МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «Додаток для відстеження та обліку робочого часу працівників»

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Дігалевич Іван Олександрович

Керівник: доктор філософії з прикладної математики, доцент, фахівець-практик
Красюк Богдан Віталійович

Рецензент: *кандидат технічних наук, доцент,*
доцент кафедри прикладної математики
Донецького національного університету
імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2025 р.

Острог, 2026

АНОТАЦІЯ

кваліфікаційної роботи

на здобуття освітнього ступеня бакалавра

Тема: Додаток для відстеження та обліку робочого часу працівників

Автор: *Дігалевич Іван Олександрович*

Науковий керівник: *Красюк Богдан Вікторович старший викладач кафедри економіко-математичного моделювання та інформаційних технологій.*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: ____*(кількість сторінок роботи) с.,* ____ *(кількість рисунків) рис.,* _____ *(кількість таблиць) табл.,* _____ *(кількість додатків) додатків,* _____ *(кількість джерел) джерел.*

Ключові слова: ВЕБЗАСТОСУНОК, ОБЛІК РОБОЧОГО ЧАСУ, HR-МЕНЕДЖМЕНТ, NEXT.JS, REACT.JS, NODE.JS, EXPRESS, PRISMA ORM, MONGODB, MERN СТЕК, АВТЕНТИФІКАЦІЯ, JWT, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, АВТОМАТИЗАЦІЯ, БАГАТОМОВНІСТЬ

Короткий зміст праці: У кваліфікаційній роботі розглянуто процес проєктування та розробки клієнт-серверного вебзастосунку для обліку робочого часу працівників, управління відпустками та лікарняними. Актуальність теми зумовлена стрімким переходом компаній на гібридний формат роботи та зростаючою потребою бізнесу у цифровізації і автоматизації HR-процесів, що приходять на зміну застарілим електронним таблицям. Метою роботи було створення комплексного рішення у форматі Full-stack monorepo, яке забезпечує безпечний, прозорий і безперервний цикл управління робочим часом та організаційною ієрархією компанії.

У теоретичній частині обґрунтовано вибір сучасного технологічного стеку та архітектурних підходів для побудови вебзастосунків. Як фундаментальну платформу серверної частини обрано середовище Node.js з використанням фреймворку Express.js для побудови REST API. Для безпечного та гнучкого зберігання облікових записів, історії нарахувань та пов’язаних даних використано документо-орієнтовану нереляційну базу MongoDB з інтеграцією об'єктно-реляційного відображення Prisma ORM. Архітектура клієнтської частини спирається на сучасний фреймворк Next.js 14 (App Router), що узгоджується з вимогами високої продуктивності, адаптивності та безпеки сучасних вебстандартів.

Практична частина присвячена реалізації ключового функціоналу продукту. У серверній логіці створено систему автентифікації та авторизації на основі JSON Web Tokens (JWT) для захисту доступу до ресурсів та розмежування ролей (ADMIN, EMPLOYEE); алгоритмізовано автоматичне нарахування днів відпустки та управління лікарняними. Клієнтський інтерфейс реалізовано на React.js із використанням Tailwind CSS: створено інтерактивний календар свят, систему багатомовності (i18n), механізми подачі та схвалення запитів на вихідні, а також дашборди зі статистикою. Серверну частину та базу даних розгорнуто на хмарних платформах Render та MongoDB Atlas, а фронтенд-застосунок — на Vercel.

Запропонована архітектура враховує вимоги до ізоляції користувацьких сесій, швидкодії та інформаційної безпеки й створює основу для подальшого масштабування та експлуатації застосунку в корпоративному середовищі.

The qualification work examines the design and development of a client-server web application for employee time tracking, vacation, and sick leave management. The topic is relevant due to the rapid shift towards hybrid work formats and the growing business demand for the digitalization and automation of HR processes, replacing outdated spreadsheets. The objective was to build a comprehensive full-stack monorepo application that supports a secure, transparent, and continuous loop of time and corporate hierarchy management.

The theoretical part justifies the choice of a modern technology stack and architectural patterns for web applications. Node.js with Express.js was selected as the server runtime for building the REST API. MongoDB combined with Prisma ORM provides flexible and type-safe storage for accounts, time-off requests, and related data. The client architecture is based on the Next.js 14 (App Router) framework, meeting modern requirements for performance, responsiveness, and security.

The practical part covers the implementation of the core product features. Server-side logic includes JWT-based authentication and role-based authorization (ADMIN, EMPLOYEE) to protect resources, plus algorithmic auto-accrual of vacation days and sick leave management. The user interface is implemented in React.js and Tailwind CSS, including an interactive holiday calendar, i18n support, time-off request workflows, and statistical dashboards. The backend and database are deployed on Render and MongoDB Atlas, while the frontend is hosted on Vercel.

The resulting architecture addresses session isolation, performance, and information security, providing a basis for further scaling and corporate operation of the application.

Зміст

ВСТУП

Інтенсивний розвиток хмарних технологій та глобальне проникнення автоматизованих систем управління персоналом (HRM) у професійну та корпоративну практику суттєво змінили спосіб, у який сучасні компанії отримують, обробляють та контролюють дані про робочий час і ресурси співробітників.

Якщо раніше облік відпусток, лікарняних та робочих змін часто покладався на використання розрізнених локальних програм або ручну роботу з веденням громіздких електронних таблиць, то сьогодні очікування бізнесу зміщуються до комплексних екосистем самообслуговування (**Employee Self-Service**): система має інтегрувати баланси днів, організаційну структуру, внутрішні комунікації та календарні графіки в єдиному цифровому просторі. Паралельно зростає вимога до швидкості відображення даних та прозорості внутрішніх HR-процесів, інтегрованих у повсякденний workflow працівників.

Веббраузер став основним робочим середовищем користувача, де концентруються корпоративні інструменти, документація та комунікаційні сервіси. Постійне перемикання між щоденними завданнями та сторонніми комунікаціями з менеджерами для з'ясування залишку відпустки чи подачі заявок порушує безперервність робочого процесу, збільшує кількість проміжних бюрократичних дій та формує додаткове навантаження на адміністративний апарат компанії.

Звідси випливає **інженерна проблема**: як надати співробітникам та адміністраторам доступний інтерактивний інструмент обліку робочого часу й управління відпустками безпосередньо через вебклієнт, зберігаючи при цьому високий рівень безпеки, гнучкість бізнес-логіки та строгу консистентність даних.

Корпоративні вебзастосунки нового покоління, зокрема ті, що базуються на архітектурі **Full-stack monorepo** та сучасних фреймворках, повинні ефективно поєднувати переваги серверного рендерингу й оптимізованої маршрутизації клієнтської частини із надійним, типізованим інструментарієм взаємодії з шаром персистентності. Для користувача-працівника це означає вимогу до швидкодії, миттєвого доступу до персональної статистики, прозорого відображення статусів

запитів та відсутності розривів між локальним станом інтерфейсу й централізованою базою даних. Для розробника постає необхідність свідомо проєктувати межі відповідальності в межах єдиного репозиторію: чітко розмежовувати реактивну логіку користувацьких дашбордів (*Employee/Admin Dashboards*) та серверні обчислювальні алгоритми.

Фундаментом стійкості такої системи залишається **серверна частина (backend)** — критичний рівень координації та збереження бізнес-правил. Саме на сервері зосереджують елементи, що визначають довгострокову якість продукту: верифікацію вхідних даних, контроль доступу на основі ролей (**RBAC**), виконання алгоритмів автоматичного нарахування вихідних днів, валідацію ієрархічних зв'язків та захист від деструктивних дій щодо критичних облікових записів адміністраторів. Централізоване управління логікою на сервері дозволяє впроваджувати чіткі бізнес-процеси, ізолювати конфіденційні дані та розвивати продукт без дублювання критичного коду в клієнтських скриптах.

Історичний досвід побудови корпоративних інформаційних систем показує, що хаотична організація взаємодії між клієнтом та базою даних призводить до надмірного навантаження на мережу та ускладнює отримання пов’язаних структур, таких як профілі користувачів разом із їхньою історією нарахувань, лімітами та даними керівників. Для сценаріїв, де фронтенд має чітко розділені за ролями представлення даних, використання структурованого **REST API** з централізованим клієнтом обміну даними дає можливість оптимізувати мережеві навантаження та спростити розробку.

У зв'язці середовища **Express.js** (сервер) та **Next.js 14 App Router** (клієнт) цей підхід підкріплюється ефективною маршрутизацією, контрольованим передаванням JWT-токенів та динамічним рендерингом інтерфейсу. Зберігання облікових записів, ієрархічних зв’язків, новин та статусів запитів доцільно реалізовувати у документо-орієнтованій нереляційній базі **MongoDB**, а програмне моделювання сутностей та строгу типізацію запитів — за допомогою **Prisma ORM**. Такий вибір ідеально узгоджується з **TypeScript**-природою сучасного вебстеку, усуває

невідповідності типів на межі бази даних і коду та спрощує ітеративний розвиток схем колекцій (наприклад, додавання історії *DaysAccrual*).

Безпека доступу до персональних та корпоративних даних потребує чіткої моделі довіри. У роботі обґрунтовується та реалізується підхід на основі **JSON Web Tokens (JWT)** як stateless-механізму автентифікації й авторизації з подальшою суворою перевіркою прав доступу ролей *ADMIN* та *EMPLOYEE* через спеціалізовані серверні middleware.

Обов'язковим елементом системи є **автоматизований розрахунок балансів**: динамічне обчислення доступних днів відпустки (із розрахунку 1.5 дня за відпрацьований місяць) та базових 10 днів лікарняних на рік з урахуванням індивідуальної дати початку роботи (*startDate*).

Клієнтська частина, побудована на **Next.js 14**, дозволяє реалізувати інтерактивний інтерфейс з урахуванням адаптивності та повної багатомовності (**i18n**) через React Context. Окремий практичний акцент робиться на інтерактивному календарі (на базі *react-calendar* та *date-fns*), який інтегрує відображення як затверджених періодів відпочинку працівника, так і офіційних українських державних свят, розрахованих за алгоритмом Гауса. Для підвищення зручності роботи з профілями впроваджено модуль обрізки зображень (*react-easy-crop*).

Розгортання серверної частини на хмарній платформі **Render** у поєднанні з хостингом фронтенду на **Vercel** та керованою базою даних **MongoDB Atlas** дозволяє наблизити інженерну реалізацію до реальних промислових умов експлуатації. Це узгоджується з загальною метою кваліфікаційної роботи — не обмежитися локальним прототипом, а продемонструвати цілісний життєвий цикл програмного продукту.

Метою кваліфікаційної роботи є системне проєктування, концептуалізація та практична реалізація відмовостійкого програмного продукту — повноцінної клієнт-серверної системи обліку робочого часу працівників **Employee Time Tracking (TimeTracker)**, орієнтованої на автоматизацію процесів нарахування та погодження відпусток, лікарняних, корпоративного інформування та управління організаційною ієрархією.

