајевима, као што су масовна ажурирања или груписање у EF, основна порука рада је да се рупа у перформансама значајно сужава код савремених ORM оквира као што су EF и *NHibernate*.

Koли, Стајнер и Асадузаман у свом раду [5] разматрају изазове који произлазе из коришћења ORM алата при превазилажењу проблема неусаглашености између објекатно-оријентисаних програмских језика и релационих база података. Аутори детаљно приказују анти-патерне узроковане коришћењем ORM-a који утичу на перформансе упита, укључујући проблеме као што су превише жељни захтеви, непотребно угњежђивање подупита и лоша читљивост упита. Иако представљају стратегије за оптимизацију, аутори истичу да је приступ решавању проблема са ORM-ом комплексан и често захтева разматрање на нивоу целокупне шеме базе података. Ово истраживање поставља значајне основе за разумевање и оптимизацију учинка ORM алата на перформансе релационих упита и подстиче даља истраживања у правцу интеграције објекатно-оријентисаних и релационих приступа.

Освртањем на неколико радова може се закључити да међу научним колективом влада подељено мишљење о употреби алата за објектно-релационо мапирање. У наредном потпоглављу, биће пружен осврт на неке од недостатака у литератури које ће овај рад покушати да попуни.

## Уочени недостаци у литератури

На основу анализе релевантних радова, примећени су неки кључни недостаци и ограничења у постојећој литератури о ORM-у који треба узети у обзир:

### Непотпуност

Многи извори пропуштају дискусију о релевантним аспектима мапирања и њиховом утицају на карактеристике квалитета система, услед чега је отежано разумевање предности и мана разуличих стратегија мапирања. Уколико би се читаоци фокусирали само на неколико радова, не би се могла створити шира слика о свим потенцијалним компромисима који се морају направити приликом коришћења алата.

### Одсуство анализе Python ОРМ алата

Иако је Python један од најпопуларнијих програмских језика данашњице, чини се да литература занемарује ORM алате специфичне за овај језик. С обзиром на растућу употребу Python-a у развоју веб апликација, потребна је дубља анализа и рецензија ових алата у контексту мапирања.

Иако постојећа литература обухвата велики део аспеката објектно-релационог мапирања, ипак постоје значајне рупе. У наредним поглављима биће пружена посебна на пажња на изабране алате за објектно-релационо мапирање, са циљем да се попуне неке од уочених недостатака у литератури

# Преглед изабраних ОРМ алата

У претходним поглављима, представљене су основе објектно-релационог мапирања и проблеми који ови алати решавају. Постоји много различитих алата за објектно-релационо мапирање, али, због своје велике заступљености у развоју модерних апликација, изабрани су *Entity Framework*, *Hibernate* и *SQLAlchemy* за детаљну анализу начина на који су они имплементирани.

Сваки од ових алата има своју јединствену архитектуру, начине извршавања упита и механизме оптимизације. Циљ овог поглавља је да пружи техничке детаље, предности и могуће изазове који произилазе из коришћења наведених алата.

Разумевање техничких аспеката ових алата, попут начина на који обрађују упите, управљају кеширањем и врше оптимизације, може значајно помоћи програмерима да максимално искористе њихове потенцијале и избегну типичне замке.

Анализа ће започети детаљима о имплементацији *Entity Framework-a*, затим ће се прећи на *Hibernate* и завршити детаљним прегледом *SQLAlchemy* радног оквира.

## *Еntity Framework*

Entity Framework (ЕF) представља интегрални део .NET платформе и представља водећи алат за објектно-релационо мапирање унутар *Microsoft* екосистема. Од своје појаве, EF је доживео бројне ревизије и промене, при чему је свака верзија доносила побољшања у функционалностима и флексибилности.

У овом поглављу ће бити представљене архитектурне основе *Entity Framework-a*, како он трансформише LINQ упите у SQL, и који механизми стоје иза управљања подацима и оптимизације. Такође, биће представљени различити аспекти који утичу на перформансе и добре праксе за достизање оптимизованих резултата при раду са EF.

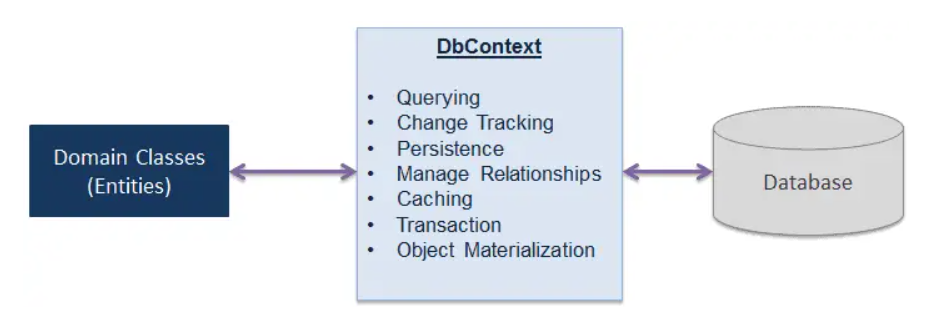
### Aрхитектура алата

#### DbContext и DbSet

DbContext представља основу класу одговорну за интеракцију са базом података. Ова класа служи као интерфејс између доменских класа и табела у бази података.

Одговорности DbContexta у раду са базом података су:

* **Управљање упитима**: Превођење *LINQ-to-Entities* упита у SQL упит који се шаље ка бази података
* **Праћење промена ентиета**: DbContext представља имплементацију *Unit of Work* шаблона, стога ова класа прати све промене које се дешавају над ентитетима и осигурава да база података буде у конзистентном стању
* **Перзистенција података**: На основу испраћеног стања ентитета, DbContext извршава одговарајуће DML наредбе над базом података
* **Кеширање**: Пружа могућност кеширањa на првом нивоу и, за трајање свог животног циклуса, чува ентитете који су били добављени неким од претходних упита
* **Материјализација**: Процес током којег се подаци добављени из базе конвертују у објекте ентитета

[[1]](#footnote-1)Слика 4.1 Позиција *DbContext* класе у *Entity Framework-y*

DbSet је класа која представља интерфејс помоћу које се врши комуникација ка једној табели у бази података. Пружа могућност вршења Add, Remove, Find, Attach операција помоћу којих се генеришу неопходне DML операције над базом података. Такође, помоћу метода Local, SqlQuery и DbQuery пружа могућност, приступу локалним подацима у кеш меморији DbContexta, директно писање SQL упита, као и писање LINQ упита над ентитетима. DbSet се најчешће декларише као поље класе која наслеђује DbContext, чиме се остварује мапирање ентитета класа на записе у табелама релационе базе података и координише свим инстанцама.

public class BenchmarkDbContext : DbContext

{

public BenchmarkDbContext() {}

public DbSet<User> Users { get; set; }

public DbSet<Profile> Profiles { get; set; }

}

Листинг 4.1 Препоручен начин коришћења DbContext и DbSet класа

#### Entity Data Model

*Entity Data Model* (EDM) је скуп концепата који описују структуру података независно од начина на који се они чувају. ЕДМ је, заправо, *in-memory* репрезентација свих мета података неопходних за објектно-релационо мапирање и састоји се од: концептуалног модела, модела складишта (*storage model*) и мапирањима између њих. Овакав концепт омогућава логици апликације да се фокусира на домен, изолојући је од детаља приступа бази.

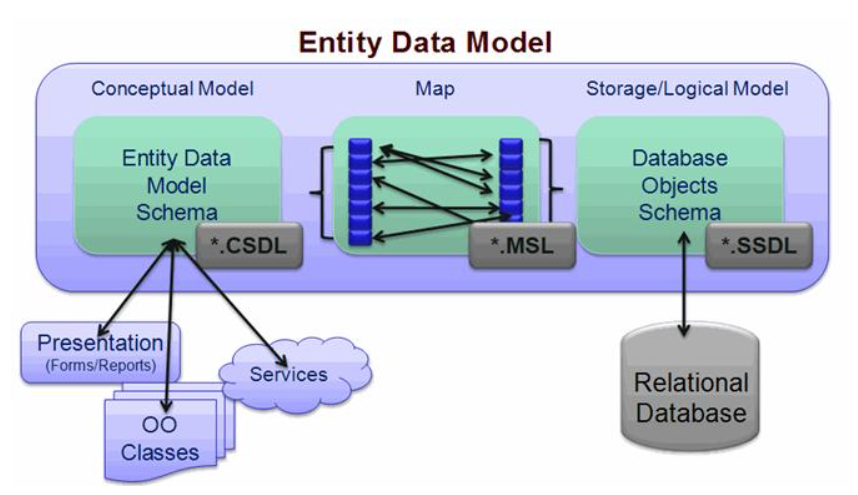
Концептуални модел чине доменске класе апликације, *DbContext* класа и *DbSet* колекције. Складишни модел заснован је на шеми базе података са којом је апликација повезана. Информације о мапирањима су неопходне како би се могло ефикасно врши пребацивање из концептуалног модела у складишни и обрнуто.

