

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут фізики, математики та інформаційних  
технологій


Кафедра інформаційних технологій та систем

**Сінько Олександр Володимирович**

**ДОСЛІДЖЕННЯ ПІДХОДІВ РЕАЛІЗАЦІЇ JAVA PERSISTENCE API В  
ВЕБ-ДОДАТКАХ**

**кваліфікаційна робота  
здобувача вищої освіти другого (магістерського) рівня  
освітньої програми «Мультимедійні системи»  
за спеціальністю 121 „Інженерія програмного забезпечення”**

Особистий підпис \_\_\_\_\_ Олександр СІНЬКО

Науковий керівник  \_\_\_\_\_ Світлана ПЕРЕЯСЛАВСЬКА,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій та  
систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук,  
доцент кафедри інформаційних  
технологій та систем

Полтава – 2023

## АНОТАЦІЯ

**Сінько О.В.**

**Тема:** Дослідження підходів реалізації Java Persistence API в веб-додатках.

**Спеціальність:** 121 "Інженерія програмного забезпечення".

**Установа:** ДЗ ЛНУ імені Тараса Шевченка, 2023 р.

**Кваліфікаційна робота містить:** 59 стор., 19 рис., 22 джерел, 1 додаток.

**Об'єкт дослідження** – специфікація Java Persistence API.

**Предмет дослідження** – веб-додаток з реалізацією Java Persistence API.

**Мета роботи** – дослідження підходів реалізації Java Persistence API при опрацювання реляційних баз даних в веб-додатках.

**Методи дослідження:** *теоретичні:* аналіз наукової літератури, узагальнення та систематизація теоретичних положень про створення веб-сервісів, та шляхів роботи з БД, на мові програмування Java з використанням певних бібліотек та фреймворків; *експериментальні:* тестування розробленого додатка.

**Результати роботи.** Досліджено сучасні технології роботи з базами даних в Java веб-додатках. Розроблено веб-сервіс на мові програмування Java, який виконує основні CRUD методи та реалізує Java Persistence API.

**Ключові слова:** веб-додаток, Java Persistence API, DAO, ORM, база даних.

## ABSTRACT

**Sinko O.V.**

**Theme:** Researching Java Persistence API Implementation Approaches in Web Applications.

**Specialty:** 121 "Software engineering".

**Institution:** Taras Shevchenko National University of Luhansk, 2023.

**Qualification work contains:** 59 pages., 13 figures, 22 sources, 1 appendices.

**Object of research** – Java Persistence API specification.

**Subject of research** - web application with Java Persistence API implementation.

**Purpose of work** – research of Java Persistence API implementation approaches when working with relational databases in web applications.

**Methods of research:** *theoretical*: analysis of scientific literature, generalization and systematization of theoretical provisions on the creation of web services and ways of working with databases, in the Java programming language using certain libraries and frameworks; *experimental*: testing the developed application.

**Results of work.** Modern technologies for working with databases in Java web applications are studied. A web service has been developed in the Java programming language, which performs basic CRUD methods and implements the Java Persistence API.

**Keywords:** web application, Java Persistence API, DAO, ORM, database.

# ЗМІСТ

Перелік умовних скорочень .....	6
Вступ.....	7
Розділ 1. Теоретичні основи реалізації Java Persistence API в веб-додатках ....	9
1.1. Архітектура та структура веб-додатка на платформі Java.	8
1.2. Розгляд архітектури REST при розробці веб-додатка.....	11
1.3. Принципи роботи з базою даних у Java веб-додатках .....	12
1.3.1. JDBC .....	12
1.3.2. JPA, Hibernate .....	17
1.3.3. Spring Data.....	23
1.3.4. Порівняльний аналіз існуючих підходів до роботи з БД.....	24
1.4. Структура досліджуваного додатка.....	26
Висновки до розділу 1 .....	28
Розділ 2. Програмна розробка веб-додатка з реалізацією Java Persistence API .....	29
2.1. Характеристика веб-додатка.....	29
2.1.1. Цілі, завдання проекту.....	29
2.1.2. Огляд необхідних залежностей веб-додатку.....	30
2.1.3. Конфігурація веб-додатка .....	30
2.1.4. Огляд “cross-cutting” логіки розробленої для додатка .....	35
2.1.5. Модель бази даних веб-додатка .....	36
2.2. Реалізація технології Java Persistence API у досліджуваному додатку...	39
2.2.1. Розробка модулю Entity.....	39
2.2.2. Розробка модулю DAO.....	42
2.3. Реалізація інших необхідних модулів.....	48
2.3.1. Розробка модулю Controller .....	48
2.3.2. Розробка модулю View.....	50

2.4. Аналіз отриманих результатів .....	53
Висновки до розділу 2 .....	56
Висновки .....	57
Список використаних джерел .....	59
Додаток А.....	60

## **ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ**

OOP – Object-Oriented Programming

MVC – Model View Controller

SOLID – Single responsibility, Open–closed, Liskov substitution, Interface segregation, Dependency inversion

ACID – Atomicity, Consistency, Isolation, Durability

URL – Uniform Resource Locator

REST – Representational State Transfer

JVM – Java Virtual Machine

JRE – Java Runtime Environment

JDK – Java Development Kit

SQL – Structured Query Language

HTML - HyperText Markup Language

## ВСТУП

**Актуальність роботи.** Організувати взаємодію з базою даних у програмі на Java можна у різний спосіб. Найпростіший і- використовувати API JDBC (Java Database Connectivity), в даному випадку код буде містити вбудовані запити. Але у разі змін схеми бази даних розробникам доведеться адаптувати вихідний код додатків, щоб отримати доступ до змінених елементів. Більш складним, зате гнучкішим рішенням, є можливість скористатися специфікацією Java Persistence API та певними ORM-фреймворками, що реалізують перетворення концепцій програми (класів, методів та атрибутів і т.д.) в концепції бази даних (таблиці, стовпці, рядки та і т.д.). ORM забезпечують високий рівень абстракції реляційної бази даних, що дозволяє використовувати мову програмування замість операторів SQL і процедур, що зберігаються. Як наслідок, взаємодія між прикладною програмою та базою даних може стати динамічнішим, але і більше складним для розуміння. Виходячи зі сказаного, можна зробити висновок про пріоритетність Java Persistence API в додатках, що містять бази даних, та, як наслідок, актуальність досліджуваної теми кваліфікаційної роботи.

**Мета роботи** – дослідження підходів реалізації Java Persistence API при опрацювання реляційних баз даних в веб-додатках.

**Об’єкт дослідження** – специфікація Java Persistence API.

**Предмет дослідження** – веб-додаток з реалізацією Java Persistence API

Відповідно до предмета дослідження і мети, були виділені основні завдання дослідження:

- дослідити теоретичні основи реалізації Java Persistence API в веб-додатках.
- розглянути архітектуру та структуру веб-додатка на платформі Java
- проаналізувати принципи роботи з базою даних у Java веб-додатку.
- розробити веб-сервіс на мові Java, який реалізує Java Persistence API.

Для вирішення завдань дослідження використано такі **методи дослідження**: *теоретичні*: аналіз наукової літератури, узагальнення та систематизація теоретичних положень про створення веб-сервісів, та шляхів роботи з БД, на мові програмування Java з використанням певних бібліотек та фреймворків; *експериментальні*: тестування розробленого додатка.

**Практичне завдання розробки** – Досліджено сучасні технології роботи з базами даних в Java веб-додатках. Розроблено веб-сервіс на мові програмування Java, який реалізує Java Persistence API.

**Структура дипломної роботи.** Робота складається з пояснювальної записки, списку використаних джерел, додатків. Обсяг роботи становить 59 сторінок, обсяг використаної літератури - 22 джерел. До складу роботи входять два розділи. Перший розділ присвячено теоретичним аспектам технологій роботи з базами даних в Java веб-додатках. Окремим питанням розглядається Java Persistence API. Другий розділ присвячено розробці веб-сервісу, який реалізує Java Persistence API.



## **РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РЕАЛІЗАЦІЇ JAVA PERSISTENCE API В ВЕБ-ДОДАТКАХ**

### **1.1. Архітектура та структура веб-додатка на платформі Java**

Багато сучасних програмістів віддають перевагу у своїй роботі користуванню такої мови програмування як Java. Пов'язано це з тим, що ця мова є універсальною. Вона зручна, зрозуміла та практична. З самого свого виникнення мова Java стала активно розвиватися та удосконалюватися. Зараз Java зручно використовувати як для комп'ютерних програм, так і для мобільних платформ. Це повноцінна мова, підтримуюча об'єктно-орієнтоване програмування.

Ключова особливість Java – можливість створювати веб-додатки та розширення. Java має наступні особливості, по перше це наявність функціоналу для ООП, подруге зрозумілий та відносно простий синтаксис і також достатня кількість документації на усіх мовах. Java відмінно “працює” з Мережею. Також варто відзначити – згадана мова є кросплатформною. Перенести програму з однієї ОС в іншу вдається в найкоротші терміни без втрати якості вихідної кодифікації.

Java є гарним інструментом та платформою для розробки веб-додатків. Веб-додаток на Java – це спеціальна програма, заснована на принципі роботи по типу клієнт-сервіс. В веб-додатку клієнт буде взаємодіяти з веб-сервісом, з допомогою допоміжних утиліт. Їх називають браузерами. Логіка розподілу між сервером та клієнтом. Зберігання інформації виготовляється в основному на сервері. Обмін даними забезпечується Мережею. Основна перевага такого контенту – це те, що клієнти не залежатимуть від тієї чи іншої операційної системи. Не важливо, яку ОС встановлено на задіяному пристрої. Веб-додатки працюватимуть скрізь.

Архітектура веб-додатків включає в себе клієнтську та серверну частину. За рахунок цього втілюється технологія під назвою “клієнт-сервер” [12]. Клієнт відповідає за реалізацію користувацького інтерфейсу. Також він формує серверні запити та обробляє відповіді які отримують від відповідних “команд”.

Серверна частина отримує заданий клієнтом запит, здійснює необхідні обчислення, та формує відповідь, клієнту для майбутніх маніпуляцій, після чого здійснює відправку отриманої відповіді по Мережі використовуючи протоколи HTTP.

Також існує третій “компонент” який використовується, це місце для зберігання важливих даних, таких як інформація о користувачі, повідомлення, теги, коментарі, і так далі. Для цього потрібна завжди оновлювана база даних. Тобто веб-додаток складається з трьох основних частин, це користувацький інтерфейс, серверна частина для роботи з даними та формування відповіді на запит інтерфейсу, і база даних де зберігаються усі необхідні данні.

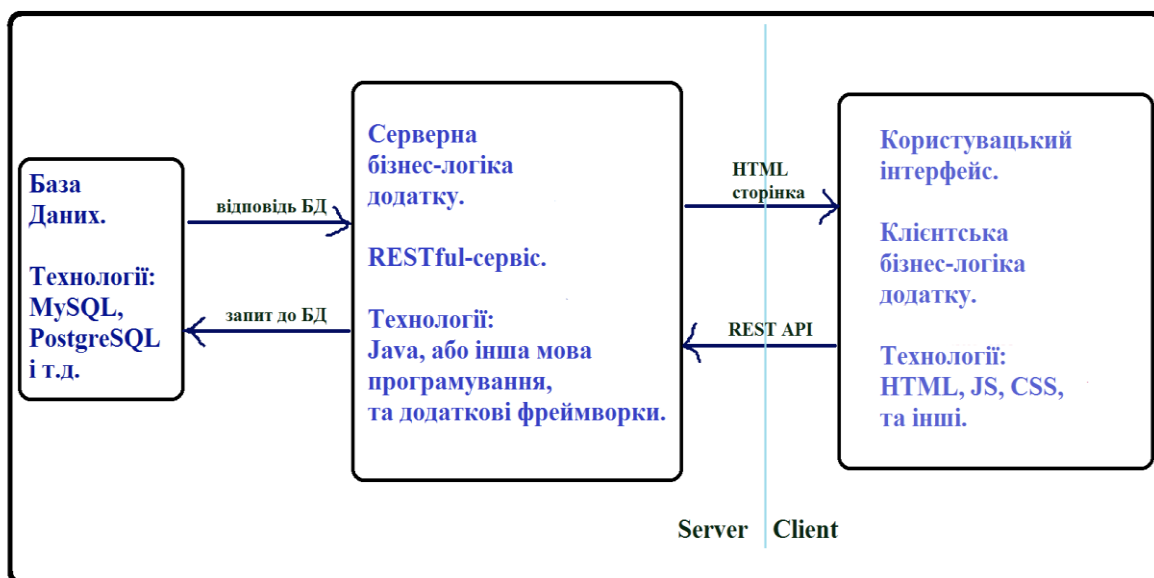


Рис. 1.1. Архітектура веб-додатку

## **1.2. Розгляд архітектури REST при розробці веб-додатка**

Коли є взаємодія з будь-якою веб-сторінкою, це включає запит і відповідь через сторінку HTML. Аналогічно, веб-сервіси також включають запит та відповідь, але у формі XML чи JSON. Java, будучи відповідною мовою для взаємодії на стороні сервера, забезпечує взаємодії між різними програмами на різних платформах.

Веб-сервіс в Java є відповідним середовищем для поширення зв'язку між клієнтськими та серверними програмами у мережі. Це програмний модуль, який призначений для виконання певного набору завдань, таким чином, веб-сервіси можна шукати через мережу, а також викликати їх відповідним чином. Підчас запиту веб-сервіс зможе надати функціональність клієнту, який викликає цей веб-сервіс. Функції та переваги веб-сервісів бувають різними, це може бути надання бізнес-функцій у мережі, взаємодія між програмами, тобто веб-служби які дозволяють різним програмам взаємодіяти один з одним, та інші сервіси які можуть принести користь користувачам.

Як правило, існує два типи веб-сервісів Java, це Soap Web Service, або RESTful web Service. У майбутньому веб-додатку буде використовуватися використовуючи тип REST. REST означає Representational State Transfer (передача стану уявлення), це популярний архітектурний підхід для створення API у сучасному світі. Тобто REST це дуже зручний спосіб спілкування між додатками. Він описує стандарти, використовуючи які Клієнт взаємодіє з Сервером за допомогою HTTP, де виклики REST API можуть здійснюватися за допомогою HTTP. REST API зручний тим, що він не змушує використовувати якусь конкретну мову програмування. Клієнтська та Серверна частини нашої програми можуть бути написані різними мовами програмування. Також REST API зручний тим що він не прив'язаний до конкретного типу передачі даних, тобто для передачі даних можна використовувати не тільки JSON, але і будь-який інший формат даних, але найпоширенішим і дуже зручним є формат JSON. JSON розшифровується як

JavaScript Data Notation. Формат JSON є текстовою інформацією. JSON не прив'язаний до конкретної мови програмування та використовується повсюдно. JSON використовуються для зберігання та особливо для обміну інформацією.