Для досягнення мети поставлено та декомпозовано такі завдання:

1. Проаналізувати предметну область HRM-систем та інструментів обліку часу, зіставити вимоги сучасного бізнесу до прозорості процесів з обмеженнями ручного обліку, обґрунтувати доцільність власного інтегрованого рішення у форматі monorepo.
2. Спроєктувати слабкозв’язану клієнт-серверну архітектуру: REST API сервер (Node.js + Express) та SPA/SSR клієнт (Next.js 14 App Router) з централізованим шаром мережевої взаємодії.
3. Розробити концептуальну та логічну модель даних для MongoDB, реалізувати схеми Prisma ORM з урахуванням цілісності, строгої типізації, зв'язків моделей користувачів, запитів на відпустки, новин та історії нарахувань.
4. Реалізувати REST API на Express.js як основний інтерфейс обміну даними; забезпечити централізовану обробку HTTP-маршрутів та розрахунок алгоритмів автоматичного нарахування вихідних днів працівників.
5. Імплементувати механізми автентифікації на базі JWT Bearer токенів та авторизації з розділенням ролей (*ADMIN, EMPLOYEE*) на рівні серверних middleware.
6. Реалізувати адаптивний користувацький інтерфейс на Next.js 14 з підтримкою тем, i18n локалізації (uk/en) через React Context, інтерактивним календарем свят та компонентами обробки зображень аватарів.
7. Забезпечити інтеграцію алгоритмів побудови організаційного дерева компанії на основі числових рівнів ієрархії (*gradeLevel*) та захисту від циклічних залежностей підлеглих.
8. Розгорнути клієнтську та серверну частини у хмарному середовищі (Vercel, Render, MongoDB Atlas), задокументувати конфігурацію, змінні оточення та підходи до супроводу.

Об’єкт дослідження — процеси алгоритмізації обліку робочого часу, управління станами в інтерфейсах корпоративних self-service порталів та реалізація безпечної клієнт-серверної комунікації в розподілених інформаційних системах.

Поняттєвий обсяг об’єкта охоплює маршрутизацію користувацьких та адмінських запитів, асинхронний розрахунок балансів відпусток і лікарняних днів, узгодження стану клієнтських дашбордів із централізованою базою даних, а також підтримання організаційної консистентності дерева підпорядкування працівників компанії.

Предмет дослідження — інженерні архітектурні моделі клієнт-серверної взаємодії у форматі monorepo, патерни побудови інтерфейсів на Next.js 14 App Router, інструментальні засоби серверної розробки на Node.js/Express, зокрема реалізація шару доступу через Prisma ORM, моделей MongoDB та механізмів безпеки на базі JWT і криптографічного хешування паролів.

Практична цінність роботи полягає у відтворюваному наборі архітектурних та програмних рішень для створення сучасної системи автоматизації HR-процесів компанії з прозорими алгоритмами нарахування часу, захищеною моделлю доступу та масштабованим інтерфейсом, що готовий до експлуатації та подальшого розширення.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ РОЗРОБКИ

1.1 Опис предметного середовища

Предметним середовищем даної кваліфікаційної роботи є сфера розробки корпоративних інформаційних систем, зокрема клієнт-серверних вебзастосунків для **управління персоналом (HRM)** та **автоматизованого обліку робочого часу (Time Tracking)**.

У сучасному інформаційному суспільстві та умовах переходу багатьох компаній на гібридний або повністю віддалений формат роботи, управління життєвим циклом співробітника еволюціонувало від простого кадрового діловодства до необхідності створення інтегрованих цифрових екосистем. Ефективність роботи компанії багато в чому залежить від того, наскільки швидко та прозоро працівник може взаємодіяти з організаційною структурою: дізнатися свій баланс відпустки, подати запит на лікарняний або переглянути корпоративні новини.

Традиційний підхід — використання розрізнених електронних таблиць (Excel, Google Sheets), паперових заяв та обмін десятками повідомлень у месенджерах — призводить до **десинхронізації даних**: розривається зв’язок між фактичною відсутністю працівника та розрахунками бухгалтерії, зростає кількість проміжних бюрократичних дій і знижується загальна продуктивність. Відповідно, інженерно доцільним є розробка централізованого вебзастосунку, який надає інструменти самообслуговування (**Employee Self-Service**) безпосередньо у браузері, зберігаючи при цьому контрольований канал валідації та доступу до бази даних через серверну інфраструктуру.

Створення такого продукту вимагає комплексного підходу: керування станом інтерфейсу та дашбордів, безпечна автентифікація (зокрема на основі **JWT**), узгоджена комунікація клієнта з backend через **REST API**, персистентне збереження облікових записів та історії нарахувань у **MongoDB** (через **Prisma ORM**), а також дотримання принципів адаптивного дизайну. Головна мета предметного середовища — забезпечити компанію надійним інструментом обліку часу та відпусток в інтегрованому робочому циклі, мінімізуючи рутину для HR-менеджерів та адміністраторів.

1.1.1 Аналіз ринку систем управління персоналом та обліку часу

Сучасний ринок HRM-систем переживає структурну трансформацію, пов’язану з переходом бізнесу на хмарні (Cloud-based) рішення. Новий стандарт посилює вимоги до доступності: системи повинні швидко працювати на будь-яких пристроях, мати інтуїтивно зрозумілий інтерфейс та інтегруватися з сучасними вебтехнологіями.

Паралельно відбувається відмова від важких, монолітних ERP-систем на користь легких, сфокусованих вебзастосунків. Користувачі очікують не просто таблиці з даними, а **контекстної візуалізації** — інтерактивних календарів, автоматичного розрахунку доступних вихідних на основі дати працевлаштування та миттєвого оновлення статусів запитів. Ринок формує попит на рішення, які поєднують швидкодію, прозору модель доступу (рольові моделі, журналювання ручних нарахувань), збереження історії у хмарі та стабільний UX.

Для інженерної реалізації це означає необхідність поєднати можливості сучасних React-фреймворків (зокрема **Next.js 14**) з надійним backend-шаром на **Node.js**, що обробляє бізнес-логіку нарахування днів та нормалізує відповіді для клієнта.

1.1.2 Функціональна модель системи та основні актори

Для розмежування прав доступу, мінімізації ризиків витоку корпоративних даних і прозорого моделювання відповідальності в межах застосунку **Employee Time Tracking** виокремлюються три ключові ролі (актори): **неавторизований користувач (Гість)**, **Працівник (EMPLOYEE)** та **Адміністратор (ADMIN)**.

1. **Актор «Гість» (Unauthenticated User)** — неавторизований користувач. Його можливості суворо обмежені сценаріями підготовки облікового запису та входу в систему: реєстрація та авторизація. Доступ до корпоративних новин, календаря, ієрархії та функцій подачі запитів без валідної сесії повністю заблокований на рівні серверної політики (middleware).
2. **Актор «Працівник» (EMPLOYEE)** — ідентифікований клієнт із підтвердженою сесією, що пред’являє валідний **JWT** у заголовку Authorization при зверненнях до REST API. Цей актор отримує доступ до основної цінності продукту: перегляд власної статистики вихідних днів, подача запитів на відпустку/лікарняний, взаємодія з календарем свят, читання новин та перегляд організаційного дерева (read-only).
3. **Актор «Адміністратор» (ADMIN)** — користувач із розширеними привілеями. Окрім базових функцій працівника, адміністратор має доступ до спеціалізованого дашборду. Його роль полягає в управлінні системою: схвалення або відхилення запитів співробітників, управління користувачами (CRUD-операції, зміна посад, рівнів ієрархії), ручне нарахування додаткових

днів відпустки/лікарняного, а також створення корпоративних новин і публікація державних свят.

1.1.3 Моделювання основного бізнес-процесу «Обробка запитів на відпустку/лікарняний»

Основним бізнес-процесом, що формує ключову користувацьку цінність, є **цикл подачі та опрацювання запиту на відсутність** (Time-Off Request). Процес описується як послідовність узгоджених кроків із чіткими точками контролю бізнес-правил та збереження даних.

На першому етапі працівник (EMPLOYEE) вибирає дати у модулі інтерфейсу та заповнює форму запиту (вказуючи тип: VACATION або SICK_LEAVE, та причину). Клієнтська частина на **Next.js** ініціює HTTP POST-запит до відповідного ендпоінту API, передаючи корисне навантаження та **JWT** для ідентифікації сесії. Система фіксує запит у базі даних зі статусом **PENDING**.

На другому етапі запит обробляє бекенд у середовищі **Node.js/Express**: виконується верифікація токена та валідація дат (дати не можуть бути в минулому, дата завершення має бути більшою або рівною даті початку). Запит стає видимим в панелі Адміністратора.

На третьому етапі адміністратор розглядає запит і викликає PATCH-метод для зміни статусу на **APPROVED** або **REJECTED**. Якщо запит схвалено, система автоматично враховує ці дні у загальній статистиці використаних днів працівника (Used days), динамічно зменшуючи його залишок (Available).

На завершальному етапі оновлені дані повертаються клієнту; React оновлює UI, і працівник бачить свої затверджені вихідні зеленим кольором на загальному корпоративному календарі.

1.2 Огляд та аналіз аналогічних програмних рішень

Сегмент HRM-систем та застосунків для трекінгу часу є висококонкурентним. Для підвищення якості розроблюваного продукту **Employee Time Tracking** доцільно виконати критичний огляд поширених рішень, виявити типові архітектурні чи продуктові вади та сформувати вимоги, які адресують специфічні потреби малого та середнього бізнесу (МСБ).

1.2.1 Аналіз функціоналу конкурентів (BambooHR, Jira, Hurma, Excel)

- **BambooHR** демонструє величезний спектр можливостей для HR, включаючи рекрутинг та оцінку продуктивності. Проте для невеликих команд продукт є

перевантаженим функціонально (overengineered) і надзвичайно дорогим через модель оплати "за кожного користувача" (per-seat).

- **Jira (з плагінами Time Tracking)** орієнтована на управління завданнями (task-tracking) та розробку. Її інтерфейс чудово підходить для логування годин під конкретний тікет, але є вкрай незручним для загального кадрового планування, розрахунку балансу відпусток чи публікації корпоративних новин.
- **Традиційні таблиці (Excel / Google Sheets)** залишаються найпопулярнішим безкоштовним методом. Їхня критична вада — **повна відсутність автоматизації**, високий ризик людської помилки (випадкове видалення формул), відсутність безпеки на рівні ролей та неможливість інтерактивної роботи з календарем.

1.2.2 Порівняльний аналіз технологічних стеків та архітектурних підходів аналогів

Багато класичних ERP та HR-систем історично будувалися як великі моноліти на базі PHP, Java або .NET із серверним рендерингом кожної сторінки (повне перезавантаження). Такий підхід часто спричиняє затримки в роботі інтерфейсу, особливо при завантаженні складних календарних сіток чи ієрархічних дерев.

Сучасні вимоги користувачів роблять пріоритетним поєднання **SPA/SSR-фреймворків** (наприклад, Next.js) та надійного API-шару. Використання сучасних ORM (наприклад, **Prisma**) на сервері замість застарілих підходів прямого написання запитів забезпечує строгу типізацію та захист від ін'єкцій. Саме ця архітектурна комбінація (MERN-стек нового покоління) лягає в основу Employee Time Tracking.

1.2.3 Виявлення недоліків існуючих рішень та обґрунтування доцільності розробки власного продукту

За результатами аналізу можна виокремити ключові проблемні тенденції ринку:

1. **Надлишкова складність (Overengineering)** та перевантаженість UI у комерційних платформах, що ускладнює онбординг нових працівників.
2. **Висока вартість володіння** через підпискові моделі.
3. **Відсутність гнучкої локалізації** (наприклад, неможливість легко інтегрувати алгоритми обчислення українських державних свят).
4. **Ручний фактор та розсинхронізація** при використанні електронних таблиць.