Како би ово мапирање било ефикасно, ЕF користи три језика заснована на XML језику: *Conceptual Schema Definition Language* (CSDL), *Store Schema Definition Language* (SSDL) и *Mapping Specification Language* (MSL).

**CSDL** се фокусира на концептуални део апликације, он описује ентитете и њихове односе, без узимања у обзир специфичних детаља или структуре конкретне базе података.

**SSDL** описује логичку структуру података базе података повезане са апликацијом, укључујући табеле, колоне, везе и остале релационе атрибуте.

Да би се осигурало да оба света могу да комуницирају без ометања, **MSL** постоји као мост између ова два језика. Он дефинише правила и начине на које се концептуални модели из CSDL-a мапирају на енитете у бази података описаних путем SSDL-a.

[[2]](#footnote-2)Слика 4.2 Приказ архитектуре ЕДМ-а

#### Руковање конекцијама

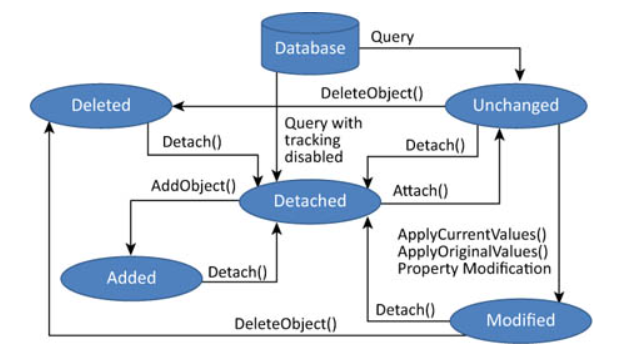
У раду са базом података, битно je разумети како се управља конекцијама, почевши од њиховог успостављања до затварања, како не би дошло до ситуација у којима су угрожене перформансе или безбедност система. ЕF пружа ефикасан и оптимизован систем за руковање конекцијама и он се састоји од неколико кључних сегмената:

* **Успостављање конекције:** Када се креира DbContext инстанца, EF не отвара одмах конекцију са базом података. Уместо тога, конекција ће бити отворена само када је то потребно, приликом извршења упита или када је потребно сачувати податке.
* **Животни циклус конекције:** У својој основној поставци, ЕF користи краткотрајне конекције, што значи да се конекција отвара само када је потребно и брзо се затвара после коришћења чиме се минимизују ресурси које конекција заузима. Међутим, у неким случајевима, као што су трансакције, конекција може да буде отворена дужи временски период.
* **Затварање и ослобађање конекције:** Када је рад са базом података завршен, EF аутоматски затвара конекцију како би осигура да не дође до непотребног трошења ресурса. Такође, препоручено је да се користи using исказ приликом рада са инстанцом DbContext класе како би се осигурало правилно ослобађање ресурса.

#### Праћење промена ентитета

*Entity Framework* користи механизам праћења промена како би идентификовао и записао све измене над ентитетима од последњег учитавања из базе података. Ова способност је кључна за рад са ОРМ системима и осигурава ефикасност и усаглашеност при ажурирању базе података.

Ентитет може имати једно од пет стања која одређују како ће се на њега гледати при следећем позиву методе SaveChanges инстанце класе DbContext:

* **Unchanged (Непромењен)**: Ентитет је учитан из базе података и није било измена на њему од тада.
* **Modified (Измењен)**: Након учитавања из базе података, вршене су промене на ентитету. Ова променаће бити сачувана у бази података при следећем позиву методе *SaveChanges*.
* **Deleted (Обрисан)**: Ентитет је обележен за брисање и биће уклоњен из базе података при следећем позиву методе *SaveChanges*.
* **Added (Додат)**: Ентитет је додат у контекст али још увек не постоји у бази података. Биће креиран нови запис у бази података при следећем позиву *SaveChanges* методе.
* **Detached (Одвојен)**: Ентитет је у контексту али не прати промене на њему. Ниједна промена овог ентитета неће бити рефлектована у бази података.

[[3]](#footnote-3)Слика 4.3 Животни циклус ентитета

### Извршавање упита

#### Language Integrated Query (LINQ)

LINQ представља интегрисан систем за упите који је основни део .NET радног оквира. Омогућава директно писање упита у C# програмском језику, користећи синтаксу која је читљива и интуитивна. Применом LINQ језика могу се лако изводити различите операције као што су селекција, филтрирање, сортирање, груписање и спајање података из различитих извора. *LINQ to Entities* је врста LINQ jeзика намењена за слање упита ка бази података користећи концептуалне моделе и саставни је део *Еntity Framework-a*.

#### Превођење LINQ упита

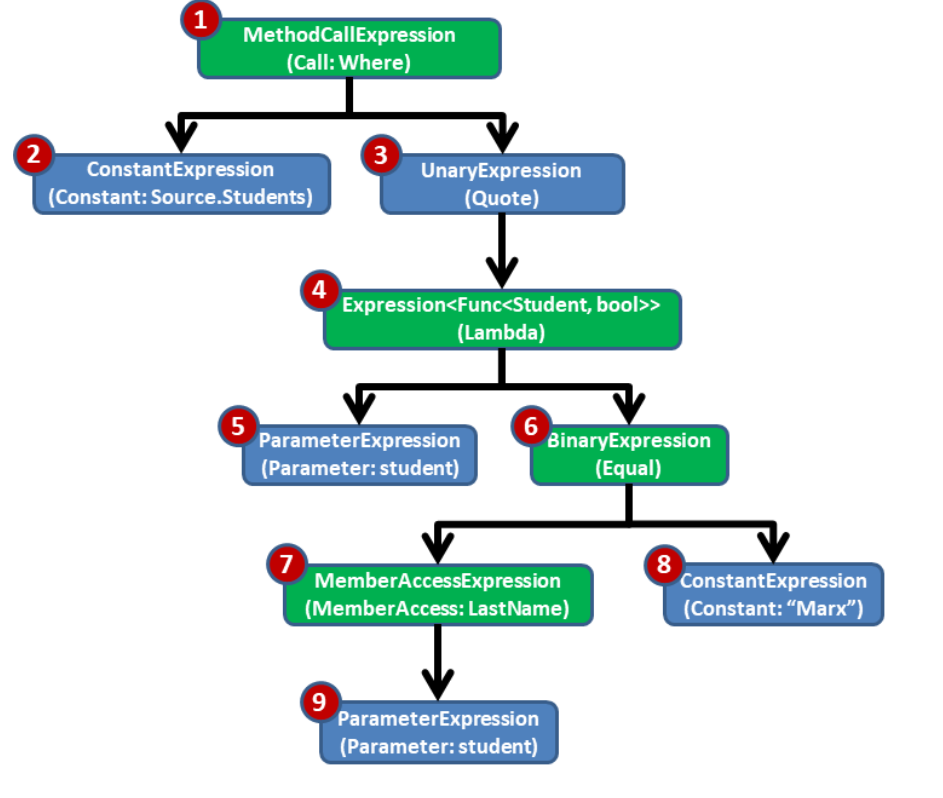
Интерно, LINQ упити се представљају као изразна стабла (*expression trees*), структуре података које представљју упите као скуп објеката. На овај начин се омогућава алату да "разуме" упит на вишем нивоу апстракције, чиме се омогућава оптимизација упита пре превођења у SQL. Када LINQ упит треба да се изврши, изразно стабло се прослеђује упитном пајплајну *Entity Framework-a*. Oвај пајплајн садржи компоненте које анализирају изразно стабло са циљем генерисања одговарајућег SQL упита. Генерисани SQL упит се затим прослеђује провајдеру базе података (енг. *database provider*), библиотеци намењеној за комуникацију са одговарајућом базом података, који се повезује са базом и извршава упит.

За упит наведен у листингу испод:

Source.Students

.Where(student=>student.LastName == “Marx);

Листинг 4.2 Пример LINQ to Entities упита

[[4]](#footnote-4)Изгенерисаће се следеће изразно стабло:

Слика 4.3 Изразно стабло LINQ упита

#### Maтеријализација

Након што се SQL упит изврши и врати резултате, Entity Framework мора да преведе те резултате у инстанце објеката који су дефинисани у апликацији. Овај процес је познат као материјализација. Он укључује креирање објеката, попуњавање њихових својстава са вредностима из резултата упита и, у неким случајевима, повезивање објеката у сложене структуре као што су листе или графови.

Након разматрања процеса превођења LINQ упита у SQL у одељку 4.1.2, у следећем одељку 4.1.3 детаљно ће се размотрити стратегије и методе за оптимизацију и подешавање перформанси упита.

