Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

*На правах рукопису*

УДК 004.896

# ВИПУСКНА КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

Тема «Розроблення автоматизованої системи роботи з персоналом на підприємствах»
Спеціальність 121 "Інженерія програмного забезпечення"

**ПОЯСНЮВАЛЬНА ЗАПИСКА**
**МР.ІПЗм**

Студент
__________________ Кирій Д.В.
(підпис) (розшифровка підпису) (дата)

Науковий керівник
доц. ___________ Супрун О.М.
(посада) (підпис) (розшифровка підпису) (дата)

Допускається до захисту
з питань нормоконтролю
Завідувач кафедри

______________ Бичков О.С.
(підпис) (розшифровка підпису) (дата)

2021

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Спеціальність 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ:

Завідувач кафедри програмних систем і технологій

___________________(О.С.Бичков)

"___"_________________ 20___р.

**ЗАВДАННЯ**

**НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ**

**СТУДЕНТУ**

**Кирію Дмитру Віталійовичу**

(прізвище, ім'я, по батькові)

1. Тема випускної кваліфікаційної магістерської роботи: "Розроблення автоматизованої системи роботи з персоналом на підприємствах"

Керівник проекту: Супрун Ольга Миколаївна

Затверджені протоколом кафедри № 6 від 11.11.2020

2. Строк здачі студентом закінченої роботи 21.05.2021.

3. Вихідні дані до роботи: інтернет-ресурси з питань роботи з персоналом та побудови веб-застосунків. Технології розробки клієнт-серверних додатків та інструменти побудови UML-діаграм.

4. Зміст пояснювальної записки: дослідження чинників, які впливають на вибір автоматизованої системи роботи з персоналом, зручність існуючих систем з подібним функціоналом, можливість користувачів взаємодіяти один з одним в рамках такої системи. Визначення особливостей архітектурного рішення для

автоматизованої системи роботи з персоналом, розробка прототипу такої системи з необхідним функціоналом.

5. Дата видачі завдання  “12” жовтня 2020 р.

Керівник ___________________________

(підпис)

Завдання прийняв до виконання _________________________

(підпис)

**КАЛЕНДАРНИЙ ПЛАН**

| Номер і назва етапів бакалаврської роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|
| 1. Постановка задачі | 12.11.2020 | |
| 2. Уточнення постановки задачі | 12.11.2020 | |
| 3. Дослідження предметної області | 13.11.2020-09.12.2020 | |
| 4. Проектування рішення<br>4.1. Документування та обґрунтування запропонованого рішення<br>4.2. Створення робочого прототипу за першим сценарієм. | 10.12.2020-16.02.2021 | |
| 5. Дослідження проблематики та уточнення рішення | 17.02.2021-24.02.2021 | |
| 6. Програмна реалізація<br>6.1. Уточнення кожного блоку, модуля та документування деталізованих вимог<br>6.2. Розроблення та тестування кожного блоку, модуля<br>6.3. Внесення правок в документацію за необхідності | 25.02.2021-05.04.2021 | |
| 7. Створення пакету доставки програмного забезпечення та інструкції з розгортання | 06.04.2021-16.04.2021 | |
| 8. Оформлення і друк пояснювальної записки | 17.04.2021-19.04.2021 | |
| 9. Отримання рецензій | 20.04.2021-13.05.2021 | |

| 10. Оформлення презентацій | 14.05.2021-20.05.2021 | |
|---|---|---|
| 11. Захист | 25.05.2021 | |

Студент – магістр ______________ ________________________
(підпис) (розшифровка підпису)

Керівник роботи ________________ ______________________
(підпис) (розшифровка підпису)

# АНОТАЦІЯ

**Випускна кваліфікаційна магістерська робота:** 71 с., 4 рис., 4 додатки, 6 джерел.

**Тема:** Розроблення автоматизованої системи роботи з персоналом на підприємствах.

**Мета:** Покращення та спрощення процесу роботи з персоналом на підприємствах за рахунок розробки відповідного програмного забезпечення.

**Задачі дослідження**:

- дослідити теоретичні основи побудови і програмної реалізації систем управління персоналом на підприємствах;
- проаналізувати існуючі технології для розробки програмного забезпечення;
- проаналізувати вимоги ринку та можливості конкурентів;
- розробити та запропонувати до впровадження вдосконалену автоматизовану систему роботи з персоналом на підприємствах.

**Об'єкт дослідження:** Засоби, принципи та методології проектування систем у веб середовищі.

**Предмет дослідження:** Автоматизована система роботи з персоналом на підприємствах.

**Результат дослідження:** Удосконалена автоматизована система роботи з персоналом на підприємствах, яка одночасно вирішує проблему управління співробітниками, а також надає зручні інструменти роботи з процесами компанії для всіх кадрів.

# АННОТАЦИЯ

**Выпускная квалификационная магистрская работа**: 71 с., 4 рис., 4 приложения, 6 источников.

**Тема:** Разработка автоматизированной системы работы с персоналом на предприятиях.

**Цель:** Улучшение и упрощение процесса работы с персоналом на предприятиях за счет разработки соответствующего программного обеспечения.

**Задачи исследования:**

- исследовать теоретические основы построения и программной реализации систем управления персоналом на предприятиях;
- проанализировать существующие технологии для разработки программного обеспечения;
- проанализировать требования рынка и возможности конкурентов;
- разработать и предложить к внедрению усовершенствованную автоматизированную систему работы с персоналом на предприятиях.

**Объект исследования:** Средства, принципы и методологии проектирования систем в веб среде.

**Предмет исследования:** Автоматизированная система работы с персоналом на предприятиях.

**Результат исследования:** Усовершенствованная система работы с персоналом на предприятиях, которая одновременно решает проблему управления сотрудниками, а также предоставляет удобные инструменты работы с процессами компании для всех кадров.

# ANNOTATION

**Final master's thesis:** 71 pages, 4 pictures, 4 additions, 6 sources.

**Subject:** Development of an automated system for work with personnel at companies.

**Objective:** To improve and simplify the process of working with staff at companies through the development of appropriate software.

**Research objectives:**

- to explore the theoretical foundations of construction and software implementation of personnel management systems at companies;
- analyze technologies for software development;
- analyze market requirements and opportunities of competitors;
- develop and propose for implementation an improved automated system of work with personnel at companies.

**Object of research:** Tools, principles and methodologies for designing systems in a web environment.

**Subject of research:** Automated system of work with personnel at enterprises.

**The result of the study:** Improved system for work with personnel at companies, which simultaneously solves the problem of employee management, as well as provides convenient tools for working with company processes for all employees.

# ЗМІСТ

ЗМІСТ ........ 9
ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ ........ 11
ВСТУП ........ 12
1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ЗАСТОСУНКІВ ........ 14
1.1. Клієнт-серверна архітектура ........ 14
1.1.1. Клієнт-серверна комунікація ........ 15
1.1.2. Порівняння клієнт-серверного підходу до peer-to-peer комунікації ........ 16
1.2. Високонавантажені веб-застосунки ........ 18
1.2.1. Принципи побудови високонавантаженої архітектури ........ 18
1.2.2. Розподілений кеш у високонавантажених системах ........ 22
1.3. Висновок до розділу 1 ........ 22
2. ТЕХНОЛОГІЇ РОЗРОБКИ ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ЗАСТОСУНКІВ ........ 24
2.1. Клієнтська частина ........ 24
2.1.1. JavaScript, HTML, CSS ........ 24
2.1.2. React ........ 31
2.2. Серверна частина ........ 33
2.2.1. Node.js ........ 33
2.2.2. NestJs ........ 37
2.2.3. MySQL ........ 38
2.2.4. Docker ........ 39
2.3. Обґрунтування вибору технологій ........ 40
2.3.1. Переваги React ........ 40
2.3.2. Переваги Node.js та NestJS ........ 41
2.3.3. Переваги MySQL ........ 42

2.4. Висновок до розділу 2 .......... 42

3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ТА РЕЗУЛЬТАТИ НАУКОВОГО ДОСЛІДЖЕННЯ .......... 43

3.1. Визначення вимог до програмного забезпечення .......... 43

3.2. Проектування програмного забезпечення .......... 44

3.2.1. Архітектури клієнту .......... 44

3.2.2. Архітектура серверу .......... 46

3.2.3. Проектування бази даних та створення міграції .......... 46

3.3. Життєвий цикл програмного забезпечення .......... 47

3.4. Результати наукового дослідження .......... 48

3.4.1. Аналіз конкурентів .......... 48

3.4.2. Порівняння з розробленим ПЗ .......... 50

3.5. Висновок до розділу 3 .......... 50

ВИСНОВКИ .......... 51

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .......... 52

ДОДАТКИ .......... 53

Додаток А .......... 53

Додаток Б .......... 54

Додаток В .......... 55

Додаток Д .......... 59

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Веб-сервер - це сервер, що приймає запити та віддає відповідь клієнту.

Клієнт – програмний компонент, який надсилає запити до серверу.

HTTP - протокол передачі даних від клієнта до серверу, та навпаки, у текстовому вигляді.

JS - динамічна, об'єктно-орієнтована, прототипна мова програмування.

TS - мова програмування, яка розширюється можливості JavaScript. Основною перевагою є можливість типізації даних.

Автентифікація - перевірка достовірності пред'явленого користувачем ідентифікатора.

WebSocket - протокол передачі даних в реальному часі.

API – програмний інтерфейс, що надає доступ до певного функціоналу.

HTML – мова розмітки, яка надає основні інструменти для побудови структурної частини веб - сторінки.

DOM - специфікація прикладного програмного інтерфейсу для роботи зі структурованими документами.

# ВСТУП

Під час створення і розвитку компанії виникає проблема швидкого та зручного управління кадрами на підприємствах, не вирішення якої збільшує рівень невдоволеності та зменшує продуктивність роботи. Для усунення даної проблеми існують рішення, що дають змогу працювати з персоналом, керувати внутрішніми процесами компанії. Але, існуючі рішення акцентують увагу на строгому контролі співробітників, що зменшує рівень якості праці. Після проведеного дослідження було визначено, що ефективність працівників напряму залежить від кількості обмежень та правил, які застосовуються. Наприклад, в більшості застосунків для управління персоналом вбудований трекінг робочих годин, система покарань, у випадку виходу за встановлені рамки, тощо. Даний підхід не дозволяє побудувати адекватну корпоративну культуру, яка б позитивно впливала на бажання довести поставлені задачі до якісного результату. Для вирішення даної проблеми було побудовано систему, яка дозволяє як і ефективно керувати без введення обмежень, опираючись тільки на результати завершеної роботи, так і надає додаткові інструменти взаємодії з керівництвом працівникам, що спрощує такі процеси, як ініціювання перегляду заробітної плати та зручного планування відпусток. Дане рішення дозволить покращити загальний настрій на підприємствах користувачів та вирішити проблему управління людськими ресурсами одночасно.

**Актуальність дослідження:**

Актуальність дослідження полягає у вдосконаленні систем роботи з персоналом за рахунок акцентування уваги на перевагах, які надає рішення не тільки керівництву, а й співробітникам, що дозволить збільшити ефективність роботи та покращити загальний настрій персоналу.

**Об'єкт дослідження:** Засоби, принципи та методології проектування систем у веб середовищі.

**Метод дослідження:** Аналіз особливостей розробки веб-застосунків, а також дослідження вдоволеності персоналу наявними рішеннями та розробленим

прототипом.

**Новизна одержаних результатів:** Удосконалена система для роботи з персоналом на підприємствах, яка не тільки надає керівництву потрібний для ефективного управління кадрами функціонал, а також інструменти для покращення взаємодії співробітників з процесами компанії.

# 1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ЗАСТОСУНКІВ

Веб-застосунок - це прикладне програмне забезпечення, яке працює на веб-сервері, на відміну від комп’ютерних програм, що запускаються локально в операційній системі пристрою. Доступ до веб-додатків здійснюється користувачем через веб-браузер з активним підключенням до мережі. Ці програми програмуються за допомогою змодельованої структури клієнт-сервер – користувачеві (клієнту) надаються послуги через зовнішній сервер, розміщений іншою стороною. Приклади часто використовуваних веб-програм включають: веб-пошту, роздрібні продажі в Інтернеті, Інтернет-банкінг та Інтернет-аукціони.

## 1.1. Клієнт-серверна архітектура

Клієнт-серверна архітектура – це архітектура додатків, що дозволяє розділити навантаження між провайдерами ресурсу чи послуги, які називаються серверами, та отримувачами послуг, що називаються клієнтами. Зазвичай клієнти та сервери спілкуються між собою використовуючи комп’ютерну мережу, вони знаходяться на окремому обладнанні, але і клієнт, і сервер можуть перебувати в одній системі.[1] Хост сервера запускає одну або кілька серверних програм, які надають свої ресурси клієнтам. Клієнт, зазвичай, не надає доступ до своїх ресурсів, але він запитує дані чи послуги у сервера. Отже, користувачі ініціюють сеанси спілкування із серверами, які очікують на вхідні запити та реагують на них тим чи іншим чином (див. Рис. 1.1.).