Розробка власного продукту **Employee Time Tracking** є обґрунтованою, оскільки дозволяє цілеспрямовано реалізувати сфокусовану архітектуру (Full-stack

monorepo), REST API шар на **Express.js**, інтерактивний клієнт на **Next.js 14**, а також кероване збереження даних і зв'язків у **MongoDB** з моделюванням через **Prisma ORM**. Додатково, хмарне розгортання на платформах **Render** та **Vercel** наближає систему до реальних умов корпоративної експлуатації.

1.3 Постановка задачі на розробку

Створення відмовостійкого вебзастосунку для обліку робочого часу потребує формалізації вимог: **функціональних**, **специфічних** (UX/продуктових) та **нефункціональних** (безпека, продуктивність, розгортання).

1.3.1 Формулювання основних функціональних вимог

Основними функціональними вимогами до програмного продукту є:

- **Автентифікація та управління ролями:** реєстрація користувачів (за замовчуванням — EMPLOYEE), безпечний вхід за електронною поштою та паролем; захист доступу до API-ресурсів через **JWT**; розділення прав доступу між працівником та адміністратором (ADMIN).
- **Модуль запитів (Time-Off):** створення запитів на відпустку або лікарняний; зміна статусів запитів адміністратором (PENDING, APPROVED, REJECTED).
- **Бізнес-логіка балансу днів:** автоматичне нарахування відпустки (1.5 дня за кожен відпрацьований місяць) та базових лікарняних (10 днів/рік); можливість ручного нарахування додаткових днів адміністратором зі збереженням історії операцій (DaysAccrual).
- **Корпоративний портал:** модулі для публікації та перегляду новин, а також управління організаційною ієрархією користувачів (менеджер — підлеглі).

1.3.2 Формулювання специфічних функціональних вимог

До специфічних вимог, що формують зручність продукту, належать:

- **Інтерактивний календар:** інтеграція компонента календаря (react-calendar) з візуальним кольоровим кодуванням державних свят, корпоративних подій та особистих затверджених вихідних.
- **Багатомовність інтерфейсу (i18n):** динамічна зміна мови UI (українська / англійська) через React Context без перезавантаження сторінки та зі збереженням вибору в localStorage.
- **Адаптивний UI/UX та обробка медіа:** використання **Tailwind CSS** для створення сучасного, чуйного інтерфейсу (card-surface, градієнти); інтеграція модуля обрізки аватарів (react-easy-crop) перед завантаженням у профіль.

- **Розділені дашборди:** створення ізольованих робочих просторів — загального дашборду для працівника (Мої запити, Мої вихідні) та панелі управління для адміністратора.

1.3.3 Формулювання нефункціональних вимог

Нефункціональні вимоги визначають якість, безпеку та експлуатаційну придатність системи:

- **Архітектурні стандарти:** використання патерну Full-stack monorepo; поділ на незалежні backend (REST API) та frontend (SPA/SSR) застосунки.
- **Безпека даних:** паролі не зберігаються у відкритому вигляді (використовується хешування **bcryptjs**); сесії керуються виключно через **JWT**-токени з перевіркою терміну дії; неможливість видалення власного акаунта адміністратором для запобігання втраті доступу до системи.
- **Типізація та надійність бази даних:** використання **TypeScript** на всіх рівнях стеку; застосування **Prisma ORM** для гарантії цілісності схеми бази даних MongoDB та запобігання помилкам на рівні запитів.
- **Розгортання та відтворюваність:** конфігурація промислового середовища з використанням змінних оточення (.env); розміщення серверної частини на **Render**, клієнтської — на **Vercel**, та бази даних у керованому хмарному кластері **MongoDB Atlas**.

РОЗДІЛ 2

ПРОЄКТУВАННЯ СЕРВЕРНОЇ АРХІТЕКТУРИ ТА ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ (BACKEND)

2.1 Проєктування інформаційного забезпечення та бази даних

Проєктування інформаційного забезпечення є визначальним етапом, від якого залежать швидкодія, масштабованість і надійність усієї системи. У контексті розробки системи **Employee Time Tracking**, яка оперує пов’язаними даними користувачів, історією їхніх запитів, графіками та організаційною ієрархією, вибір правильної моделі зберігання визначає простоту подальшої підтримки. Документо-орієнтована модель чудово узгоджується з JSON-природою сучасного вебстеку та дозволяє гнучко розширювати профілі співробітників без складних міграцій.

2.1.1 Обґрунтування вибору документо-орієнтованої СУБД

Фундаментом інформаційної моделі обрано нереляційну хмарну базу даних **MongoDB Atlas**. Збереження даних у форматі BSON забезпечує структурну сумісність із середовищем Node.js. Це зменшує вартість перетворень на межі шару персистентності й полегшує агрегацію даних (наприклад, підрахунок загальної статистики вихідних для дашборду адміністратора). Для академічного обґрунтування важливо також підкреслити використання сучасних інструментів взаємодії з БД: замість традиційного Mongoose у проєкті використано **Prisma ORM**. Prisma забезпечує строгу типізацію (type-safety) запитів та автоматично генерує клієнт на основі декларативної схеми, що нівелює типові помилки NoSQL баз даних, пов'язані з невідповідністю типів на етапі виконання.

2.1.2 Логічна модель даних на базі Prisma ORM

Для валідації документів, опису схем і керування життєвим циклом даних на рівні застосунку використано декларативну схему Prisma. Логічну модель поділено на взаємопов’язані колекції:

1. **Колекція User (users):** облікові дані користувача, хеш пароля, роль (Role: ADMIN | EMPLOYEE), дата початку роботи (startDate) та додаткові ручні нарахування відпусток (vacationDays) і лікарняних (sickLeaveDays). Для побудови організаційної структури використовується рекурсивний зв'язок — поле managerId.

2. **Колекція TimeOffRequest (time_off_requests):** журнал запитів співробітників на вихідні. Документ містить посилання на requesterId, тип (VACATION або SICK_LEAVE), дати початку/кінця, причину та поточний статус (PENDING, APPROVED, REJECTED).
3. **Колекції News та Holiday:** глобальні сутності для корпоративного порталу. Містять статті новин (із прив'язкою до автора) та дати державних або корпоративних свят.
4. **Колекція DaysAccrual (days_accruals):** аудиторський слід (історія) ручного нарахування або списання днів адміністратором.

Така декомпозиція спрощує політику доступу та дозволяє формувати складні агреговані звіти, залишаючи при цьому код чистим і зрозумілим.

2.1.3 Моделювання ієрархії та зв'язків (Self-relations)

Для систем управління персоналом критично важливо розуміти, хто кому підпорядковується. Завдяки можливостям Prisma, в колекції користувачів реалізовано зв'язок "один-до-багатьох" на саму себе через поле managerId типу ObjectId. Це дозволяє рекурсивно витягувати дерево підлеглих для будь-якого менеджера.

Лістинг 2.1. Приклад Prisma-схеми користувача та запитів на вихідні

```
enum Role {
  EMPLOYEE
  ADMIN
}

enum RequestStatus {
  PENDING
  APPROVED
  REJECTED
}

model User {
  id             String           @id @default(auto()) @map("_id")
@db.ObjectId
  name           String
  email          String           @unique
  password       String
  role           Role             @default(EMPLOYEE)
  createdAt      DateTime         @default(now())
  startDate      DateTime?
  vacationDays   Float            @default(0)
```

```
  sickLeaveDays  Float            @default(10)
  jobTitle       String?
  gradeLevel     Int              @default(0)
  managerId      String?          @db.ObjectId

  requests       TimeOffRequest[] @relation("UserRequests")
  accruals       DaysAccrual[]    @relation("UserAccruals")

  @@map("users")
}

model TimeOffRequest {
  id          String        @id @default(auto()) @map("_id")
@db.ObjectId
  type        String        // VACATION або SICK_LEAVE
  startDate   DateTime
  endDate     DateTime
  status      RequestStatus @default(PENDING)
  reason      String?
  requesterId String        @db.ObjectId
  requester   User          @relation("UserRequests", fields:
[requesterId], references: [id])
  createdAt   DateTime      @default(now())

  @@map("time_off_requests")
}
```

2.2 Архітектура серверного застосунку та API-рівень

Серверна частина реалізується в середовищі Node.js. Для підвищення якості супроводу й зменшення класу помилок доцільно використовувати **TypeScript** у режимі суворої перевірки типів. Базовим HTTP-каркасом для інтеграції middleware, маршрутизації та обробки CORS виступає мікрофреймворк **Express.js**.

2.2.1 Парадигма REST API у контексті HRM-системи

Архітектурною базою комунікації проєкту є архітектура **REST API**. На відміну від складних запитів, де структура даних постійно змінюється, дашборди TimeTracker мають чітко визначені контракти (наприклад, отримати всі запити — GET /api/time-off, створити запит — POST /api/time-off, змінити статус — PATCH /api/time-off/:id). Використання стандартизованого REST-підходу ідеально узгоджується з централізованим клієнтом fetch на фронтенді (frontend/lib/api.ts). Усі маршрути чітко розділені на публічні (/api/auth), маршрути працівників та суворо захищені маршрути адміністраторів (/api/admin/*).

2.2.2 Шарувата архітектура (Layered Architecture)

Сервер проєктується за принципом розділення відповідальності:

- **Рівень маршрутизації (Routes):** визначення HTTP-методів та підключення middleware для перевірки токенів (authenticateToken, requireAdmin).
- **Рівень контролерів/бізнес-логіки:** валідація вхідних даних, розрахунок балансів, обробка статусів.
- **Рівень доступу до даних (Prisma Client):** взаємодія з MongoDB Atlas (створення, читання, оновлення, видалення документів).

Лістинг 2.2. Приклад контролера для зміни статусу запиту адміністратором

```
import { Request, Response } from 'express';
import prisma from '../lib/prisma';

export const updateRequestStatus = async (req: Request, res: Response)
=> {
  try {
    const { id } = req.params;
    const { status } = req.body; // PENDING, APPROVED, REJECTED

    // Валідація статусу...

    const request = await prisma.timeOffRequest.update({
      where: { id },
      data: { status },
      include: { requester: true }
    });

    res.json({ success: true, data: request });
  } catch (error) {
    res.status(500).json({ error: 'Помилка оновлення статусу запиту'
});
  }
};
```

2.3 Алгоритми бізнес-логіки, автентифікації та безпеки

Безпека Employee Time Tracking базується на поєднанні криптографічно стійких практик зберігання облікових даних і stateless-моделі сесій. Паролі ніколи не зберігаються у відкритому вигляді: на етапі реєстрації застосовується бібліотека **bcryptjs** із сіллю (10 раундів), що підвищує стійкість бази у разі її теоретичної компрометації.

2.3.1 Механізм JWT-сесій та контроль ролей

Після успішної перевірки облікових даних сервер генерує **JSON Web Token (JWT)**, підписаний секретним ключем (JWT_SECRET). Токен містить ідентифікатор користувача (id) та його роль (role) і діє 7 днів. Клієнт (Next.js) зберігає цей токен у localStorage і передає в заголовку Authorization: Bearer <token> при кожному запиті. Спеціальні middleware на сервері розшифровують токен: authenticateToken допускає всіх авторизованих, а requireAdmin жорстко блокує доступ (HTTP 403 Forbidden), якщо role !== 'ADMIN'. Крім того, в системі реалізовано захист від самовидалення: адміністратор не може видалити власний акаунт або понизити роль останнього адміністратора в системі.