### Aспекти побољшања перформанси

#### Lazy Loading и Eager Loading

EF подразумевано користи *Lazy Loading* стратегију за добављање података, што значи да се повезани подаци са циљаним ентитетом не преузимају, одмах, приликом добављања главног ентитета, већ по експлицитном захтеву. То се постиже тако што ЕF креира посредничке (енг. *proxy*) објекте, који, на захтев, комуницирају са базом како би добавили повезане податке, који још нису преузети. Као што је већ напоменуто, ова стратегија учитавања може да изазове N+1 проблем, стога ЕF нуди могућност избора *Eager Loading* стратегије. Oво се постиже коришћењем Include методе у LINQ-y изразy, што се може видети у наредном примеру:

return context.Users.Include(x => Posts).ToList();

Листинг 4.3 Избор *Eager Loading* стратегије

Oвим се у једном упиту учитавају и подаци о главном ентитету (*User*), као и о повезаном ентитету (*Post*). Meђутим, треба још једном напоменути да прекомерно коришћење ове методе може негативно утицати на перформансе.

#### Индексирање

Ефикасно индексирање може драстично побољшати перформансе упита. Стварање индекса на колонама које се често користе у упитима или приликом операција спајања, може помоћи у бржем добављању података. EF нуди могућност креирања основних и композитних индекса над атрибутима ентитета и то се може извршити на два начина: уз помоћ анотација или коришћењем *FluentAPI-a*, посебног начина за конфигурацију модела.

public class Tag

{

public int Id {get; set;}

[Index]

Public string Name {get; set;}

}

Листинг 4.4 Креирање индекса уз помоћ анотација

protected override void

OnModelCreating (DbModelBuilder builder)

{

builder.Entity<Tag>()

.Property(t => t.Name)

.HasColumnAnnotation(

“Index”,

new IndexAnnotation(

new IndexAttribute(“Name\_Index”)

));

}

Листинг 4.5 Креирање индекса уз помоћ *FluentAPI-ja*

#### Профилисање

Профилисање у EF алату представља битан инструмент за оптимизацију и надзор комуникације апликације са базом података. Омогућава програмерима да идентификују неефикасне упите, пружа увид у преведене SQL упите и помаже у дијагностиковању потенцијалних проблема у перформансама. У својој основи, *Entity Framework* омогућава логовање путем којег се могу приказати информације о извршеном упиту, као што су време извршавања, добављена количина података, као и преведена LINQ наредба у нативни SQL.

За постављен LINQ упит:

return context.Users.Where(x => x.DoB >= year).ToList();

Листинг 4.6 Пример упита за који се врши профилисање

Профилисањем се утврђује одговарајући SQL упит који је *Entity Framework* генерисао:

SELECT u.user\_id, u.date\_of\_birth

FROM Users AS u

WHERE u.date\_of\_birth >= @\_\_year\_0

Листинг 4.7 Генерисан SQL упит од стране EF-a

#### Кеширање

*Entity Framework* омогућава неколико облика кеширања података:

* **Кеширање објеката** (eнг. *Object caching*): Посебан ObjectStateManager објекат, који је уграђен у контекст класу, води рачуна о објектима који су претходно добављени и налазе се у меморији. Ово је познатије и као кеширање првог нивоа (енг. *First level cache*).
* **Keширање плана извршавања упита**: Чување изгенерисане SQL наредбе за упите који се понављају неколико пута.
* **Кеширање метаподатака** - Служи за дељење метаподатака о моделу између неколико различитих конекција.

Приликом кеширања првог нивоа, ObjectStateManager објекат води рачуна да ли се ентитет, који се тражи упитом, већ налази у меморији. Ако се у меморији налази ентитет са идентичним кључем који је требао да се добави, Entity Framework ће га укључити одмах у резултат упита. Треба напоменути да ће Entity Framework свакако извршити упит ка бази података, али ће се заобићити фаза материјализације објекта чиме се штеди на ресурсима. Са друге стране, треба узети у обзир да уколико се превише ентитета налази у меморији, то може негативно да утиче на перформансе упита.

Кеширање метаподатака је битно за ефикасну интеракцију са моделом података у EF. Метаподаци представљају структурне и конфигурационе информације о моделу, као што су дефиниције ентитета и везе између ентитета, као и подаци неопходни за успостављање конекције са базом података. Приликом креирања инстанце DbContext класе, EF интерпретира ове метаподатке и додељује одговарајући објекат конекције контексту. С обзиром на чињеницу да је животни век контекст класе везан за трајање трансакције, EF кешира метаподатке како би брзо приступио конфигурационим информацијама приликом креирања нових инстанци контекста. Такође, уколико се креира нови објекат конекције који има исти конекциони стринг, као и неки од постојећих објеката конекције, EF ће, ради ефикасности, искористити кеширане метаподатке.

### Oстале кључне особине

#### Миграције

Кључна способност коју нуди *Entity Framework* је инкрементална миграција података за ажурирање шеме базе података, уз промене у доменском моделу апликације. EF пружа функцију миграција која омогућава систематско развијање шеме базе података у складу с променама направљеним у ентитетским класама у коду.

Миграције су обично *code-first*, што значи да је ентитетски модел у коду сматран извором истине и база података треба да буде ажурирана да би се успоставила сагласност с њим. Када се направе промене у ентитетским класама, као што су додавање или уклањање атрибута, EF може генерисати миграциону скрипту која садржи кораке потребне за ажурирање базе у складу с тим. Свака миграција има Up() методу која примењује промене шеме и Down() методу која их враћа, што омогућава да се, ефикасно, измењена шема базе података, врати на претходно стање.

Миграције садрже само инкременталне промене, а не целокупну шему. EF прати до сада примењене миграције преко историјске табеле. Пре него што примени даље миграције, упоређује снимак (енг. *snapshot*) у историји са тренутним моделом да би применио само потребне промене. За ове потребе, ЕF садржи посебан фајл под називом *Snapshot* у којем се налази тренутно стање шеме базе података. Миграције могу креирати, брисати или модификовати структуралне компоненте базе података попут табела, колона, индекса и ограничења да би се шема усагласила.

Командна линија EF алата пружа команде за генерисање, примену и враћање миграција. За постојеће базе података, EF Core такође пружа функционалност за инверзно инжењерство модела и миграција из шеме базе података. У глобалу, миграције омогућавају управљање променама шеме на систематизован и аутоматизован начин у различитим окружењима за релационе базе података као што су SQL Server, PostgreSQL и MySQL.

#### Контрола конкурентног приступа

С обзиром на чињеницу да је база података дељени ресурс у времену, што значи да више корисничких процеса могу да јој приступају истовремено, *Entity Framework* пружа механизме којима се може контролисати вишекориснички режим рада. Најчешћи метод је оптимистичка контрола конкуренције путем верзионисања реда у табели, што се постиже увођењем посебне колоне која представља верзију ентитета. Ово повезује временску ознаку или инкрементирајући број верзије реда са сваким записом, који се ажурира при свакој измени. При чувању, EF проверава да ли је верзија промењена од добављања, и уколико је то био случај, изазове се *DbUpdateConcurrencyException* изузетак указујући на застарелу верзију података. Са друге стране, песимистичка контрола конкуренције користи закључавање базе података да би спречила конфликте ажурирања. Закључавање може бити имплицитно спроведено од стране *Entity Framework-a* или експлицитно наведено за одређене ентитете или упите. Ипак, треба узети у обзир да, иако песимистичко закључавање спречава настајање конфликата ажурирања, оно умањује могући ниво конкурентности при комуникацији са базом података.

public class Person

{

public int Id {get; set;}

[Timestamp]

public byte[] Version {get; set;}

}

Листинг 4.8 Оптимистичко закључавање

*Entity Framework* представља снажан алат за управљање релационим базама података у .NET апликацијама. Кроз различите могућности као што су миграције података, контрола конкуренције и интегрисане оптимизације упита, EF омогућава програмерима да ефикасно мењају шему базе података и управљају подацима у својим апликацијама. Са правилном имплементацијом и коришћењем најбољих пракси, максимално се могу искористити предности које овај ORM радни оквир нуди.

## Hibernate

*Hibernate* представља алат за објектно-релационо мапирање, развијен за Јава програмски језик. Захваљујући својој стабилности и поузданости, као и квалитетној документацији и активној заједници програмера који раде на његовом константном унапређењу, истакао се као један од најпопуларнијих објектно-релационих мапера за Јава екосистем.

С друге стране, JPA (*Java Persistence API*) је Јава спецификација за управљање перзистенцијом података. *Hibernate* је, у ствари, имплементација JPA спецификације, чиме је омогућено да развојни тимови користе JPA стандарде и анотације у *Hibernate* апликацијама.