Формат даних JSON містить колекцію пар ключ-значення. JSON Data Binding або JSON Mapping – це прив'язка JSON до об'єкта Java. Прив'язка здійснюється для того, щоб можна було з JSON отримати об'єкт Java і навпаки перетворити Java об'єкт в JSON формат. Для того щоб робити це перетворення Spring за лаштунками використовує проєкт Jackson, тобто не потрібно писати в ручну перетворення даних, Spring Framework робить все сам. Перетворення даних з Java об'єкта в JSON відбувається за допомогою getter'ів, і навпаки перетворення даних з JSON в Java об'єкт використовується setter'и.

REST API не змушує використовувати певну мову програмування. Клієнтська та серверна частини нашої програми можуть бути написані різними мовами програмування. Для передачі інформації можна використовувати не тільки JSON, а й будь-який інший формат даних.

REST фокусується на ресурсах і на тому, наскільки ефективно ви виконуєте операції з ними, використовуючи HTTP. HTTP визначає структуру запиту, тобто це протокол який використовується для передачі даних у мережі. Якщо написати певний запит у браузер, то цим відбувається посилка HTTP request, після чого у веб-браузері виникає http response(відповідь), в такому випадку клієнтом є браузер, в ньому користувач запитує інформацію, тим самим посилає запит і сервер браузера повертає запитувану інформацію.

### **1.3. Принципи роботи з базою даних у Java веб-додатках**

Для взаємодії та роботи з БД у певному додатку, наприклад веб-додатку який буде представлений у цій роботі в якості прикладу, існують способи(persistence api), основні з яких буде розглянуто, проаналізовано та

досліджено, а саме це: JDBC, JPA Hibernate та Spring Data. Цей список варіантів роботи називаються технологіями для роботи з базою даних, деякі з яких можуть використовуватися самі по собі, а деякі тільки у зв'язці з іншими, наприклад JDBC можливо використовувати окремо від інших технологій у переліку який буде розглянуто у цьому дослідженні, а Hibernate не має такої можливості тому що він використовує JDBC “під капотом”, та доповнює цю технологію дуже важливим функціоналом. Більш детально роботу цих(JDBC, Hibernate, JPA, SpringData) технологій буде розглянуто, досліджено та проаналізовано далі у роботі.

### **1.3.1. JDBC**

#### **JDBC, Java Data Base Connectivity(з'єднання з базами даних на Java)**

– це професійний стандарт для взаємодії Java-додатків з різними СУБД. JDBC заснований на концепції драйверів, які дозволяють отримувати з'єднання з базою даних спеціальним URL. При завантаженні драйвера, він реєструє себе у системі і далі автоматично викликається, коли програма просить URL адрес, який містить протокол, за який цей драйвер відповідає. JDBC має певні переваги у використанні, по перше, це легкість розробки, тобто розробник може не знати специфіки бази даних, з якою він працює, по друге, це тощо код який написав розробник майже не змінюється при переході, наприклад, компанії на другу базу даних, а також можливо підключитися до будь якої бази даних за допомогою певного URL.

URL являє собою певну адресу яка складається з протокола “jdbc:”, підпротокола, а саме це назва БД яка планується використовуватися “mysql:”, та останній фрагмент URL'у, це ідентифікатор бази даних, це локальне ім'я БД, тобто як розробник сам назвав базу яку він використовує. Приклад JDBC URL для підключення до такої БД як MySQL – “jdbc:mysql://localhost:3306/my\_db”, назва “my\_db”, розташований за адресою localhost, та очікуючий підключення до порту 3306, де localhost та 3306 це стандартні значення у адресі які можуть використовуватися у роботі з БД на локальному хості.

JDBC складається з двох частин, це – jdbc api, тобто це набір класів та інтерфейсів, потрібний для отримання доступу до бази даних. У Java ці методи та класи розташовані у двох певних пакетах(`java.sql`, `javax.sql`). Друга складова частина, це JDBC-драйвер, це компонент, специфічний для кожної існуючої бази даних.

JDBC має класи та інтерфейси для взаємодії з БД, основні це:

- `java.sql.DriverManager` – клас, який дозволяє завантажити та зареєструвати необхідний JDBC-драйвер, а потім отримати з'єднання з базою даних.

- `java.sql.DataSource` – інтерфейс, який вирішує ті самі завдання, що і `DriverManager`, але більш зручним та універсальним чином.

- `java.sql.Connection` – забезпечує формування запитів до джерела даних та управління транзакціями.

- `java.sql.Statement/PreparedStatement` – ці інтерфейси дозволяють надіслати запит до джерела даних.

- `java.sql.ResultSet` – оголошує методи, які дозволяють переміщатися по набору даних та зчитувати значення окремих полів у поточному записі.

Основні етапи роботи с базою даних при використанні JDBC. По перше це реєстрація драйверів, по друге це встановлення з'єднання з базою даних, далі створення запиту до БД, далі виконання запиту до БД, потім обробка результатів та закриття з'єднання з БД. Для того щоб зареєструвати JDBC драйвер потрібно виконати певну команду, а саме викликати метод `newInstance()` за допомогою метода `forName("ім'я класу драйвера")`, а цей метод викликати за допомогою класу `"Class"`.

Далі з'являється можливість встановити з'єднання з БД, для цього потрібно використати статичний метод `java.sql.DriverManager.getConnection (...)`, у якості параметру може передаватися наприклад URL бази даних. В результаті виклику цього методу буде встановлено з'єднання з БД та створен об'єкт класу `java.sql.Connection`

– це своєрідна сесія, всередині контексту якої і буде проходити подальша робота з базою даних.

Далі починається робота з транзакцій, де транзакція – це певний вплив на базу даних, що переводить її з одного цілостного стану в інший, що виражається у зміні даних, що зберігаються у базі даних. Також транзакції мають деякі рівні ізолюваності, основні це – `read uncomited` або `dirty read`, тобто читання непідтверджених/незафіксованих даних у транзакції. Немає гарантії того що дані змінені другими транзакціями, не будуть у будь-який момент часу змінені у результаті їх відкату, тому таке читання може бути потенційним джерелом помилок.

Наступний рівень ізолюваності транзакцій це `read committed` – читання тільки зафіксованих транзакцій, тобто після виконання команди `commit`, наступний це `repeatable read` – а саме, це, читання усіх змін своєї транзакції, цей рівень транзакції каже про те що в рамках однієї транзакції, один і той самий запит, наприклад, `select` виконаний двічі, повинен давати один і той самий результат.

І останній рівень, це `serializable` – результат паралельного виконання серіалізованої транзакції с іншими транзакціями повинен бути логічно еквівалентний результату їх послідовного виконання, тобто `serializable` потребує послідовного виконання транзакцій. Чим більше “lock’ів” накладається на транзакцію тим вона буде більш швидкою, відповідно `read uncommitted` повільніше усіх, а `serializable` найшвидша.

Існують деякі правила для написання більш якісних транзакцій розробником, разом ці правила називаються ACID:

**Atomicity**(атомарність) – гарантує, що ніяка транзакція не виконається та не зафіксується у системі частково, тобто, будуть виконані усі підоперації транзакції, або не виконано ні одної.

**Consistency**(Узгодженість) – транзакція, що досягає свого нормального завершення і тим саме фіксує свої результати, зберігає узгодженість бази

даних. Загалом цей принцип говорить про те, що одна транзакція повинна переводити базу даних з одного консистентного стану в інший.

Isolation(Ізольованість) – під час виконання транзакції паралельні транзакції не повинні впливати на її результат.

Durability(Долговічність) – незалежно від проблем на “нижніх” рівнях(наприклад, знеструмлення системи чи збої в обладнанні) зміни, зроблені успішно завершеною транзакцією, повинні залишатися збереженими після повернення системи до роботи.

Для формування запитів до бази даних у Java використовуються три основні інтерфейси, по перше, це `java.sql.Statement` – для операторів SQL без параметрів, по друге, це `java.sql.PreparedStatement` – для операторів SQL з параметрами, та останній це `java.sql.CallableStatement` – для виконання збережених у базі процедур.

Запит до БД та обробка результатів реалізується за допомогою певних методів. Виконання запитів здійснюється за допомогою виклику методів об’єкта, реалізуючого інтерфейс `java.sql.Statement`:

- `executeQuery()` – для запитів, результатом яких є один набір значень, наприклад `SELECT`. Результатом виконання є об’єкт класу `java.sql.ResultSet`.

- `executeUpdate()` – для виконання операторів `INSERT`, `UPDATE`, `DELETE`, а також для операторів `DDL`(Data Definition Language). Метод повертає ціле число, яке показує, скільки записів було модифіковано.

- `execute()` – виконує SQL команди, які можуть повертати різні результати. Наприклад, можна використовувати для операції `CREATE TABLE`. Повертає `true`, якщо перший результат містить `ResultSet` і `false`, якщо перший результат – кількість модифікованих записів або результат відсутній. Щоб отримати перший результат, необхідно викликати метод `getResultSet()` або `getUpdateCount()`. Інші результати доступні через виклик `getMoreResults()`, який при необхідності може бути здійснений багаторазово.

Об’єкт з інтерфейсом `java.sql.ResultSet` зберігає в собі результат запиту до бази даних – деякий набір даних, всередині якого є “курсор”, що вказує на



один із елементів набору даних, тобто на поточний запис. Використовуючи цей “курсор”, можна переміщатися по набору даних за допомогою методу `next()`.

Закриття з’єднання за базою даних, відбувається за допомогою виклику методу `close()`, у відповідного об’єкта `java.sql.Connection` або за допомогою використання механізму `try-with-resources` при створенні такого об’єкта.

### **1.3.2. JPA, Hibernate**

Розбираючи тему роботи з базою даних у Java веб-додатках. Ця тема важлива з декількох причин, по перше майже всі проекти мають базу даних, більшість проектів мають реляційну базу даних, якщо говорити про enterprise, і більшість проектів використовують якийсь ORM фреймворк для того щоб працювати с БД. Hibernate це один з найпопулярніших фреймворків. Більшість проблем з перформенсом які є на проектах це проблеми з persistence layer’ом, тобто на рівні роботи с базою даних. Взагалі такі проблеми беруться по тій причині що, хтось пише код, та не розуміє як він працює “під капотом”.

Якщо говорити про ORM, Object-Relation Mapping(Об’єктно-реляційне відображення), то мова йде про мапінг і відповідність, або не відповідність, з одного боку об’єктно орієнтованою, а з іншого боку реляційною моделлю. Між цими двома моделями є невідповідності, і завдання ORM фреймворка в першу чергу це вирішити проблему не відповідності між двома моделями, це потрібно робити для того щоб у коді будуючи persistence layer(рівень роботи з базою даних) могли працювати в об’єктно орієнтованій моделі, а не в реляційній, тобто є фронт енд, є бізнес логіка і в якийсь момент потрібно звертатися до даних, працювати с даними, зберігати, обновляти, видаляти і т.д. Для того щоб працювати с даними не в реляційній моделі, тобто щоб не писати SQL код кожного разу, а щоб працювати в об’єктно орієнтованій моделі, так як робота відбувається у бізнес логіці. Так ось, які є проблеми невідповідності, а є їх багато:

- Associations – якщо в Java потрібно зробити асоціацію між двома класами, тоді просто створюємо поле і робимо посилання. Для того щоб

зробити асоціацію в базі даних, то потрібно використовувати “foreign key” і якщо наприклад мова йде про “один до одного” чи “один до багатьох”, то достатньо створити просто одну колонку, і якщо ми говоримо про “багато до багатьох”, потрібна ще ціла окрема таблиця, тоді як в Java можна просто створити список користувачів, і створити список задежностей, і все.

- Identity – ще одна проблема, це проблема ідентичності, тобто якщо відбувається порівняння через “==”, або через метод “equals()”, у базі даних порівняння відбувається через ідентифікатор “id”.

- Navigation – проблема з навігацією, де в Java є можливість використати методи getName, getList і т.п., тобто є можливість через get’ери, дійти до будьякого місця, де потрібно щось зробити, це в Java, а якщо говорити про навігацію в базі, то не має можливості використовувати get() методи, тобто в базі даних подібна проблема вирішується, або ще одним запитом, або за допомогою join’ів.

- Granularity – в Java коли є один тип, наприклад User, він може містити поле типу примітива, а може містити поле якогось складного типу, наприклад Adress, тобто може бути “тип у типі”. В базі даних, класу відповідає таблиця, але таблиця не може мати значення складних типів, тобто не має можливості створити в базі даних якийсь тип і зробити колонку типу Adress, в базі даних у такому випадку потрібно буде працювати з асоціацією, тобто створювати нову таблицю типу Adress, та роботи “foreign key” на таблицю User.

- Inheritance – ще одна проблема, це проблема наслідування, в Java ми можемо сказати, наприклад, Admin extends User і у Admin автоматично всі поля User з’являються. В базі даних ми не можемо сказати, таблиця Admin extends таблиця User.

Ці усі проблеми мають велике значення, тому що в Java є можливість працювати з асоціаціями, з наслідуванням, з підтипами, по одному, а в базі даних вони будуть реалізовані по іншому, усі ці речі є можливість реалізовувати в ручну через SQL як у JDBC, тобто працюючи через реляційну модель. Але є такі інструменти як ORM фреймворки, завдання яких “під

капотом” вирішувати, або надавати різні опції вирішення цих основних проблем.

JPA, Java Persistence API – це специфікація ORM фреймворку в Java.

Hibernate ORM – це реалізація JPA, тобто реалізація ORM фреймворку в Java, тобто Hibernate ORM, це ORM фреймворк який вирішує невідповідності між об’єктною і реляційною моделю, дозволяє працювати з базою даних в об’єктно орієнтованій моделі, і перетворює зміну стану об’єктів в DML, тобто в Data Manipulation Language. Оскільки Hibernate ORM вирішує проблему між об’єктно орієнтованою і реляційною моделю і дозволяє працювати з даними в об’єктній моделі, то наприклад, коли в об’єктній моделі треба зробити якусь транзакцію, наприклад викликати якийсь метод, то Hibernate повинен після цього “під капотом” послати в базу даних запит update, і в таблиці, в рядку зробити зміни, тобто це робота Hibernate ORM.