2.3.2 Алгоритм автоматичного нарахування вихідних днів

Ключовою бізнес-цінністю серверної частини є динамічний алгоритм calculateAutoAccruedDays(). Замість щоденного чи щомісячного запуску "важких" фонових задач (cron jobs), баланс розраховується "на льоту" під час запиту статистики:

1. **Відпустка (VACATION):** Якщо дата початку роботи (startDate) припадає на поточний рік, система розраховує кількість повних відпрацьованих місяців і множить їх на 1.5. Якщо працівник працює більше року, він отримує фіксовано 18 днів. До цієї бази додаються дні, нараховані адміністратором вручну (vacationDays).
2. **Лікарняний (SICK_LEAVE):** Базово нараховується 10 днів на рік, до яких додаються ручні коригування (sickLeaveDays).
3. **Використані дні (Used):** Система агрегує всі запити зі статусом APPROVED за поточний рік.
4. **Доступний залишок:** Формується за формулою: Загальний баланс - Використані дні.

Цей підхід гарантує абсолютну консистентність даних і відсутність розсинхронізації, характерної для ручних розрахунків.

2.4 Інтеграція алгоритмів корпоративного порталу

Окрім обліку часу, сервер забезпечує функціонування корпоративного простору компанії.

2.4.1 Управління календарем свят та новин

Для коректного відображення доступних робочих днів інтегровано скрипт-сідер (seedHolidays.ts), який заповнює БД українськими державними святами на поточний та наступний роки (наприклад, дата Великодня розраховується за динамічним алгоритмом Гауса). Адміністратор також може додавати події типу company_event. Це дозволяє клієнтському інтерфейсу підсвічувати ці дні в календарі різними кольорами та не враховувати їх як порушення графіку відпусток.

Лістинг 2.3. Логіка розрахунку тривалості затвердженої відпустки (фрагмент)

```
const calculateUsedDays = (startDate: Date, endDate: Date): number => {
  const diffTime = Math.abs(endDate.getTime() - startDate.getTime());
  const diffDays = Math.ceil(diffTime / (1000 * 60 * 60 * 24));
  return diffDays + 1; // +1, бо день початку також враховується
};
```

2.5 Контейнеризація та розгортання серверної інфраструктури (DevOps)

Для забезпечення безперервної доступності застосунку та наближення розробки до умов промислової експлуатації (Production) було налаштовано сучасний процес CI/CD.

2.5.1 Розгортання на Render та робота з MongoDB Atlas

Промислове розгортання бекенду виконано на хмарній платформі **Render**. Процес налаштовано таким чином: будь-яке оновлення коду в гілці репозиторію автоматично ініціює збірку (npm install && npx prisma generate && npm run build), компіляцію TypeScript у JavaScript та запуск процесу Node.js. База даних **MongoDB Atlas** виступає керованим кластером з налаштованою авторизацією через Connection String. Важливим інженерним фактором при використанні безкоштовного тарифу Render (Free tier) є обробка стану *cold start*: після простою платформа зупиняє контейнер, і перший запит може займати 30–60 секунд. Це враховано на клієнті через імплементацію відповідних екранів завантаження (LoadingScreen). Згідно з планом розвитку продукту, наступним етапом модернізації серверної архітектури є повна локальна ізоляція середовищ за допомогою **Docker** та **Docker Compose**, що дозволить стандартизувати процес розробки між різними машинами програмістів.

РОЗДІЛ 3

ПРОГРАМНА РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ АРХІТЕКТУРИ ТА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ (FRONTEND)

3.1 Проєктування та імплементація базової клієнтської архітектури

Розробка клієнтської частини сучасної системи обліку робочого часу потребує модульної, високопродуктивної та масштабованої архітектури, яка коректно обробляє приватні корпоративні дані та забезпечує безперервний цикл управління запитами. Фронтенд проєкту **Employee Time Tracking (TimeTracker)** спроєктовано як односторінковий застосунок (SPA) з підтримкою серверного рендерингу (SSR) на базі фреймворку **Next.js 14**, із чітким розмежуванням доменних модулів (Dashboard Працівника, Admin Panel, Календар, Новини) та єдиним централізованим каналом обміну даними з backend через REST API.

3.1.1 Вибір фреймворку та управління станом (Next.js 14 App Router)

Фундаментальною технологією побудови інтерфейсу обрано React 18 у складі фреймворку Next.js версії 14 з використанням архітектури App Router. Вибір зумовлений гнучкою моделлю маршрутизації, можливістю оптимізації завантаження сторінок та зрілою екосистемою інструментів. У корпоративному середовищі, де користувач очікує миттєвого відображення статистики своїх вихідних, важливо мінімізувати накладні витрати на клієнтський рендеринг: Next.js дозволяє рендерити частину HTML безпосередньо на сервері (Vercel).

Реалізація базується виключно на функціональних компонентах і системі React Hooks (useState, useEffect, useMemo, useCallback, useContext), що забезпечує інкапсуляцію локальних станів форм та зменшує "вагу" коду. Для управління глобальними параметрами (мова інтерфейсу) використовується React Context API, а для мережевого шару — централізований API-клієнт (frontend/lib/api.ts).

3.1.2 Інструментарій маршрутизації та захисту сторінок

На відміну від застарілих підходів до маршрутизації, у Next.js 14 App Router файлова система напряму визначає маршрути (app/login/page.tsx, app/dashboard/page.tsx тощо). Архітектура проєкту передбачає три рівні доступу:

1. **Публічні сторінки (Public):** /login та /register. Якщо користувач уже авторизований (наявний токен у localStorage), відбувається автоматичний редирект на відповідний дашборд.

2. **Сторінки працівника (Auth / Employee):** /dashboard, /profile, /calendar, /news, /hierarchy.
3. **Сторінки адміністратора (Admin):** Маршрути, префіксовані /admin/* (наприклад, /admin/requests, /admin/users).

Для захисту роутів створено обгортки вищого порядку (HOC) або перевірки на рівні компонентів, що перевіряють наявність JWT і роль користувача, не дозволяючи звичайному працівнику перейти за URL адмін-панелі.

Лістинг 3.1. Приклад типової перевірки автентифікації на клієнті

```
import { useEffect, useState } from 'react';
import { useRouter } from 'next/navigation';
import { api } from '@/lib/api';

export function useRequireAuth(requiredRole?: 'ADMIN' | 'EMPLOYEE') {
  const router = useRouter();
  const [isAuthorized, setIsAuthorized] = useState(false);

  useEffect(() => {
    const token = localStorage.getItem('token');
    if (!token) {
      router.replace('/login');
      return;
    }

    api.get('/auth/me').then(({ user }) => {
      if (requiredRole && user.role !== requiredRole) {
        router.replace(user.role === 'ADMIN' ? '/admin/dashboard' :
'/dashboard');
      } else {
        setIsAuthorized(true);
      }
    }).catch(() => router.replace('/login'));
  }, []);

  return isAuthorized;
}
```

3.1.3 Управління мережевими запитами через централізований клієнт

Для комунікації з REST API (Node.js/Express) використано нативний fetch API, інкапсульований в єдиний модуль (frontend/lib/api.ts). Це позбавляє розробника необхідності щоразу додавати заголовок Authorization: Bearer <token> і базовий URL (NEXT_PUBLIC_API_URL) до кожного компонента. У контексті проєкту це знижує

ризик неавторизованих запитів, спрощує обробку помилок (наприклад, централізоване розлогінення при отриманні HTTP 401 Unauthorized) і робить поведінку мережі передбачуваною.

3.2 Розробка адаптивного користувацького інтерфейсу (UI/UX)

Інтерфейс системи Employee Time Tracking має бути зручним, читабельним і адаптивним до мобільних пристроїв та десктопів, не створюючи зайвого візуального шуму.

3.2.1 Парадигма utility-first за допомогою Tailwind CSS

Стилізацію реалізовано через Tailwind CSS у парадигмі utility-first: атомарні класи задають відступи, типографіку, flex/grid-розкладку, градієнти та тіні безпосередньо в JSX. Це зменшує ризик глобальних конфліктів і розростання власних CSS-файлів. Інтерфейс побудовано з використанням концепції card-surface (картки на світлому фоні), що підкреслює чистоту та сучасність дизайну (наприклад, поділ сторінки "Dashboard" на секції "Мої запити" та "Мої вихідні").

3.2.2 Інтеграція календаря та візуалізації подій

Основним інструментом візуалізації відпусток є компонент CalendarComponent. У його основі лежить бібліотека react-calendar, яку доповнено пакетом date-fns для форматування дат. Для покращення UX розроблено алгоритм кольорового кодування дат:

- **Червоний колір:** офіційні державні свята (підтягуються з API GET /api/holidays).
- **Синій колір:** внутрішні корпоративні події (company_event).
- **Зелений/жовтий колір:** особисті затверджені (APPROVED) або очікувані (PENDING) періоди відпустки працівника.

Для дат зі святами реалізовано спливаючі підказки (tooltips), щоб працівник одразу розумів, чому конкретний день підсвічено.

3.3 Інтеграція спеціалізованих клієнтських модулів (i18n, Avatar Crop)

Застосунок містить модулі, що забезпечують багатомовність та взаємодію з медіафайлами користувача.

3.3.1 Розробка кастомної системи багатомовності (i18n)

З метою повного контролю над локалізацією та мінімізації залежностей реалізовано власне рішення i18n на базі React Context API. Базовими мовами обрано українську (uk — за замовчуванням) та англійську (en). Словники зберігаються у форматі JSON (frontend/lib/translations/uk.json, en.json). Контекст (LanguageProvider) надає функцію t(key) для перекладу тексту в компонентах. Обрана мова записується у localStorage.language, що зберігає вибір користувача між сесіями. Окремо розроблено мапер errorTranslations.ts, який перехоплює стандартні повідомлення про помилки від бекенду та конвертує їх у локалізований текст для відображення у повідомленнях (Toasts/Alerts).

Лістинг 3.2. Глобальний контекст багатомовності (спрощений фрагмент)

```
import React, { createContext, useContext, useState, useEffect } from
'react';
import uk from '../lib/translations/uk.json';
import en from '../lib/translations/en.json';

const translations = { uk, en };
const LanguageContext = createContext<any>(null);

export function LanguageProvider({ children }: { children:
React.ReactNode }) {
  const [lang, setLang] = useState<'uk' | 'en'>('uk');

  useEffect(() => {
    const saved = localStorage.getItem('language') as 'uk' | 'en';
    if (saved && ['uk', 'en'].includes(saved)) setLang(saved);
  }, []);

  const t = (key: string) => {
    return key.split('.').reduce((o, i) => (o ? o[i] : null),
translations[lang]) || key;
  };

  const changeLanguage = (newLang: 'uk' | 'en') => {
    setLang(newLang);
    localStorage.setItem('language', newLang);
  };

  return (
    <LanguageContext.Provider value={{ lang, changeLanguage, t }}>
      {children}
    </LanguageContext.Provider>
  );
}

export const useLanguage = () => useContext(LanguageContext);
```

3.3.2 Механізм обрізки зображень профілю (Avatar Crop)

Для стандартизації профілів працівників (що важливо для відображення в організаційному дереві та списку новин) впроваджено можливість зміни аватарки. Замість надсилання "сирих" фотографій на сервер, що може перевантажити канал і пам'ять, клієнтська частина використовує бібліотеку react-easy-crop. Вона дозволяє користувачу у модальному вікні кадрувати зображення у пропорції 1:1, після чого браузер генерує оптимізований Data URL (Base64), який зберігається у профілі (PUT /api/auth/profile).