У оквиру овог поглавља, детаљно ће се размотрити архитектура *Hibernate* алата, његове основне концепте и функционалности, као и начине којима се овај алат интегрише са SpringJPA, делом *Spring* радног оквира који олакшава комуникацију са базама података у Јава апликацијама.

### Архитектура алата

#### ЈDBC

JDBC (*Java Database Connectivity*) представља програмски интерфејс развијен у Јава програмском језику за комуникацију са релационим базама података. Он омогућава Јава апликацијама да упућују упите према бази података користећи стандардизоване SQL наредбе, без обзира на конкретну базу података која се користи. Такође, обезбеђује драјвере који служе као мост између апликације и базе података, што доприноси преносивости кода. Омогућава извршавање CRUD операцијa (Create, Read, Update, Delete), управљање трансакцијама и многе друге операције које се односе на базе података. При томе, JDBC пружа могућност за ефикасно и сигурно управљање ресурсима, конекцијама са базом и грешкама које могу настати при комуникацији.

#### SessionFactory

*Hibernate* SessionFactory објекат је одговоран за конфигурисање интеракције са базом података. Основни задатак овог објекта је да управља конекцијама са базом, као и метаподацима неопходним за мапирање ентитета релационе базе података на доменске објекте апликације. Такође, игра кључну улогу у креирању Session објеката којима се врши интеракција ка бази. Конструисање SessionFactory објекта је скупа операција, стога се, приликом покретања апликације, креира једна инстанца која се користи током њеног рада.

Традиционални приступ конфигурацији SessionFactory објекта представља декларативну конфигурацију, уз помоћ XML датотека. Ове датотеке садрже конекциони УРЛ ка бази података, креденцијале приступа, информације о драјверу и дијалекту SQL језика базе података, метаподатке мапирања и подешавања механизма кеширања. *Hibernate*, при покретању, учитава конфигурационе датотеке како би конструисао SessionFactory објекат. Са друге стране, могуће је, уз помоћ *FluentAPI-ја*, у програму конструисати овај објекат употребом SessionFactoryBuilder објекта и уланчавањем позива метода којима се постављају претходно поменути параметри конфигурације.

A screen shot of a computer code

Description automatically generatedСлика 4.4 XML конфигурација *Hibernate-a*

SessionFactory factory = new SessionFactoryBuilder()

.setProperty(„hibernate.dialect“,

„org.hibernate.dialect.MySQLDialect“)

.setProperty(„hibernate.connection.ur“,

„jdbc:mysql://localhost:3306/mydb“)

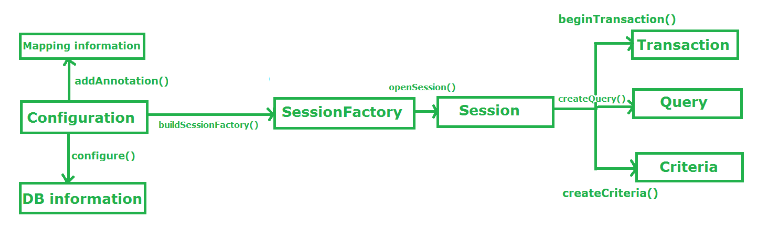
.configure()

.buildSessionFactory();

Листинг 4.9 Конфигурација *Hibernate-a* коришћењем *FluentAPI-ja*

#### Session

Сесија (Session) представља имплементацију *Unit of Work* обрасца за рад са базом података и обавија JDBC Connection интерфејс. Пружа интерфејс за извршавање CRUD операција над ентитетима, као и за припрему и извршавање упита. Такође, управља механизмом кеширања ентитета који су учитани у току једне сесије. Поред тога, има кључну улогу у креирању трансакционих објеката и управља променама стања у животном циклусу ентитета у оквиру јединице рада.

Слика 4.5 Архитектура *Hibernate* алата

#### *EntityManager* у *SpringJPA*

EntityManager у Спринговој JPA имплементацији пружа интерфејс за управљање ентитетима који прати стандарде JPA спецификације. Представља омотач (eнг. wrapper) oко *Hibernate* Session објекта, пружајући сличне могућности перзистенције и креирања упита. Такође, као што постоји SessionFactory објекат уз помоћ којег се управља конфигурацијом комуникације са базом података и креираjy објекти сесије, EntityManagerFactory је одговоран за креирање EntityManager oбјеката и управљање конфигурацијом. Такође, као и SessionFactory објекат, креира се једном, приликом покретања апликације, путем XML или програмске конфигурације.

Када је потребно, помоћу инстанце EntityManagerFactory класе, креира се EntityManager објекат, помоћу којег се врше операције приступа подацима у оквиру једне трансакције. Попут објекта сесије, представља контекст у оквиру којег се врши конверзија ентитета, задужен је за кеширање и управљање променом стања у оквиру животног циклуса ентитета у апликацији.

#### Животни циклус перзистенције ентитета

Апликација која има могућност трајног чувања података мора да комуницира са механизмом за перзистенцију, сваки пут када је неопходно да се тренутно стање пренесе из меморије у базу података, или обрнуто. Ово значи да је потребно позвати *Hibernate* интерфејсе како би се сачували или учитали подаци.

Приликом интеракције са објектно-релационим мапером, неопходно је да се води рачуна о стањима инстанце ентитета у односу на његову перзистенцију. Ово се назива **животни циклус перзистенције** и представља скуп стања кроз који прође инстанца ентитета током свог живота у апликацији. Како апликација не би морала да се брине о праћењу промена стања, механизам перзистенције нуди услугу којом се памте све промене стања ентитета. Ово се назива **контекст перзистенције**. Ентитети у *Hibernate* aлату пролазе кроз неколико различитих стања. Иницијално, након креирања, a пре повезивања са сесијом и базом, ентитет се налази у **транзијентном** стању (*Transient*). У овом стању, ентитет још увек није уписан у базу и не постоји његова идентификациона ознака. Када се ентитет сачува први пут, помоћу сесије, он прелази у **перзистентно** стање (*Persistent*) где постаје управљан од стране *Hibernate* сесије. Док је у перзистентном стању, све измене ентитета се прате и синхронизују са базом података. Taкође, ентитет може бити обрисан, што га ставља у **уклоњено** стање (*Removed*) при чему он више не постоји у бази. Након затварања сесије, ентитет постаје **одвојен** (Detached) од сесије али и даље садржи податке из базе, уколико није уништен из меморије. Објекат у одвојеном стању може поново да се припоји новој сесији у оквиру које може бити ажуриран или обрисан. Треба напоменути да, уколико дође до поништавања трансакције, ефекти синхронизације са базом података не морају бити трајни.

A diagram of a computer

Description automatically generatedСлика 4.6 Животни циклус перзистенције

### Мапирање ентитета

Како би апликација могла да користи перзистиране податке неопходно је, најпре, извршити мапирање класа ентитета на табеле у релационој бази података. У овом поглављу, детаљно ће бити дискутовано o начинима на који *Hibernate* учитава и чува инстанце класа доменског модела. Иако постоји неколико начина путем којих је могуће извршити ово мапирање, посебна пажња ће бити пружена на мапирање помоћу анотација. Анотације представљају специјална обележја у Јава коду које указују на то како треба интерпретирати одређене делове кода приликом приступа бази података.

#### Основе

Како би било могуће мапирати Јава класу на табелу у релационој бази података, постоји неколико анотација које треба узети у обзир:

* **@Entity**: Ова анотација се користи како би се Јава класа обележила као ентитет, што указује да ће њене инстанце бити мапиране на табелу у бази података
* **@Table**: Користи се за дефинисање имена табеле на коју ће се ентитет мапирати. Уколико се ова анотација не користи, име Јава класе ће бити коришћено као име табеле
* **@Id**: Означава примарни кључ ентитета, при чему сваки ентитет мора имати једно поље обележено овом анотацијом
* **@Column**: Користи се за мапирање конкретног поља на колону у бази података. Такође, може се користити за дефинисање својстава колоне, као што су дужина, јединственост итд.

За успешно мапирање ентитета на табелу, неопходно је навести барем **@Entity** и **@Id** анотације.

@Entity

@Table(name = “MyEntity”)

public class MyEntity {

@Id

@GeneratedValue

Private Long id;

@Column(name = “first\_name”)

private String firstName;

}

Листинг 4.10 Мапирање уз помоћ анотација

#### Композитни кључ

Композитни кључ је примарни кључ који се састоји из две или више колона у табели базе података. У Hibernate, композитни кључеви се могу креирати коришћењем посебним анотацијама:

* **@Embeddable**: Ова анотација се користи да обележи Java класу која представља композитни кључ. Класа која је обележена овом анотацијом не сме имати своју табелу; уместо тога, њена поља представљају колоне у табели ентитета који је користи.
* **@EmbeddedId**: Ова анотација се користи у ентитетској класи да обележи атрибут који представља композитни кључ. Атрибут мора бити типа класе која је обележена @Embeddable анотацијом.