Перевага ORM і Hibernate у тому що, по перше, якщо порівнювати з JDBC, це робота з об’єктно орієнтованою моделю, по друге зникає “boilerplate” код який потрібно писати у JDBC, усі ці try-catch, запити і т.д., і продуктивність написання логіки використовуючи Hibernate збільшується, а також підтримка такого коду значно покращується, того що, наприклад, є у коді усюди прописаний у ручну SQL, і припустимо є якась таблиця, і припустимо до цієї таблиці потрібно додати ще одну колонку, і виходить що усюди де є “prepared statement”, “statement”, де є “захардкожений” SQL, усюди можуть виникнути проблеми у коді, тобто простіше якщо писати не напряму, а використовувати Hibernate ORM.

Якщо говорити про продуктивність, то Hibernate ORM працює повільніше ніж JDBC, тому що це ще один рівень логіки поверх JDBC. Але краще отримати усі переваги ORM фреймворку, ніж виграти трохи перформенсу, в тих випадках, де перформенс критичний, де потрібно писати SQL через JDBC, то з Hibernate без проблем можна перейти на JDBC.

При використанні Hibernate ORM та JPA, виникають залежності, функціонал яких потрібно буде використати у проекті, це:

- JPA.

- Hibernate Core.

Далі, для того щоб працювати з Hibernate і JPA, то потрібно це налаштувати у проєкті. Наприклад можна працювати з Hibernate без JPA, але на практиці краще працювати з JPA як з специфікацією, з Hibernate використати як реалізацію.

**Hibernate та JPA** – це дві сутності які працюють у взаємодії, де перша, Hibernate Framework, а саме – одна з найпопулярніших реалізацій ORM-моделі, де ORM(Об’єктно-реляційне відображення) описує відносини між програмними об’єктами та записами у БД [2]. Object-Relational Mapping(ORM) це технологія програмування, яка зв’язує бази даних з концепціями об’єктно-орієнтованих мов програмування. ORM – це насправді концепція, яка говорить про те що, Java об’єкт можна як уявити як данні у БД так і навпаки. Вона, ORM, знайшла втілення у вигляді специфікації JPA(Java Persistence API). Тобто JPA це специфікація, а Hibernate – це ORM фреймворк/набір бібліотек, яка сумісна з JPA [2] . Тобто Hibernate буду брати об’єкт і робити все для того щоб зміни у об’єктно орієнтованій моделі відображалися як зміни в реляційній моделі, базі даних.

Об’єкт, клас, який також називається сутністю, то в JPA цей об’єкт називається entity(сутність), тобто це клас, який не має бізнес логіки, а є тільки поля, геттери, сеттери та конструктори, цей клас є для того тільки, щоб зберігати дані, і задача Hibernate буде ці дані зберігати в базі, удаляти, оновлювати, і т.д.. Тому, для цього потрібно використовувати певні речі, для того щоб зв’язатися з базою даних, і створити усі конфігураційні потрібні класи для Hibernate, з’являється код в якому пишуться конфіги, і друге “мапінг”. Де конфігурація, це URL, username, password, також діалект SQL(MySQL, PostgreSQL, H2 і т.п.). А “мапінг”, це певні команди(анотації) які кажуть Hibernate, де певна entity, яку треба зберігати, в яку колонку і т.д. тобто треба зв’язати за допомогою анотацій, створити зв’язок, між об’єктно орієнтованою моделлю і реляційною. Маючи конфігурацію і мапінг,

з'являється можливість починати писати DAO(Data Access Layer) який “під капотом” буде використати не JDBC, а Hibernate.

Для створення проекту і до отримання інформації з БД, потрібно виконати кроки, такі як:

1. Створити бізнес модель, і класи у Java.
2. Створити для цих класів таблиці у БД.
  - для кожної таблиці має бути primary key.
  - також, перевагу треба віддавати герерованим id, тобто sequence.
  - дотримуватися naming convention створюючи таблицю.
  - реалізувати зв'язок між таблицями з допомогою foreign key.
  - “покривати” таблиці констрейнтами(nullable, unic).

3. реалізувати mapping і логіку persistence layer'a, передусім підключати dependency для jpa та hibernate.

4. не забути створити конфігурацію, у xml форматі, або за допомогою java класів.

5. розробити с початку основні CRUD(create, read, update, delete) операції, потім додавати той функціонал що потрібен у проекті.

6. притримуватися правил при створенні сутностей та persistence layer'a.

- вимикати dirty cheking для усіх read операцій.
- уникати непотрібних викликів persist та merge методів. Е так

званий save redundant анти патерн, який часто зустрічається у проектах, тобто для update entity в Hibernate використовувати dirty cheking, якщо є persistence context не read only, то зміни у entity автоматично роблять зміни до бази даних, але також може викликатися entity manager і метод save, який “під капотом” викликає метод merge. Тобто що змінилося dirty cheking, потім ще раз можна зберігають через репозиторій. Merge операція небезпечна тим, що вона в будь-якому випадку створює копію того, що ви може передаватися розробником у persistence context, тобто ось цей redundant save він створює додаткову роботу для hibernate, яка жодної логіки не виконує, а просто змушує процесор ще раз

скопіювати/перегнати дані. Тобто, висновок у тому, що потрібно уникати redundant save і для оновлень використовувати dirty cheking.

Процеси що проходять під капотом у Hibernate:

- dirty cheking

- lazy loading – працює тільки у рамках persistence context, якщо не вивантажили lazy, але при цьому вийшли за persistence context і зробили get, то буде lazy initialization exception.

- flush – запити в БД які робляться, відбуваються і викликаються не в момент коли ми просимо hibernate їх зробити, а в момент flush. Тобто коли ми просимо hibernate їх зробити, а в момент flush. Тобто коли викликається команда entityManager.persist або entityManager.remove, це не означає що hibernate в цей момент надсилає запит до бази даних, ці апити йдуть в БД на момент flush, де flush за замовчуванням відбувається на команді commit.

- Action Queue. Під капотом у hibernate все реалізовано на базі певних подій(тобто відбуваються якісь події і hibernate якось на них реагує), а усі свої дії hibernate робить у якості actions(дій), і ці всі actions складаються у певну чергу. Тобто коли hibernate каже persist, remove, merge, update, і т.д., то hibernate робить це все відразу, він оформляється в так званий action і зберігається у чергу. А у момент flush він бере цю чергу і виконує її, але виконує він їх не в тому порядку, в якому ми їх додаємо, а в порядку пріоритетності.

Hibernate - усі оперції, які викликає розробник, зберігає у чергу, сортує за пріоритетом операцій, і на момент flush викликає по черзі всі операції відповідно до їх пріоритету. Це означає що наприклад операція зберігання persist, тобто insert має вищий пріоритет ніж операція update і remove.

**Dirty checking** - це прилад що працює тільки в рамках persistence context'а, відповідає він за оновлення даних тобто update. Hibernate перевіряє чи змінюється entity, якщо вона змінюється, то він посилає в базу даних update.

Як це працює:

- У момент коли entity завантажується в persistence context, створюється її снєпшотна копія, яка виглядає як масив типу Object, в якому кожен елемент це по суті значення в рядку даних, тобто рядок entity завантажується в масив типу Object. Для кожної entity створюється такий масив. Потім у певний момент відбувається перевірка кожного fild'a entity зі значенням яке у снєпшоті тобто у цьому масиві типу Object. І якщо якесь значення змінилося генерується update. Перевірка відбувається у момент котрий називається flush. Flush це момент коли hibernate посилає sql запрів у свою базу даних. І цей ось flush відбувається за умовчанням під час комміту. Але flush не означає коміт транзакції, він означає, що просто sql пішов у базу даних. Тобто якщо insert пішов у базу його можна відкатити, тобто зробити rollback, якщо помилку вибило. У всіх місцях коли робляться select'и і не робимо update'и, то потрібно вимикати dirty cheking, тому що він включений за замовчуванням. Тобто у всіх місцях, де дані тільки читаються, потрібно вимикати dirty cheking, за допомогою анотації read-only. Можна вимкнути для конкретної entity, а можна вимкнути для цілого entity menager.

**Lazy loading** - це механізм, який дозволяє hibernate вивантажувати дані з бази не відразу, а у момент коли ми до них звертаємося.

FetchType Lazy говорить hibernate не витягувати дані відразу.

FetchType Eager говорить hibernate витягувати дані відразу.

Це означає, що в будь-якому місці де є релейшин (відношення) @OneToMany або @ManyToOne, коли потрібно буде витягнути ентіті, всі її релейшини відразу не підвантажуватимуться, тобто вони будуть lazy. Всі відносини @OneToOne або @ManyToOne будуть вивантажуватись відразу.

### 1.3.3. Sring Data/ Spring Data JPA

Якщо потрібно писати persistence layer, то можна звертатися до Spring Data, суть у тому що, якщо говорити взагалі про Spring, то більшість його проектів це покращення якихось уже існуючих фреймворків. Тобто існує багато різних технологій де можна зберігати данні(MySQL, PostgreSQL, MongoDB та інші), і Spring Data покриває їх усіх, тобто це великий проект, і

він надає схожий API тобто інтерфейс не дуже зважаючи увага чи то реляційна база даних, чи нереляційна.

Якщо говорити про реляційну базу даних, наприклад MySQL, або PostgreSQL, то використовуючи Spring Data можна працювати з цими базами через посередників таких як JDBC, Hibernate, або там через інші існуючі схожі варіанти. Тобто Spring Data в реляційну базу даних має теж різні підпроекти, чи там Spring Data JPA, або Spring Data JDBC.

Існують декілька шаблонні операції при написанні CRUD додатку, але їх вже потрібно прописувати самостійно так як не в кожному проекті використовуються наприклад:

- Pagation – який дозволяє вивантажувати дані не шматком, а вивантажувати їх по сторінках, ну або невеликими шматками, наприклад, коментарі можна завантажувати/вивантажувати невеликими шматками.

- Sorting – можна написати метод, який вивантажуватиме дані відсортовані за алфавітом або за віком тощо. Наприклад, якщо метод findAll(), тобто “знайти усіх” вивантажує дані просто, без порядку, то можна написати метод який робить це в певному порядку.

Однак! Написання такого шаблонного функціоналу можна уникнути використовуючи проект Spring Data REST.

#### **1.3.4. Порівняльний аналіз існуючих підходів до роботи з БД**

Після того як було визначено основні шляхи існуючих підходів до роботи з БД, таких як JDBC, JPA у роботі з Hibernate, Spring Data, то має сенс проаналізувати ці шляхи. Перш за все важливим визначенням є те що не має якогось універсального шляху який можливо використовувати повсемірно, кожний із шляхів має певні особливості. Після опрацювання матеріалу по шляхам роботи з БД, та уважного розгляду кожного із цих варіантів, стає зрозуміло те що кожний шлях це певний рівень, який має свою певну



реалізацію та використовується при певних вимогах, які потрібні при розробці проекту. Так самим найнижчим рівнем при роботі з БД є JDBC, по-перше це тому що цей рівень не є якимось ORM фреймворком, JDBC це певний набір бібліотек, і він працює у реляційній моделі даних, через те що це найнижчий рівень, то він має найменшу кількість складної логіки, і тому працює швидше, ніж інші рівні, але має також і мінус, у вигляді того що JDBC працює у реляційній моделі даних, і не є досить зручним.

Наступний шлях це JPA у роботі з Hibernate, де JPA є специфікацією, а Hibernate-Core, є реалізацією цієї специфікації [9]. JPA та Hibernate можуть працювати окремо, але це не є зручним та ефективним шляхом при роботі з БД, тому у великій кількості проектів, та взагалі у більшості де використовується цей рівень, то JPA та Hibernate-Core працюють у зв'язці. Цей шлях роботи з БД, а саме JPA у роботі з Hibernate-Core, є більш зручним ніж шлях роботи з БД JDBC, з тієї причини що це ORM фреймворк, тобто працює цей шлях у роботі з БД не в реляційній моделі даних, а в об'єктній, через це цей шлях легше у використанні для розробника, та проекти при використанні цього шляху, є більш якісними та гнучкими, але через те що це ще один рівень логіки, то цей шлях працює трохи повільніше, але це не є критичним, тому що переваги перебивають усі недоліки.

Наступний рівень це Spring Data і це по суті ще один рівень логіки, який у собі має усі переваги попередніх рівнів, та усі недоліки також, з тієї причини, що по суті коли використовується кожний наступний рівень, то він по факту використовує і попередні, тобто якщо JDBC це перший рівень логіки роботи з БД, а JPA у зв'язці з Hibernate-Core, це другий, то при роботі з Hibernate, по факту використовується і JDBC під капотом Hibernate, а при використанні Spring Data який є певною “коробкою” над якимось шляхом у роботі з БД, то стає зрозуміло, що найзручнішим та найякіснішим і ефективнішим є саме робота з Spring Data яка накладає своїми перевагами JPA з Hibernate, який у свою чергу накладає JDBC, але не треба забувати і про недоліки, тобто що кожний наступний рівень це певний шар логіки який накладається на

попередні, та займає пам'ять, та трохи уповільнює роботу. У додатку який планується розробити, буде використовуватися шлях роботи з БД, а саме JPA з Hibernate-Core, тому що при використанні JDBC, висвітлюється тільки робота у реляційній моделі і це не дає певну змогу показати як є зв'язується весь проект починаючи з БД, та завершуючи роботу зі View. Але не планується і використовувати Spring Data через те що він робить код проекту достатньо абстрактним. Тому у додатку який планується розробляти буде використовуватися певна “середина”, а саме JPA у зв'язці з Hibernate-Core.

#### 1.4. Структура досліджуемого додатка

При написанні роботи метою якої є дослідження варіантів роботи з базою даних, буде розроблено веб-додаток, який виконуватиме основні CRUD операції, використовуючи REST API.