3.4 Розгортання клієнтського застосунку (Vercel)

Для забезпечення швидкості доставки контенту кінцевим користувачам (CDN) та автоматизації CI/CD процесу, розгортання фронтенд-частини виконано на хмарній платформі **Vercel**, яка є нативним середовищем для Next.js.

3.4.1 Конфігурація середовища (Production Build)

При підключенні GitHub-репозиторію до Vercel, платформа автоматично розпізнає Next.js проєкт. На етапі production збірки (npm run build) Vercel виконує мініфікацію JS/CSS коду та статичний аналіз, генеруючи оптимізовані статичні HTML-файли для маршрутів, що не потребують постійних динамічних даних. Ключовим елементом налаштування є підключення змінної оточення NEXT_PUBLIC_API_URL, яка вказує на розгорнутий на Render бекенд (наприклад, https://api.timetracker.onrender.com/api). Це дозволяє фронтенду коректно маршрутизувати всі fetch-запити без виникнення помилок CORS (за умови коректного whitelist-налаштування на стороні Express.js).

РОЗДІЛ 4

ТЕСТУВАННЯ ТА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ

4.1 Стратегія тестування та забезпечення надійності системи

У процесі розробки складних клієнт-серверних систем, зокрема таких, що оперують корпоративними даними, обліком робочого часу та організаційною ієрархією (облікові записи, залишки відпусток, лікарняні), автоматизоване тестування є невід'ємною складовою життєвого циклу розробки програмного забезпечення (SDLC).

Стратегія тестування застосунку **Employee Time Tracking (TimeTracker)** базується на класичній піраміді тестування: переважають модульні (unit) тести як найдешевші за вартістю підтримки, доповнені інтеграційними перевірками критичних шарів (REST API, доступ до БД через Prisma ORM) та обмеженим набором наскрізних (E2E) сценаріїв для найбільш ризикованих користувацьких потоків (автентифікація, подача запиту на відпустку, погодження запиту адміністратором).

Головна мета автоматизації — запобігання регресіям під час рефакторингу (зокрема при зміні алгоритмів автонарахування вихідних), верифікація бізнес-правил в ізольованих середовищах і підтвердження коректності авторизаційних потоків (перевірка JWT, відмова в доступі без токена, перевірка ролей ADMIN/EMPLOYEE). Окремо враховується специфіка розгортання монорепозиторію: ізольоване тестування бекенду (Express.js) та фронтенду (Next.js), що гарантує стабільність контракту між ними.

4.2 Тестування серверної частини (Backend)

Серверна частина є критичним вузлом: саме вона реалізує криптографічні аспекти зберігання паролів (bcryptjs), авторизацію доступу до REST-маршрутів, виконання логіки автонарахування днів (1.5 дня/місяць) та персистенцію в MongoDB. Тому тести backend-рівня повинні підтримувати асинхронність, передбачувану ізоляцію залежностей бази даних та відтворювані сценарії помилок.

4.2.1 Інструментарій тестування: Jest, ts-jest та Supertest

Основним test runner обрано **Jest**. Оскільки серверна реалізація ведеться з TypeScript, для трансформації та узгодженого запуску тестів використовується

ts-jest. Це дозволяє запускати тести без ручного попереднього кроку компіляції в окремий каталог для кожного локального прогону.

Для перевірки HTTP-шару без обов'язкового відкриття реального мережевого сокета застосовується **Supertest**: запити надсилаються безпосередньо до інстансу Express-додатка, що істотно прискорює інтеграційні тести й спрощує їх інтеграцію в конвеєри CI/CD.

4.2.2 Модульне тестування бізнес-логіки та мокування (Mocking)

Під час тестування доменних сервісів застосовується ізоляція залежностей. Наприклад, для перевірки алгоритму розрахунку залишку відпусток доцільно не підключати реальну базу MongoDB у unit-тестах: клієнт Prisma підміняється за допомогою бібліотеки jest-mock-extended, що дозволяє перевіряти гілки логіки (наприклад, розрахунок для співробітника, який працює менше року, або перевірка балансу при існуючих схвалених запитах) із детермінованими відповідями.

Окремим інженерним викликом є перевірка роутів адміністратора: тест імітує виклики з токеном звичайного співробітника та перевіряє, чи коректно спрацьовує middleware requireAdmin, повертаючи HTTP 403 Forbidden.

Лістинг 4.1. Приклад інтеграційного тесту REST API через Supertest

```
import request from 'supertest';
import app from '../src/server';
import { generateMockJwt } from './utils/testHelpers';

describe('Time-Off API — запити користувача', () => {
  it('повертає список запитів співробітника за валідного JWT', async
() => {
    const token = generateMockJwt({ id: 'mock-user-id', role:
'EMPLOYEE' });

    const response = await request(app)
      .get('/api/time-off')
      .set('Authorization', `Bearer ${token}`);

    expect(response.status).toBe(200);
    expect(response.body.success).toBe(true);
    expect(Array.isArray(response.body.data)).toBe(true);
  });

  it('повертає 401 UNAUTHORIZED без токена', async () => {
    const response = await request(app)
      .get('/api/time-off');

    expect(response.status).toBe(401);
```

```
    expect(response.body.error).toBe('Доступ заборонено. Токен
відсутній.');
  });

  it('забороняє доступ до адмін-маршруту для звичайного працівника',
async () => {
    const token = generateMockJwt({ id: 'mock-user-id', role:
'EMPLOYEE' });

    const response = await request(app)
      .get('/api/admin/users')
      .set('Authorization', `Bearer ${token}`);

    expect(response.status).toBe(403);
  });
});
```

4.3 Тестування клієнтської частини (Frontend)

Тестування UI для застосунків на базі Next.js 14 зміщує акцент з «чистої» валідації даних на перевірку поведінки компонентів, реакцій на дії користувача (заповнення форми запиту на відпустку), коректності відображення кольорів у календарі та ізоляції дашбордів.

4.3.1 Середовище тестування: Jest та jsdom

Для клієнтської частини TimeTracker, побудованої на Next.js, стандартно застосовується комбінація **Jest** та середовища **jsdom**. Воно емулює браузерне середовище (DOM) у терміналі, що дозволяє швидко рендерити React-компоненти без необхідності запуску реального браузера (як у Selenium або Cypress) на етапі модульного тестування.

4.3.2 Тестування інтерфейсів за допомогою React Testing Library

Компонентні тести пишуться у стилі **React Testing Library (RTL)**: пріоритет мають запити за ролями, плейсхолдерами та текстом, доступні користувачу, а не перевірка внутрішнього стану компонентів. Це знижує зв’язаність тестів із реалізацією та підвищує стійкість до рефакторингу Tailwind-класів.

Складність створюють глобальні провайдери (наприклад, LanguageProvider для i18n). Якщо компонент викликає функцію перекладу t() поза відповідним провайдером, React генерує помилку. Тому впроваджується патерн renderWithProviders: єдина обгортка, яка під час тестового рендеру інжектує необхідні контексти та підміняє роутер Next.js (через мокування next/navigation).

Лістинг 4.2. Кастомний рендер для компонентів, залежних від Context API (i18n)

```
import { render } from '@testing-library/react';
import { LanguageProvider } from '../lib/contexts/LanguageContext';

// Мокування маршрутизатора Next.js App Router
jest.mock('next/navigation', () => ({
  useRouter() {
    return {
      push: jest.fn(),
      replace: jest.fn(),
      prefetch: jest.fn(),
    };
  },
}));

export function renderWithProviders(ui: React.ReactElement) {
  return render(
    <LanguageProvider>
      {ui}
    </LanguageProvider>
  );
}

export * from '@testing-library/react';
export { renderWithProviders as render };
```

4.4 Тестування безпеки, ізоляції даних та бізнес-правил

Окрім функціональних тестів, архітектура Employee Time Tracking потребує перевірок, що відображають специфіку корпоративного порталу:

1. **Ізоляція після виходу (Logout cleanup).** Критичний тест безпеки: функція виходу з системи (logout()) має обов'язково видаляти JWT-токен із localStorage та очищати будь-які закешовані дані профілю, після чого редиректити на /login. Це знижує ризик несанкціонованого доступу, якщо працівник використовує спільний комп'ютер.
2. **Валідація дат у формах.** Перевіряється логіка компонента TimeOffRequestForm. Тест імітує введення дат і підтверджує, що система блокує спробу подати запит, де endDate передує startDate, або якщо дати знаходяться у минулому, не дозволяючи відправити такий HTTP-запит на сервер.
3. **Захист від циклічної ієрархії.** При тестуванні бекенду адмін-панелі перевіряється алгоритм managerId. Тест повинен гарантувати, що адміністратор не може призначити користувача "А" керівником користувача "Б", якщо "Б" вже є керівником "А" (запобігання нескінченним циклам в організації).

4. **Коректність відображення i18n.** Перевіряється, чи змінюються тексти в інтерфейсі (наприклад, з "Мої запити" на "My Requests") при зміні мови через LanguageProvider без перезавантаження сторінки.

Комплексний підхід (Backend + Frontend + перевірка бізнес-логіки автонарахувань) підвищує ймовірність стабільної роботи Employee Time Tracking у production-середовищі (Vercel, Render) та зменшує вартість супроводу при подальшому масштабуванні функціоналу системи.

РОЗДІЛ 5

ІНФОРМАЦІЙНА БЕЗПЕКА, ЗАХИСТ ПЕРСОНАЛЬНИХ ДАНИХ ТА ВІДМОВОСТІЙКІСТЬ СИСТЕМИ

5.1 Аналіз загроз інформаційній безпеці у корпоративних системах HRM

Впровадження інформаційних систем для управління персоналом та обліку робочого часу (Time Tracking) неминуче пов’язане з обробкою та зберіганням великих обсягів конфіденційної інформації. До таких даних у системі Employee Time Tracking належать: особисті ідентифікатори користувачів (email, імена), організаційна структура компанії, паролі у хешованому вигляді, а також дані про відсутність на робочому місці (зокрема лікарняні листи, що можуть опосередковано розкривати інформацію про стан здоров'я).

Враховуючи критичність цих даних, розроблений вебзастосунок повинен відповідати базовим критеріям тріади інформаційної безпеки: конфіденційності (доступ до даних мають лише авторизовані особи), цілісності (захист від несанкціонованої зміни балансу днів відпустки) та доступності (гарантія безперебійної роботи порталу). Основними векторами атак на подібні клієнт-серверні застосунки є:

Brute-force атаки на механізми автентифікації з метою перехоплення облікових записів адміністраторів.

Міжсайтовий скриптинг (XSS) та підробка міжсайтових запитів (CSRF) для викрадення JWT-токенів сесії.

NoSQL-ін'єкції, спрямовані на маніпуляцію запитами до бази даних MongoDB.

Несанкціонований доступ (Broken Access Control) — спроби звичайних працівників (EMPLOYEE) отримати доступ до ендпоінтів адміністратора.

5.2 Реалізація багаторівневого захисту на рівні застосунку (Application Security)

Для протидії вищезазначеним загрозам архітектура застосунку Employee Time Tracking включає комплекс захисних механізмів на рівні мережі, сервера та клієнта.