Характеристика "клієнт-сервер" описує взаємозв'язок взаємодіючих програм у застосунку. Сервер дозволяє отримати доступ до функції чи послуги одному або багатьом клієнтам, які запитують такі послуги. Сервери класифікують відповідно до послуг, що надаються. Наприклад, веб-сервер займається обробкою веб-сторінок, а файловий - комп'ютерних. Спільним ресурсом може бути будь-яке програмне забезпечення серверного комп'ютера та електронні компоненти,

починаючи від програм та даних, закінчуючи процесорами та запам'ятовуючими пристроями. Спільне використання ресурсів сервера є послугою.

Чи комп’ютер є клієнтом, сервером чи обома, визначається характером програми, яка вимагає сервісних функцій. Наприклад, на одному комп’ютері може одночасно працювати веб-сервер та програмне забезпечення файлового сервера, щоб обслуговувати різні дані клієнтам, які роблять різні типи запитів. Клієнтське програмне забезпечення також може взаємодіяти із серверним програмним забезпеченням на тому самому комп'ютері. Зв'язок між серверами, наприклад, для синхронізації даних іноді називають міжсерверним або server-to-server.

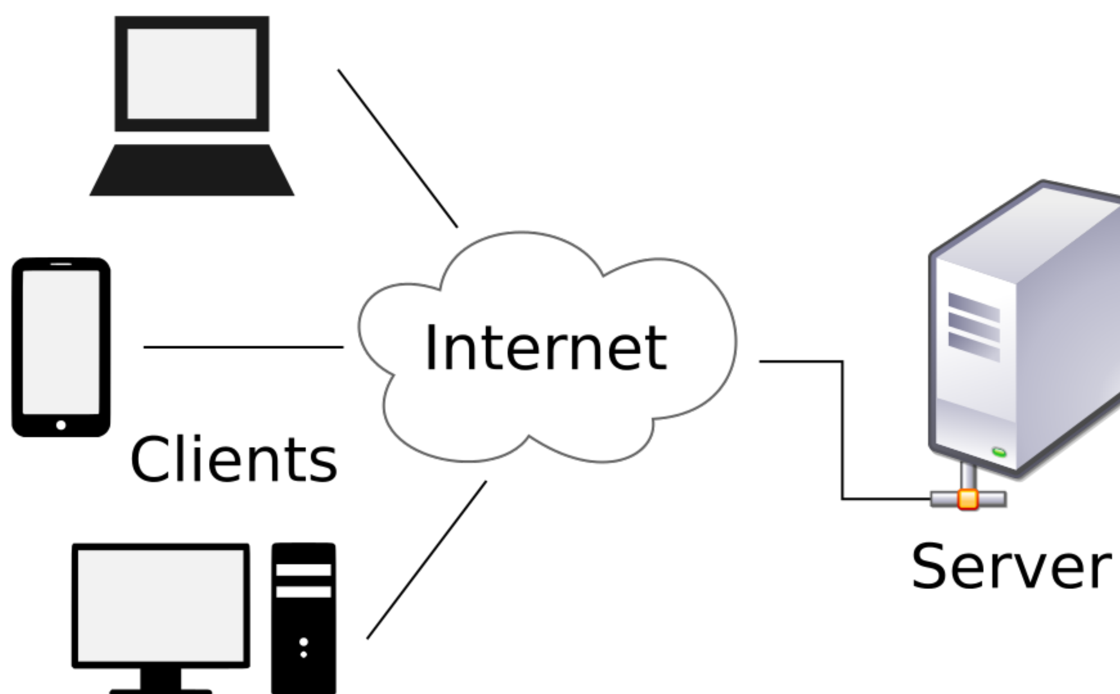



Рис. 1.1. Базова схема клієнт-серверної архітектури

### 1.1.1. Клієнт-серверна комунікація

Загалом, послуга - це абстракція комп'ютерних ресурсів, і клієнт не повинен знати про те, як працює сервер під час виконання запиту та надання відповіді.

Клієнт повинен розуміти відповідь лише на основі протоколу програми, тобто вмісту та форматування даних для запитуваної послуги.

Клієнти та сервери обмінюються повідомленнями за шаблоном запит-відповідь. Клієнт надсилає запит, а сервер повертає відповідь. Цей обмін повідомленнями є прикладом міжпроцесорного спілкування. Для спілкування комп'ютери повинні мати спільну мову, і вони повинні дотримуватися правил, щоб і клієнт, і сервер знали, чого очікувати. Мова та правила спілкування визначені протоколом зв'язку. Всі протоколи працюють на рівні програми. Протокол рівня додатків визначає основні схеми діалогу. Щоб ще більше формалізувати обмін даними, сервер може реалізувати інтерфейс прикладного програмування (API).

API - це рівень абстракції для доступу до послуги. Обмеження спілкування певним форматом вмісту, полегшує синтаксичний аналіз. Абстрагування доступу полегшує міжплатформенний обмін даними. Сервер може отримувати запити від багатьох різних клієнтів за короткий проміжок часу. В будь-який момент комп'ютер може виконувати лише обмежену кількість завдань і покладається на систему планування, для визначення пріоритетів вхідних запитів клієнтів для їх задоволення. Щоб запобігти зловживанням та максимізувати доступність, серверне програмне забезпечення може обмежити доступність для клієнтів. Атаки на службу використовують зобов'язання сервера обробляти запити, перевантажуючи його надмірною частотою запитів. Шифрування слід застосовувати, якщо конфіденційна інформація повинна передаватися між клієнтом та сервером.

### 1.1.2. Порівняння клієнт-серверного підходу до peer-to-peer комунікації

На додаток до моделі клієнт-сервер, розподілені обчислювальні програми часто використовують архітектуру однорангових (P2P) додатків (див. Рис. 1.2.). У моделі клієнт-сервер - сервер часто призначений для роботи як централізована система, яка обслуговує багатьох клієнтів. Вимоги до обчислювальної потужності та пам'яті сервера повинні масштабуватися відповідно до очікуваного

навантаження. Системи балансування навантаження та системи відмови часто використовуються для масштабування сервера за межі однієї фізичної машини.

Балансування навантаження визначається як методичний та ефективний розподіл мережевого або додаткового трафіку на декількох серверах у фермі серверів. Кожен балансир навантаження знаходиться між клієнтськими пристроями та серверними серверами, отримуючи та розподіляючи вхідні запити на будь-який доступний сервер, здатний їх виконати.[2]

У одноранговій мережі два або більше комп’ютери об’єднують свої ресурси та спілкуються в децентралізованій системі. Піри - це рівноправні вузли в неієрархічній мережі. На відміну від клієнтів у мережі клієнт-сервер або клієнт – черга – клієнт, однорангові комунікації здійснюють безпосередню взаємодію між собою. У однорангових мережах алгоритм у протоколі комунікації балансує навантаження і навіть комп’ютери зі скромними ресурсами можуть допомогти розподілити навантаження. Якщо вузол стає недоступним, його спільні ресурси залишаються доступними до тих пір, поки їх пропонують інші партнери. В ідеалі, рівноправній особі не потрібно досягати високої доступності, оскільки інші надлишкові однорангові зв’язки компенсують час простою будь-якого ресурсу. Оскільки доступність та завантажуваність комп’ютерів змінюються, протокол перенаправляє запити.

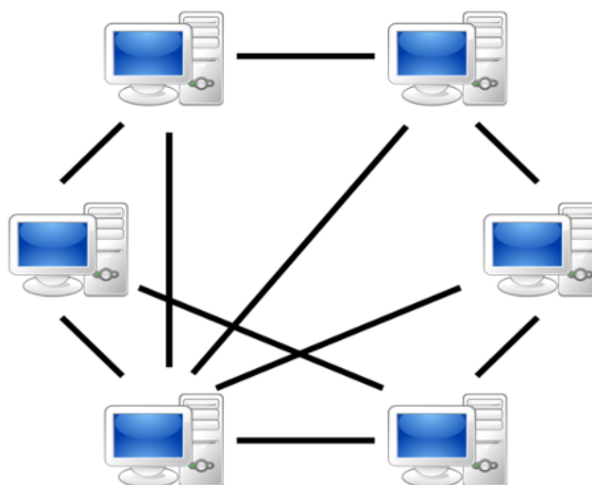


Рис.1.2. Схема peer-to-peer мережі

### 1.2. Високонавантажені веб-застосунки

У розробці веб-додатків потрібно зосереджуватись на створенні високопродуктивної масштабованої архітектури. Навіть якщо проект досить невеликий, в якийсь момент кількість користувачів може сильно збільшитись або може знадобитися масштабування. Якщо система не гнучка і не може витримувати великі навантаження, вона має великі шанси на вихід з ладу.

#### 1.2.1. Принципи побудови високонавантаженої архітектури

Навантаження починається, коли один фізичний сервер стає не в змозі ефективно здійснювати обробку даних. Якщо один екземпляр одночасно обслуговує 10000 підключень - це велике навантаження. Highload - це одночасне обслуговування тисяч і мільйонів користувачів. Що зазвичай відбувається, коли система не витримує великих навантажень:

1. Повільне або нескінченне завантаження сторінки;
2. Випадкові помилки;
3. Відключені з'єднання з веб-сервером;
4. Часткове завантаження вмісту (наприклад, немає деяких зображень);
5. Зменшення активності аудиторії користувачів та втрати клієнтів як результат.

З цих причин доводиться докладати чимало зусиль для підтримки та масштабування веб-програм, витрачаючи тим самим час, витрати та енергію, а також втрачаючи клієнтів. Архітектура веб-додатку визначає 95% успіху всієї роботи. Ось чому так важливо створити легко масштабовану архітектуру сервера, яка зможе витримувати великі навантаження. Для розробки успішних великомасштабних веб-додатків потрібно зрозуміти принципи побудови високопродуктивного програмного забезпечення. Принципи розробки високопродуктивних додатків:

- Динаміка та гнучкість. Ніколи не відомо наскільки може змінитись кількість користувачів у той чи інший період часу. Можливо, кількість користувачів збільшиться в 10 разів. Або якась незначна функція продукту почне набирати популярність тоді, коли ніхто не очікує. Як результат, перед розробником постануть проблеми, пов’язані з необхідністю розширення проекту.
- Створюючи великомасштабні веб-програми, основну увагу слід звертати на гнучкості, яка дозволить легко впроваджувати зміни та розширення. Гнучкість, відсутність попереднього планування всіх аспектів, є найважливішою характеристикою будь-якої швидкозростаючої системи.
- Поступове зростання проекту. Важко передбачити розмір аудиторії на наступні роки, тому краще зосередити увагу на масштабованості. Те саме стосується архітектури додатків. Поступові рішення є основою для успішної розробки власних веб-додатків.
- Якщо маємо запущений додаток, немає сенсу негайно надавати інфраструктуру, яка може витримати мільйони користувачів. Використовуючи хмару для розміщення нових проектів, ми можемо зменшити вартість сервера та спростити управління ними. Крім того, багато служб хмарного хостингу надають послуги приватної мережі, що дозволяє програмістам безпечно використовувати кілька серверів у хмарі та зробити систему масштабованою.[3]

Масштабування будь-якої веб-програми - це поступовий процес, що включає 4 кроки:

1. Аналіз навантаження.

Визначення площ, які найбільше постраждали від навантажень. Перенесення цих областей в окремі вузли та їх оптимізація

2. Створення високопродуктивної масштабованої архітектури.

Як правило, нова програма запускається на одному сервері, працює веб-сервер, база даних і сама програма. Такий підхід до розробки користувацьких веб-додатків є обґрунтованим, оскільки допомагає заощадити час та зменшити витрати.

З самого початку не будується великий проект (якщо він не передбачає багато користувачів), потрібно зосередитись на масштабованості продукту та обрати потужний сервер, який зможе обробляти великі навантаження, якщо це потрібно. Однак метою багатьох проектів є залучення тисяч і мільйонів користувачів, надання багатого користувацького досвіду та якісних послуг і заробіток. Рішення для електронної комерції, портали для клієнтів, ігрові додатки AR, онлайн-консалтингові послуги, рішення для соціальних мереж, додатки для побачень. Усі ці приклади припускають наявність великої аудиторії і вимагають використання різних оптимізацій, що дозволяють витримувати великі навантаження. Перш за все, розглянемо способи, які допоможуть створювати масштабні та високопродуктивні веб-додатки.

- Поділ бази даних.

Найчастіше, перший вузол, який навантажується - це база даних. Кожен запит від користувача до програми зазвичай складає від 10 до 100 запитів до бази даних. Розгалуження бази даних на окремому сервері збільшить її продуктивність та зменшить негативний вплив на інші компоненти.

- Міграція бази даних.

У деяких випадках переміщення бази даних на інший сервер може стати проблемою для робочого додатку. Для міграції бази даних можна зробити такі речі: скористатись простим рішенням - розмістити оголошення про заплановану роботу на застосунку. Краще це робити вночі, коли активність аудиторії користувачів мінімальна.