Архітектура програми виглядає таким чином (рис. 1.2):

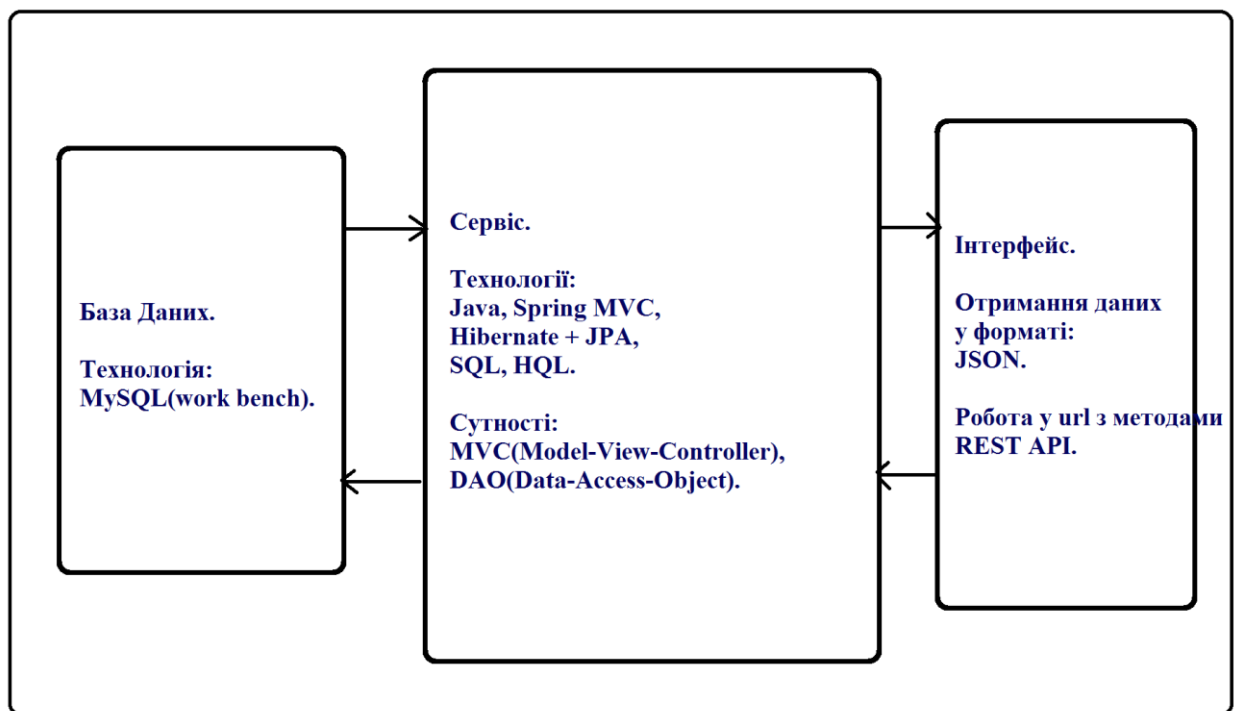


Рис. 1.2. Архітектура досліджуемого додатка

1. База даних, яка зберігає співробітників певних департаментів, з короткою витримкою інформації про цих співробітників.

2. Сервіс, а саме back end частина веб-додатка, яка виконує функцію обробки даних одержуваних з бази даних та передачу цих даних у браузер, у форматі JSON , де дані можна отримати за допомогою введення певного URL в адресний рядок браузера. У даному веб-додатку сервіс використовує такі сутності як DAO(Data access object) в якому виконується логіка отримання даних з бази даних, потім отримані дані передаються в додаткову сутність яка іменується як сервіс, і в цій сутності проводиться обробка даних, отриманих з DAO, і в результаті чого сервіс спрощує ці дані до рівня методів які будуть використовуватися в наступній сутності, а саме в контролері, де контролер це одна з сутностей такого патерна як MVC, в результаті чого контролер виконує безліч завдань, а саме вміщує в собі методи, які виконують конкретно задані завдання залежно від того які анотації спринг фреймворку проставлені над методом контроллера і які методи викликаються всередині цих методів. Прикладом одного з таких методів є метод над яким стоїть анотація “@GetMapping(‘/employees’)”, і всередині такого методу викликається функція з сервісу з отримання співробітників з бази даних, де потім контролер повертає посилання, що зберігає ці дані про співробітників, і потім після запуску серверу виникає можливість прописати конкретний URL адрес для отримання у браузері на екран даних о співробітниках.

3. Сервер - який запускає додаток, а саме Apache Tomcat server 8.5.

## Висновки до розділу 1

У данному розділі були розглянуті теоретичні основи реалізації Java Persistence API в веб-додатках. Було розглянуто та досліджено побудову архітектури та структури при реалізаціях веб-додатків. Розглянута архітектура REST, яка буде використовуватися при майбутній розробці додатку.

Також було досліджено принципи роботи роботи з базою даних у Java веб-додатку, проведено порівняльний аналіз існуючих підходів до роботи з БД. Встановлено, що існує багато технологій організації взаємодії з базою даних у програмі на Java. Однак, більшої переваги набувають технології, пов'язані з Java Persistence API, що реалізують перетворення концепцій програми (класів, методів та атрибутів і т.д.) в концепції бази даних (таблиці, стовпці, рядки та і т.д.). Відповідно до цього було розглянуто ORM-фреймворк Hibernate, що реалізує концепцію Java Persistence API та можливості його застосування з фреймворком Spring.

Також, під час дослідження визначено ролі основних компонентів які мають використовуватися при розробці веб-додатка, проаналізовано основні методології і принципи які мають використовуватися при розробці додатка. Крім того, розглянуто основні можливості шляхів взаємодії з БД, які можуть використовуватися при розробці додатків які працюють з базою даних на джава платформі.

## РОЗДІЛ 2. ПРОГРАМНА РОЗРОБКА ВЕБ-ДОДАТКА З РЕАЛІЗАЦІЄЮ JAVA PERSISTENCE API

### 2.1. Характеристика веб-додатка

#### 2.1.1. Цілі, завдання проекту

У ході дослідження шляхів для роботи з базою даних на платформі Java, був структурований та досліджений певний матеріал, а саме це – створення веб-додатків, робота з базою даних, основні технології при роботі з БД(JDBC, Hibernate у зв’язці з JPA та Spring Data).

Також було визначено цілі та завдання розробленого додатка. Перш за все основною **ціллю розробленого додатка**, це продемонструвати роботу з базою даних, додаток повинен під’єднуватись до БД, вміти виконувати певний основний для будьякого додатка функціонал у роботі з БД, а саме це отримувати дані, оновлювати їх, додавати, та видаляти дані. Тобто ціль додатка, це бути прикладом того як виглядає правильна реалізація роботи з persistence api. Основним **завданням проекту** було продемонструвати роботу на стороні сервісу, тобто “back end” частину, яка по своїй сутності є модулем для роботи з базою даних, тобто завдання проекту полягає у дослідженні шляхів роботи з БД на платформі Java, та основних моментів зв’язаних з цим.

У ході цього дослідження було вирішено розробити веб-додаток який був би прикладом роботи з базою даних, демонстрував основні етапи створення веб-додатку, та саме головне це те що розроблений додаток надавав би можливість ознайомитися з тим, як проходить робота з БД використовуючи певний шлях(Hibernate з JPA) при створенні додатку.

Для розробки додатку використовуються основний сучасний набір інструментів, які використовуються у більшості “enterprise” проектах, тобто при розробці великих, надійних, якісних проектах, а саме це – Java, Spring(Core, MVC), JDBC, Hibernate + JPA, SQL, HQL, MySQL, Servlets, JSTL, c3p0, aspectjweaver. Також у розробці подібних додатків використовуються

певні концепції, для більш якісного виконання тієї роботи для якої було розроблено додаток, і саме у додатку який розроблений, також було використано такі концепції як – OOP, SOLID, MVC, ACID, lazy loading, dirty cheking, IoC, DI, code conventions.

### 2.1.2. Огляд необхідних залежностей веб-додатку

Перед створенням веб-додатку, потрібно визначити з якими сутностями потрібно буде мати справу, для того щоб була можливість реалізувати необхідний функціонал, який би можна було називати веб-додатком. Так як розробка веб-додатку виконується на платформі Java, то розробка програм та додатків на джава повина виконуватися за допомогою якогось інструменту, а саме за допомогою “середовища розробки”, одним із таких є IntelliJ Idea, який надає та містить у собі багато потрібного функціоналу для комфортної розробки додатків, та одним із таких функцій є можливість в окремому файлі визначати потрібні сутності, з якими планує працювати розробник, ці сутності прийнято називати dependency, або залежності, для розроблення веб-додатку.

Файл який зберігає у собі dependency має назву pom.xml. Містить pom.xml у собі dependency. Кожна залежність обромлюється у певний scope(область) тегів: **<dependency>** інформація о залежності яка додається у проект**</dependency>**, у досліджуємому проекті цих dependency, тобто залежностей є сім штук, усі вони обромлюються у основний тег **<dependencies>** тут зберігаються залежності**</dependencies>**, який групує усі необхідні залежності.

#### **<dependencies>**

1. Перша залежність, це spring-webmvc, вона потрібна для змоги розробляти веб-додатки, та комфортно створювати та працювати з трьома такими модулями(слоями), як model(модель, або шаблон об'єкту з яким працюємо), далі view(уявлення, тобто інтерфейс), та контроллер(який передає інформацію до view о моделі з можливим використанням методів які

приходять з модуля service), більш детально ці модулі будуть розглянуті та досліджені далі у роботі.

```
<!--      https://mvnrepository.com/artifact/org.springframework/spring-  
webmvc -->
```

```
<dependency>
```

```
<groupId>org.springframework</groupId>
```

```
<artifactId>spring-webmvc</artifactId>
```

```
<version>5.3.23</version>
```

```
</dependency>
```

2. Наступна друга залежність, це javax.servlet/jstl, яка дозволяє працювати з сервлетами та допомагає розробляти інтерфейс.

```
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
```

```
<dependency>
```

```
<groupId>javax.servlet</groupId>
```

```
<artifactId>jstl</artifactId>
```

```
<version>1.2</version>
```

```
</dependency>
```

3. Далі третя залежність, це org.hibernate/hibernate-core, ця залежність дозволяє працювати з таким фреймворком як Hibernate, а саме з його основним проектом Hibernate Core, також ця залежність надає певні анотації для більш якісної роботи з цим фреймворком. Де Hibernate-core дозволяє працювати з даними у об'єктній моделі замість реляційної, змінюючи стани об'єктів з реляційної моделі у об'єктно-орієнтовану. Тобто це ORM фреймворк, який у більшості своїй працює у зв'язці з JPA, де JPA є специфікацією для Hibernate-core, тим самим Hibernate-core для JPA є реалізацією.

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
```

```
<dependency>
```

```
<groupId>org.hibernate</groupId>
```

```
<artifactId>hibernate-core</artifactId>
```

```
<version>5.6.10.Final</version>
```

**</dependency>**

4. Наступна залежність, четверта у досліджуємому веб-додатку, це mysql-connector-java, ця залежність дозволяє приєднуватися до такої бази даних як MySQL.

**<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<version>8.0.31</version>**

**</dependency>**

5. Далі п'ята залежність, це com.mchange/c3p0, вона дозволяє працювати у view, тобто у html файлах з такими конструкціями як “forloop, foreach”, тобто з циклами різного рода, та взагалі надає широкий функціонал який дає змогу працювати з змінними, тобто використовується ця залежність у файлах view.

**<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->**

**<dependency>**

**<groupId>com.mchange</groupId>**

**<artifactId>c3p0</artifactId>**

**<version>0.9.5.5</version>**

**</dependency>**

6. Наступна, шоста залежність це org.springframework/spring-orm, яка надає функціонал фреймворку Spring, дозволяє працювати з основними анотаціями цього фреймворку, тобто накриває веб-додаток своєрідним каркасом додаткового корисного функціоналу.

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->**

**>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-orm</artifactId>**

**<version>5.3.23</version>**



**</dependency>**

7. Остання залежність яка використовується у розробленому веб-додатку, це org.aspectj/aspectjweaver, вона дозволяє розробляти аспекти, тобто надає можливість створювати сквозну логіку, яка працює паралельно основній. Аспекти по більшості додаються у додатки для більш комфортної роботи розробнику, тобто цей функціонал виконує багато “boilerplate” роботи, яку потрібно би було робити розробнику, це усі які перевірки, додаткова інформація та інше.

**<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->**

**<dependency>**

**<groupId>org.aspectj</groupId>**

**<artifactId>aspectjweaver</artifactId>**

**<version>1.9.9.1</version>**

**</dependency>**

**</dependencies>**

### 2.1.3. Конфігурація веб-додатка

Після вирішення того які dependencies(залежності) будуть використовуватися при розробці веб-додатку, треба написати певну конфігурацію, яка би відповідала тим інструментам якими планується користуватися при розробці додатка. Тобто, **конфігурація** – це певна сутність, яка вміщає у собі перелік того, з якими об’єктами доведеться працювати. З програмної точки зору конфігурація, це Java клас(додаток розробляється за допомогою мови програмування Java) у якому прописан певний код. По-перше указуються пакети, які повинні скануватися, тобто це дає змогу функціоналу усіх залежностей працювати у будьякому пакеті, і при цьому не має бути помилок при використанні певних бібліотек, анотацій и т.п. По-друге, у конфігурації потрібно створити біни(об’єкти), тих залежностей які були вказані у pom.xml файлі, для того щоб була можливість працювати з цими об’єктами після імпорту бібліотек, певно законфігурованих, тобто з тим

набором характеристик, які потрібні у поточному додатку розробнику. У даному веб-додатку у конфігурації створюються такі біни як – InternalResourceViewResolver, dataSource, sessionFactory, transactionManager.

Де InternalResourceViewResolver – указує у якому пакеті містяться файли інтерфесу, та якого типу вони. Виглядає конфігурація цього біна таким чином:

```
<bean  
class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

Наступний бін dataSource потрібен для того щоб налагодити зв'язок з базою даних, у цьому біні указується уся потрібна інформація(url, driverClass, username, password) для успішного з'єднання з БД. Конфігурація цього біна виглядає так:

```
<bean id="dataSource"  
class="com.mchange.v2.c3p0.ComboPooledDataSource"  
    destroy-method="close">  
    <property name="driverClass" value="com.mysql.cj.jdbc.Driver" />  
    <property name="jdbcUrl"  
value="jdbc:mysql://localhost:3306/my_db?useSSL=false&serverTimezone=  
UTC" />  
    <property name="user" value="root" />  
    <property name="password" value="root" />  
</bean>
```

Далі третій створений бін потрібний для роботи з базою даних, а саме для створення сесій, за допомогою яких виникає можливість створювати, та дизайнити певні необхідні транзакції, називається цей бін sessionFactory, та виглядає у кодї наступним чином:

```
<bean id="sessionFactory"  
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
```

```

        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan"
value="com.alexandrsinko.spring.mvc_hibernate_aop.entity" />
        <property name="hibernateProperties">
            <props>
                <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

```

Та останій бін який конфігурується, це transactionManager, який використовує sessionFactory, та потрібен для менеджменту роботи з транзакціями, виглядає код цього біна, таким чином:

```

    <bean id="transactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

```

#### 2.1.4. Огляд “cross-cutting” логіки розробленої для додатка

У додатку використовується скрізна (cross-cutting) логіка, тобто, реалізован певний функціонал, який виконується паралельно основному функціоналу, але не є якоюсь головною частиною логіки додатку. Для написання такої логіки використовується SpringAOP, який дозволяє відокремлювати “кроскатинг” логіку, від бізнес логіки, та описувати її у декларативному стилі у вигляді аспектів.