5.2.1 Захист мережевого шару (CORS та Rate Limiting)

Для запобігання несанкціонованим запитам зі сторонніх доменів у застосунку налаштовано строгу політику CORS (Cross-Origin Resource Sharing). Серверна частина на базі Express.js приймає HTTP-запити виключно з доменів, внесених до білого списку (whitelist), який визначається змінними оточення FRONTEND_URL (наприклад, продакшен-домен на Vercel). Крім того, для захисту від brute-force атак та автоматизованого перебору паролів доцільно впровадити обмеження кількості запитів (Rate Limiting) на критичні ендпоінти, такі як /api/auth/login.

Лістинг 5.1. Приклад налаштування політики CORS та базового захисту в Express.js

```
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';

const app = express();

// Захист HTTP-заголовків за допомогою Helmet
app.use(helmet());

// Налаштування CORS: доступ лише для дозволеного фронтенду
const allowedOrigins = process.env.FRONTEND_URLS?.split(',') || [process.env.FRONTEND_URL];

app.use(cors({
  origin: (origin, callback) => {
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error('Блокування політикою CORS: несанкціонований домен'));
    }
  },
  credentials: true,
}));

app.use(express.json({ limit: '1mb' })); // Захист від переповнення пам'яті великими payload
```

5.2.2 Протидія ін'єкціям та XSS-атакам

Традиційні вебзастосунки часто стають жертвами SQL/NoSQL ін'єкцій через недостатню валідацію вхідних даних. У розробленому проєкті цей ризик

мінімізовано завдяки використанню Prisma ORM. Prisma абстрагує логіку взаємодії з базою даних, автоматично екрануючи (sanitizing) усі вхідні параметри перед формуванням BSON-запитів до MongoDB.

Для захисту від XSS (Cross-Site Scripting) на клієнтській стороні використовується механізм рендерингу React. Фреймворк Next.js автоматично екранує всі текстові змінні перед їх вставкою в DOM-дерево, унеможливлюючи виконання шкідливого JavaScript-коду, який зловмисник міг би спробувати впровадити через текстове поле (наприклад, у причині запиту на відпустку).

5.3 Відповідність стандартам захисту персональних даних (GDPR Compliance)

Сучасні HRM-системи повинні проєктуватися з урахуванням глобальних стандартів захисту даних, зокрема Загального регламенту про захист даних (GDPR). Архітектура TimeTracker реалізує ключові принципи приватності:

Мінімізація даних (Data Minimization): Збираються лише ті дані, які критично необхідні для ведення обліку (ім'я, email, посада, залишок днів). Система не вимагає домашніх адрес, паспортних даних чи фінансової інформації.

Право на забуття (Right to be Forgotten): Адміністратор має інструмент для повного видалення користувача з системи (DELETE /api/admin/users/:userId), що каскадно видаляє або анонімізує його запити на відпустку.

Захист даних у стані спокою (Data at Rest): Паролі користувачів хешуються алгоритмом bcrypt з використанням солі, що робить неможливим відновлення оригінального пароля навіть у разі фізичного витоку бази даних.

5.4 Забезпечення відмовостійкості та резервного копіювання

Інформаційна безпека також охоплює забезпечення безперебійної роботи сервісу. Оскільки система розгорнута в хмарному середовищі, відмовостійкість досягається на рівні інфраструктури:

Захист бази даних: Використання керованого кластера MongoDB Atlas забезпечує автоматичне реплікування даних на кілька фізичних вузлів. Це гарантує доступність бази даних навіть у разі виходу з ладу одного з серверів. Atlas також автоматично створює резервні копії (Automated Backups), що дозволяє відновити стан системи на будь-яку точку в часі (Point-in-Time Recovery) у разі критичного збою або випадкового видалення даних адміністратором.

Відмовостійкість фронтенду: Деплой на платформу Vercel автоматично підключає застосунок до глобальної мережі доставки контенту (CDN) Vercel Edge Network. Це не лише прискорює завантаження інтерфейсу для користувачів, але й надає базовий захист від DDoS-атак на рівні інфраструктури провайдера.

Моніторинг: Впровадження спеціального ендпоінту GET /api/health дозволяє налаштувати автоматичні перевірки (Health Checks) для платформи Render. Якщо процес Node.js зависає або втрачає з'єднання з MongoDB, платформа автоматично перезапустить контейнер, мінімізуючи час простою (downtime).

Таким чином, розроблений комплекс заходів інформаційної безпеки забезпечує надійний захист корпоративних даних, протидіє поширеним вебвразливостям та створює відмовостійке середовище для безперебійного обліку робочого часу працівників.

РОЗДІЛ 6

ІНФРАСТРУКТУРНЕ ЗАБЕЗПЕЧЕННЯ, МАСШТАБУВАННЯ ТА DEVOPS-ПРАКТИКИ

6.1 Автоматизація процесів розгортання (CI/CD)

У сучасній інженерії програмного забезпечення процес ручного перенесення коду на сервери (через FTP або прямий доступ по SSH) вважається застарілим, неефективним та вкрай схильним до людських помилок. Для забезпечення надійності, передбачуваності та швидкості доставки нових функцій у системі Employee Time Tracking впроваджено базові практики безперервної інтеграції та безперервного розгортання (Continuous Integration / Continuous Deployment — CI/CD).

Архітектура монорепозиторію (monorepo), де клієнтська (Next.js) та серверна (Express.js) частини зберігаються спільно в одному Git-репозиторії, вимагає чіткого розділення процесів збірки. Процес розгортання автоматизовано через глибоку інтеграцію системи контролю версій (GitHub) із хмарними платформами хостингу. Пайплайн розгортання складається з наступних етапів:

Локальні перевірки (Pre-commit hooks): Перед тим як код потрапляє до віддаленого репозиторію, інструменти на кшталт Husky ініціюють локальні перевірки. Запускається статичний аналіз коду (ESLint) та форматування (Prettier), що гарантує дотримання єдиного стилю кодування (Code Style) всією командою розробників і запобігає потраплянню синтаксичних помилок у репозиторій.

Безперервна інтеграція (CI): Кожен новий коміт (commit) у головну гілку (main) ініціює автоматичну перевірку цілісності коду на серверах CI/CD. На рівні бекенду виконується встановлення залежностей (npm install), генерація клієнта бази даних (npx prisma generate) та сувора перевірка типів TypeScript (tsc --noEmit). На рівні фронтенду Next.js виконує власний статичний аналіз, оптимізацію зображень та валідацію маршрутів. Якщо хоча б один з етапів завершується помилкою (наприклад, невідповідність типів Prisma та схеми БД), розгортання блокується, захищаючи production-середовище від збою.

Безперервне розгортання (CD): У разі успішної збірки, платформи хостингу автоматично підтягують оновлення через Webhooks. Vercel оновлює статичні файли (HTML/CSS) та Serverless-функції фронтенду. Водночас Render виконує збірку Node.js-процесу із новою версією REST API.

Zero Downtime Deployment: Обидві платформи використовують концепцію розгортання без простою. Нова версія застосунку розгортається у фоновому ізольованому контейнері. Лише коли сервер успішно запустився і відповів на health-check запит, балансувальник навантаження перемикає трафік користувачів зі старої версії на нову. Користувачі не помічають моменту оновлення системи, що є критично важливим для корпоративних застосунків.

6.2 Хмарна архітектура та ізоляція середовищ

Система TimeTracker побудована за принципами хмарної архітектури (Cloud-Native). Це означає відмову від монолітних фізичних серверів на користь гнучких хмарних рішень, що дозволяє легко масштабувати окремі вузли (базу даних, фронтенд або бекенд) незалежно один від одного.

6.2.1 Серверна інфраструктура (PaaS)

Бекенд застосунку розміщено на платформі як послуга (Platform as a Service — PaaS) від Render. Такий підхід позбавляє розробника необхідності вручну адмініструвати операційні системи (наприклад, оновлювати ядро Linux) на класичних віртуальних машинах (IaaS). Render бере на себе інфраструктурні задачі: автоматично генерує та оновлює SSL/TLS-сертифікати (Let's Encrypt), керує балансуванням мережевого навантаження та забезпечує зворотне проксіювання (Reverse Proxy) трафіку до середовища виконання Node.js.

Важливим аспектом архітектури є управління секретами. Конфігурація середовища керується виключно через змінні оточення (Environment Variables). Рядок підключення до бази даних (DATABASE_URL), параметри CORS (FRONTEND_URL) та секретний ключ для підпису токенів (JWT_SECRET) не зберігаються у коді. Вони ін’єктуються безпосередньо в процес пам'яті під час запуску контейнера, що унеможливлює їх витік через публічний чи корпоративний репозиторій.

Для обробки проблеми «холодного старту» (cold start), характерної для Serverless та PaaS-платформ (коли після 15 хвилин простою сервер "засинає"), впроваджено спеціальний Health-Check ендпоінт (GET /api/health). Він не лише перевіряє активність Node.js процесу, але й пінг-запитом перевіряє доступність кластера MongoDB через Prisma, гарантуючи, що бекенд повністю готовий до обробки запитів співробітників.

6.2.2 Фронтенд та Serverless-обчислення

Клієнтська частина розгорнута на спеціалізованій платформі Vercel. Використання фреймворку Next.js 14 дозволяє застосовувати гібридну стратегію рендерингу. Сторінки, що рідко змінюються (наприклад, сторінка входу /login або загальна інформація), генеруються статично під час збірки (Static Site Generation — SSG), що забезпечує миттєве завантаження. Натомість динамічні дашборди (статистика вихідних, список PENDING запитів) використовують серверний рендеринг (SSR) або рендеринг на стороні клієнта (CSR) для забезпечення максимальної актуальності даних.

Інфраструктура Vercel автоматично розподіляє скомпільований код та статичні ресурси (JS, CSS, зображення) через глобальну мережу доставки контенту (Edge CDN). Це гарантує, що незалежно від географічного розташування користувача (чи він працює з офісу в Києві, чи віддалено з-за кордону), інтерфейс завантажуватиметься з найближчого до нього кеш-вузла (Edge Node), забезпечуючи мінімальну затримку (latency).

6.3 Контейнеризація за допомогою Docker

Для забезпечення абсолютної відтворюваності середовища розробки та остаточного усунення класичної проблеми «це працює лише на моєму комп'ютері» (works on my machine), стратегія архітектурного розвитку системи передбачає глибоке впровадження технології контейнеризації Docker.

Хоча поточний деплой реалізовано через хмарні керовані платформи (Render/Vercel), контейнеризація є критичною для локальної розробки, стандартизації CI-процесів та потенційного перенесення застосунку в закриті корпоративні кластери (наприклад, on-premise розгортання в Kubernetes для банківських або державних установ, які вимагають зберігання даних на власних серверах).

Docker дозволяє запакувати застосунок разом з усіма його залежностями, конфігураційними файлами (package.json) та фіксованою версією середовища виконання (наприклад, node:18-alpine) у єдиний незмінний і стандартизований образ (Image). Використання образу на базі Alpine Linux суттєво зменшує кінцевий розмір контейнера та мінімізує поверхню для потенційних кібератак.

Впровадження оркестратора локального середовища (docker-compose.yml) дозволяє розробникам та тестувальникам однією командою (docker-compose up) підіймати локально бекенд, фронтенд та локальну копію бази даних. При цьому

налаштовується "прокидання" томів (Volume mapping), щоб зміни в коді на комп'ютері розробника миттєво відображалися у працюючому контейнері (Hot Module Replacement), зберігаючи швидкість розробки ідентичною до середовища продакшену.