- Використання реплікації для синхронізації даних з одного сервера на інший.

Після налаштування слід змінити IP-адресу бази даних у додатку на новий сервер. А потім - вимкнути старий сервер.

Далі слід відокремити веб-сервер, виділення якого в окремий вузол дозволить залишити більше ресурсів для програми. Слід налаштувати розгортання програми на сервері Nginx, так і на програмному сервері, що представляє дані.

Тоді Nginx сам видасть статичні файли, а сервер буде зайнятий лише обробкою сценаріїв. Nginx дозволяє підключення до серверної бази за IP-адресою. Коли навантаження збільшується, веб-програма починає працювати повільніше. У якийсь момент причина криється вже в самій реалізації. Щоб уникнути таких проблем, слід використовувати декілька серверів.

Встановлюючи бекенди, потрібно переконатись, що вони мають однакову конфігурацію. Використання Nginx дозволяє збалансувати навантаження між ними. Для цього слід визначити список серверів та використовувати його в конфігурації. Як тільки починається використання кількох серверних систем, запити від одного і того ж користувача будуть надсилатися на різні сервери. Для цього знадобиться єдине сховище для всіх сеансів, наприклад розподілений кеш.

Черги завдань дозволяють виконувати важкі операції асинхронно, не сповільнюючи роботу основної програми. Поки сервер черг буде отримувати завдання від програми, інші сервери обробляють завдання. Якщо середня кількість завдань у черзі збільшується, слід збільшити і кількість серверів. Це допоможе збалансувати навантаження.

DNS підтримує балансування на основі Round Robin, що дозволяє вказати кілька IP-адрес приймаючих веб-серверів, які називаються frontends. Тут потрібно встановити кілька однакових інтерфейсів, щоб DNS давав різні IP-адреси різним клієнтам. Таким чином, ви забезпечуєте балансування між фасадами.

Завантаження та обробка файлів зазвичай відбувається з внутрішньої сторони. Коли існує кілька резервних копій, це стає незручним, оскільки розробники повинні пам’ятати, на який сервер вони завантажують кожен файл. Крім того, це може суттєво знизити продуктивність сервера. Щоб уникнути таких труднощів, слід використовувати окремі сервери для завантаження, зберігання та обробки файлів.

### 1.2.2. Розподілений кеш у високонавантажених системах

Кеш є апаратним або програмним компонентом, що зберігає дані, щоб у майбутньому запити на ці дані могли швидше обслуговуватися. Дані, що зберігаються в кеш-пам'яті, можуть бути результатом попереднього обчислення або копією даних, що зберігаються в інших місцях. Звернення до кешу відбувається, коли запитані дані можуть бути знайдені в сховищі. Зчитування даних з кешу відбувається швидше, ніж переобчислення результату або зчитування з повільнішого сховища даних. Таким чином, чим більше запитів може бути подано з кешу, тим швидше працює система.

Щоб бути економічно ефективними та забезпечити ефективне використання даних, кеші повинні бути відносно невеликими. Тим не менше, кеші зарекомендували себе у багатьох областях обчислень, оскільки типові комп'ютерні програми отримують доступ до даних з високим рівнем швидкості. Такі шаблони доступу демонструють тимчасову локальність, де запитуються дані, які вже були нещодавно запрошені, та просторову локальність, коли запитуються дані, які зберігаються фізично близько до даних, які вже були запрошені.

Розподілений кеш є продовженням традиційної концепції кеш-пам'яті, що використовується в одному кластері. Розподілений кеш може охоплювати декілька серверів, щоб він міг збільшуватися в розмірі та збільшувати транзакційну здатність. В основному він використовується для зберігання даних додатків, що знаходяться в базі та даних веб-сеансів. Ідея розподіленого кешування стала реалістичною, оскільки основна пам'ять стала дешевою, а мережеві карти стали швидкими. Крім того, розподілений кеш добре працює на машинах з нижчою вартістю, які зазвичай використовуються для веб-серверів, на відміну від серверів баз даних, які потребують дорогого обладнання.

## 1.3. Висновок до розділу 1

В цьому розділі було розглянуто теоретичні відомості, які стосуються

побудови високонавантажених веб-додатків. Було виділено основні техніки, стандарти, системи та технології, задля подальшого їх використання у створенні прототипу удосконаленої автоматизованої системи роботи з персоналом на підприємствах, а також для зручності розробки серверної та клієнтської частин цього програмного продукту, що являтимуть собою масштабовані, гнучкі та прості у використанні програмні компоненти системи.

# 2. ТЕХНОЛОГІЇ РОЗРОБКИ ВИСОКОНАВАНТАЖЕНИХ ВЕБ-ЗАСТОСУНКІВ

Для розроблення клієнтської частини веб-застосунків найчастіше використовуються 3 основні технології HTML, CSS, JavaScript. Серверна частина дає нам більшу варіативність за рахунок кількості мов, підходів та фреймворків для її імплементації. В даному розділі буде наведено опис кожної технології, яка була використана для розробки автоматизованої системи управління персоналом та їхні переваги в порівнянні з іншими варіантами.

## 2.1. Клієнтська частина

Для розробки веб-застосунку є висока варіативність технологій, які являються надбудовую JavaScript і полегшують побудову масштабованої архітектури. Для розробки даного додатку було обрано React, як основну бібліотеку для візуалізації даних та налаштування взаємодії з ними.

### 2.1.1. JavaScript, HTML, CSS

**JavaScript**, часто скорочується як JS - це мова програмування, яка відповідає специфікації ECMAScript. JavaScript є високорівневим та багатопарадигменним. Він має динамічну типізацію, об’єктно орієнтоване програмування на основі прототипів та функції першого класу.[4]

Поряд з HTML та CSS, JavaScript є однією з основних технологій Всесвітньої мережі. Понад 97% веб-сайтів використовують його на стороні клієнту для опису поведінки веб-сторінок, часто включаючи сторонні бібліотеки. Усі основні веб-браузери мають спеціальний двигун для виконання коду на пристрої користувача. Як мова багатопарадигменна, JavaScript підтримує керовані подіями, функціональні та імперативні стилі програмування. Мова має інтерфейси прикладного програмування для роботи з текстом, датами, регулярними виразами, стандартними структурами даних та об'єктною моделлю документа (DOM). Стандарт ECMAScript

не включає жодного вводу / виводу, наприклад, мережевих, сховищних чи графічних засобів. На практиці веб-браузер або інша система виконання забезпечує API для вводу-виводу. Спочатку двигуни JavaScript використовувались лише у веб-браузерах, але зараз вони є основними компонентами інших програмних систем, зокрема серверів та різноманітних додатків.

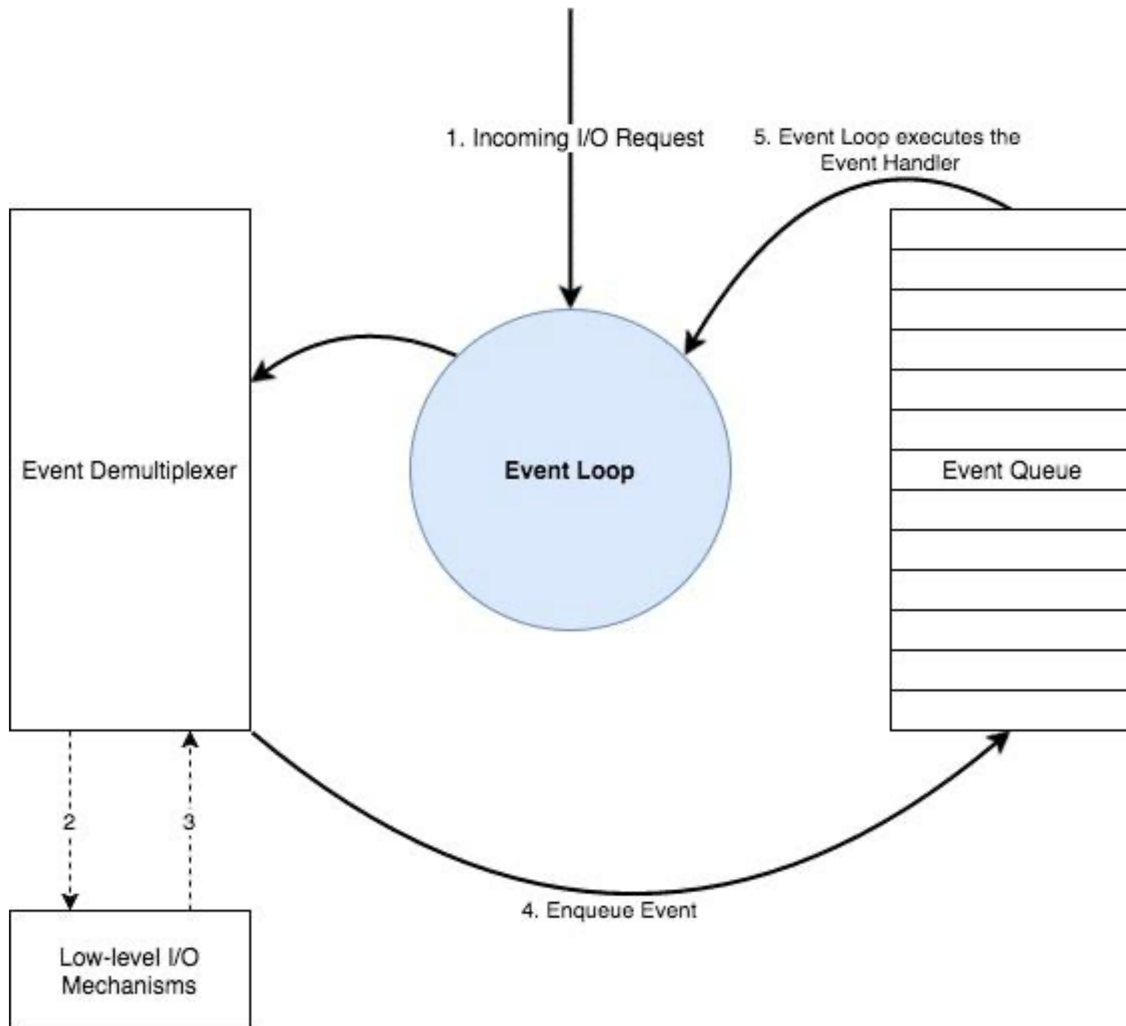



Рис. 2.1. Принцип роботи асинхронних механізмів JavaScript.

Наступні функції є загальними для всіх відповідних реалізацій ECMAScript, якщо прямо не вказано інше.

- Імперативний та структурований

JavaScript підтримує більшу частину синтаксису структурованого програмування з C (наприклад, оператори if, while, do while цикли тощо).

- Слабко типізований

JavaScript слабко типізований, що означає, що певні типи неявно передаються залежно від використовуваної операції.

- Об’єкти

У JavaScript об'єкт - це асоціативний масив, доповнений прототипом. Кожен ключ надає ім'я властивості об'єкта, і існує два синтаксичні способи вказати таке ім'я: крапковий запис (obj.x = 10) і нотація в дужках (obj ['x'] = 10). Властивість може бути додана або видалена під час виконання.

- Прототипи

JavaScript використовує прототипи, на відміну від інших об'єктно-орієнтованих мов, які використовують класи для успадкування.

- Функції як конструктор об’єктів

Функції виконують роль конструктору об'єктів разом із їх типовою поведінкою. Префікс виклику функції new створить екземпляр прототипу, успадковуючи властивості та методи від конструктора (включаючи властивості від прототипу Object). ECMAScript 5 пропонує метод Object.create, дозволяючи явне створення екземпляра без автоматичного успадкування від прототипу Object. Властивість прототипу конструктора визначає об'єкт, який використовується для внутрішнього прототипу нового об'єкта. Нові методи можна додати, змінивши прототип функції, що використовується як конструктор. Вбудовані конструктори JavaScript, такі як Array або Object, також мають прототипи, які можна змінювати. Незважаючи на те, що можливо змінити прототип Object, це, як правило, вважається поганою практикою, оскільки більшість об'єктів у JavaScript успадковують методи та властивості прототипу Object, і вони можуть не очікувати модифікації прототипу.

- Функції як методи

На відміну від багатьох об'єктно-орієнтованих мов, не існує різниці між визначенням функції та визначенням методу. Швидше, відмінність відбувається під час виклику функції; коли функція викликається як метод об'єкта, this, ключове

слово, прив'язується до цього об'єкта, для цього виклику.

JavaScript і DOM дають можливість зловмисникам доставляти скрипти для запуску на клієнтському комп'ютері через Інтернет. Автори браузера мінімізують цей ризик, використовуючи два обмеження. По-перше, скрипти працюють у пісочниці, в якій вони можуть виконувати лише дії, пов'язані з Інтернетом, а не загальні завдання програмування, такі як створення файлів. По-друге, сценарії обмежуються політикою того самого походження: скрипти з одного веб-сайту не мають доступу до такої інформації, як імена користувачів, паролі чи файли cookie, що надсилаються на інший сайт. Більшість помилок безпеки, пов’язаних з JavaScript, є порушенням тієї самої політики походження або пісочниці.