Пример ентитета са композитним кључем приказан је у листингу 4.11:

@Embeddable

public Class CompositeKey implements Serializable {

@Column(name = “first”)

private Long first;

@Column(name = “second”)

private Long second;

}

@Entity

@Table(name = “MyEntity”)

public class MyEntity {

@EmbeddedId

private CompositeKey id;

}

Листинг 4.11 Пример дефинисања композитног кључа

#### Мапирање асоцијација

Мапирање асоцијација *у Hibernate-у* означава процес повезивања двa или више ентитета. Ове везе могу бити једносмерне или двосмерне и обично одговарају релационим везама у бази података, као што су „један-на-један“, „један-на-више“ и „више-на-више“.

Основне анотације које се користе за мапирање асоцијација укључују @OneToOne, @OneToMany, @ManyToOne и @ManyToMany. Ове анотације нуде различите конфигурације, укључујући и одређивање смера везе (једносмерна или двосмерна) и каскадне операције.

Уколико се два ентитета налазе у односу "један-на-један" или "један-на-више" конфигурација мапирања се врши на следећи начин:

@Entity

public class Person {

@Id

private Long id;

@OneToOne

@JoinColumn(name = “document\_id”)

private Document document;

}

Листинг 4.12 Веза један-на-један између два ентитета

Са друге стране, уколико је однос између два ентитета "више-на-више", приступа се стандардном решењу у релационим базама података, креирању посебне табеле везе. Помоћу *Hibernate-a*, то се постиже на следећи начин:

@Entity

public class Student {

@Id

private Long id;

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name =

"course\_id")

)

private Set<Course> courses = new HashSet<>();

}

Листинг 4.13 Мапирање односа више-на-више у *Hibernate-y*

Приликом мапирања асоцијација, често је потребно одредити начин на који се повезани ентитети учитавају. Мапирање може бити конфигурисано тако да користи *Lazy Loading* или *Еаger Loading* стратегија добављања повезаних ентиета. То се постиже на следећи начин:

@OneToMany(fetch=FetchType.EAGER)

public Set<Order> getOrders() {

return orders;

}

Листинг 4.14 Конфигурација *Eager Loading* стратегије

Такође, битно је напоменути да је *Lazy Loading* подразумевани начин добављања повезаних ентитета.

### Извршавање упита

#### Hibernate Query Language (HQL)

За разлику од стандардног SQL језика који је прилагођен за релационе базе података, HQL је објектно-оријентисани језик за писање упита, специфично развијен за Hibernate. HQL oмогућава писање упита над ентитетима користећи доменске моделе и њихове атрибуте, чиме се пружа могућност писања интуитивнијих упита над базом података.

query = session.createQuery(„from User where id = :id“);

query.setLong(„id“, 3);

User u = (User) query.uniqueResult();

Листинг 4.13 Пример HQL упита

#### *Criteria Queries*

*CriteriaAPI* нуди могућност за програмско конструисање упита коришћењем Јава објеката и метода. За разлику од HQL упитa који се конструишу помоћу стрингова, *Criteria* упити се генеришу скупом метода које нуди програмски интерфејс *CriteriaAPI*. Главна предност овог приступа је што *CriteriaAPI* омогућава динамичко креирање упита, као и могућност ранијег откривања грешака, током компилације. Ипак, битно је да се напомене да упити засновани на HQL стринговима и *Criteria*API програмским интерфејсом имају исту ефикасност и перформансе.

CriteriaBuilder cb = session.getCriteriaBuilder();

CriteriaQuery<User> cr = cb.createQuery(User.class);

Root<User> root = cr.from(User.class);

cr.select(root)

Query<User> query = session.createQuery(cr);

List<User> users = query.getResultList();

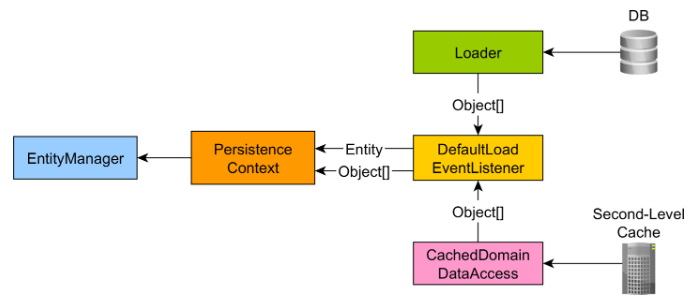
Листинг 4.14 Пример *CriteriaAPI* упита

#### Генерисање SQL наредбе

Без обзира на приступ упиту, *Hibernate* генерише SQL упит тако што парсира HQL или *Criteria*API упит и трансформише га у оптимизоване SQL исказе на основу метаподатака о мапирању ентитета и дијалекта базе података, који су енкапсулирани у SessionFactory објекту. Такође, на основу конфигурационих података, *Hibernate* може да оптимизује упит, узимајући у обзир информације о индексима, кеширању и другим факторима оптимизације. Резултујући SQL упит се прослеђује бази података, где се извршава.

#### Хидрација објекта

Хидрација објекта је процес у којем *Hibernate* преводи редове добијене из базе података у објекте Јава апликације. Ово је виталан процес објектно-релационог мапирања и oтпочиње тако што се прикупе подаци генерисани упитом, који се најчешће налазе у табеларној форми, у оквиру JDBC скупа резултата (*ResultSet*). Затим, за сваки ред резултата, Hibernate интерпретира конфигурационе метаподатке како би утврдио који Јава објекат одговара датом реду и која поља тог објекта треба да се попуне вредностима реда. Уколико је објекат повезан са другим објектима, *Hibernate* ће индефиковати и иницијализовати ове асоцијације, уз могуће коришћење *Lazy Loading* стратегије. Једном када је објекат хидриран, он се ставља у контекст перзистенције у виду EntityEntry објекта којем је могуће пратити промене стања и синхронизовати га са базом података, по потреби.

Слика 4.7 Хидрација објекта

### Keширање

Кеширање је битан механизам за побољшање перформанси апликације засноване на перзистенцији података. У *Hibernate-у*, постоје два основна нивоа кеширања: кеш првог нивоа (*First Level Cache*) и кеш другог нивоа (*Second Level Cache*).

#### Кеш првог нивоа

Први ниво кеша је увек активан и не може се онемогућити. Он је асоциран са *Hibernate* објектом сесије и осигурава да се, у току комуникације, исти објекат не учитава више пута из базе података.

#### Кеш другог нивоа

Док је први ниво кеша везан за конкретну сесију, други ниво кеша је везан за SessionFactory и дељен је између свих сесија. Он може кеширати ентитете, колекције и упите.

За активирање кеша другог нивоа потребно је конфигурисати *Hibernate* да користи конкретан пружалац кеша, као што је *EhCache* или *Hazelcast*. Уколико се користи конфигурација помоћу XML датотеке, активирање кеша се може извршити на следећи начин:

<property name="hibernate.cache.use\_second\_level\_cache">

true

</property>

<property name="hibernate.cache.region.factory\_class">

org.hibernate.cache.ehcache.EhCacheRegionFactory

</property>

Листинг 4.14 Koнфигурација кеша другог нивоа у *Hibernate-y*

Када је конфигурисан, ентитети се могу кеширати користећи анотацију @Cache:

@Entity

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY)

public class Person

Листинг 4.15 Употреба кеша другог нивоа

Механизам кеширања у *Hibernate-у* може знатно побољшати перформансе апликације, али треба бити пажљив са његовом употребом. Неправилно конфигурисан кеш може довести до неконзистентних података или деградације перформанси.

## SQLAlchemy

SQLAlchemy је алат за објектно-релационо мапирање развијен за потребе апликација направљених у Пајтон програмском језику. Oд свог настанка, 2005. године, тежи да пружи комплетан систем за рад са релационим базама података у Пајтону, ослањајући се на DB API програмски интерфејс за интеракцију са базама података. Због својих кључних карактеристика, попут способности за ефикасну обраду сложених SQL упита и имплементације обрасца јединице рада (*Unit of Work*), *SQLAlchemy* обезбеђује високо аутоматизован механизам за перзистенцију података и сматра се де факто стандардом за рад са релационим базама података у Пајтону.

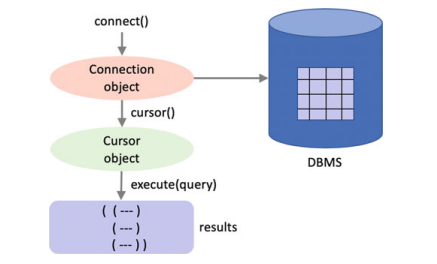
У наредном поглављу, детаљно ће се ући у имплементацију битних механизама за објектно-релационо мапирање у SQLAlchemy алату, почевши са прегледом његове архитектуре и механизмима за праћење стање ентитета. Након тога, биће пружен увид у начине путем који се врши мапирање класа ентитета на табеле релационе базе података. Такође, детаљно ће се разматрати процес извршавања упита над базом података, од декларисања до креирања објеката над добављеним подацима. Поглавље се завршава, кратким увидом у аспекте и механизме којима се долази до побољшања перформанси упита извршаваних над базом података.