6.4 Стратегії оптимізації продуктивності та кешування

Масштабування корпоративних систем обліку часу неминуче стикається з проблемою інтенсивних запитів до бази даних. Пікові навантаження виникають циклічно: на початку робочого дня (коли співробітники масово перевіряють пошту та новини) або наприкінці звітного місяця (коли адміністратори та HR-менеджери генерують масові звіти щодо відпусток).

Для забезпечення стабільно високої швидкодії та уникнення деградації продуктивності бази даних розроблено багаторівневу стратегію оптимізації:

6.4.1 Індексація та профілювання MongoDB

Для колекцій User та TimeOffRequest застосовано одиночні та складені (Compound) індекси на полях, за якими найчастіше виконується пошук або фільтрація. Наприклад, окрім індексу на поле email, створено складений індекс { requesterId: 1, status: 1 } для колекції запитів. Оскільки адміністратор або застосунок найчастіше шукає запити конкретного працівника зі статусом APPROVED (для підрахунку балансу), такий індекс перетворює повільний лінійний пошук (Collection Scan) на швидкий логарифмічний (Index Scan), зменшуючи навантаження на процесор кластера бази даних в десятки разів.

6.4.2 Інвалідація кешу у Next.js (On-Demand Revalidation)

Використання внутрішніх механізмів кешування App Router у Next.js 14 дозволяє радикально зменшити кількість GET-запитів до бекенду. Наприклад, список державних свят, який змінюється вкрай рідко, кешується фронтендом назавжди. Натомість для динамічних даних застосовується On-Demand Revalidation (ревалідація на вимогу). Коли адміністратор схвалює запит на відпустку, бекенд не лише оновлює базу даних, але й повідомляє клієнту про необхідність оновити конкретний тег (наприклад, revalidateTag('requests')), що гарантує постійну актуальність даних без зайвих опитувань сервера (polling).

6.4.3 Перспективи In-Memory кешування (Redis)

Згідно з архітектурним планом (розділ «Що не реалізовано / Плани»), наступним масштабним кроком оптимізації є впровадження in-memory бази даних Redis. Розрахунок балансу відпусток є складною обчислювальною операцією, оскільки вимагає агрегації всієї історії запитів та ручних нарахувань працівника. Замість того, щоб щоразу навантажувати MongoDB та CPU бекенду розрахунками, цей результат (Total, Used, Available) зберігатиметься в оперативній пам'яті Redis.

Реалізація планується за патерном Cache-Aside: при запиті статистики сервер спочатку перевіряє Redis; якщо дані є (Cache Hit) — вони миттєво повертаються клієнту (час відгуку < 5 мс). Якщо даних немає (Cache Miss) — сервер вираховує баланс через Prisma/MongoDB, повертає його клієнту і паралельно записує в Redis. Перерахунок та інвалідація кешу відбуватиметься виключно у момент додавання нового запиту на відпустку або при ручному нарахуванні днів адміністратором.

Таким чином, інфраструктурний базис Employee Time Tracking є повністю готовим до експоненційного зростання навантажень. Поєднання автоматизованого CI/CD розгортання, строгих DevOps-практик управління секретами, контейнеризації та глибоко продуманої стратегії багаторівневого кешування формує сучасний, Enterprise-ready підхід до розробки корпоративних вебзастосунків.

РОЗДІЛ 7

АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ, СЦЕНАРІЇ ВИКОРИСТАННЯ ТА ЕКСПЛУАТАЦІЙНА ДОКУМЕНТАЦІЯ СИСТЕМИ

7.1 Математична та алгоритмічна модель розрахунку балансу робочого часу

Одним із найскладніших та найважливіших компонентів системи Employee Time Tracking є ядро автоматичного обчислення доступних вихідних днів для кожного працівника. У більшості існуючих систем такий розрахунок виконується за допомогою статичних фонових процесів (cron jobs), що оновлюють базу даних раз на добу або раз на місяць. У розробленому вебзастосунку реалізовано динамічний підхід: баланс обчислюється в режимі реального часу («на льоту») при кожному зверненні до маршрутів статистики (/api/time-off/stats), що гарантує абсолютну консистентність даних та усуває ризики часткового оновлення.

7.1.1 Алгоритм нарахування щорічної відпустки (VACATION)

Базовим правилом бізнес-логіки є надання працівнику 18 днів оплачуваної відпустки на рік, що пропорційно розподіляються як 1.5 дня за кожен відпрацьований календарний місяць.

Функція calculateAutoAccruedDays() отримує на вхід об'єкт користувача з бази даних (через Prisma) та аналізує поле startDate (дата працевлаштування). Алгоритм працює за наступними граничними умовами:

Відсутність дати працевлаштування: Якщо startDate дорівнює null (наприклад, профіль щойно створено адміністратором і ще не заповнено), система автоматично повертає 0 авто-днів, блокуючи можливість подати запит на відпустку.

Працевлаштування в минулих роках: Якщо рік, вказаний у startDate, менший за поточний календарний рік (працівник працює більше одного року), алгоритм автоматично нараховує повний річний баланс: 18 днів.

Працевлаштування у поточному році: Якщо рік збігається з поточним, алгоритм вираховує різницю в місяцях між поточним місяцем та місяцем працевлаштування (включно). Формула має вигляд:

AutoAccrued = ((CurrentMonth - StartMonth) + 1) * 1.5.

Загальний доступний баланс відпустки формується як сума автоматичних нарахувань та ручних коригувань адміністратора (поле vacationDays), від якої віднімається сума тривалостей усіх запитів зі статусом APPROVED у поточному році:

AvailableVacation = AutoAccrued + ManualVacationDays - UsedVacationDays.

7.1.2 Алгоритм розрахунку тривалості лікарняних (SICK_LEAVE) та відпусток

Лікарняні дні розраховуються за спрощеною моделлю: кожен працівник за замовчуванням має 10 днів на рік, які оновлюються 1 січня.

Складність полягає у підрахунку використаних днів (Used days). Коли користувач створює запит, він вказує startDate та endDate. Оскільки відпустка вимірюється в календарних днях, алгоритм на сервері виконує перетворення часових міток (timestamps) у дні:

Days = Math.ceil(Math.abs(endDate - startDate) / (1000 * 60 * 60 * 24)) + 1.

Одиниця додається для врахування того факту, що межі відпустки є включними (наприклад, відпустка з 1 по 2 число триває 2 дні).

7.1.3 Алгоритмізація календаря свят (метод Гауса для Великодня)

Для забезпечення точної роботи інтерактивного календаря та візуального інформування співробітників, система повинна знати дати офіційних державних свят. Більшість свят мають фіксовану дату (наприклад, 1 січня, 24 серпня). Однак свята, прив'язані до місячного календаря (Великдень, Трійця), є перехідними.

Для їх автоматичного обчислення в скрипті ініціалізації бази даних (seedHolidays.ts) імплементовано класичний алгоритм Гауса обчислення дати православного Великодня.

Алгоритм виконує наступні математичні операції над поточним роком Y:

a = Y % 19

b = Y % 4

c = Y % 7

d = (19 * a + 15) % 30

e = (2 * b + 4 * c + 6 * d + 6) % 7

f = d + e

Якщо f <= 9, то Великдень святкується 22 + f березня. Якщо f > 9, свято припадає на f - 9 квітня. Отримана дата зберігається в колекції holidays MongoDB зі статусом public_holiday, що дозволяє клієнтському компоненту react-calendar підсвічувати цей день червоним кольором та блокувати його врахування як звичайного робочого дня.

7.2 Керівництво з розгортання та адміністрування середовища

Для забезпечення можливості передачі проєкту іншим розробникам або розгортання на нових серверах, розроблено детальну експлуатаційну специфікацію. Застосунок розроблено у форматі Monorepo, тому розгортання складається з двох незалежних етапів.

7.2.1 Системні вимоги та підготовка бази даних

Для локального запуску та модифікації коду необхідні наступні компоненти:

Середовище виконання: Node.js версії 18.x або вище.

Менеджер пакетів: npm (від версії 9.x).

База даних: Активний кластер MongoDB Atlas (або локальний екземпляр MongoDB Server).

Перед запуском необхідно створити файл конфігурації .env у директорії backend/ із наступними критичними ключами:

DATABASE_URL — рядок підключення до MongoDB (формату mongodb+srv://<user>:<password>@cluster...).

JWT_SECRET — криптографічно стійкий випадковий рядок (мінімум 32 символи) для підпису токенів сесії.

FRONTEND_URL — адреса клієнтського застосунку для налаштування CORS (локально це http://localhost:3000).

7.2.2 Ініціалізація та запуск Backend-частини

Перехід у директорію сервера: cd backend.

Встановлення залежностей: npm install.

Генерація клієнта Prisma ORM на основі поточної схеми: npx prisma generate.

Синхронізація індексів та структури з MongoDB: npx prisma db push.

Наповнення бази даних святами: npm run seed:holidays.

Запуск сервера у режимі розробника з підтримкою гарячого перезавантаження (hot reload) через tsx: npm run dev.

7.2.3 Ініціалізація та запуск Frontend-частини

Перехід у директорію клієнта: cd frontend.

Встановлення залежностей: npm install.

Створення .env файлу зі змінною NEXT_PUBLIC_API_URL=http://localhost:5000/api.

Запуск середовища Next.js: npm run dev.

Після цього застосунок стає доступним у браузері за адресою http://localhost:3000. Для тестування функцій адміністратора розробнику необхідно спочатку зареєструвати новий акаунт (який автоматично отримає роль EMPLOYEE), відкрити Prisma Studio та змінити значення поля role на ADMIN.

7.3 Деталізовані сценарії використання (User Cases) для ролі «Працівник»

Інтерфейс працівника (EMPLOYEE) спроєктовано за принципом мінімізації кліків для доступу до ключової інформації. Після успішної авторизації через сторінку /login (з обробкою помилок через мапер errorTranslations.ts, який локалізує відповіді сервера), працівник потрапляє на головний екран — Dashboard.

Сценарій 1: Аналіз особистої статистики.

На дашборді реалізовано дві основні вкладки. У вкладці «Мої вихідні» працівник бачить компонент DaysOffStats. Цей компонент робить авторизований GET /api/time-off/stats запит і рендерить кругові або стовпчасті діаграми. Інтерфейс

чітко розмежовує: Загальний баланс (Total), Використані дні (Used) та Доступний залишок (Available). Працівник також бачить розбивку на автонараховані дні та ті, що були додані адміністратором вручну.

Сценарій 2: Подача запиту на відпустку.

У вкладці «Мої запити» користувач взаємодіє з формою TimeOffRequestForm.

Працівник обирає тип відсутності у випадаючому списку (Відпустка або Лікарняний).

Використовуючи інтегрований date-picker, обирає дату початку та кінця. Клієнтська валідація негайно блокує спробу вибрати дати у минулому.

Працівник вводить текстове обґрунтування (reason).

Після натискання кнопки "Надіслати", Next.js виконує POST-запит до API.

Запит з'являється у списку TimeOffRequestsList зі статусом PENDING (жовтий індикатор). До моменту розгляду адміністратором, баланс днів працівника не зменшується.

Сценарій 3: Редагування профілю та завантаження аватара.

Перейшовши на сторінку /profile, працівник має можливість змінити пароль або завантажити фото профілю. Для забезпечення консистентності дизайну (щоб усі фото в системі мали квадратну пропорцію), застосовується модуль react-easy-crop. Користувач завантажує фото, відкривається модальне вікно AvatarCropModal, де за допомогою миші або жестів масштабується область обличчя. Після підтвердження, клієнтський браузер конвертує виділену область у стиснутий Data URL (Base64) розміром до 350 КБ і надсилає на сервер через PUT /api/auth/profile.