Виглядає реалізація аспектно-орієнтованого програмування таким чином:

```
@Component
```

```

@Aspect

public class LoggingAspect {

    @Around("execution(*
com.alexandrsinko.spring.mvc_hibernate_aop.dao.*.*(..))")
    public Object aroundAllRepositoryMethodsAdvice(
        ProceedingJoinPoint proceedingJoinPoint
    ) throws Throwable{

        MethodSignature methodSignature =
            (MethodSignature) proceedingJoinPoint.getSignature();
        String methodName = methodSignature.getName();
        System.out.println("Begin of " + methodName);
        Object targetMethodResult = proceedingJoinPoint.proceed();
        System.out.println("End of " + methodName);
        return targetMethodResult;
    }
}

```

Цей код реалізований у додатку виконує роботу паралельно основній, а саме при виконанні якогось методу, наприклад отримання співробітників, тобто виконання методу `getEmployees()`, у консоль буде виведено інформацію проте який метод почав свою роботу, та коли метод завершує роботу, тоді у консоль буде виведено інформацію про те що метод завершив роботу, такий функціональ у своїй більшості потрібен розробникам для того щоб краще розуміти архітектуру розробляемого додатка, та відстежувати шаг за шагом який блок або модуль, або метод – як наприклад у даному додатку, працює.

### **2.1.5. Модель бази даних веб-додатка**

Модель це певний контейнер для зберігання даних, знаходячись у контролері додатку, є можливість додавати дані у модель, а потім використовувати ці дані у інтерфейсі додатку. Модель у додатку представлення у вигляді класу який описує співробітника, тобто його характеристики, та точки доступу до цих характеристик та полей, за

допомогою певних функцій, таких як конструктор, та геттери і сеттери. Модель у більшості своїй не зберігає у собі інформацію, або якісь дані, вона лише містить поля які зв'язані з таблицею у базі даних, де база даних як раз і зберігає дані об моделі у собі, завдяки функціоналу Hibernate-core представленному у вигляді анотацій. Тобто модель це певний шаблон об'єкту, де об'єкт повинен мати певні характеристики які повинстю визначені моделю(Java-класом). Виглядає модель у вигляді спрощеної діаграми так:

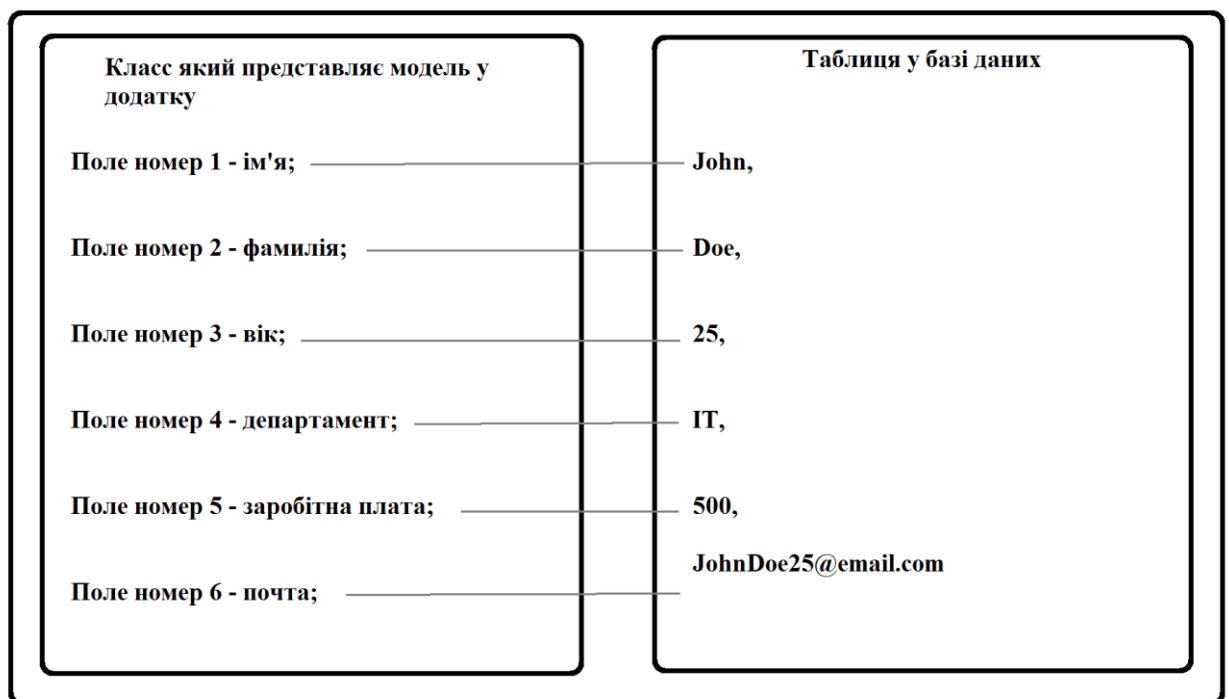


Рис. 2.1. Співвідношення моделі та таблиці у базі даних

Код моделі, а саме клас, у розробленому додатку знаходиться у пакеті entity, клас має назву Employee, має потрібні анотації для зв'язку з базою даних(@Entity, @Table(name="employees")), також поля такі як id, name, surname, department, salary, та геттери і сеттери для доступу до цих полів. У БД цей клас зі своїми полями представляється у якості певного SQL запиту, та виглядає наступним чином:

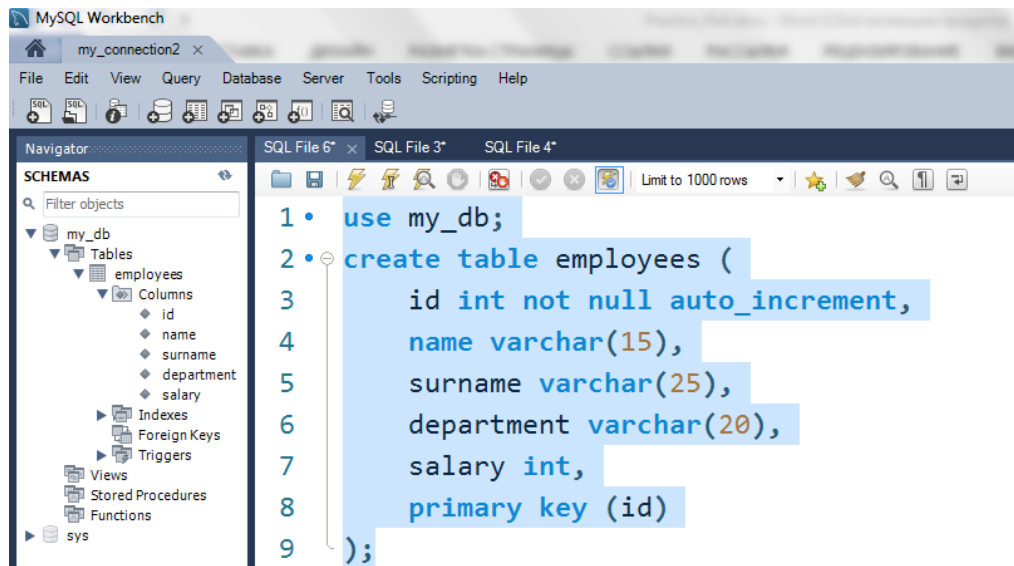


Рис. 2.2. Створення та визначення моделі у БД(MySQL)

Після створення маєця змога побачити за допомогою ще одно SQL запиту те як дістати дані із таблиці, для того щоб побачити що в ній зберігається, виглядає це таким чином:

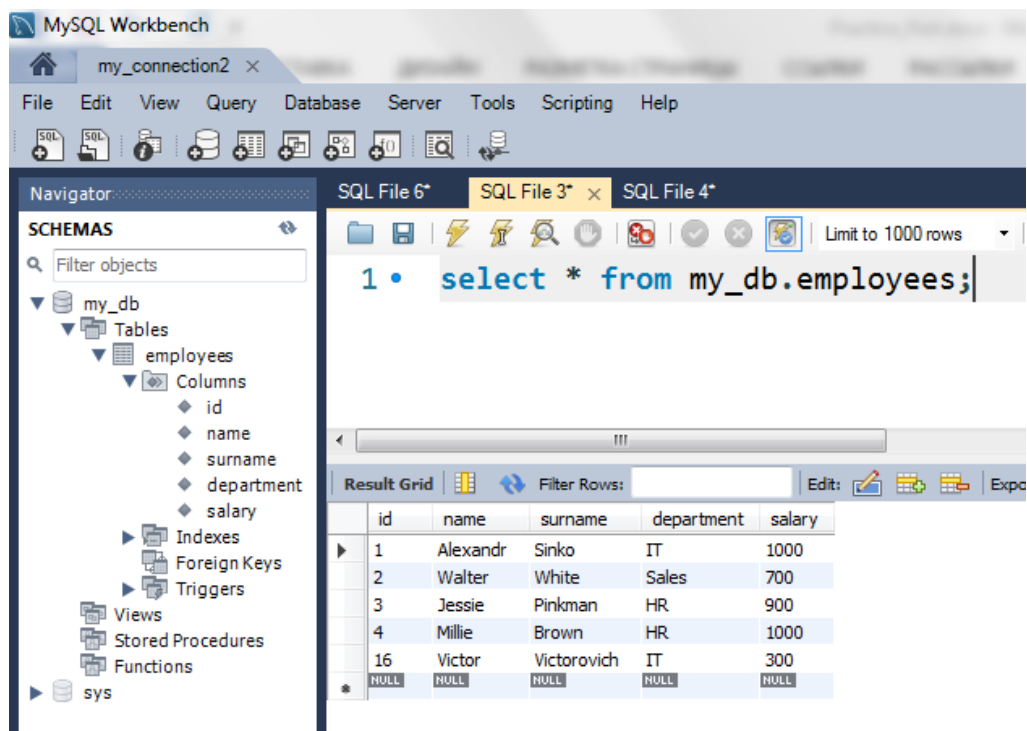


Рис 2.3. Дані які зберігаються у БД

## 2.2. Реалізація технології JAVA PERSISTENCE API у досліджуємому додатку

### 2.2.1. Розробка модулю Entity

Представником моделі у досліджуємому додатку є сутність, або з програмної точки зору клас, який прийнято називати Entity, при цьому використовуючи Hibernate-core фреймворк. Entity, це сутність яка має певний стан та поведінку, залежну від цього стану, а також правила для взаємодії з даною сутністю.

З програмної точки зору Entity, це набір певних характеристик, таких як поля, члени цього класу, та методи для роботи з цією сутністю, але у досліджуємому веб-додатку, методи для роботи з сутністю виносяться у інший слой(клас, або набір класів), та реалізуються там, робити так потрібно для розробки більш якісних додатків, тобто дотримуючись певних патернів у розробці, наприклад патерну SOLID, який говорить саме про ці речі.

Для того щоб була змога називати клас, який є представником моделі, як Entity, потрібно використовувати функціонал який надає Hibernate-core фреймворк, а саме певний набір анотацій, по перше це потрібно використовувати анотацію “@Entity” над класом, який планується бути моделю, та анотацію “@Id” над полем Id, тобто над ідентифікатором. Також використовуються інші анотації, наприклад такі як “@Table(name=”Employee”)", “@Column(name=”userName”)", і т.д, але це вже більш другорядні анотації, тобто без використання цих анотацій клас який є представником моделі все одно можна називати Entity, за рахунок таких головних анотацій як “@Entity” та “Id”. Далі більш конкретно проведемо огляд Entity у розробленому веб-додатку.

1. По-перше потрібно додати певні потрібні бібліотеки, для використання анотацій, основні Entity та Id, а далі теж важливі, але більш другорядні, це Table, Column, GeneratedValue, GenerationType.

```
import javax.persistence.Entity;
```

```
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
```

2. Далі над класом Employee тобто над Entity додатка, вказуються анотації @Entity, Table(name="employees").

```
@Entity
@Table(name = "employees")
public class Employee {
```

3. Наступним кроком створюється поле id над яким вказується анотація @Id та анотація @Column(name="id") яка вказує що поле id є у таблиці, та вказується анотація з параметром @GeneratedValue(strategy=GenerationType.IDENTITY), де сама анотація вказує на те що поле є генерованим, а параметр анотації вказує на те за яким правилом поле повинно генеруватися, у даному веб-додатку, id буде просто інкрементуватися. Далі створюються інші поля, прототипи яких є у таблиці, над полями вказується анотація @Column та ім'я поля у таблиці.

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
@Column(name = "name")
private String name;
@Column(name = "surname")
private String surname;
@Column(name = "department")
private String department;
@Column(name = "salary")
private int salary;
```



4. Створюються необхідні методи класу для доступу до полів Entity, і власне до даних цих полів, це конструктор, та геттери і сеттери, а також метод toString(), якщо буде необхідність скористуватися цим методом.

```
public Employee() {  
  
}  
  
public Employee(String name, String surname, String department, int  
salary) {  
    this.name = name;  
    this.surname = surname;  
    this.department = department;  
    this.salary = salary;  
}  
public int getId() {  
    return id;  
}  
public void setId(int id) {  
    this.id = id;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public String getSurname() {  
    return surname;  
}  
public void setSurname(String surname) {  
    this.surname = surname;  
}
```

```

    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee{" +
            "id=" + id +
            ", name=" + name + "\" +
            ", surname=" + surname + "\" +
            ", department=" + department + "\" +
            ", salary=" + salary +
            "'}";
    }
}

```

Далі Entity(Employee) буде використовуватися у інших шарах додатку, таких як DAO, Service, Controller, та навіть View.

### 2.2.2. Розробка модулю DAO

Модуль DAO є найбільш складним, у всьому додатку, по суті своїй це і є “бек-енд” веб-додатка, якщо виходити з того поняття що “бек-енд” це

обробка запитів які приходять з фронту, далі робота з базою даних якщо в запиті потрібно скористуватись БД, після чого це обробка даних які приходять з бази даних, та повертання даних, у вигляді певної відповіді назад до фронту додатка. DAO є акронімом, та розшифровується як Data Access Object, тобто об'єкт доступу до даних, і саме DAO є першим шаром логіки роботи з базою даних, і основна робота з даними які приходять з БД, або посилаються до неї, проходять у DAO.

Цілю магістерської роботи є можливість дослідити роботу з базою даних, у даному веб-додатку, для взаємодії з БД використовується ORM фреймворк, а саме Hibernate-Core, який також є реалізацією JPA, яка є специфікацією до цього фреймворку, звісно JPA теж використовується. А також у процесі дослідження Hibernate-Core фреймворку, стало зрозуміло що він “під капотом” використовує JDBC, тобто висновком з цього твердження є те що у даному досліджуємому веб-додатку використовується Hibernate-Core, JPA та JDBC який виконує свою роботу поза лаштунками основної логіки роботи Hibernate-Core.

Досліджуючи роботу з БД було розроблено додаток, де приклади взаємодії з БД буде досліджено у пунктах які описують код, нижче. Основний шар логіки роботи з БД – DAO, у додатку, складається з двох типів класів, а саме з публічного класу і інтерфейсу, де інтерфейс є певним набіром методів, а клас який реалізує цей інтерфейс є реалізацією цього інтерфейсу.

1. Перший приклад коду, це код інтерфейса, у якому є два імпорти, по-перше це імпорт Entity з якою звісно доводиться працювати, та по-друге це імпорт списку, який використовується щоб зберігати певну кількість об'єктів типу Employee, тобто типу Entity.