### Архитектура алата

#### DBAPI

*Python Database API*, или DB API скраћено, представља стандардизовани интерфејс за начине којима Пајтон модули интерагују са базама података. DB API спецификација пружа скуп метода и конвенција којих се драјвери база података морају придржавати како би били у складу са DB API стандардом. Својом апстракцијом приступа базама података, DB API има за циљ да обезбеди конзистентни интерфејс за остваривање конекције са базом, извршавање упита, као и руковање са добављеним подацима, чиме се омогућава aпликацијама развијеним у Пајтону да лако комуницирају са било којим системима за управљање базама података.

Постоји неколико кључних елемената у DB API-jу:

* **Конекциона функција** - Помоћу које се остварује иницијално повезивање са базом података. Као повратну вредност враћа објекат конекције (Connection).
* **Објекат конекције** - Омогућава приступ бази података. Основни задатак објекта конекције је да пружи могућност слања упита, што се остварује кроз креирање посебних курсорских објеката (Cursor) oбјеката. Такође, помоћу њега се управља трансакцијама, кроз commit() и rollback() методе, и затвара конекција са базом података.
* **Курсорски објекат** - Представља контекст у оквиру којег се извршавају упити ка бази података и добављају подаци који су резултат извршавања. Eнкапсулира интеракцију за базом података и апстракцију курсора базе података. Уз помоћ метода execute() и executemany() пружа могућност извршавања упита, као и управљање подацима добављеним подацима у скупу резултата кроз методе fetchone(), fetchmany() и fetchall() . Након што се заврши рад са базом података, неопходно је затворити курсоре како би се могли потврдити или понишитити ефекти трансакција.

Слика 4.8 Елементи DBAPI програмског интерфејса

#### *Engine*

Engine je oсновна компонента архитектуре *SQLAlchemy* aлата, која представља капију између *SQLAlchemy-ја* и саме базе података. Задатак ове компоненте је да управља повезивањем са базом података, користећи одговарајући DBAPI драјвер и да одржава базен конекција (*connection pool*) са циљем остваривања ефикасне комуникације са базом података током животног века апликације. Поред тога, нуди интерфејс којим се енкапсулирају разлике специфичне за дијалекте база података, услед којих је могуће вршити комуникацију са различитим системима за управљање базама података на јединствен начин. Такође, чува метаподатке о шеми базе података и мапирањима између табела базе и класа апликације. Руковањем свим овим одговорностима, Engine ствара основу за ефикасно функционисање *SQLAlchemy-ja*.

Како би се креирао *Engine* објекат, неопходно је проследити конекциони стринг, који садржи све информације неопходне за повезивање са базом података. Битно је напоменути да се, приликом инстанцирања овог објекта, конекција са базом података не отвара одмах, већ по експлицитном позиву методе connect() прописаном путем DB API стандарда.

Tипичан облик конекционог стринга, прописан је RFC-1738 стандардом, и приказан је у листингу 4.16:

dialect+driver://username:password@host:port/database

Листинг 4.16 Пример конекционог стринга

Информације о дијалекту специфичном за повезану базу података, енкапсулиране су унутар Dialect објекта чија референцa се налази унутар Engine oбјекта, док се управљање над базеном конекција врши кроз посебан Pool објекат.

A diagram of a pool and a dialogue

Description automatically generatedСлика 4.9 Архитектура Engine објекта

#### Session

За разлику од *Engine* објекта који упоставља комуникацију са базом података и омогућава мапирање између класа и табела, кроз *Session* објекат се реализује сва комуникација са базом података и он представља зону надзора (енг. *holding zone*) за све учитане објекте, којима прати и управља променама у стањима животног циклуса. Као и у одговарајућим примерима претходно наведених објектно-релационих мапера, *Session* објекат представља имплементацију обрасца јединице рада (*Unit of Work*). Такође, *Session* представља и имплементацију *Identity Map* шаблона, чиме се спречава учитавање редундантних објеката у меморију.

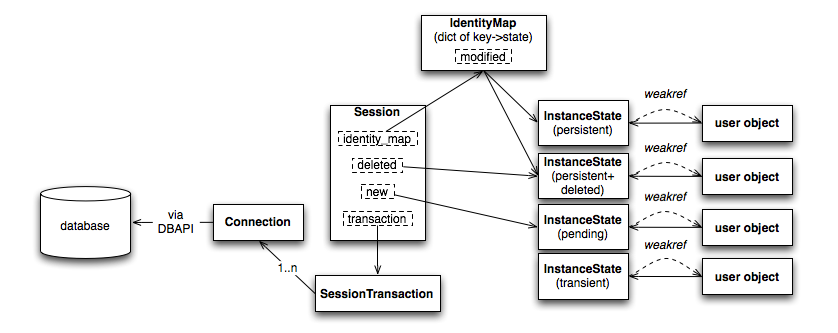
Приликом инстанцирања, објекат сесије је без стања (енгл. *stateless*), што значи да још није асоциран ни са једном конекцијом ка бази података. Приликом слања упита, сесија, од асоцираног Engine објекта, добија ресурс конекције уз помоћ ког се отпочиње трансакција ка бази података. Ефекти трансакције се затим, на експлицитан захтев, кроз посредство сесије, потврђују или поништавају.

Приликом било које измене над објектима који се налазе у сесији, генерише се догађај промене који се записује у сесији. Ове измене су енкапсулиране у оквиру InstanceState објекта, чија се референца налази у оквиру објекта сесије, са циљем заштите модификованих или обрисаних објеката од механизма garbage collectiona. Уколико се шаље упит ка бази података, или треба да се изврши потврда трансакције, објекат сесије, најпре, "почисти" (енгл. *flush*) све измене записане у меморији и прослеђује их ка бази података ради синхронизације. Овај процес синхронизације се врши кроз неколико корака:

* **Преглед промена**: Сесија прво прегледа све објекте које прати у зони надзора и идентификује све промене које треба да примени на базу података
* **Формирање SQL наредби**: За сваку детектовану промену, сесија формира одговарајућу SQL наредбу. На пример, уколико је нови објекат додат у сесију, генерисаће се INSERT наредба
* **Извршавање наредби**: Једном када су све SQL наредбе формиране, сеисја их извршава над базом података у одговарајућем редоследу

Важно је напоменути да, иако објекат сесије синхронизује промене у бази података, то не значи да су промене трајне, јер оне могу бити опозване уколико се поништи трансакција у оквиру које су ове измене начињене.

Објекте, које прати сесија, могуће је откачити (енгл. *detach*) од сесије, чиме се процес аутоматске синхронизације за њих обуставља. Након што се откаче, могуће их је и даље користити у оквиру апликације, с тим што се промене над њима неће рефлектовати у бази података. Уколико има потребе за поновном синхронизацијом са базом података, могуће их је поново повезати са новом или постојећом сесијом, чиме њихово стање поново бива одражено у бази кроз објекат сесије.

Слика 4.10 Илустрација објекта сесије

#### Животни циклус ентитета

Kao што је већ наведено, објекти ентитета могу проћи кроз неколико стања током свог животног циклуса у апликацији. Основна стања у којима објекат може бити су:

* **Transient**: Представља инстанцу која се не налази у сесији и још није сачувана у бази података. Једина веза коју објекат у транзијентном стању има са ОРМ-ом је путем метаподатака који повезују његову класу са одговарајућом табелом у бази.
* **Pending**: Након што се инстанца у транзијентном стању дода у сесију, она прелази у стање чекања, чиме се означава да ће, приликом следећег позива *flush* механизма, она бити уписана у базу података.
* **Persistent**: Инстанца ентитета у перзистентном стању присутна је у сесији и за њу постоји одговарајући запис у бази података. Ентитет може прећи из стања чекања у перзистентно стање, уколико се његове измене синхронизују са базом података, позивом flush() методе. Такође, сви ентитети добављени упитима над базом података су иницијално у перзистентном стању.
* **Deleted**: Обрисано стање ентитета означава да се захтев за брисање инстанце у апликацији синхронизовао са базом података, али ефекти трансакције и даље нису потврђени. Уколико дође до потврде трансакције, објекат из обрисаног, прелази у одвојено (енгл. *detached*) стање, чиме се више не налази у сесији. Са друге стране, уколико дође поништавања ефеката трансакције, објекат се враћа из обрисаног у перзистентно стање.
* **Detached**: Представља инстанцу која се односи, или се односила, на запис у бази података, али тренутно није везана ни за једну сесију, што означава да се промене над њеним стањем неће рефлектовати у бази података, процесом синхронизације.