**HTML** - це стандартна мова розмітки документів, призначених для відображення у веб-браузері. Веб-браузери отримують документи HTML з веб-сервера або з локальної пам’яті. HTML описує структуру веб-сторінки семантично та включає ознаки зовнішнього вигляду документа.

**Елементи HTML** – це такі елементи, що дозволяють будувати HTML-сторінок. Використовуючи дані HTML конструкції зображення та інші об’єкти, такі як інтерактивні форми, можуть бути вбудовані у сторінку. Елементи HTML записуються у вигляді тегів з використанням кутових дужок. Такі теги, як <video /> та <input type=”checkbox” />, безпосередньо додають новий вміст на веб-сторінку. Інші теги, такі як <h1>, оточують та надають інформацію про текст документа і можуть включати інші теги як вкладені елементи. HTML забезпечує створення структурованих документів, визначаючи структурну семантику тексту, наприклад посилання, абзаци, заголовки, цитати, списки та інші елементи. Браузери не відображають теги HTML, але використовують їх для побудови фінального вмісту сторінки.

HTML може вбудовувати скрипти, написані мовою програмування, наприклад JavaScript, що впливає на поведінку та вміст фінальної веб-сторінок. Включення CSS визначає те, як сторінка буде виглядати та впливає на

розташування елементів.

Семантичний HTML - це спосіб написання HTML, який підкреслює значення закодованої інформації над її поданням. HTML включав семантичну розмітку з самого початку, але також включав презентаційну розмітку, таку як теги <font>, <i> та <center>. Існують також семантично нейтральні теги span і div. Коли каскадні таблиці стилів починали працювати в більшості браузерів, розробникам веб-застосунків пропонувалося уникати використання презентаційної розмітки HTML з метою логічного розділення презентації та вмісту.

Існують певні програмні агенти, що залежать від семантичної чіткості знайдених ними веб-сторінок, оскільки використовують різні техніки та алгоритми для читання та індексування мільйонів веб-сторінок на день та надання веб-користувачам засобів пошуку, без яких корисність Всесвітньої павутини значно зменшиться. Важливим типом веб-агенту, що виконує сканування та читання сторінок автоматично, без попереднього знання їх вмісту та того, що він може знайти, є веб-сканер або павук пошукової системи.

Для того, щоб павуки пошукових систем могли оцінювати значення певних фрагментів тексту, що знаходяться у документах HTML, а також для тих, хто створює змішувачі та інші гібриди, для більш автоматизованих агентів у міру їх розробки, існуючі семантичні структури в HTML потрібно широко використовувати та рівномірно застосовувати, щоб показати семантичне значення опублікованого тексту.

Презентаційні теги розмітки застаріли в поточних рекомендаціях HTML та XHTML. Більшість презентаційних функцій попередніх версій HTML більше не дозволяються, оскільки вони призводять до погіршення доступності, вищих витрат на обслуговування сайту та більших розмірів документів.

Хороший семантичний HTML також покращує доступність веб-документів. Наприклад, коли засіб зчитування з екрана або аудіо браузер може правильно встановити структуру документа, він не витратить час для людей із вадами зору,

читаючи повторну або неактуальну інформацію, якщо він був правильно розмічений.

**Каскадні таблиці стилів (CSS)** - це мова, яка використовується для опису зовнішнього вигляду документа, написаного мовою розмітки. CSS було створено для логічного розділення вигляду та вмісту, включаючи кольори, макет, шрифти. Це розділення дозволяє покращити лаконічність вмісту, забезпечити вищу гнучкість, дозволити декільком веб-сторінкам спільно використовувати форматування, вказавши відповідний CSS в окремому файлі .css, що зменшує складність та повторюваність структурного вмісту, а також дозволяє кешувати стилі, щоб покращити швидкість завантаження між сторінками, що ділять файл, та його форматування. Поділ форматування та вмісту також робить можливим подання однієї і тієї ж сторінки розмітки в різних стилях для різних методів візуалізації, таких як екран, друк, голосом (через мовний браузер або зчитувач екрана) та на основі Брайля, тактильні пристосування. CSS також має правила альтернативного форматування, якщо доступ до вмісту здійснюється на мобільному пристрої.

Каскадне ім'я походить із зазначеної пріоритетної схеми, щоб визначити, яке правило стилю застосовується, якщо більше одного правила відповідає певному елементу. Ця схема каскадного пріоритету передбачувана.

Кожен веб-браузер використовує механізм макетування для візуалізації веб-сторінок, і підтримка функціональності CSS. Оскільки браузери не аналізують CSS ідеально, було розроблено декілька методів кодування для орієнтування на певні браузери з обхідними шляхами (загальновідомими як хакі CSS або фільтри CSS). Впровадженню нових функціональних можливостей у CSS може заважати відсутність підтримки у основних браузерах. Наприклад, Internet Explorer повільно додавав підтримку багатьох функцій CSS 3, що сповільнювало використання цих функцій та погіршувало репутацію браузера серед розробників. Щоб забезпечити незмінний досвід для своїх користувачів, веб-розробники часто тестують свої сайти

в декількох операційних системах, браузерах та версіях браузерів, збільшуючи час розробки та складність. Такі інструменти, як BrowserStack, створені для зменшення складності обслуговування цих середовищ. На додаток до цих інструментів тестування, багато сайтів ведуть списки підтримки браузера для певних властивостей CSS, включаючи CanIUse та Mozilla Developer Network. Крім того, CSS 3 визначає функціональні запити, які забезпечують директиву @supports, яка дозволить розробникам націлювати браузери з підтримкою певної функціональності безпосередньо в їх CSS. CSS, який не підтримується старими браузерами, іноді також може бути виправлений за допомогою поліфілів JavaScript, які є фрагментами коду JavaScript, призначеними для послідовної поведінки браузерів. Ці обхідні шляхи - і необхідність підтримувати резервні функціональні можливості - можуть ускладнити проекти розробки, і, отже, компанії часто визначають список версій браузера, які вони підтримуватимуть і не підтримуватимуть.

Оскільки веб-сайти приймають нові стандарти коду, несумісні зі старими браузерами, ці браузери можуть бути відключені від доступу до багатьох ресурсів. Багато найпопулярніших веб-сайтів в Інтернеті не просто візуально погіршуються у старих браузерах через погану підтримку CSS, але і взагалі не працюють.

Деякі зазначені обмеження поточних можливостей CSS включають:

- Селектори не можуть піднятися

На даний момент CSS не пропонує жодного способу обрати батька чи предка елемента, який відповідає певним критеріям. CSS Selectors Level 4, який все ще перебуває у статусі Working Draft, пропонує такий селектор, але лише як частину повного профілю селектора, а не швидкий "живий" профіль, що використовується в динамічному стилі CSS. Більш вдосконалена схема вибору дозволить отримати більш складні таблиці стилів. Основні причини того, що Робоча група CSS раніше відхиляла пропозиції батьківських селекторів, пов’язані з продуктивністю браузера та проблемами додаткового рендерингу.

- Не можна явно оголосити нову область дії незалежно від позиції

Правила масштабування для властивостей, таких як z-index, шукають найближчий батьківський елемент із позицією. Це дивне зчеплення має небажані ефекти.

- Динамічна поведінка псевдокласу некерована

CSS реалізує псевдокласи, які дозволяють певний рівень зворотного зв'язку користувачів шляхом умовного застосування альтернативних стилів. Один псевдо-клас CSS, ":hover", є динамічним і має потенціал для неправильного використання, але CSS не має можливості для клієнта вимкнути його або обмеження її ефектів.

- Неможливо назвати правила

Неможливо назвати правило CSS, яке дозволило б клієнтським сценаріям посилатися на правило, навіть якщо його селектор змінюється.

- Не можна включати стилі з правила в інше правило

Стилі CSS часто повинні дублюватися у декількох правилах, щоб досягти бажаного ефекту, викликаючи додаткове обслуговування та вимагаючи більш ретельного тестування. Для вирішення цієї проблеми було запропоновано деякі нові функції CSS, але згодом від них відмовились. Неможливо націлити конкретний текст без зміни розмітки.

### 2.1.2. React

**React** - це відкрита, інтерфейсна бібліотека JavaScript для побудови користувацьких інтерфейсів або компонентів інтерфейсу. Він підтримується Facebook та спільнотою окремих розробників та компаній. React можна використовувати як основу при розробці односторінкових або мобільних додатків. Однак React займається лише управлінням станом та наданням цього стану DOM, тому для створення додатків React зазвичай потрібно використовувати додаткові бібліотеки для маршрутизації, а також певну функціональність на стороні клієнта.

Віртуальний DOM - ще одна помітна особливість. Її суть полягає у

використанні віртуальної об'єктної моделі документа або віртуального DOM. React створює кеш-структуру даних в пам'яті, обчислює отримані різниці, а потім ефективно оновлює відображуваний DOM браузера. Цей процес називається примиренням. Це дозволяє програмісту писати код так, ніби вся сторінка відображається при кожній зміні, тоді як бібліотеки React відображають лише підкомпоненти, які насправді змінюються. Цей вибірковий рендеринг забезпечує значний приріст продуктивності. Він економить зусилля з перерахунку стилю CSS, макета сторінки та рендерингу для всієї сторінки.

**React hooks** - це функції, які дозволяють розробникам "підключатись" до стану реагування та функцій життєвого циклу з функціональних компонентів. Хуки не працюють всередині класів - вони дозволяють використовувати React без класів.

React пропонує кілька вбудованих хуків, таких як useState, useContext, useReducer і useEffect. useState, useReducer та useEffect, які найчастіше використовуються, призначені для контролю стану та побічних ефектів відповідно. Існують правила хуків, які описують характерний шаблон коду, на який покладаються хуки. Це сучасний спосіб обробки стану за допомогою React.

1. Хуки слід викликати лише на верхньому рівні (не всередині циклів або операторів if);
2. Хуки слід викликати лише з функціональних компонентів React, а не з нормальних функцій або компонентів класу.

Хоча ці правила не можуть бути застосовані під час виконання, інструменти аналізу коду, такі як лінтери, можна налаштувати для виявлення багатьох помилок під час розробки. Правила застосовуються як до використання хуків, так і до реалізації спеціальних хуків, які можуть викликати інші вбудовані хуки.

React не намагається надати повну "бібліотеку програми". Він розроблений спеціально для побудови користувацьких інтерфейсів, тому не включає багато інструментів, які деякі розробники можуть вважати необхідними для створення додатка. Це дозволяє вибрати, яку бібліотеку розробник хоче виконувати для

вирішення завдання, забезпечення доступу до мережі або локального зберігання даних.

Для підтримки концепції React про односпрямований потік даних, архітектура Flux була розроблена як альтернатива популярній архітектурі модель-вигляд-контролер. Flux має дії, які надсилаються через центральний диспетчер до сховища, а зміни в сховищі передаються назад до компонентів. При використанні з React це розповсюдження здійснюється завдяки властивостям компонентів. З моменту своєї концепції Flux був замінений такими бібліотеками, як Redux та MobX. Потік – це варіант патерну спостерігач. Під час використання архітектури Flux компонент не модифікує отриману ним інформацію, але повинен передавати функції, що створюють дії, які відправляються для того, щоб тим чи іншим чином модифікувати сховище. Дія - це певний об'єкт, що описує подію, яка відбулася: наприклад, дія, що описує одного користувача, який додав іншого до друзів, може містити унікальний ідентифікатор користувача, ідентифікатор нового друга та тип USER_ADDED_FRIEND. Сховища, що зазвичай сприймаються як моделі, часто змінюють себе у відповідь на певну подію, отриману від диспетчера. З моменту його створення було створено багато реалізацій описаного архітектурного підходу, найвідомішим є Redux, який має одне сховище, що часто називається єдиним джерелом істини.

### 2.2. Серверна частина

Для побудови серверної частини було обрано Node.js та фреймворк Nest.js, який наближує побудову серверів використовуючи JavaScript до відомих та підтверджених часом підходів створення масштабованих серверних програм.

#### 2.2.1. Node.js

**Node.js** - це міжплатформене  середовище виконання JavaScript із відкритим вихідним кодом, яке працює на двигуні V8 і виконує код JavaScript поза веб-

браузером. Node.js дозволяє розробникам використовувати JavaScript для написання інструментів командного рядка та для сценаріїв на стороні сервера - запуску сценаріїв на стороні сервера для створення динамічного вмісту веб-сторінки до того, як сторінка буде відправлена у веб-браузер користувача. Отже, Node.js представляє парадигму "JavaScript скрізь", яка об'єднує розробку веб-додатків навколо однієї мови програмування, а не різних мов для сценаріїв на стороні сервера та клієнта.