```
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;  
import java.util.List;
```

2. Далі приклад імплементації абстрактних методів, тобто не реалізованих у інтерфейсі, це getAllEmployees() який після реалізації повинен надавати змогу отримати список об'єктів типу Employee, тобто співробітників,

далі метод `saveEmployee` який повинен надавати змогу зберігати співробітника, наступний метод це `getEmployee`, завдяки якому повина бути можливість отримати одного конкретного співробітника по його ідентифікатору, та останній метод це `deleteEmployee` який надає можливість видалити певного співробітника по його `id`.

```
public interface EmployeeDAO {  
    List<Employee> getAllEmployees();  
    void saveEmployee(Employee employee);  
    Employee getEmployee(int id);  
    void deleteEmployee(int id);  
}
```

Наступний приклад коду, це реалізація інтерфейсу, яка розміщена у публічному класі.

1. По-перше імпорти усіх необхідних сутностей, які далі будуть використовуватися для реалізації логіки роботи DAO. Перший імпорт це клас `Employee`, тобто модель, дані якої будуть використовуватися у методах які реалізуються у методах, у роботі з БД. Далі імпортація класів `Session` та `SessionFactory` для отримання сесій, а далі роботи з транзакціями, і імпорт класу `Query` для написання певних запитів у БД. Також імплементация `Autowired` для впровадження залежності, а саме біну `SessionFactory`, для роботи з одноіменним класом. Та останній імпорт `Repository` для використання анотації `@Repository` – яка указує спрінгу що цей шар логіки потрібно сканувати, та визначати його як шар для роботи з БД.

```
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.query.Query;  
import org.springframework.beans.factory.annotation.Autowired;  
import java.util.List;  
import org.springframework.stereotype.Repository;
```

@Repository

```
public class EmployeeDAOImpl implements EmployeeDAO {  
    private SessionFactory sessionFactory;  
    @Autowired  
    public EmployeeDAOImpl(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
}
```

2. По-друге реалізація методів оголошених у інтерфейсі. Для того щоб написати певний функціонал методу який повинен якось взаємодіяти з БД, перш за все це потрібно створити певну локальну сесію, потім реалізувати логіку метода, і потім важливо закрити сесію.

Відкривається сесія за допомогою певної стрічки коду, такої як – `EntityManager entityManager = emf.createEntityManager();`. Та завершити транакцію можливо використовуючи таку стрічку коду як – `entityManager.getTransaction().commit();`. Але саме цих стрічок коду не має у цьому шарі, він виконується у іншому шарі логіки називаємому як `service`, який буде досліджено далі у роботі.

```
entityManager.getTransaction().begin();
```

@Override

```
public List<Employee> getAllEmployees() {  
    Session session = sessionFactory.getCurrentSession();  
    List<Employee> allEmployees = session  
        .createQuery("from Employee", Employee.class)  
        .getResultList();  
    return allEmployees;  
}
```

@Override

```
public void saveEmployee(Employee employee) {  
    Session session = sessionFactory.getCurrentSession();  
    session.saveOrUpdate(employee);  
}
```

```

    }
    @Override
    public Employee getEmployee(int id) {
        Session session = sessionFactory.getCurrentSession();
        Employee employee = session.get(Employee.class, id);
        return employee;
    }
    @Override
    public void deleteEmployee(int id) {
        Session session = sessionFactory.getCurrentSession();
        Query<Employee> query = session.createQuery("delete from Employee
" +
        "where id =:employeeId");
        query.setParameter("employeeId", id);
        query.executeUpdate();
    }
}

```

### 2.2.3. Розробка модулю Service

Модуль Service це наступний шар логіки у розробленому додатку, який вже не працює з базою даних на пряму, але використовує методи які приходять з DAO, тобто у більшості своїй у сервісі знаходиться так звана бізнес логіка, яка доповнює та розширює функціонал бекенду, за допомогою використання методів DAO.

Але також модуль Service може бути якось зв'язаним, скоріше не з самою базою даних, а функціоналом який виконується у DAO і це не рахується за помилку, якщо це вносить позитивні зміни, у даному розробленому додатку, який використовується у якості прикладу, модуль сервіс зв'язан невеликою долею функціоналу, а саме він використовує не тільки методи DAO, але ще й завдяки анотації `@Transactional` починає роботу транзакції за допомогою команди `entityManager.getTransaction().begin()`, та припиняє роботу за

допомогою команди `entityManager.getTransaction().commit();` і робе це все автоматично, тобто не потрібно прописувати цей код, а потрібно лише використати анотацію зазначену вище над методами сервісу які використовують методи із DAO. Код у сервісі виглядає таким чином:

1. Код інтерфейсу:

```
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;
import java.util.List;

public interface EmployeeService {

    List<Employee> getAllEmployees();

    void saveEmployee(Employee employee);

    Employee getEmployee(int id);

    void deleteEmployee(int id);

}
```

2. Реалізація методів інтерфейсу, де використовуються методи які приходять з DAO, тому у кодї є впровадження залежності біну DAO, та сама анотація `@Transactional`, яка починає та завершує роботу транзакції.

```
import com.alexandrsinko.spring.mvc_hibernate_aop.dao.EmployeeDAO;
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDAO employeeDAO;

    @Autowired

    public EmployeeServiceImpl(EmployeeDAO employeeDAO) {

        this.employeeDAO = employeeDAO;

    }

    @Override
```

```

@Transactional
public List<Employee> getAllEmployees() {
    return employeeDAO.getAllEmployees();
}

@Override
@Transactional
public void saveEmployee(Employee employee) {
    employeeDAO.saveEmployee(employee);
}

@Override
@Transactional
public Employee getEmployee(int id) {
    return employeeDAO.getEmployee(id);
}

@Override
@Transactional
public void deleteEmployee(int id) {
    employeeDAO.deleteEmployee(id);
}
}

```

## 2.3. Реалізація інших необхідних модулів

### 2.3.1. Розробка модулю Contrller

Після реалізації модулів працюючих з базою даних, потрібно розробити модуль, який має приймати запити з “фронту”, обробляти їх, та надсилати відповідь назад до view. Тобто стає необхідність створити певний шар логіки який буде спілкуватися з нижніми(бекенд), та верхніми(фронтенд) шарами логіки, та контролювати взаємодію між ними, таким модулем є контроллер, у



розробленому та зараз досліджуємому додатку є створений спеціальний клас який називається Controller, і він відповідає саме за вище перелічену роботу.

Реалізація класу Controller виглядає таким чином:

1. Необхідні імпорти, які потрібні контролеру для обробки запитів, та реалізації самого контролера.

```
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;
import
com.alexandrsinko.spring.mvc_hibernate_aop.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import java.util.List;
@Controller
public class EmployeeController {
```

2. Реалізація dependency injection, а саме впровадження залежності біну сервіса.

```
private EmployeeService employeeService;
@Autowired
public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
}
```

3. Реалізація методів контролеру.

```
@RequestMapping(value = "/")
public String showAllEmployees(Model model) {
    List<Employee> allEmployees = employeeService.getAllEmployees();
    model.addAttribute("allEmps", allEmployees);
}
```

```

        return "all-employees";
    }
    @RequestMapping(value = "/addNewEmployee")
    public String addNewEmployee(Model model) {
        Employee employee = new Employee();
        model.addAttribute("employee", employee);
        return "employee-info";
    }
    @RequestMapping(value = "/saveEmployee")
    public String saveEmployee(@ModelAttribute("employee") Employee
employee) {
        employeeService.saveEmployee(employee);
        return "redirect:/";
    }
    @RequestMapping(value = "/updateInfo")
    public String updateEmployee(@RequestParam("empId") int id, Model
model){
        Employee employee = employeeService.getEmployee(id);
        model.addAttribute("employee", employee);
        return "employee-info";
    }
    @RequestMapping(value = "/deleteEmployee")
    public String deleteEmployee(@RequestParam("empId") int id) {
        employeeService.deleteEmployee(id);
        return "redirect:/";
    }
}

```

### 2.3.2. Розробка модулю View

Модуль View є складовою більшою фронтенду, тобто користувацького інтерфейсу, і не є цілю дослідження, але роздивитися його потрібно, заради розуміння загальної картини роботи веб-додатка цілком.

Реалізація View не є пріоритетною, тому цей модуль не має обширної реалізації, а лише певний потрібний функціонал, для коректної роботи додатка.

Складається реалізація View з двох файлів, де перший виводить на екран перелік співробітників з необхідними баттонами для маніпуляції даними, та другий файл надає змогу бачити інтерфейс певного співробітника, та маніпулювати даними цього конкретного співробітника. Виглядає реалізація View(користувацького інтерфейсу) таким чином:

1. View(користувацький інтерфейс) першого файлу.

```
<% @ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<body>
<h2>All Employees</h2>
<br>
<table>
  <tr>
    <th>Name</th>
    <th>Surname</th>
    <th>Department</th>
    <th>Salary</th>
    <th>Operations</th>
  </tr>
  <c:forEach var = "emp" items="${allEmps}">
    <c:url var="updateButton" value="/updateInfo">
      <c:param name="empId" value="${emp.id}"/>
```

```

</c:url>
<c:url var="deleteButton" value="/deleteEmployee">
  <c:param name="empId" value="${emp.id}"/>
</c:url>
<tr>
  <td>${emp.name}</td>
  <td>${emp.surname}</td>
  <td>${emp.department}</td>
  <td>${emp.salary}</td>
  <td>
    <input type="button" value="Update"
      onclick="window.location.href='${updateButton}'"/>
    <input type="button" value="Delete"
      onclick="window.location.href='${deleteButton}'"/>
  </td>
</tr>
</c:forEach>
</table>
<br>
<input type="button" value="Add"
  onclick="window.location.href='addNewEmployee'"/>
</body>
</html>

```

2. View(користувачський інтерфейс) другого файлу.

```

<% @ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<!DOCTYPE html>
<html>
<body>
<h2>

```