7.4 Деталізовані сценарії управління для ролі «Адміністратор»

Акаунти з роллю ADMIN мають розширене навігаційне меню, що веде до секції /admin/*. Цей простір є суворо захищеним; будь-яка спроба доступу без відповідного токена призводить до переадресації.

Сценарій 1: Управління організаційною структурою (Hierarchy).

Створення організаційної ієрархії дозволяє компанії візуалізувати структуру підпорядкування. В адмін-панелі на сторінці /admin/hierarchy адміністратор бачить список усіх працівників.

Редагуючи профіль співробітника, адміністратор може встановити числове значення gradeLevel (де 0 — найвищий рівень, наприклад, CEO; 1 — топ-менеджмент; 2 — лінійні керівники тощо), вказати текстову посаду (jobTitle) та призначити керівника (managerId).

Ключова архітектурна особливість: при призначенні керівника клієнтська та серверна частини виконують валідацію на запобігання циклічним залежностям (захист від циклів у дереві). Система перевіряє ланцюг підпорядкування (pathSet), щоб унеможливити ситуацію, коли працівник А підпорядковується працівнику Б, а працівник Б — працівнику А. Після збереження даних клієнтський інтерфейс автоматично перемальовує ієрархічне дерево компанії.

Сценарій 2: Ручне нарахування днів та аудит.

Стандартних автонарахованих днів може бути недостатньо (наприклад, компанія надає бонусні вихідні за понаднормову роботу або весілля). На сторінці управління користувачем (/admin/users) адміністратор відкриває модальне вікно AccrualForm.

Він вказує кількість днів, тип (відпустка або лікарняний) та обов'язкову причину. При відправці POST /api/admin/users/:userId/accrual, сервер виконує дві транзакційні дії:

Оновлює поле vacationDays або sickLeaveDays у колекції User.

Створює незмінний запис у колекції DaysAccrual, де фіксується хто (adminId), кому (userId), скільки днів і з якої причини нарахував. Цей аудиторський слід (Audit Trail) дозволяє в майбутньому вирішувати суперечки щодо балансу відпусток і є доступним для перегляду через GET-маршрут історії нарахувань.

Сценарій 3: Публікація корпоративних новин.

Для інформування компанії адміністратор переходить у /admin/news та використовує компонент CreateNewsForm. Форма приймає заголовок, текстовий контент та опціональне зображення (розміром до 1 МБ). При відправці форми сервер прив'язує новину до authorId поточного адміністратора та встановлює часову мітку

publishedAt. Працівники миттєво отримують доступ до новини на своїй сторінці /news.

7.5 Забезпечення багатомовності та локалізація

Враховуючи, що система може використовуватися міжнародними або змішаними командами, жорстке "хардкодування" тексту в компонентах є неприпустимим. Тому в архітектуру глибоко інтегровано систему багатомовності (i18n).

Реалізація спирається на патерн "Контекст провайдера" (Context Provider) у React. Вся текстова інформація винесена у два конфігураційні JSON-файли: uk.json (українська) та en.json (англійська).

Обгортка LanguageProvider оточує весь застосунок у файлі layout.tsx (Next.js 14). Вона перевіряє об'єкт localStorage на наявність ключа language. Якщо ключ відсутній, за замовчуванням встановлюється uk.

Коли користувач натискає на перемикач мови у верхній панелі навігації (Navbar), LanguageProvider оновлює свій стан. Оскільки це стан рівня всього застосунку, React миттєво ініціює рендеринг усіх компонентів (перемальовування Virtual DOM). Усі компоненти, що використовують хук useLanguage() та функцію t('key'), отримують нові рядки з відповідного JSON-словника.

Окремою проблемою локалізації є повідомлення про помилки, які генерує backend (наприклад, "Користувача з таким email не знайдено"). Оскільки backend не знає про поточну мову клієнта, він завжди повертає стандартизовані ключі або україномовні відповіді. На клієнті імплементовано функцію-перехоплювач errorTranslations.ts, яка порівнює відповідь сервера зі словником помилок і перекладає її безпосередньо перед відображенням у Toast-повідомленні, забезпечуючи бездоганний користувацький досвід незалежно від обраної мови інтерфейсу.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи досягнуто поставлену мету: сконцептуалізовано, спроєктовано та практично реалізовано відмовостійкий програмний продукт — клієнт-серверну систему обліку робочого часу **Employee Time Tracking (TimeTracker)**, орієнтовану на автоматизацію нарахування відпусток, управління лікарняними та корпоративного інформування. Отримано цілісну екосистему у форматі Full-stack monorepo, яка забезпечує безперервний цикл взаємодії співробітників та адміністраторів у єдиному цифровому просторі, поєднуючи сучасні архітектурні підходи (SPA/SSR) із високими стандартами безпеки та консистентності даних.

На етапі аналізу предметної області з’ясовано, що поширені ринкові рішення (наприклад, глобальні ERP-системи) нерідко демонструють перевантаженість інтерфейсу та високу вартість підписок, тоді як традиційні електронні таблиці призводять до розсинхронізації даних і помилок у розрахунках балансу днів. Обґрунтовано доцільність власного сфокусованого продукту, який спирається на чітке розмежування ролей (EMPLOYEE та ADMIN), механізми автоматичного розрахунку доступних вихідних та інтуїтивно зрозумілий інструментарій самообслуговування.

У частині серверної архітектури (backend) сформовано надійний інформаційний фундамент: для стандартизації обміну даними та централізованої обробки запитів застосовано архітектуру **REST API** на базі середовища **Node.js** та мікрофреймворку **Express.js**. Персистентне зберігання облікових записів, історії нарахувань днів і статусів запитів реалізовано в документо-орієнтованій хмарній базі **MongoDB Atlas** з інтеграцією сучасного інструменту **Prisma ORM**, що гарантує строгу типізацію та цілісність ієрархічних зв'язків користувачів. Забезпечено багаторівневу безпеку: stateless-автентифікацію на основі **JWT**, криптографічне хешування паролів (**bcryptjs**) та суворий контроль доступу на рівні middleware. Алгоритмічне ядро підсилено механізмом динамічного автонарахування вихідних днів (1.5 дня відпустки за кожен відпрацьований місяць), що усуває необхідність ручного втручання та мінімізує вплив людського фактору.

Клієнтську частину (frontend) реалізовано на базі сучасного фреймворку **Next.js 14 (App Router)** та бібліотеки **React 18**, що забезпечує високу продуктивність завдяки оптимізованій маршрутизації. Інтерфейс побудовано в адаптивній парадигмі з використанням **Tailwind CSS**, що гарантує коректне відображення як на десктопних, так і на мобільних пристроях. Вагомим інженерним результатом стало впровадження інтерактивного календаря (**react-calendar**) для візуалізації розрахованих державних свят і затверджених відпусток, а також інтеграція модуля обробки зображень (**react-easy-crop**) для корпоративних аватарів. Додатково реалізовано багатомовність (**i18n**) на базі React Context із збереженням вибору в localStorage, що підвищує зручність роботи з системою для багатонаціональних команд.

Для відтворюваності середовищ і спрощення розгортання застосовано сучасні практики хмарного хостингу. Якість продукту забезпечується строгою типізацією **TypeScript** та стратегією автоматизованого тестування: інтеграційні перевірки серверного шару (Jest, Supertest для HTTP-маршрутів) та клієнтські тести з використанням jsdom і React Testing Library. Це суттєво зменшує ризик регресій при зміні бізнес-логіки та UI-компонентів. Налаштовано безперервний цикл розгортання (CI/CD): фронтенд розміщено на **Vercel**, а серверну логіку — на платформі **Render**.

Таким чином, розроблена система **Employee Time Tracking** відповідає сформульованим вимогам і сучасним індустріальним орієнтирам побудови корпоративних вебзастосунків. Запропонована архітектура створює міцну основу для подальшого масштабування: впровадження Redis-кешування, розширення покриття автотестами, додавання системи сповіщень (Push/Email) про статуси запитів, експорту PDF-звітів, а також інтеграції з зовнішніми сервісами на кшталт Google Calendar.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Google Developers. (2023). Chrome Extensions Documentation: Manifest V3. URL: https://developer.chrome.com/docs/extensions/mv3/ (Дата звернення: 10.03.2026).
2. React Documentation. (2024). React: A JavaScript library for building user interfaces. URL: https://react.dev/ (Дата звернення: 12.03.2026).
3. Vitejs. (2024). Vite: Next Generation Frontend Tooling. URL: https://vitejs.dev/guide/ (Дата звернення: 15.04.2026).
4. GraphQL Foundation. (2024). GraphQL: A query language for your API. URL: https://graphql.org/learn/ (Дата звернення: 18.03.2026).
5. Apollo GraphQL. (2024). Apollo Server Documentation. URL: https://www.apollographql.com/docs/apollo-server/ (Дата звернення: 19.03.2026).
6. Apollo GraphQL. (2024). Apollo Client Documentation. URL: https://www.apollographql.com/docs/react/ (Дата звернення: 19.03.2026).
7. MongoDB Inc. (2024). The MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (Дата звернення: 22.03.2026).
8. Mongoose. (2024). Mongoose ODM v8.0.0 Documentation. URL: https://mongoosejs.com/docs/guide.html (Дата звернення: 23.03.2026).
9. Node.js Foundation. (2024). Node.js v20.x Documentation. URL: https://nodejs.org/en/docs/ (Дата звернення: 25.03.2026).
10. Microsoft. (2024). TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/intro.html (Дата звернення: 26.03.2026).
11. Tailwind Labs. (2024). Tailwind CSS Documentation. URL: https://tailwindcss.com/docs/installation (Дата звернення: 28.03.2026).
12. Radix UI. (2024). Radix Primitives: Unstyled, accessible components for building high-quality design systems. URL: https://www.radix-ui.com/docs/primitives/overview/introduction (Дата звернення: 02.04.2026).

13. Internet Engineering Task Force (IETF). (2015). RFC 7519: JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519 (Дата звернення: 05.04.2026).
14. OWASP Foundation. (2023). REST Security Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (Дата звернення: 08.04.2026).
15. Google AI for Developers. (2024). Gemini API Documentation. URL: https://ai.google.dev/docs (Дата звернення: 10.04.2026).
16. Resend. (2024). Resend API Reference: Email for developers. URL: https://resend.com/docs/api-reference/introduction (Дата звернення: 12.04.2026).
17. Docker Inc. (2024). Docker Documentation. URL: https://docs.docker.com/ (Дата звернення: 14.04.2026).
18. MDN Web Docs. (2024). Web Storage API (localStorage). URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (Дата звернення: 16.04.2026).
19. MDN Web Docs. (2024). Web Speech API (Text-to-Speech). URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API (Дата звернення: 18.04.2026).
20. W3C. (2023). Web Content Accessibility Guidelines (WCAG) 2.1. URL: https://www.w3.org/TR/WCAG21/ (Дата звернення: 20.04.2026).
21. Vitest. (2024). Vitest: A blazing fast unit test framework powered by Vite. URL: https://vitest.dev/guide/ (Дата звернення: 22.04.2026).
22. Testing Library. (2024). React Testing Library Documentation. URL: https://testing-library.com/docs/react-testing-library/intro/ (Дата звернення: 23.04.2026).
23. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
24. Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

25. Render. (2024). Render Cloud Hosting Documentation. URL: https://render.com/docs (Дата звернення: 25.04.2026).