Хоча .js - це стандартне розширення імені файлу для коду JavaScript, назва "Node.js" не посилається на певний файл у цьому контексті, а є лише назвою продукту. Node.js має керовану подіями архітектуру, здатну до асинхронного вводу-виводу. Ці варіанти дизайну спрямовані на оптимізацію пропускної здатності та масштабованості у веб-додатках з багатьма операціями введення / виведення, а також для веб-додатків у режимі реального часу (наприклад, програм спілкування в реальному часі та браузерних ігор). Раніше розподілений проект розробки Node.js керувався Node.js Foundation, і тепер він об’єднався з JS Foundation, щоб сформувати OpenJS Foundation.

Node.js дозволяє створювати веб-сервери та мережеві інструменти з використанням JavaScript та колекції "модулів", які обробляють різні основні функціональні можливості. Модулі передбачені для введення-виведення файлової системи, мережевих (DNS, HTTP, TCP, TLS / SSL або UDP), двійкових даних (буфери), криптографічних функцій, потоків даних та інших основних функцій. Модулі Node.js використовують API, призначений для зменшення складності написання серверних додатків. JavaScript є єдиною мовою, яку Node.js підтримує спочатку. Node.js в основному використовується для побудови мережевих програм, таких як веб-сервери. Найбільш істотна різниця між Node.js та PHP полягає в тому, що більшість функцій у блоці “до завершення” (команди виконуються лише після закінчення попередніх команд), тоді як функції Node.js не блокуються, команди виконуються одночасно або навіть паралельно.

Node.js офіційно підтримується на Linux, macOS та Microsoft Windows 8.1 та Server 2012. Наданий вихідний код також може бути побудований на операційних системах, подібних до тих, які офіційно підтримуються, або модифікований третіми сторонами для підтримки інших, таких як ОС NonStop та сервери Unix.

Node.js - це середовище виконання JavaScript, яке обробляє вхідні запити в циклі, що називається циклом подій.

- Внутрішні компоненти

Node.js використовує libuv для обробки асинхронних подій. Libuv - це абстрактний рівень для функціонування мережі та файлової системи як на Windows, так і на системах на базі POSIX, таких як Linux, macOS, OSS на NonStop та Unix.

- Потоки

Node.js працює на однопотоковому циклі подій, використовуючи неблокуючі виклики вводу-виводу, що дозволяє йому підтримувати десятки тисяч одночасних з'єднань, не несучи витрат на переключення контексту потоку. Дизайн спільного використання одного потоку серед усіх запитів, які використовують шаблон спостерігача, призначений для побудови дуже швидко реагуючих додатків, де будь-яка функція, що виконує введення-виведення, повинна використовувати зворотній виклик. Для розміщення однопотокового циклу подій Node.js використовує бібліотеку libuv, яка, у свою чергу, використовує пул потоків фіксованого розміру, який обробляє деякі неблокуючі асинхронні операції вводу-виводу.

Пул потоків обробляє виконання паралельних завдань у Node.js. Виклик головної функції потоку відправляє завдання до загальної черги завдань, з якої вони тягнуться і виконуються. За своєю суттю неблокуючі системні функції, такі як мережеві, перекладаються в неблокуючі сокети на стороні ядра, тоді як за блокуючі системні функції, такі як введення / виведення файлів, виконуються блокуючим способом у власних потоках. Коли потік у пулі потоків виконує завдання, він інформує про це основний потік.

Недоліком цього однопотокового підходу є те, що Node.js не дозволяє масштабуватися вертикально, збільшуючи кількість ядер процесора машини, на якій він працює, без використання додаткового модуля, такого як кластер, StrongLoop Process Manager, або pm2. Однак розробники можуть збільшити кількість потоків за замовчуванням у пулі потоків libuv. Серверна операційна система, ймовірно, розподілить ці потоки між кількома ядрами. Інша проблема полягає в тому, що тривалі обчислення та інші завдання, пов'язані з процесором, заморожують весь цикл подій до завершення.

**V8** - це механізм виконання JavaScript, який спочатку був створений для Google Chrome. Потім він був відкритий для Google у 2008 році. Написаний на C ++, V8 компілює вихідний код JavaScript до власного машинного коду під час виконання. Починаючи з 2016 року, він також включає інтерпретатор байт-коду.

**npm** - це попередньо встановлений менеджер пакетів для серверної платформи Node.js. Він встановлює програми Node.js з реєстру npm, організовуючи встановлення та управління сторонніми програмами Node.js. Пакети в реєстрі npm можуть варіюватися від простих допоміжних бібліотек, таких як Lodash, до запуску завдань, таких як Grunt.

Node.js підтримує WebAssembly, а станом на Node 14 має експериментальну підтримку WASI, системного інтерфейсу WebAssembly. Node.js забезпечує спосіб створення "аддонів" за допомогою API на основі C, що називається N-API, який можна використовувати для створення завантажуваних модулів .node із вихідного коду, написаного на C / C ++. Модулі можна безпосередньо завантажувати в пам'ять і виконувати з середовища JS як прості модулі CommonJS. Реалізація N-API покладається на внутрішні об'єкти C / C ++ Node.js і V8, що вимагають від користувачів імпорту конкретних заголовків Node.js у власний вихідний код. Оскільки платформа Node.js постійно розвивається, сумісність API може змінюватися і іноді може порушуватися новою версією. Для вирішення проблеми треті сторони ввели обгортки з відкритим кодом C / C ++ поверх API, які частково

полегшують проблему. Вони спрощують інтерфейси, але як побічний ефект вони також можуть внести складність, з якою доводиться мати справу операторам. Незважаючи на те, що основна функціональність Node.js знаходиться у вбудованій бібліотеці JavaScript, модулі, написані на C ++, можуть бути використані для поліпшення можливостей.

### 2.2.2. NestJs

**Nest (NestJS)** - це фреймворк для створення ефективних, масштабованих серверних додатків Node.js. Він використовує сучасний JavaScript та повністю підтримує TypeScript та поєднує елементи OOP (об'єктно-орієнтоване програмування), FP (функціональне програмування) та FRP (функціонально реактивне програмування).

Під капотом Nest використовує надійні HTTP Server імплементації, такі як Express (за замовчуванням), і за бажанням може бути налаштований на використання Fastify.

Nest забезпечує рівень абстракції над цими бібліотекамиNode.js, а також надає їх API безпосередньо розробнику. Це дає свободу використовувати безліч сторонніх модулів, доступних для базової платформи.

В останні роки, завдяки Node.js, JavaScript став мовою як для фронтових, так і для бекенд програм. Це призвело до створення чудових проектів, таких як Angular, React та Vue, які покращують продуктивність розробників і дозволяють створювати швидкі та розширювані фронтенд програми. Хоча для Node існує безліч чудових бібліотек, помічників та інструментів (і на стороні сервера JavaScript), жоден з них ефективно не вирішує головну проблему - архітектуру.

Nest надає нестандартну архітектуру додатків, яка дозволяє розробникам та командам створювати високотестовані, масштабовані, слабко пов'язані та легко обслуговувані додатки. Архітектура натхнена Angular.

### 2.2.3. MySQL

**MySQL** - це система управління реляційними базами даних із відкритим кодом. Його назва - це поєднання "My", імені дочки співзасновника Майкла Віденія та "SQL", скорочення від Structured Query Language. Реляційна база даних організовує дані в одну або кілька таблиць, в яких типи даних можуть бути пов'язані між собою. Ці відносини допомагають структурувати дані. SQL - це мова, яку програмісти використовують для створення, модифікації та вилучення даних з реляційної бази даних, а також контролю доступу користувачів до бази даних. На додаток до реляційних баз даних та SQL, СУБД, як MySQL, працює з операційною системою для реалізації реляційної бази даних у системі зберігання комп'ютера, управляє користувачами, забезпечує доступ до мережі та полегшує тестування цілісності бази та створення резервних копій.[5]

MySQL - це безкоштовне програмне забезпечення з відкритим кодом на умовах Загальної публічної ліцензії GNU, а також доступне під різними патентованими ліцензіями. MySQL належав і фінансувався шведською компанією MySQL AB, яку придбала Sun Microsystems. У 2010 році, коли Oracle придбала Sun, Widenius роздвоїв проект MySQL з відкритим кодом для створення MariaDB.

MySQL має окремі клієнти, які дозволяють користувачам безпосередньо взаємодіяти з базою даних MySQL за допомогою SQL, але частіше він використовується з іншими програмами для реалізації додатків, які потребують можливості реляційних баз даних. MySQL є компонентом стеку програмного забезпечення для веб-додатків LAMP, що є абревіатурою для Linux, Apache, MySQL, Perl / PHP / Python. MySQL використовується багатьма веб-програмами, керованими базами даних, включаючи Drupal, Joomla, phpBB та WordPress. MySQL також використовується багатьма популярними веб-сайтами, включаючи Facebook, Flickr, MediaWiki, Twitter та YouTube.

### 2.2.4. Docker

**Docker** - це набір продуктів платформи як послуги (PaaS), які використовують віртуалізацію на рівні ОС для доставки програмного забезпечення в пакетах, званих контейнерами. Контейнери ізольовані один від одного і об'єднують власне програмне забезпечення, бібліотеки та конфігураційні файли. Вони можуть спілкуватися між собою за чітко визначеними каналами. Оскільки всі контейнери користуються послугами одного ядра операційної системи, вони використовують менше ресурсів, ніж віртуальні машини.

Послуга має як безкоштовний, так і преміум-рівень. Програмне забезпечення, яке розміщує контейнери, називається Docker Engine. Він був створений у 2013 році та розроблений Docker, Inc.

Docker може упакувати програму та її залежності у віртуальний контейнер, який може працювати на будь-якому комп'ютері Linux, Windows або macOS. Це дозволяє додатку працювати в різних місцях, наприклад, локально, у відкритій хмарі або у приватній хмарі. При запуску в Linux Docker використовує функції ізоляції ресурсів ядра Linux та функціональну файлову систему, що дозволяє контейнерам запускатися в межах одного екземпляра Linux, уникаючи накладні витрати на запуск та обслуговування віртуальних машин. Docker на macOS використовує віртуальну машину Linux для запуску контейнерів.

З самого моменту створення контейнери Docker мають невелику вагу, на одному сервері може працювати кілька різних контейнерів одночасно, що дозволяє економити ресурси. Аналіз 2018 року показав, що типовий випадок використання Docker передбачає запуск восьми контейнерів на хост і що чверть аналізованих організацій запускає 18 і більше на хост. Підтримка простору імен ядра Linux в основному ізолює погляд програми на операційне середовище, включаючи дерева процесів, мережу, ідентифікатори користувачів та змонтовані файлові системи, тоді як групи ядра забезпечують обмеження ресурсів для пам'яті та центрального процесора. Починаючи з версії 0.9, Docker включає власний компонент для

безпосереднього використання засобів віртуалізації, що надаються ядром Linux, на додаток до використання абстрагованих інтерфейсів віртуалізації через libvirt, LXC та systemd-nspawn. Docker реалізує API високого рівня для забезпечення легких контейнерів, які запускають процеси ізольовано.

### 2.3. Обґрунтування вибору технологій

Для розробки проекту було обрано такі технології, як React, NestJS, MySQL, Docker. Інструменти, які вони надають, чітко співпадали з потребами додатку, який проектувався та надавали той функціонал, який був потрібен для реалізації потреб застосунку і зменшив кількість написаного коду в декілька разів, що позитивно вплинуло на розробку, та вплине на майбутню підтримку і модифікацію.

#### 2.3.1. Переваги React

- **Економність** - оскільки немає необхідності починати витрачати великі гроші з самого початку, а дана бібліотека являється абсолютно безкоштовним та швидким у розробці рішенням.
- **Чудовий досвід користувачів** - ніхто не любить стояти в черзі, і очікувати, поки веб-сайт завантажиться. Це погано впливає на взаємодію з користувачем, продукт втрачає клієнтів. React має функції, які роблять програми надзвичайно швидкими: паралельний режим, React Fiber, Suspense, віртуальний DOM.
- **Популярність** - React являється стабільною та зрілою технологією. З роками його тільки покращують і вносять нові корисні функції.
- **Підвищення продуктивності та швидкості** - коли мова заходить про такі технології, як бібліотеки чи фреймворки, продуктивність та швидкість є запорукою успіху. Якщо React може впоратися з Facebook, високонавантаженим застосунком, він може впоратися з чим завгодно. І причиною того, що React настільки швидкий, є віртуальний DOM. Замість того, щоб кожен раз будувати всі html елементи, він шукає відмінності між старим DOM і поточним та відповідно оновлює його.

- **SEO-зручний** - не тільки Facebook любить React, але і Google боти також. Хоча пошукові системи мають проблеми з читанням важких програм JavaScript, для програм React це інше. Це тому, що вони можуть працювати на сервері, тоді як віртуальний DOM дбає про рендеринг та взаємодію у браузері.
- **Скорочений час виходу на ринок (TTM)** - час - це гроші, особливо для стартапів та інших компаній, метою яких є створення цифрових продуктів. Оскільки React пришвидшує весь процес розробки, можна набагато швидше протестувати MVP на ринку та внести зміни відповідно, не витрачаючи час.