```
Employee Info
</h2>
<br>
<form:form action="saveEmployee" modelAttribute="employee">
    <form:hidden path="id"/>
    Name:<form:input path="name"/>
    <br><br>
    Surname:<form:input path="surname"/>
    <br><br>
    Department:<form:input path="department"/>
    <br><br>
    Salary:<form:input path="salary"/>
    <br><br>
    <input type="submit" value="Submit">
</form:form>
</body>
</html>
```

## 2.4. Аналіз отриманих результатів

Розроблений додаток має не складну архітектуру, але використовує сучасні методи взаємодії з базою даних, веб-додаток має шість основних модулів, починаючи з найнижчого, тобто взаємодії з БД та завершуючи верхнім рівнем, а саме представленням користувацького інтерфейсу користувачу, тобто у додатку використовуються саме такі модулі як – Entity, DAO, Service, Controller, View, та більш другорядний модуль Aspect(для роботи з аспектами), також при розробці використовувалися такі паттерни та поняття як - MVC, OOP, SOLID, ACID, lazy loading, dirty cheking, “транзакція”, “рівні ізольованості транзакцій”. Робота додатка проходить

таким чином – користувач використовуючи наданий йому функціонал посилає певний запит де цей запит приймає контроллер та реагує на нього, а саме звертається до сервісу, а сервіс у свою чергу звертається до DAO і DAO виконує свою роботу, а саме посилає певний запит у базу даних(MySQL), отримує інформацію, обробляє її, та передає назад до сервісу у вигляді певного метода, і сервіс обробляє дані додаючи додатковий функціонал і передає назад у контроллер, де контроллер викликає певну сторінку користувацького інтерфейсу, та передає дані цій сторінці і в результаті чого користувачу приходить нова сторінка, з тою логікою яку він очікував.

Працює додаток певним чином:

1. Запускається БД(MySQL), де нижче на скріншоті видно, що у базі зберігаються певні поля.

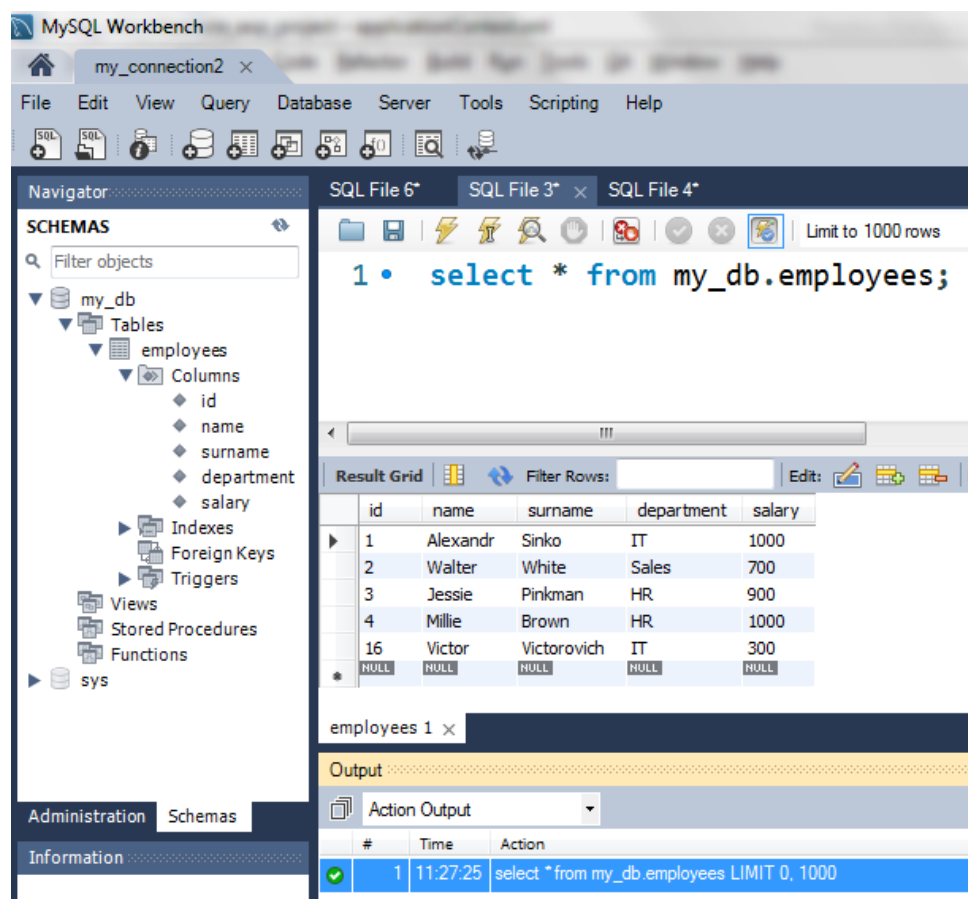


Рис.2.4. Таблиця у запущеної БД

2. Далі запускається сервер, у даному додатку сервер запускається за допомогою чистого Tomcat 8.5., тобто без використання SpringBoot. На скріншоті можна побачити, що додаток успішно запусився та функціонує.

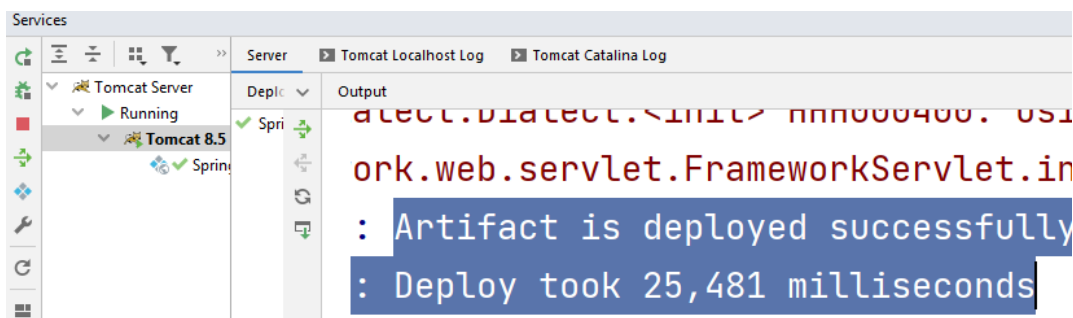


Рис.2.5. Запуск сервера

3. У результаті запуску сервера, та його сумісної роботи з БД, на екрані можна побачити користувацький інтерфейс, а саме сторінку зі списком співробітників з їх параметрами, та кнопки з функціоналом по додаванню нового співробітника, зміни інформації об співробітнику, та видалення співробітника.

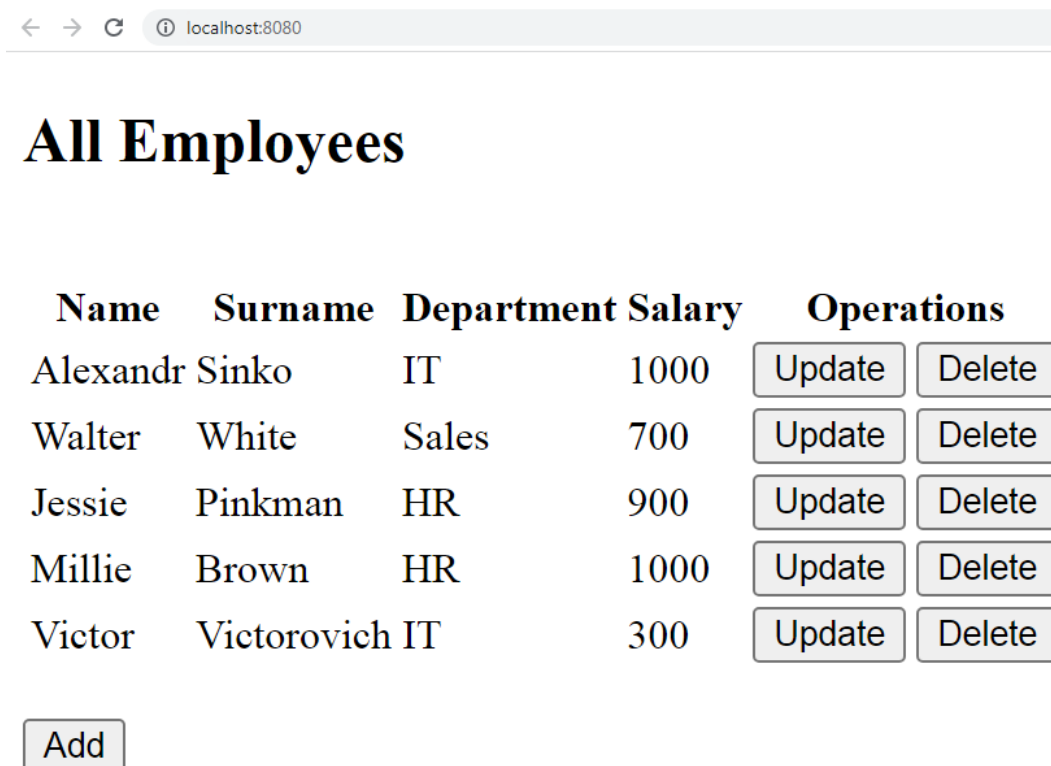
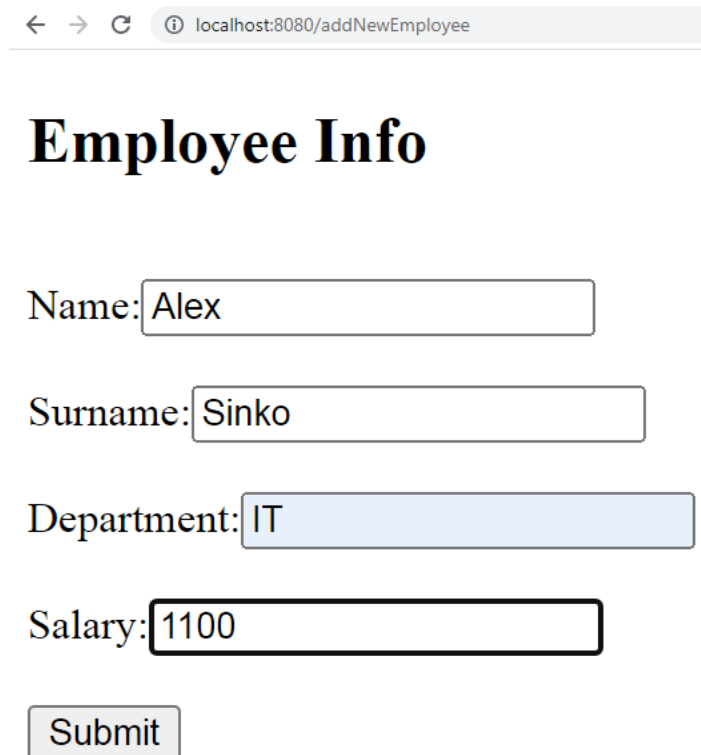


Рис.2.6. Головна сторінка веб-додатка (список співробітників)

4. Також окрім сторінки зі списком співробітників, їх характеристиками та функціями, є можливість додати нового співробітника, для цього потрібно натиснути на кнопку “add”, після чого відкриється нова сторіна, де буде можливість вказати параметри(ім’я, прізвище, департамент, заробітна плата) нового співробітника. Після додавання співробітника можна також побачити що цей співробітник з вказаними параметрами був доданий до списку на головній сторінці, а також при перевірці у БД є можливість побачити що дійсно такий співробітник був доданий у список.



← → ↻ ⓘ localhost:8080/addNewEmployee

## Employee Info

Name:

Surname:

Department:

Salary:

Рис.2.7. Сторінка додавання нового співробітника

Таким чином, проведений аналіз отриманих результатів доводить, що під час дослідження було вивконанно всі поставлені завадння й розроблено веб-додаток, що реалізує JPA у роботі з базами даних.



## Висновок до розділу 2

У результаті дослідження був розроблений веб-додаток основною задачею та цілю якого було продемонструвати взаємодію та роботу з базою даних, певними способами. Для розробки та змоги дослідити додаток та технології які використовувалися, було обрано розробити веб-додаток з використанням JDBC, Hibernate-Core з JPA, SpringMVC, Spring-Core, Tomcat, та базу даних MySQL(це основні технології які використовувалися).

У результаті дослідження додатку можна сказати що, розроблений додаток є працюючим, з використанням тих технологій які планувалося дослідити.

Розроблений додаток відповідає вимогам таких основних патернів якості розробки, як OOP, SOLID, ACID, IoC, DI, dirty cheking, lazy loading. Ідеєю розробленого додатка є демонстрування того, яким чином відбувається взаємодія бази даних, з “бекенд” частиною додатка, при цьому не залежачи від того яка база даних використовується, чи то MySQL, чи PostgreSQL та інші реляційни БД. Після розробки додатку, було проаналізовано його роботу, та вручну протестовано, де у результаті аналізу та тестування додатка, не виникло проблем з його використанням, в результаті чого стало зрозуміло що додаток відповідає і технічним характеристикам, і ідеї або цілі самої роботи.

## ВИСНОВКИ

В рамках випускної кваліфікаційної роботи було проведено дослідження підходів реалізації JPA в Java веб-додатках, а саме проаналізовано науково-технічний матеріал, аналіз якого дозволив розробити веб-сервіс, та реалізувати шлях взаємодії з БД на принципах persistence api на платформі Java. При цьому було отримано наступні висновки та результати:

У першому розділі даної роботи був проведений огляд основних компонентів, технологій, інструментів, патернів та шляхів роботи з БД, архітектури веб-додатків. Розглянуто основні можливості фреймворків, зокрема фреймворку Hibernate, що реалізує JPA, досліджено теми шляхів взаємодії з БД, та побудови архітектури самого додатка.

Проведені дослідження надали можливість визначити, що веб-додаток буде мати не складну, але показову архітектуру, з основними CRUD методами, використовуючи Spring, JDBC, Hibernate-Core з JPA, SpringMVC, Tomcat, та базу даних MySQL, патерни DAO, MVC.

В другому розділі роботи було розроблено веб-сервіс, а саме: розроблен функціонал взаємодії з БД, реалізовані основні методи, а саме CRUD, реалізован простий UI.

Під час розробки програми особлива увага приділяється реалізації саме JPA, стандартизованому інтерфейсу для Java ORM фреймворків, що надає можливість зберігати у зручному вигляді Java-об'єкти у базі даних. Вибір певних інструментів та технологій, таких як ORM-фреймворк Hibernate, фреймворк Spring, дозволяє успішно реалізовувати обрані концепції.

В результаті, розроблений веб-сервіс може служити як певний довідник по співробітниках, або як приклад того як повинна виглядати основа архітектури при побудові та розробці подібних додатків. Також програма може використовуватися у навчальному контексті, програма може доопрацьовуватися, додаючи новий функціонал з можливістю рефакторінга архітектури.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. arrogante-it : веб-сайт. URL: <https://github.com/arrogante-it>
2. Bauer C., King G. Java Persistence with Hibernate. - 2006. -880 p.
3. BetaCode. Java Basic : веб-сайт. URL: <https://betacode.net/10973/java-basic> (дата звернення 22.12.2020 р. )
4. Building Modern Business Applications: Reactive Cloud Architecture for Java, Spring, and PostgreSQL.
5. GitHub : веб-сайт. URL: <https://github.com/enhorse/java-interview/blob/master/README.md#java-collections> (дата звернення 20.12.2020 р. )
6. IntelliJ Idea. Навчальні відео : веб-сайт. URL: <https://www.jetbrains.com/ru-ru/idea/resources/> (дата звернення 20.12.2020 р.)
7. Java Code Conventions. URL: <https://www.oracle.com/technetwork/java/codeconventions-150003.pdf> (дата звернення 06.01.2021 р. )
8. JavaStudy. Spring MVC : веб-сайт. URL: <https://javastudy.ru/spring-mvc/spring-mvc-basic/> (дата звернення 12.03.2021 р. )
9. Ottinger J. Linwood J., Minter D. Beginning Hibernate. URL: [https://www.oreilly.com/library/view/beginning-hibernate-for/9781484223192/A321250\\_4\\_En\\_3\\_Chapter.html](https://www.oreilly.com/library/view/beginning-hibernate-for/9781484223192/A321250_4_En_3_Chapter.html) (дата звернення 05.12.2020 р. ).
10. Pillana B. Performance Analysis of Java Persistence API Providers. DOI:10.33107/ubt-ic.2018.101
11. Wikipedia. James Goslong : веб-сайт. URL: <https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D1%81%D0%BB%D0%B8%D0%BD%D0%B3,%D0%94%D0%B6%D0%B5%D0%B9%D0%BC%D1%81> (дата звернення 20.12.2020 р. )
12. Архітектура клієнт-сервер. Сховища даних. Реляційні бази даних як основа сховищ даних. URL: <http://educational.mariroz>

- com/InformTechVInfrastrRynku/lect/lect8.pdf (дата звернення 05.12.2020 р. ).
- 13.Блінов, І.М., Романчик, В. С. Java. Методи програмування: навчально-методичний посібник. Минск : издательство «Четыре четверти», 2013. 896с.
- 14.Блох Д. Java: ефективне програмування, 3-е изд. М.: Лори, 2019. 463с.
- 15.Гамма, Р.Хелм, Р.Джонсон, Дж. Вліссідес, Прийоми об'єктно-орієнтованого проектування. Патерни проектування. С.-Пб: Питер, 2001. 368с.
- 16.Герберт Шілдрт, Java 8. Повне керівництво, 9-е видання. М.: Вільямс. 2015. 1376с.
- 17.Історія мов програмування. Що допомогло Java увійти в кожен будинок : веб-сайт. URL: <https://habr.com/ru/post/306476/> (дата звернення 25.12.2020 р.)
- 18.Мартін Р. Чистий код: створення, аналіз і рефакторинг. К.: Фабула, 2019. 416 с.
- 19.Ноутон П., Шилдрт Г., Java 2 Наиболее полное руководство. 1102с.
- 20.Офіційний сайт IntelliJ IDEA. URL: <https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html> (дата звернення 10.12.2020 р. )
- 21.Офіційний сайт Java Platform, Standard Edition 7 API Specification : веб-сайт. URL: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html> (дата звернення 20.12.2020 р. )
- 22.Програмування на мові Java : веб-сайт. URL: <http://shtanyuk.tk/edu/nniit/java-new/html/11.html> (дата звернення 20.01.2021 р.)

## ДОДАТОК А

### Вихідний код файлу з залежностями pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.alexandrsinko.spring.mvc</groupId>

  <artifactId>spring_mvc_hibernate_aop</artifactId>

  <version>1.0-SNAPSHOT</version>

  <packaging>war</packaging>

  <name>spring_mvc_hibernate_aop Maven Webapp</name>

  <!-- FIXME change it to the project's website -->

  <url>http://www.example.com</url>

  <properties>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <maven.compiler.source>1.8</maven.compiler.source>

    <maven.compiler.target>1.8</maven.compiler.target>

  </properties>

  <dependencies>

    <dependency>
```

```

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->

<dependency>

  <groupId>org.springframework</groupId>

  <artifactId>spring-webmvc</artifactId>

  <version>5.3.23</version>

</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->

<dependency>

  <groupId>javax.servlet</groupId>

  <artifactId>jstl</artifactId>

  <version>1.2</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->

<dependency>

  <groupId>org.hibernate</groupId>

  <artifactId>hibernate-core</artifactId>

  <version>5.6.10.Final</version>