### Мапирање ентитета

Приликом отпочињања рада са *SQLAlchemy* објектно-релационим мапером, неопходно је, најпре, извршити мапирање класа, дефинисаних у апликацији, на табеле у релационој бази података. Поред тога, потребно је дефинисати мапирање веза између апликативних класа на везе између табела. Овим се формирају метаподаци који се чувају у оквиру Engine објекта којима се омогућава ефикасно претварање из инстанци класа ентитета у записе у табелама базе података, и обрнуто.

#### Класично мапирање

Класично мапирање, познато и као императивно мапирање, у *SQLAlchemy* алату се односи на мапирање већ постојећих класа у апликацији на постојеће табеле у релационој бази података. Ово значи да у *SQLAlchemy* сматра да постоје два одвојена објекта: Table објекат, који енкапсулира метаподатке о структури табеле у бази, и, у апликацији дефинисан, објекат ентитета класе, који се повезују помоћу посебне map\_imperatively методе mapper\_registry модула. Након овог повезивања, објекат ентитета класе добија нове атрибуте, кореспондентне колонама у табели:

user\_table = Table(

"user",

mapper\_registry.metadata,

Column("id", Integer, primary\_key=True),

Column("username", String(50)),

)

class User:

pass

mapper\_registry.map\_imperatively(User, user\_table)

Листинг 4.17 Класично мапирање у *SQLAlchemy-jy*

Такође, mapper функција има могућност и додавања посебних атрибута у кориснички дефинисаној класи, који су неопходни за мапирање веза између објеката. Овакав поступак се у Пајтон свету назива инструментација класа (енгл. *class instrumentation*).

Ипак, битно је напоменути да је класично мапирање застарео начин повезивања класа и табела и није више препоручен начин употребе *SQLAlchemy-ја*.

#### Декларативно мапирање

За разлику од класичног мапирања, модерна употреба *SQLAlchemy-ја* за мапирање ентитета заснива се на декларативном мапирању, налик оном у *Hibernate* алату. У оваком начину мапирања, крајњи корисник експлицитно дефинише атрибуте класа који ће се мапирати на одговарајућу табелу. Типичан начин коришћења је да се најпре дефинише DeclarativeBase наткласа, која омогућава употребу декларативних процеса у свим доменским класама које је наследе. У листингу 4.19, приказан је типичан начин декларативног дефинисања мапирања.

class Base(DeclarativeBase):

pass

class User(Base):

\_\_tablename\_\_ = "user"

id: Mapped[int] = mapped\_column(primary\_key=True)

username: Mapped[str]

Листинг 4.19 Декларативно мапирање у *SQLAlchemy-jy*

DeclarativeBase класа садржи референцу на registry објекат који садржи метаподатке о табелама у релационој бази и њиховом мапирању на класе.

#### Aнатомија мапирања

A diagram of a company

Description automatically generatedСлика 4.11 Анатомија мапирања у *SQLAlchemy-jy*

На слици 4.11, приказан је механизам мапирања који је састављен од два слоја интеракције између кориснички дефинисаних класа и метаподатака табела који су повезани. Леви слој представља претходно поменуту инструментацију класа, док су функционалности везане за базу података и мапирање представљене десним слојем.

У склопу инструментације класа, компонента ClassManager повезана је са кориснички дефинисаном класом која се мапира и садржи колекцију InstrumentedAttribute oбјеката повезаних са мапираним атрибутима мапиране класе.

У десном слоју, слоју мапирања, компонента Mapper представља везу између кориснички деифнисане класе и претраживе јединице (енгл. *selectable unit*). У највећем броју случајева је то табела релационе базе, мада може бити извршено и мапирање кориснчки дефинисане класе на поглед базе података. Мapper у себи садржи колекцију MapperProperty објеката који се брину о SQL репрезентацији кореспондентних атрибута кориснички дефинисаних класа. ColumnProperty представља варијанту претходно наведеног MapperProperty објекта и представља мапирање извршено са атрибута мапиране класе на колону табеле базе података. Са друге стране, RelationshipProperty је такође варијанта MapperProperty објекта и представља везу ка другом маперу, што осликава везе између табела.

RelationshipРroperty сваком атрибуту делегира одговарајуће понашање учитавања (енгл. *loading behaviour*), чиме се назначава која се стратегија учитавања асоцираних објеката користи. Ова информација је енкапсулирана у склопу LoaderStrategу објекта и могуће стратегије су: *lazy* (одложено), *eager* (рано) или *immeadiate* (непосредно) учитавање. Подразумевана стратегија учитавања се одређује током конфигурације мапера, уз могућност коришћења алтернативних опција током извршавања упита. Takoђе, у склопу RelationshipProperty oбјекта постоји референца на DependencyProcessor компоненту која управља асоцијацијама током процеса синхронизације, односно позива flush() методе.

Приликом креирања Mapper објеката и одговарајућих RelationshipProperty објеката, формира се структура графа, у којој су Mapper објекти чворови, а ивице су RelationshipProperty објекти. Овакав граф је кључна структура за функционисање SQLAlchemy алата, зато што игра битну улогу у операцијама које захтевају каскадну пропагацију измена ка повезаним ентитетима, приликом раног учитавања (енгл. *Eager Loading*), као и током процеса синхронизације објеката који у себи садрже асоцијације.

### Извршавање упита

#### Query API

*Query API* у *SQLAlchemy* алату представља динамични програмски интерфејс, који омогућава програмерима да једноставно и ефективно креирају и извршавају упите над базом података. Пружа мост између програмских конструкција високог нивоа и SQL наредби, чиме се ствара интуитиван интерфејс за комуникацију са базом података. У овом поглављу ће детаљно бити појашњен начин на који *Query API* функционише, почевши од декларисања упита до конструкције објеката у апликацији.

У основи *Query* програмског интерфејса је способност да се креирају упити. Ово се обично изводи кроз објекат сесије (Session). Kao што је већ наведено, сесија у *SQLAlchemy* је главна тачка интеракције апликације са базом података.

Једном када је основни упит креиран, *Query* програмски интерфејс нуди различите методе за специфицирање жељеног упита ка бази података, попут филтрирања, омогућено filter() методом, у којој се наводи услов по којем се врши филтрирање, затим сортирање, што се ради уз помоћ order\_by() методе, при чему је могуће навести по којем атрибуту се врши сортирање и у којем редоследу. Поред тога, могуће је вршити и груписање на основу специфицираних поља кроз методу group\_by().

query = (

session

.query(orders.c.customer\_id)

.filter(orders.c.quantity > 10)

.group\_by(orders.c.customer\_id)

.order\_by(orders.c.date\_ordered)

)

Листинг 4.20 *Query* API упит

Овако изграђен упит се извршава позивом метода .all(), којом се враћа листа свих резултата, .first(), којом се враћа први елемент у листи резултата или .one(), чиме се враћа тачно један резултат или изазива изузетак.

#### Изразно стабло

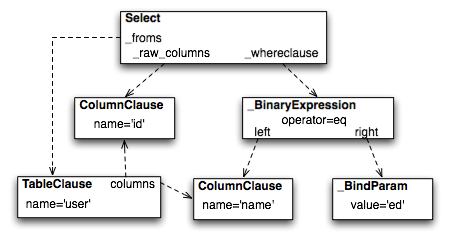
Као што је наведено у претходном поглављу, *Query* програмски интерфејс омогућава коришћење Пајтон објеката како би се представили делови SQL израза, који се, затим, комбинују како би се креирао еквивалентан *QueryАРI* израз. Oвако креирани изрази, могу бити представљени помоћу изразних стабала (енг. expression trees).

Сваки *Query* објекат, у корену изразног стабла које га представља, садржи SELECT чвор, који представља почетак одговарајуће SQL наредбе. Како се позивају друге *Query* методе, попут filter() или order\_by(), тако се изразно стабло попуњава новим чворовима са одговарајућим деловима SQL наредбе. Другим речима, позивањем метода над *Query* објектом, инкрементално се додају нови чворови у изразно стабло. Такође, уколико су у упиту специфициране операције спајања више ентитета (јоin) или је као подразумевана стратегија изабрано рано учитавање, генерисање изразног стабла се ослања и на граф Mapper објеката, поменутих у поглављу **4.3.2.3**. Приликом позива извршавања *Query* објекта, овако креирано изразно стабло се прослеђује *SQLAlchemy* компајлеру, који га преводи у одговарајући SQL израз, са постављеним параметрима упита, уз обраћање пажње на дијалект система за управљање базом података са којим се врши комуникација. Процес превођења се назива **фаза рендеровања** SQL израза (енгл. *SQL rendering*).