### 2.3.2. Переваги Node.js та NestJS

- **Швидкість**. Тести продуктивності toptal.com це підтверджують, порівнявши, як GO, PHP, Java та Node.js обробляють одночасні запити.
- **Двигун V8**. Механізм, що використовується в реалізації Node.js, спочатку був розроблений для браузера Chrome. Написаний на C++, Chrome V8 використовується для компіляції скриптів, написаних на JavaScript, у машинний код, і він виконує цю роботу з вражаючою швидкістю. Завдяки серйозним інвестиціям Google у свій двигун, V8 щороку демонструє підвищення продуктивності, а Node.js витягує з цього цілий пакет переваг.
- **Неблокуючий вхід / вихід та асинхронна обробка запитів** зробили Node.js здатним обробляти запити без будь-яких затримок. У контексті бекенда синхронна обробка передбачає, що код виконується в послідовності. Таким чином, кожен запит блокує потік, змушуючи інші запити чекати його закінчення. Асинхронна обробка дозволяє обробляти запити, не блокуючи (не блокуючи введення / виведення) потік. Отже, після обробки запиту він може витіснити зворотний дзвінок і продовжити обслуговування запитів. Це допомагає Node.js максимально використовувати один потік, що призводить до короткого часу відгуку та одночасної обробки.
- **Модель подій**. При використанні загальної мови як на стороні клієнта, так і

на сервері, синхронізація відбувається швидко, що особливо корисно для додатків реального часу на основі подій. Завдяки своїй асинхронній, однопоточній природі, Node.js є популярним вибором для онлайн-ігор, чатів, відеоконференцій або будь-якого рішення, яке вимагає постійно оновлюваних даних.

### 2.3.3. Переваги MySQL

- **MySQL безкоштовний** - можна безкоштовно завантажити ком'юніті версію MySQL або MariaDB з відкритим кодом;
- **Простий в установці** - в операційній системі, яка має диспетчер пакетів, лише за одну команду mysql буде готовий;
- **Легкість** - це означає, що субд не потребує великої кількості пам'яті вашого комп'ютера або використання центрального процесора, MySQL потребує мінімальної кількості комп'ютерних ресурсів;
- **Велика спільнота** – велика кількість технічної документації в Інтернеті, яка продовжує зростати. MySQL не застаріває та розвивається, кількість його користувачів постійно зростає і додаються деякі додаткові функції, покращуються продуктивність та безпека.

### 2.4. Висновок до розділу 2

Обрано основні технології та мови програмування для створення робочого прототипу системи, які повністю покривають всі потреби застосунку. Їх вибір обґрунтовано та описано основні плюси їх використання у побудову реального продукту.

# 3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ТА РЕЗУЛЬТАТИ НАУКОВОГО ДОСЛІДЖЕННЯ

Для коректної реалізації програмного забезпечення спочатку було визначено вимоги та спроектовано архітектуру майбутнього додатку. В цьому розділі розглянуто основні типи вимог до програмного забезпечення та конкретні вимоги до автоматизованої системи роботи з персоналом на підприємствах, архітектура клієнтської, серверної частини та бази даних, конкретна реалізація продукту з базовим функціоналом, а також результати дослідження актуальності та новизни даної імплементації.

## 3.1. Визначення вимог до програмного забезпечення

Бізнес-вимоги описують основні переваги, які нова система надаватиме її замовникам, покупцям і користувачам. Вони мають безпосередній вплив на те, які, призначені для користувача, вимоги будуть реалізовані і в якій послідовності.

Визначено наступні бізнес-вимоги:

1. Покращити управління процесами пов’язаними з персоналом на підприємстві;
2. Підвищення комфорту вирішення організаційних моментів працівниками;
3. Спрощення комунікації між співробітниками та керівництвом в умовах дистанційної праці.

При проектуванні програмного продукту важливо чітко визначити те, яким чином користувач буде працювати з додатком. Для цього створюються вимоги користувачів, що описують набір задач користувачів, які будуть вирішуватись додатком, а також способами їх вирішення у системі.

Отже, було визначено такі вимоги користувача:

1. Авторизація та реєстрація нових користувачів;
2. Видання ролей користувачам(Менеджер, Штатний співробітник, т.д.);
3. Створення vacation, salary реквестів;

4. Зміна підрозділу або команди конкретного працівника;
5. Управління поточними задачами працівників;
6. Комунікація з іншими користувачами(діалог).

Після функціональних вимог доцільно було б перейти до нефункціональних. Після визначення функціоналу, який повинен бути впроваджений, було виділено деякі системні вимоги, а саме:

1. Наявність підключення до мережі інтернет;
2. Доступ до підсистеми, яка належить конкретному підприємству;
3. Обладнання, яке може запускати сучасні веб-браузери.

При подальшому розвитку продукту також можна занести до системних вимог і знаходження надійного хостингу, який надасть доступ до сервісу користувачам по всьому світу та забезпечить швидку взаємодію із застосунком.

### 3.2. Проектування програмного забезпечення

При проектуванні важливу роль відіграє створення гнучкої та якісної архітектури. Для веб-застосунків потрібно побудувати архітектуру клієнту та архітектуру серверу. Архітектура клієнту буде описана діаграмою класів, а архітектура серверу - графічним представленням взаємодії його модулів один з одним, адже технологія, яка була обрана – нав'язує свою архітектуру.

#### 3.2.1. Архітектури клієнту

React не визначає архітектуру застосунку, тому під час розробки клієнту було обрано підхід, який має назву The Clean Architecture.[6] Використовуючи дану архітектуру застосунок ділиться на шари, які взаємодіють між собою певним чином(див. Рис. 3.1.).

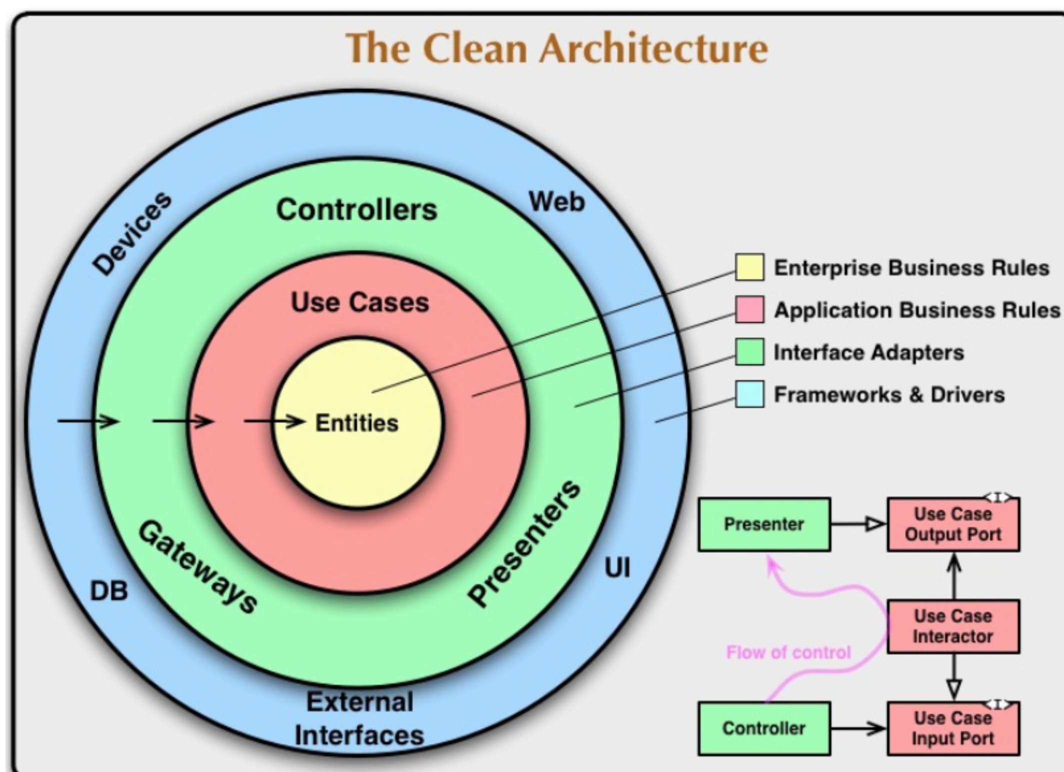



Рис. 3.1. The Clean Architecture.

Основні компоненти системи:

1. Модуль авторизації – відповідає за рівень доступу до ресурсів, авторизацію та автентифікацію користувачів, а також їх реєстрацію;
2. Модуль реквестів – відповідає за створення запитів на зміну заробітньої плати, планування відпусток.
3. Модуль задач – відповідає за поточні задачі персоналу, їх деталі та прогрес виконання.
4. Модуль заголовку програми - відповідає за заголовок;
5. Модуль діалогу - відповідає за комунікацію між користувачами;
6. Shared модуль - загальний модуль, якій зберігає в собі компоненти та сервіси, що використовуються по всій програмі.
7. App модуль - точка входу в програму, відповідає за первинні налаштування, потрібні для роботи всіх інших модулів.

Модулі, пов’язані безпосередньо із роботою з персоналом мають різні варіації для різних користувачів, адже штатний робітник не може давати задачі іншим, це

прерогатива менеджера або керівника підрозділу чи команди. Таким чином система побудована на ролях користувачів та кількості допустимих функцій для кожної з них. Детальніше архітектура клієнту передана через UML-діаграму(див. Додаток А).

### 3.2.2. Архітектура серверу

Архітектура серверу базується на модульному підході фреймворку NestJS. Таким чином наш застосунок ділиться на модулі, які відповідають кожен за свою частину функціоналу. З основних компонентів системи можна виділити модуль бази даних, який надає змогу взаємодіяти зі сховищем та виконувати SQL queries (див. Додаток Д.1.). Модуль авторизації відповідає за реєстрацію, авторизацію та автентифікацію кожного користувача (див. Додаток Д.2.).

Всі інші бізнес-модулі імпортують його та використовують для власних потреб.

Основні функціональні модулі надають програмний інтерфейс для взаємодії з об'єктами системи.

Клієнт звертається до серверу, використовуючи HTTP протокол як основний спосіб отримання та передачі даних. Також для діалогу між користувачами побудовано модуль Real Time Communication, який використовує протокол WebSocket для передачі повідомлень у реальному часі. Також ці повідомлення заносяться в базу, що надає змогу користувачам повертатись до них, та тим чи іншим чином використовувати цю інформацію.

Детальніша архітектура серверної частини представлена графічно (див. Додаток Б).

### 3.2.3. Проектування бази даних та створення міграції

Архітектура бази даних складається з основних сутностей системи та зв’язків між ними. Таблиця «користувач» зберігає інформацію про кожного

зареєстрованого юзера та його інформації. Таблиця «працівник» зберігає інформацію про команду, підрозділ, заробітну платню, позицію та роль конкретного співробітника. Таблиця «реквест» зберігає інформацію про всі створені в системі запити на відпустку чи зміню заробітної платні. Кожен запис даної таблиці зберігає ідентифікатор користувача, за яким ми визначаємо хто був автором даного реквесту. Також об’єкти в цій колекції зберігають значення, яке описує стан реквесту: pending, approved або declined. Створена міграція для наповнення бази під час локальної розробки(див. Додаток В).

### 3.3. Життєвий цикл програмного забезпечення

Під час реалізації даної системи було використано “Спіральну модель”, як життєвий цикл пз. Даний підхід ідеально підходить для даного проекту: чіткі межі не визначено і функціонал може стрімко змінюватись та поповнюватись новими інструментами, а з новим функціоналом з’являються нові ризики, які можуть вплинути на подальший розвиток продукту. Також система повинна адаптуватись під кожне конкретне підприємство тим чи іншим чином, тому з новими користувачами буде додаватись новий функціонал, який не можна було спланувати до побудови готового продукту.

Життєвий цикл розділено на етапи:

1. визначення загального концепту – потрібно визначити базовий функціонал, який буде покривати потреби більшості підприємств та реалізувати його в першу чергу;
2. планування першої ітерації – планування розробки базового функціоналу та визначення часових затрат;
3. фаза дизайну – реалізація основної частини інтерфейсу у вигляді макетів;
4. конструювання та тестування - розробка та тестування робочого прототипу;
5. підтримка – аналіз відгуків, який можна проводити з появою нових

користувачів.

### 3.4. Результати наукового дослідження

Для визначення актуальності даної системи та її переваг перед конкурентами було проведено аналіз існуючих рішень та їхніх переваг і недоліків.