```

```
</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

    <groupId>mysql</groupId>

    <artifactId>mysql-connector-java</artifactId>

    <version>8.0.31</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->

<dependency>

    <groupId>com.mchange</groupId>

    <artifactId>c3p0</artifactId>

    <version>0.9.5.5</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->

<dependency>

    <groupId>org.springframework</groupId>

    <artifactId>spring-orm</artifactId>

    <version>5.3.23</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->

<dependency>

    <groupId>org.aspectj</groupId>
```

```

    <artifactId>aspectjweaver</artifactId>

    <version>1.9.9.1</version>

</dependency>

</dependencies>

<build>

    <finalName>spring_mvc_hibernate_aop</finalName>

    <pluginManagement><!-- lock down plugins versions to avoid using Maven
defaults (may be moved to parent pom) -->

        <plugins>

            <plugin>

                <artifactId>maven-clean-plugin</artifactId>

                <version>3.1.0</version>

            </plugin>

            <!-- see http://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_war_packaging -->

            <plugin>

                <artifactId>maven-resources-plugin</artifactId>

                <version>3.0.2</version>

            </plugin>

            <plugin>

                <artifactId>maven-compiler-plugin</artifactId>

                <version>3.8.0</version>

            </plugin>

```



```
<plugin>

  <artifactId>maven-surefire-plugin</artifactId>

  <version>2.22.1</version>

</plugin>

<plugin>

  <artifactId>maven-war-plugin</artifactId>

  <version>3.2.2</version>

</plugin>

<plugin>

  <artifactId>maven-install-plugin</artifactId>

  <version>2.5.2</version>

</plugin>

<plugin>

  <artifactId>maven-deploy-plugin</artifactId>

  <version>2.8.2</version>

</plugin>

</plugins>

</pluginManagement>

</build>

</project>
```

**Вихідний код класу LoggingAspect.java:**

```
package com.alexandrsinko.spring.mvc_hibernate_aop.aspect;
```

```

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.reflect.MethodSignature;

import org.springframework.stereotype.Component;

@Component

@Aspect

public class LoggingAspect {

    @Around("execution(*
com.alexandrsinko.spring.mvc_hibernate_aop.dao.*.*(..))")

    public Object aroundAllRepositoryMethodsAdvice(

        ProceedingJoinPoint proceedingJoinPoint

    ) throws Throwable{

        MethodSignature methodSignature =

            (MethodSignature) proceedingJoinPoint.getSignature();

        String methodName = methodSignature.getName();

        System.out.println("Begin of " + methodName);

        Object targetMethodResult = proceedingJoinPoint.proceed();

        System.out.println("End of " + methodName);

        return targetMethodResult;

    }

}

```

**Вихідний код Entity, класу Employee.java:**

```
package com.alexandrsinko.spring.mvc_hibernate_aop.entity;

import javax.persistence.Entity;

import javax.persistence.Table;

import javax.persistence.Id;

import javax.persistence.Column;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

@Entity

@Table(name = "employees")

public class Employee {

    @Id

    @Column(name = "id")

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    @Column(name = "name")

    private String name;

    @Column(name = "surname")

    private String surname;

    @Column(name = "department")

    private String department;

    @Column(name = "salary")

    private int salary;
```

```
public Employee() {  
  
}  
  
public Employee(String name, String surname, String department, int salary) {  
  
    this.name = name;  
  
    this.surname = surname;  
  
    this.department = department;  
  
    this.salary = salary;  
  
}  
  
public int getId() {  
  
    return id;  
  
}  
  
public void setId(int id) {  
  
    this.id = id;  
  
}  
  
public String getName() {  
  
    return name;  
  
}  
  
public void setName(String name) {  
  
    this.name = name;  
  
}  
  
public String getSurname() {  
  
    return surname;  
  
}
```

```

    }

    public void setSurname(String surname) {

        this.surname = surname;

    }

    public String getDepartment() {

        return department;

    }

    public void setDepartment(String department) {

        this.department = department;

    }

    public int getSalary() {

        return salary;

    }

    public void setSalary(int salary) {

        this.salary = salary;

    }

    @Override

    public String toString() {

        return "Employee{" +

            "id=" + id +

            ", name=" + name + "\"" +

            ", surname=" + surname + "\"" +

```

```

        ", department='" + department + "\" +
        ", salary=" + salary +
        '}';
    }
}

```

### **Вихідний код DAO, інтерфейсу EmployeeDAO.java:**

```

package com.alexandrsinko.spring.mvc_hibernate_aop.dao;

import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;

import java.util.List;

public interface EmployeeDAO {

    List<Employee> getAllEmployees();

    void saveEmployee(Employee employee);

    Employee getEmployee(int id);

    void deleteEmployee(int id);

}

```

### **Вихідний код DAO, класу EmployeeDAOImpl:**

```

package com.alexandrsinko.spring.mvc_hibernate_aop.dao;

import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.query.Query;

import org.springframework.beans.factory.annotation.Autowired;

```

```

import java.util.List;

import org.springframework.stereotype.Repository;

@Repository

public class EmployeeDAOImpl implements EmployeeDAO {

    private SessionFactory sessionFactory;

    @Autowired

    public EmployeeDAOImpl(SessionFactory sessionFactory) {

        this.sessionFactory = sessionFactory;

    }

    @Override

    public List<Employee> getAllEmployees() {

        Session session = sessionFactory.getCurrentSession();

        List<Employee> allEmployees = session

            .createQuery("from Employee", Employee.class)

            .getResultList();

        return allEmployees;

    }

    @Override

    public void saveEmployee(Employee employee) {

        Session session = sessionFactory.getCurrentSession();

        session.saveOrUpdate(employee);

    }

```

@Override

```
public Employee getEmployee(int id) {  
  
    Session session = sessionFactory.getCurrentSession();  
  
    Employee employee = session.get(Employee.class, id);  
  
    return employee;  
  
}
```

@Override

```
public void deleteEmployee(int id) {  
  
    Session session = sessionFactory.getCurrentSession();  
  
    Query<Employee> query = session.createQuery("delete from Employee " +  
        "where id =:employeeId");  
  
    query.setParameter("employeeId", id);  
  
    query.executeUpdate();  
  
}  
  
}
```

### **Вихідний код Сервісу, інтерфейсу EmployeeService:**

```
package com.alexandrsinko.spring.mvc_hibernate_aop.service;  
  
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;  
  
import java.util.List;  
  
public interface EmployeeService {  
  
    List<Employee> getAllEmployees();  
  
    void saveEmployee(Employee employee);  
  
}
```



```
Employee getEmployee(int id);

void deleteEmployee(int id);

}
```

Вихідний код Сервісу, класу EmployeeServiceImpl:

```
package com.alexandrsinko.spring.mvc_hibernate_aop.service;

import com.alexandrsinko.spring.mvc_hibernate_aop.dao.EmployeeDAO;
import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service

public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDAO employeeDAO;

    @Autowired

    public EmployeeServiceImpl(EmployeeDAO employeeDAO) {

        this.employeeDAO = employeeDAO;
    }

    @Override

    @Transactional

    public List<Employee> getAllEmployees() {

        return employeeDAO.getAllEmployees();
    }
}
```

```

    }

    @Override

    @Transactional

    public void saveEmployee(Employee employee) {

        employeeDAO.saveEmployee(employee);

    }

    @Override

    @Transactional

    public Employee getEmployee(int id) {

        return employeeDAO.getEmployee(id);

    }

    @Override

    @Transactional

    public void deleteEmployee(int id) {

        employeeDAO.deleteEmployee(id);

    }

}

```

**Вихідний код Контрлеру, класу `EmployeeController.java`:**

```

package com.alexandrsinko.spring.mvc_hibernate_aop.controller;

import com.alexandrsinko.spring.mvc_hibernate_aop.entity.Employee;

import com.alexandrsinko.spring.mvc_hibernate_aop.service.EmployeeService;

import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

@Controller

public class EmployeeController {

    private EmployeeService employeeService;

    @Autowired

    public EmployeeController(EmployeeService employeeService) {

        this.employeeService = employeeService;
    }

    @RequestMapping(value = "/")

    public String showAllEmployees(Model model) {

        List<Employee> allEmployees = employeeService.getAllEmployees();

        model.addAttribute("allEmps", allEmployees);

        return "all-employees";
    }

    @RequestMapping(value = "/addNewEmployee")

    public String addNewEmployee(Model model) {

        Employee employee = new Employee();

```

```

        model.addAttribute("employee", employee);

        return "employee-info";
    }

    @RequestMapping(value = "/saveEmployee")

    public String saveEmployee(@ModelAttribute("employee") Employee
employee) {

        employeeService.saveEmployee(employee);

        return "redirect:/";
    }

    @RequestMapping(value = "/updateInfo")

    public String updateEmployee(@RequestParam("empId") int id, Model model){

        Employee employee = employeeService.getEmployee(id);

        model.addAttribute("employee", employee);

        return "employee-info";
    }

    @RequestMapping(value = "/deleteEmployee")

    public String deleteEmployee(@RequestParam("empId") int id) {

        employeeService.deleteEmployee(id);

        return "redirect:/";
    }

}

```

**Вихідний код модулю View, файлу all-employees.jsp:**

```
<% @ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```

<!DOCTYPE html>

<html>

<body>

<h2>All Employees</h2>

<br>

<table>

  <tr>

    <th>Name</th>

    <th>Surname</th>

    <th>Department</th>

    <th>Salary</th>

    <th>Operations</th>

  </tr>

  <c:forEach var = "emp" items="${allEmps}">

    <c:url var="updateButton" value="/updateInfo">

      <c:param name="empId" value="${emp.id}"/>

    </c:url>

    <c:url var="deleteButton" value="/deleteEmployee">

      <c:param name="empId" value="${emp.id}"/>

    </c:url>

    <tr>

      <td>${emp.name}</td>

```

```

<td>${emp.surname}</td>

<td>${emp.department}</td>

<td>${emp.salary}</td>

<td>

    <input type="button" value="Update"

    onclick="window.location.href='${updateButton}'"/>

    <input type="button" value="Delete"

        onclick="window.location.href='${deleteButton}'"/>

</td>

</tr>

</c:forEach>

</table>

<br>

<input type="button" value="Add"

    onclick="window.location.href='addNewEmployee'"/>

</body>

</html>

```

**Вихідний код модулю View, файлу employeeInfo.java:**

```

<% @ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<!DOCTYPE html>

<html>

<body>

```

<h2>

Employee Info

</h2>

<br>

<form:form action="saveEmployee" modelAttribute="employee">

<form:hidden path="id"/>

Name:<form:input path="name"/>

<br><br>

Surname:<form:input path="surname"/>

<br><br>

Department:<form:input path="department"/>

<br><br>

Salary:<form:input path="salary"/>

<br><br>

<input type="submit" value="Submit">

</form:form>

</body>

</html>

**Вихідний код конфігурації, файлу ApplicationContext.xml:**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="

    http://www.springframework.org/schema/beans

    http://www.springframework.org/schema/beans/spring-beans.xsd

    http://www.springframework.org/schema/context

    http://www.springframework.org/schema/context/spring-context.xsd

    http://www.springframework.org/schema/mvc

    http://www.springframework.org/schema/mvc/spring-mvc.xsd

    http://www.springframework.org/schema/tx

    http://www.springframework.org/schema/tx/spring-tx.xsd

    http://www.springframework.org/schema/aop

    http://www.springframework.org/schema/tx/spring-aop.xsd">

<context:component-scan base-
package="com.alexandrsinko.spring.mvc_hibernate_aop" />

<mvc:annotation-driven/>

<aop:aspectj-autoproxy/>

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

    <property name="prefix" value="/WEB-INF/view/" />

    <property name="suffix" value=".jsp" />

```



```

</bean>

<bean id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource"

    destroy-method="close">

    <property name="driverClass" value="com.mysql.cj.jdbc.Driver" />

    <property name="jdbcUrl"
value="jdbc:mysql://localhost:3306/my_db?useSSL=false&serverTimezone=
UTC" />

    <property name="user" value="root" />

    <property name="password" value="root" />

</bean>

<bean id="sessionFactory"

    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

    <property name="dataSource" ref="dataSource" />

    <property name="packagesToScan"
value="com.alexandrsinko.spring.mvc_hibernate_aop.entity" />

    <property name="hibernateProperties">

        <props>

            <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

            <prop key="hibernate.show_sql">true</prop>

        </props>

    </property>

</bean>

```

```

<bean id="transactionManager"

    class="org.springframework.orm.hibernate5.HibernateTransactionManager">

    <property name="sessionFactory" ref="sessionFactory"/>

</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

</beans>

```

### **Вихідний код конфігурації, файлу web.xml:**

```

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns="http://xmlns.jcp.org/xml/ns/javaee"

    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"

    id="WebApp_ID" version="3.1">

    <display-name>spring-cource-mvc-hibernate-aop</display-name>

    <absolute-ordering />

    <servlet>

        <servlet-name>dispatcher</servlet-name>

        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-

class>

        <init-param>

            <param-name>contextConfigLocation</param-name>

            <param-value>/WEB-INF/applicationContext.xml</param-value>

        </init-param>

```

```
<load-on-startup>1</load-on-startup>  
  
</servlet>  
  
<servlet-mapping>  
  
  <servlet-name>dispatcher</servlet-name>  
  
  <url-pattern>/</url-pattern>  
  
</servlet-mapping>  
  
</web-app>
```