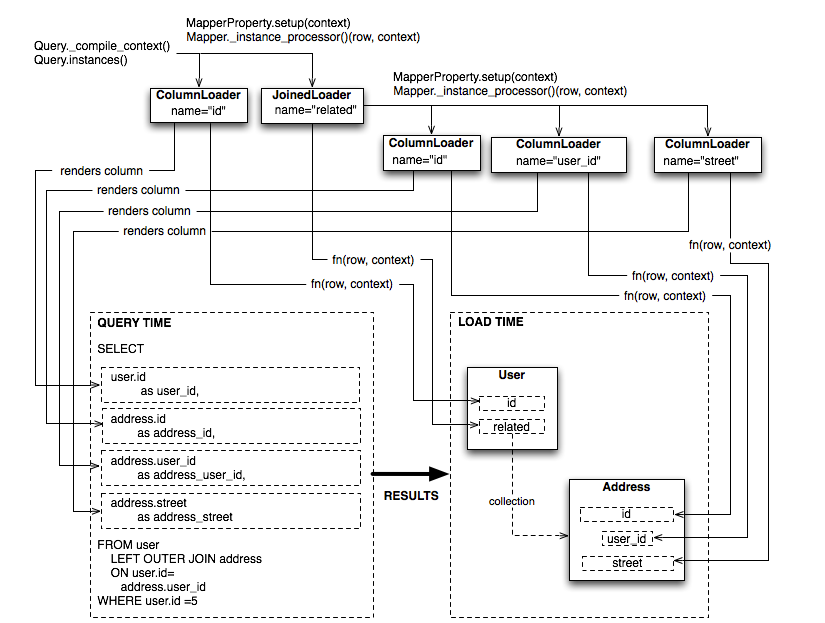
query = session.query(User.id).filter(User.name == 'ed')

Листинг 4.21 *SQLAlchemy* упит који се изражава путем изразног стабла

Израз задат у листингу 4.21 представљен је изразним стаблом на слици 4.12.

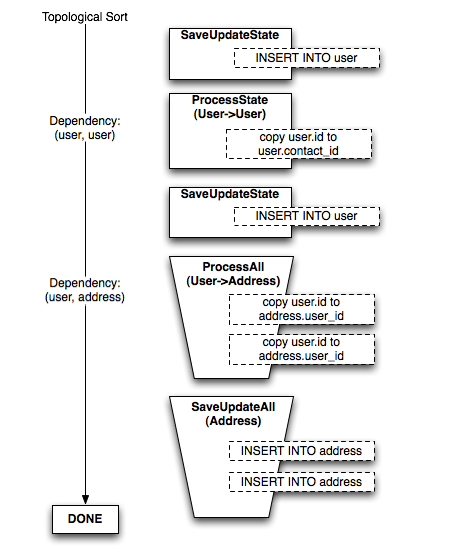
Слика 4.12 Пример изразног стабла *Query* API израза

Након што се изгенерише SQL упит и изврши над базом података, записи из базе података, захваћени упитом, се морају претворити у доменске објекте апликације. Овај процес се назива **фаза учитавања** података (енгл. *data loading*) и ослања се на рекурзивни обилазак структуре графа, формиране током фазе мапирања ентитета, при чему се узимају у обзир сви MapperProperty oбјекти и одлучује се да ли је атрибут, који је представљен овим објектом, неопходно иницијализовати подацима из базе. Ово се врши уз посредство LoaderStrategy објеката. Након што се фаза учитавања заврши, објекти су спремни за обраду у апликацији.

Слика 4.13 Фаза рендеровања и фаза учитавања

### Остале кључне особине

#### Јединица рада и тополошко сортирање

Као што је већ наведено у претходним поглављима, објекат сесије у *SQLAlchemy-jy* представља имплементацију обрасца јединице рада (*Unit of Work*), са циљем управљања променама над ентитетима. Како би се извршила синхронизација свих измена са базом података, неопходно је извршити наредбе перзистенције (DML наредбе) над базом. С обзиром на то да ентитети могу бити повезани, као и да промена над једним ентитетом мора да се пропагира на повезане ентитете, неопходно је наредбе перзистенције сортирати како не би дошло до нарушавања интегритета базе података. Као пример, може се узети брисање ентитета који је референциран путем ограничења страног кључа. Неопходно је прво, или обрисати све повезане ентитете, или извршити модификацију над њима, како се не би нарушило ограничење страног кључа. Из тог разлога, наредбе се смештају у граф зависности који се, затим, тополошки сортира како би се произвео правилан редослед извршавања. Сортирање се врши на основу претходно конструисаног графа зависности између Mapper објеката. Након што се све наредбе сортирају у правилном тополошком редоследу, оне бивају извршене над базом података како би се стања апликације и базе синхронизовали.

Слика 4.14 Тополошки сортиране DML наредбе

#### Систем догађаја

Систем догађаја (енгл. *Event System*) *SQLAlchemy* алата омогућава корисницима да креирају посебне Пајтон функције које су пријављене (енгл. *subscribe*) на oдређене догађаје у систему, чиме се омогућава да се одговарајући код изврашава током кључних фаза животног циклуса ентитета или операције. Овакве функције се називају слушаоци (енгл. *Listeners*). Tипични примери догађаја су прелази ентитета из једног стања у друго, учитавање података или синхронизација. Уопштено, систем догађаја омогућава увид у интерне механизме *SQLAlchemy-ja* и подржава проширивост и подешавање алата потребама апликације.

У претходном поглављу изложени су *Entity Framework*, *Hibernate* и *SQLAlchemy* алати за објектно-релационо мапирање. Кроз детаљну анализу стечено је разумевање о основама ових алата, њиховим битним карактеристикама и начинима на које се могу интегрисати у различите системе. С обзиром на чињеницу да је ефикасност кључна карактеристика за сваки софтверски производ, неопходно је разумети како ови алати функционишу ради остварења оптималних перформанси апликација заснованих на објектно-релационим маперима као механизмима за перзистенцију података.

Из наведеног разлога, у наредном поглављу биће приказано темељно истраживање о утицају коришћења ових алата на перформансе апликација и пружен одговор на питање које се често појављује у различитим дискусијама међу програмерима: Да ли је коришћење објектно-релационих мапера анти-шаблон?

## DEMONSTRACIJA

* Pokažite bitne elemente korišćenja aplikacije.
* Ovaj odeljak može biti poput uputstvo za korišćenje sistema.
* Opišite jedan scenario (ili više) pri korišćenju vaše aplikacije. Korak po korak prikažite kako korisnici stupaju u interakciju s vašom aplikacijom. Ubacite slike sa izgledom ekrana, koji će ilustrovati važne faze u njegovom korišćenju. Ovaj scenario (ili više njih) koji je ovde predstavljen bi bilo poželjno da bude pokriven i dinamičkim dijagramima u Poglavlju 4, i delimično ili potpuno pokriven listinzima u Poglavlju 5.

# ZAKLJUČAK

* Rekapitulacija glavnih poenti u radu:
  + Rešavani problem i motivacija za njegovo rešavanje
  + Grub opis rešenja
  + Osvrnuti se na poglavlje 2 (Pregled sličnih sistema) i zaključiti šta je to što ste vi uradili bolje ili drugačije od drugih.
* Opis mogućih pravaca daljeg proširivanja/unapređenja/otklanjanje identifikovanih nedostataka rešenja

# LITERATURA

1. *Cider* razvojno okruženje <https://cider.readthedocs.io/en/latest/> [Datum pristupa 13.08. 2016].
2. Abraham, A., 2005. Rule‐Based expert systems. *Handbook of measuring system design*.
3. Gabriel, R.P. and Pitman, K.M., 1988. Endpaper: Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, *1*(1), pp.81-101.

# BIOGRAFIJA

Ovde navedite osnovne informacije o sebi koje će predsednik komisije pročitati prilikom otvaranja odbrane završnog/diplomskog rada

Na primer: “Petar Petrović je rođen dd.mm.gggg. u Novom Sadu, gde je stekao svoje osnovno i srednje obrazovanje. Školske xxxx/yy godine se upisuje na Fakultet tehničkih nauka na studijski program X. Položio je sve ispite predviđene planom i programom i stekao uslov za odbranu završnog rada.“

1. Слика преузета са https://www.entityframeworktutorial.net/entityframework6/ [↑](#footnote-ref-1)
2. Слика преузета са: https://www.codemag.com/article/0711051/Introducing-ADO.NET-Entity-Framework [↑](#footnote-ref-2)
3. Слика преузета из [20] [↑](#footnote-ref-3)
4. Слика преузета са: https://www.codeproject.com/Articles/1240553/LINQ-Part-An-Introduction-to-IQueryable [↑](#footnote-ref-4)