#### 3.4.1. Аналіз конкурентів

- **Square**

Square, відомий своїми гнучкими інструментами, остання реалізація пропонує функції HR. Співробітники можуть заходити з будь-якого пристрою. Платні та неоплачувані перерви відстежуються, понаднормові роботи розраховуються автоматично, а часові картки точно підраховуються. Одним з основних плюсів використання Square є величезний асортимент варіантів інтеграції на їхньому ринку: WooCommerce, GoDaddy, Xero, Quickbooks, OpenCart, Chargify, Appointy, Sku IQ, Craver, GoGoGuest, Homebase, Tsheets, JotForm, EventOffice, Eventzilla, Mailchimp та багато іншого. Через залежність від смартфонів та інших портативних пристроїв Square може не підходити для галузей високого ризику через проблеми безпеки. Його можна використовувати безкоштовно, хоча вони зберігають частину кожного зібраного вами платежу.

- **Sagar Informatics**

Sagar Informatics - це програмне забезпечення для управління персоналом, яке містить запис призначених завдань, аналіз даних співробітників, моніторинг співробітників, централізовану базу даних співробітників, управління робочим аркушем та табелями робочого часу та відстеження часу очікування. Дане програмне забезпечення має хорошу підтримку, проблеми вирішуються негайно електронною поштою або телефоном, і вони мають кілька номерів, за якими можна проконсультуватись або отримати допомогу. Інтерфейс користувача Sagar Informatics трохи незграбний, незграбно забарвлений та застарілий у своєму

зовнішньому вигляді. Sagar Informatics пропонує пакети в різній ціновій категорії та має безкоштовну демо версію.

- **Sap SuccessFactor**

Sap SuccessFactor - це інструмент управління персоналом, який пропонує гнучкі опитування співробітників, інформаційні панелі та звітність на основі ролей та програми винагород співробітників. Програмне забезпечення також має безкоштовну пробну версію, гарантуючи, що потенційні клієнти платять лише за той інструмент, який їм треба. Існує певна проблема узгодженості з різними модулями, які вони пропонують, які колись були окремими продуктами. Зв'язок від одного елемента модуля до наступного часто є незграбним, що призводить до додаткового меню або входу. Платна версія Sap SuccessFactors починається з $8 за користувача на місяць.

- **Workday**

Workday – це система управління працівниками для середнього та великого бізнесу, яка пропонує інструменти для набору та розвитку талантів команди з такими функціями, як візуалізація підрахунку голосів, аналіз фінансових тенденцій та підтримка таких норм, як випробувальний термін. Workday надзвичайно зручний і має чистий та зручний інтерфейс.

- **PeopleSoft**

PeopleSoft - це рішення SaaS, що належить Oracle, з інструментами для управління відсутністю співробітників, компенсацією, довідковою службою з питань персоналу, інтерфейсом з оплатою праці, адмініструванням пенсій, ефективністю та наймом, а також відстеженням часу та праці. PeopleSoft надзвичайно простий у використанні, з мінімальним тренуванням, необхідним для запуску програмного забезпечення. Крім того, можуть виникати деякі проблеми з частими простоями та тайм-аутом, що іноді призводить до неможливості увійти до продукту.

### 3.4.2. Порівняння з розробленим ПЗ

На відміну від розробленого ПЗ більшість аналогів фокусується на часовому трекінгу. Такий підхід тільки накладає обмеження, що зменшує ефективність кадрів. Розроблене ПЗ акцентує увагу на взаємодії персоналу з різними сферами робочого життя (відпустка, заробітна платня, підписування документів, т.д.) і одночасно надає змогу керівництву формувати задачі, керувати складом підрозділів та команд і спілкуватись зі своїми співробітниками напряму, використовуючи вбудований чат. Таким чином, реалізовуємо внутрішню взаємодію кадрів та компанії зручнішої не тільки для керівництва, а й для самих робітників, що дозволяє побудувати ефективну команду.[7]

## 3.5. Висновок до розділу 3

Було спроектовано та реалізовано автоматизовану систему роботи з персоналом на підприємствах, яка надає змогу користувачам спілкуватися між собою, створювати запити на відпустку та перегляд заробітної платні, оформлювати задачі, слідкувати за їх виконанням, контролювати склад відділів та команд в компанії. Систему було збудовано достатньо гнучко, завдяки модульному підходу, тому при додаванні нового функціоналу та розширенні старого істотних архітектурних проблем виникнути не повинно. Також було проведено аналіз наявних рішень, а також досліджено вплив різних обмежень, які накладаються конкурентами на ефективність працівників.

# ВИСНОВКИ

У процесі виконання випускної кваліфікаційної магістерської роботи було досліджено актуальність розроблюваного застосунку. Основними перевагами реалізованої системи є:

1. Орієнтованість на задоволення побажань працівників, що призводить до більш ефективної роботи.
2. Гнучкість архітектури, яка дозволяє системі налаштуватись на продуктивну взаємодію користувачів та функцій продукту на будь-якому підприємстві.
3. Заздалегідь закладена ідея використання системи в умовах дистанційної роботи, що являється актуальним для сучасного світу.

Проаналізовано функціонал основних конкурентів та визначено слабкі місця, які дозволили покращити розроблюваний продукт.

В ході виконання роботи:

1. досліджено теоретичні основи побудови і програмної реалізації веб-застосунків;
2. проаналізовано рішення та технології систем для роботи з персоналом на ринку;
3. розроблено прототип програмної системи для автоматизації роботи з персоналом на підприємствах з використанням наступних технологій: React, NestJS, MySQL, Node.js.

В ході реалізації програмного продукту отримано практичний досвід роботи з NestJS для побудови гнучких та швидких систем на основі однопоточного рішення Node.js. Розроблений застосунок може працювати на більшості операційних систем, адже для його роботи потрібен лише браузер.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Web Application Architecture [Електронний ресурс]. – Режим доступу: https://www.altexsoft.com/blog/engineering/web-application-architecture- how-the-web-works/#:~:text=Web%20application%20architecture%20is%20a,established %20by%20web%20application%20architecture.

2) Web Apps Details [Електронний ресурс]. – Режим доступу: - https://hackr.io/blog/web-application-architecture-definition-models-types- and-more

3) Дино Эспозито Разработка современных веб-приложений: анализ предметных областей и технологий: Вильямс, 2017. – 464 с.

4) JS [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/JavaScript

5) MySQL [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/MySQL

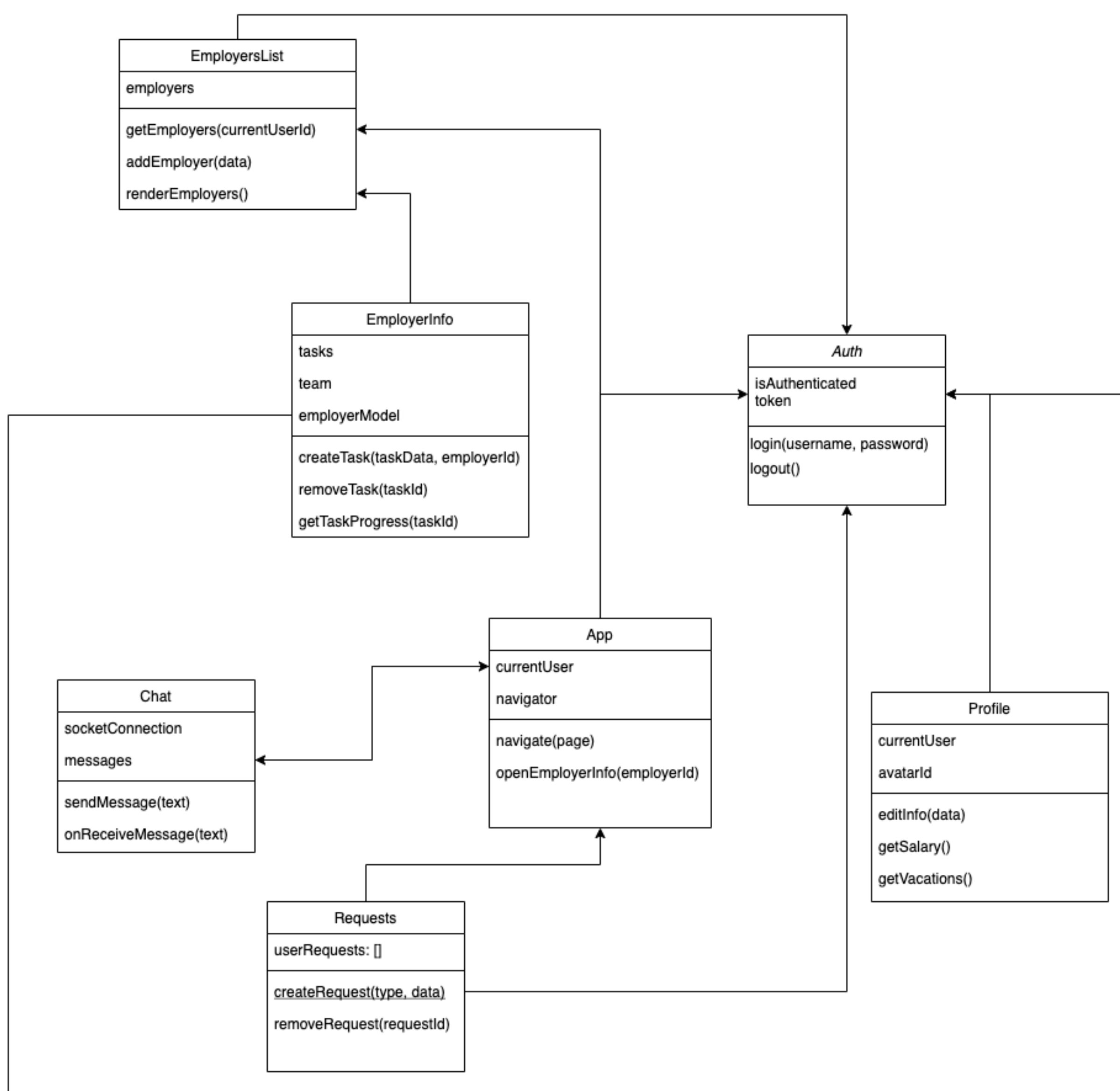
6) The Clean Architecture [Електронний ресурс]. – Режим доступу: https://habr.com/ru/post/499078/

7) Employee management [Електронний ресурс]. – Режим доступу: https://biz30.timedoctor.com/employee-management/

# ДОДАТКИ

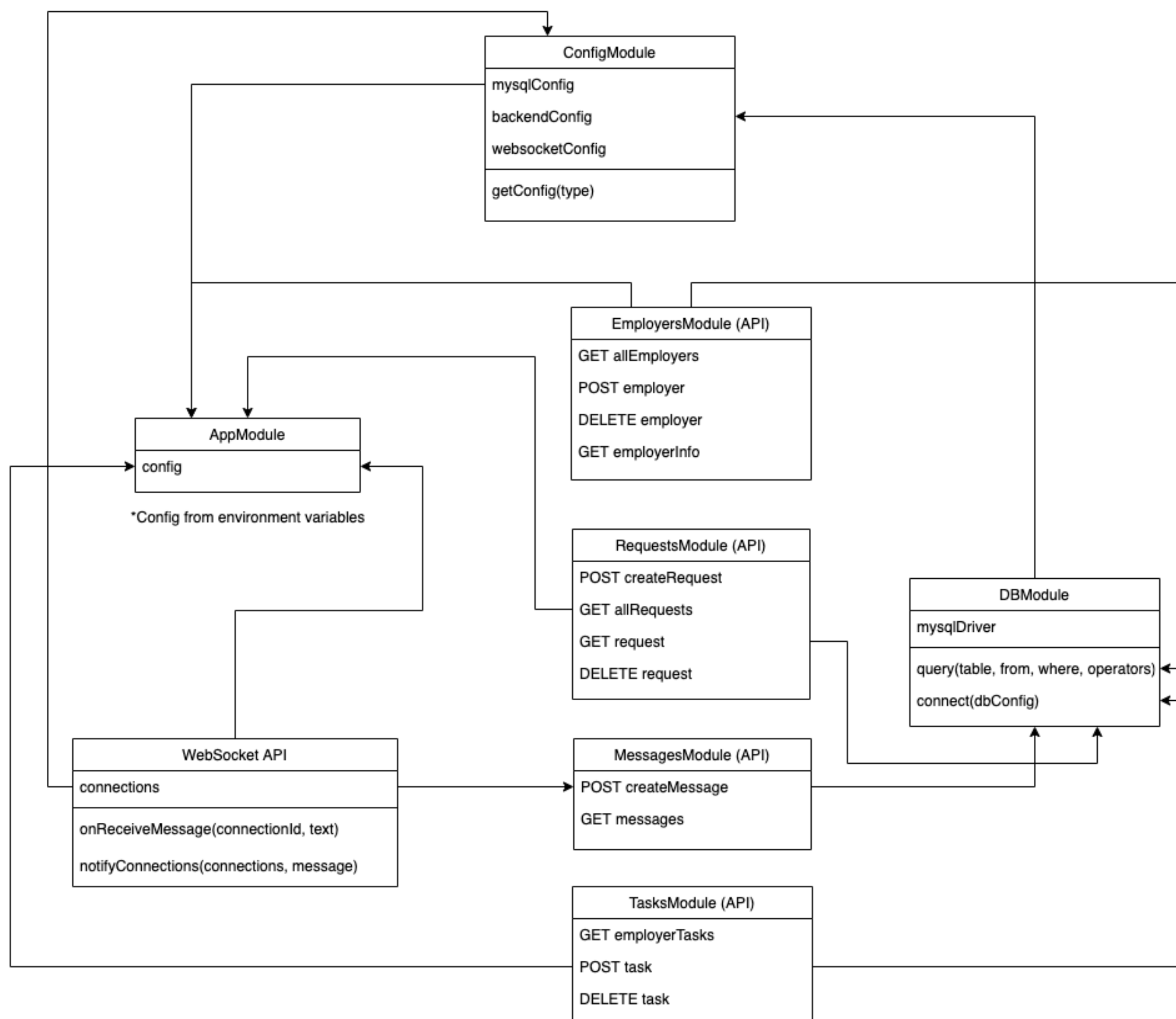
## Додаток А

## Архітектура клієнту



# Додаток Б

## Архітектура серверу



**Додаток В**

**Міграції бази даних**

```
CREATE DATABASE IF NOT EXISTS logistics;
GRANT ALL ON logistics.* TO 'logistics'@'%';

USE logistics;

CREATE TABLE USERS
(
   id       varchar(150) primary key,
   email    varchar(100) unique not null,
   phone    varchar(30) unique  not null,
   password varchar(200)        not null,
   isAdmin  bool DEFAULT FALSE
);
INSERT INTO USERS(id, email, password, phone, isAdmin)
VALUES (uuid(),
        'test@gmail.com',

'ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87621b6c233b548baeae6956
df346ec8c17f5ea10f35ee3cbc514797ed7ddd3145464e2a0bab413',
        '+380000000000',
        DEFAULT),
       (uuid(),
        'test2@gmail.com',
```

```
'ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87621b6c233b548baeae6956
df346ec8c17f5ea10f35ee3cbc514797ed7ddd3145464e2a0bab413',
        '+380000000001',
        DEFAULT),
       (uuid(),
        'admin@gmail.com',

'ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87621b6c233b548baeae6956
df346ec8c17f5ea10f35ee3cbc514797ed7ddd3145464e2a0bab413',
        '+380000000002',
        TRUE);

CREATE TABLE EMPLOYERS
(
    id         varchar(150) primary key,
    address    varchar(200),
    department varchar(100),
    birthday   date,
    role       varchar(100),
    managerId  varchar(150),
    salary     varchar(100),
    FOREIGN KEY (id) REFERENCES USERS (id)
);
INSERT INTO EMPLOYERS(id, address, department, role, birthday, managerId, salary)
VALUES ((SELECT id from users where email = 'test@gmail.com'),
        'Kyiv, Lomonosova 47',
        'department 1',
```

```
    'manager',
    STR_TO_DATE('20,5,1992', '%d,%m,%Y'),
    (SELECT id from users where email = 'admin@gmail.com'),
    '5000'),
   ((SELECT id from users where email = 'test2@gmail.com'),
    'Kyiv, Lomonosova 47',
    'department 2',
    'driver',
    STR_TO_DATE('12,8,1999', '%d,%m,%Y'),
    (SELECT id from users where email = 'test@gmail.com'),
    '650'),
   ((SELECT id from users where email = 'admin@gmail.com'),
    'Kyiv, Lomonosova 47',
    'department 1',
    'ceo',
    STR_TO_DATE('11,11,1984', '%d,%m,%Y'),
    null,
    '65000');

CREATE TABLE REQUESTS
(
  id               varchar(150) primary key,
  type             varchar(100) not null,
  createdAt        date,
  employerId       varchar(150),
  managerId        varchar(150),
  requestedSalary  varchar(100),
  vacationStartDate date,
```

```
   vacationEndDate   date,
   status     varchar(100) not null
);
INSERT INTO REQUESTS(id, type, createdAt, employerId, managerId,
requestedSalary, status)
VALUES (uuid(),
       'SALARY',
       CURRENT_DATE(),
       (SELECT id from users where email = 'test@gmail.com'),
       (SELECT employers.managerId from employers inner join users on employers.id =
users.id where users.email = 'test@gmail.com'),
       '1000',
       'PENDING');
```

**Додаток Д**

**Реалізація ключових модулів**

**Д.1.**

**Модуль бази даних**

```
import { DynamicModule, Global, Module } from '@nestjs/common';
import { AppConfig } from '../shared/interfaces/app-config.interface';
import { createConnection } from 'mysql2/promise';
import { DB_CONNECTION } from './constants';
import { UsersRepository } from '../users/users.repository';
import { DBQueryService } from './db-query.service';
import { EmployersRepository } from '../employers/employers.repository';
import { RequestsRepository } from '../requests/requests.repository';

@Global()
@Module({})
export class DatabaseModule {
  static async forRoot(config: AppConfig): Promise<DynamicModule> {
    const connectionProvider = {
      provide: DB_CONNECTION,
      useValue: await createConnection(config.mysql),
    };

    return {
      module: DatabaseModule,
      providers: [
        connectionProvider,
        DBQueryService,
```

```
        UsersRepository,
        EmployersRepository,
        RequestsRepository,
      ],
      exports: [
        UsersRepository,
        EmployersRepository,
        RequestsRepository,
      ],
    };
  }
}
import { Inject, Injectable } from '@nestjs/common';
import { DB_CONNECTION } from './constants';
import { Connection } from 'mysql2/promise';

export interface DBQueryApi {
  select: (options: SelectOptions) => Promise<any[]>;
  update: (options: UpdateOptions) => Promise<any>;
  insert: (options: InsertOptions) => Promise<any>;
}

interface JoinOptions {
  type?: string,
  tableToJoin: string,
  on: {
    firstTableColumn: string,
    secondTableColumn: string,
```

```
  }
}

interface SelectOptions {
  where?: { [key: string]: string },
  fields?: string[],
  clauses?: { [key: string]: string | number },
  join?: JoinOptions,
}

interface UpdateOptions {
  set: { [key: string]: string },
  where?: { [key: string]: string },
}

interface InsertOptions {
  insert: { [key: string]: string },
}

@Injectable()
export class DBQueryService {
  constructor(@Inject(DB_CONNECTION) private connection: Connection) {
  }

  query(table): DBQueryApi {
    return {
      select: async (options) => {
        const { where, clauses, join, fields } = options || {};
```

```
  const query = `
    SELECT ${DBQueryService.adaptFields(fields)}
    FROM \`${table}\`
    ${DBQueryService.adaptWhereMap(where)}
    ${DBQueryService.adaptJoin(join, table)}
    ${DBQueryService.adaptClausesMap(clauses)}
  `;
  return this.connection.query(
    query,
    where && [...Object.values(where)],
  ).then(([rows]) => rows as any[]);
},
update: async (options) => {
  const { where, set } = options;
  const query = `
    UPDATE \`${table}\`
    SET ${DBQueryService.adaptSetColumnsMap(set)}
    ${DBQueryService.adaptWhereMap(where)}
  `;
  return this.connection.query(
    query,
    set && [...Object.values(set), ...Object.values(where || {})],
  );
},
insert: async (options) => {
  const { insert } = options;
  const columns = Object.keys(insert);
  const query = `
```

```
        INSERT INTO \`${table}\` (${columns.reduce((str, col) => `${str}, ${col}`)})
        VALUES (${columns.map(() => '?').join(', ')})
      `;
      return this.connection.query(
        query,
        [...Object.values(insert)],
      ).then(([rows]) => rows as any[]);
    }
  };
}

private static adaptClausesMap(clauses) {
  if (!clauses) return '';
  return Object.keys(clauses)
    .reduce((str, clauseName) => `${str} ${clauseName.toUpperCase()}
${clauses[clauseName]}`, '');
}

private static adaptJoin(join: JoinOptions, table: string) {
  if (!join) return '';
  return `
    ${join.type?.toUpperCase() || 'INNER'} JOIN
    ${join.tableToJoin} ON ${table}.${join.on.firstTableColumn} =
${join.tableToJoin}.${join.on.secondTableColumn}
  `;
}

private static adaptFields(fields) {
```

```
    if (!fields) return '*';
    return fields.reduce((query, field, index) => {
      if (index === 0) return `${field}`;
      return `${query}, ${field}`;
    }, '');
  }

  private static adaptWhereMap(obj): string {
    if (!obj) return '';
    return Object.keys(obj).reduce((query, field, index) => {
      if (index === 0) return `${query} \`${field}\` = ?`;
      return `${query} AND \`${field}\` = ?`;
    }, 'WHERE ');
  }

  private static adaptSetColumnsMap(set): string {
    return Object.keys(set).reduce((str, column, index) => {
      const setColumnQuery = `${column}=?`;
      if (index === 0) return setColumnQuery;
      return `${str}, ${setColumnQuery}`
    }, '')
  }
}
```

**Д.2.**

**Модуль авторизації**

```
import { DynamicModule, Module } from '@nestjs/common';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { UsersModule } from '../users/users.module';
import { AuthService } from './auth.service';
import { JwtConfig } from '../shared/interfaces/jwt-config.interface';
import { JWT_CONFIG } from './constants';
import { JwtStrategy } from './jwt.strategy';
import { AuthController } from './auth.controller';
import { LocalStrategy } from './local.strategy';

@Module({})
export class AuthModule {
  static forRoot(jwtConfig: JwtConfig): DynamicModule {
    return {
      module: AuthModule,
      imports: [
        UsersModule,
        PassportModule,
        JwtModule.register({
          secret: jwtConfig.secret,
          signOptions: jwtConfig.signOptions,
        }),
      ],
      controllers: [AuthController],
      providers: [
```

```
      { provide: JWT_CONFIG, useValue: jwtConfig },
      AuthService,
      LocalStrategy,
      JwtStrategy,
    ],
    exports: [AuthService, JwtModule, JwtStrategy],
  };
 }
}
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { UsersService } from '../users/users.service';
import { User } from '../users/User';
import { sha512 } from '../shared/utils/crypto';

@Injectable()
export class AuthService {

 constructor(private usersService: UsersService, private jwtService: JwtService) {
 }

 async validateUser(email: string, password: string): Promise<User> {
  const user = await this.usersService.getUserByEmail(email);
  const hashedPassword = sha512(password);
  if (user && hashedPassword === user.password) {
   const { password, ...res } = user;
   return res;
  }
```

```
import { Injectable } from '@nestjs/common';
import { UserModel } from '../../users/User';
import { DBQueryApi, DBQueryService } from '../db-query.service';
import { TABLE } from '../constants';

@Injectable()
export class UsersRepository {
  private table = TABLE.users;
  private queryApi: DBQueryApi;

  constructor(private queryService: DBQueryService) {
    this.queryApi = this.queryService.query(this.table);
  }

  async getUserById(id: string): Promise<UserModel[]> {
    return this.queryApi.select({ where: { id } });
  }

  async getUserByEmail(email: string): Promise<UserModel[]> {
    return this.queryApi.select({ where: { email } });
  }

  async getUsers(): Promise<UserModel[]> {
    return this.queryApi.select();
  }
}
import { Injectable } from '@nestjs/common';
```

```
import { TABLE } from '../constants';
import { DBQueryApi, DBQueryService } from '../db-query.service';
import { Request, REQUEST_STATUS } from '../../requests/Request';

@Injectable()
export class RequestsRepository {
  private readonly table = TABLE.requests;
  private readonly queryApi: DBQueryApi;

  constructor(private dbQueryService: DBQueryService) {
    this.queryApi = this.dbQueryService.query(this.table);
  }

  async getEmployerRequests(employerId: string): Promise<Request[]> {
    return this.queryApi.select({ where: { employerId } });
  }

  async getPendingEmployerRequests(employerId: string): Promise<Request[]> {
    return this.queryApi.select({ where: { employerId, status:
REQUEST_STATUS.pending } });
  }

  async getManagerPendingRequests(managerId: string): Promise<Request[]> {
    return this.queryApi.select({ where: { managerId, status:
REQUEST_STATUS.pending } });
  }

  async changeRequestStatus(requestId: string, status: REQUEST_STATUS) {
```

```
    const where = { id: requestId };
    await this.queryApi.update({ where, set: { status } });
    return this.queryApi.select( { where });
  }

  async createRequest(request: Request) {
    await this.queryApi.insert({ insert: request as {} });
    return this.queryApi.select( { where: { id: request.id } });
  }
}import { Injectable } from '@nestjs/common';
import { TABLE } from '../constants';
import { DBQueryApi, DBQueryService } from '../db-query.service';
import { Chat } from '../../chats/Chat';

@Injectable()
export class ChatsRepository {
  private readonly table = TABLE.chats;
  private readonly queryApi: DBQueryApi;

  constructor(private dbQueryService: DBQueryService) {
    this.queryApi = this.dbQueryService.query(this.table);
  }

  async createChat(chat: Chat) {
    await this.queryApi.insert({ insert: chat as {} });
    return this.queryApi.select( { where: { id: chat.id } });
  }
```