Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

кафедра «Системний аналіз та обчислювальна математика»
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр
(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ ТА ПЕРСОНАЛОМ ДЛЯ КОМАНДИ РОЗРОБНИКІВ
(назва теми)

Виконав(ла): студент(ка) 2 курсу,
групи КНТ-814м _

Спеціальності _____124 - Системний аналіз
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Інтелектуальні технології та прийняття рішень в складних системах

ВОРОНА В.Р.
(ПРІЗВИЩЕ та ініціали)

Керівник ТЕРЕЩЕНКО Е.В.
(ПРІЗВИЩЕ та ініціали)

Рецензент ЛОЗОВСЬКА Л.І.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет___Комп’ютерних наук та технологій___________________________
Кафедра____«Системний аналіз та обчислювальна математика»_____________
Ступінь вищої освіти __Магістр____
Спеціальність____124 Системний аналізу_________________________________
(код і найменування)
Освітня програма (спеціалізація)_______Інтелектуальні технології та прийняття рішень в складних системах____
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри ТЕРЕЩЕНКО Е.В
«10» Листопада 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

______________________Ворона Вадим Романович______________________
(ПРІЗВИЩЕ, ім’я, по батькові)

1. Тема проєкту (роботи)_Дослідження та розробка інформаційної системи управління проєктами та персоналом для команди розробників._____________
керівник проєкту (роботи) Зав. кафедри, доцент Терещенко Еліна Валентинівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім’я, по батькові)
затверджені наказом закладу вищої освіти від «_10_»_11_2025 року №506
2. Строк подання студентом проєкту (роботи)____________________________
3. Вихідні дані до проєкту (роботи)__розроблено програмний застосунок інформаційної системи управління проєктами та персоналом, орієнтований на команду розробників, що функціонують у парадигмі офшорного аутсорсингу.____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)__1. Дослідження проблематики та формулювання вимог до систем управління проєктами та персоналом для команди розробників. 2. Вибір та обґрунтування програмних засобів реалізації. 3. Проєктування та реалізація інформаційної системи управління проєктами та персоналом для команди розробників.____
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, кількість слайдів, плакатів)_14 слайдів презентації________________________

6. Консультанти розділів проєкту (роботи)

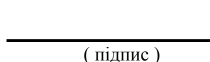
Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1	Зав. кафедри, доцент Терещенко Е. В.	03.11.25	05.12.25
2	Зав. кафедри, доцент Терещенко Е. В.	10.11.25	05.12.25
3	Зав. кафедри, доцент Терещенко Е. В.	10.11.25	05.12.25
Нормоконтроль	Доцент Широкорад Д. В.	10.11.25	12.12.25

7. Дата видачі завдання «10» Листопада 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області	1 тиждень	
2	Розробка та затвердження технічного завдання	2 тиждень	Технічне завдання
3	Проектування структури системи	3 тиждень	Структурна схема
4	Розробка схеми функціонування системи та модулів	4 тиждень	Функціональна схема. Модулі
5	Створення програмного коду системи	5 тиждень	Текст програми
6	Тестування та відлагодження системи	6 тиждень	Працездатна система
7	Розробка розділів ПЗ	6 тиждень	ПЗ
8	Розробка слайдів презентації	7 тиждень	Слайди презентації
9	Захист випускної роботи	8 тиждень	

Студент(ка) ____________ __ВОРОНА В.Р.__
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)  ____________ _ТЕРЕЩЕНКО Е.В._
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 77 с., 17 рис., 1 табл., 1 дод., 15 джерел

Мета роботи – розробка системи моніторингу й оптимізації робочих завдань з метою підвищення індивідуальної та командної ефективності.

Об’єктом дослідження є процес управління робочими завданнями та ефективністю розподілених команд в умовах офшорної розробки програмного забезпечення.

Предметом дослідження є методи, моделі та інформаційні технології моніторингу й оптимізації виконання робочих завдань для підвищення продуктивності персоналу.

Для досягнення поставленої мети необхідно виконати наступні задачі:

– проаналізувати предметну галузь управління робочими завданнями в офшорній розробці програмного забезпечення;

– виконати огляд сучасних систем-аналогів, оцінити їх функціональні можливості, обмеження;

– сформувати вимоги до системи моніторингу й оптимізації завдань;

– розробити структуру програмного застосунка;

– реалізувати програмний прототип, що забезпечує моніторинг стану завдань, облік часу, автоматичну оцінку навантаження та інструменти оптимізації робочого процесу;

– виконати тестування та оцінювання ефективності, перевірити працездатність системи, стабільність, точність розрахунків та зручність використання.

Проєкт системи моніторингу й оптимізації робочих завдань розроблено з використанням фреймворку Yii2, що працює на мові PHP. Окрім збору та зберігання інформації, система забезпечує відображення даних у вигляді таблиць і графіків у режимі реального часу, формування різноманітних звітів, а також інструменти для архівації та захисту інформації.

ПРОЄКТ, СИСТЕМА МОНІТОРИНГУ, БАЗА ДАНИХ

ЗМІСТ

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ..………………..….…2

ВСТУП 8

1. ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ ТА ФОРМУЛЮВАННЯ ВИМОГ ДО СИСТЕМ УПРАВЛІННЯ ПРОЄКТАМИ ТА ПЕРСОНАЛОМ ДЛЯ КОМАНДИ РОЗРОБНИКІВ 11

1.1 Проблематика, яку покликані вирішити системи управління проєктами та персоналом для команди розробників 11

1.2 Огляд комерційних систем управління проєктами 12

1.2.1 Atlassian Jira 13

1.2.2 Microsoft Azure DevOps 15

1.2.3 Вузькоспеціалізовані системи Тайм-Трекінгу 17

1.2.4 Noko Time Tracking 19

1.3 Висновки по першому розділу 21

2. ВИБІР ТА ОБҐРУНТУВАННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ 23

2.1 Мова програмування PHP для серверної частини веб-систем 23

2.2 Характеристика та обґрунтування використання формату JSON 26

2.3 База даних PostgreSQL 28

2.4 Наслідування в PostgreSQL 30

Висновки по другому розділу 31

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ ТА ПЕРСОНАЛОМ ДЛЯ КОМАНДИ РОЗРОБНИКІВ 33

3.1 Системний аналіз операційних процесів управління проєктами та персоналом 33

3.2 Обґрунтування архітектури системи управління проєктами та персоналом 36

3.3 Розроблення архітектури програмного застосунку 37

3.4 Розробка БД PostgreSQL 42

3.5 Програмна реалізація веб застосунку 45

3.6 Адміністративний контур 46
3.7 Користувацький контур 51
3.8 Тестування застосунку 53
Висновки по третьому розділу 54
ВИСНОВКИ 56
ПЕРЕЛІК ПОСИЛАНЬ 57
ДОДАТОК А ТЕКСТ ПРОГРАМИ 59

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ ТА ТЕРМІНІВ

ПЗ – Програмне забезпечення

ПК – Персональний комп'ютер

ОС – Операційна система

ОП – Оперативна пам'ять

СКБД – Система керування базами даних

PostgreSQL – Об’єктно-реляційна система керування базами даних; альтернатива комерційним та відкритим СКБД

PHP – Серверна мова сценаріїв, що вбудовується безпосередньо в HTML-код

JSON - JavaScript Object Notation, текстовий формат для обміну даними

ВСТУП

В умовах глобалізації та зростаючого попиту на цифрові продукти, офшорна модель розробки програмного забезпечення та веб-додатків набула критичного значення для забезпечення конкурентоспроможності вітчизняних IT-компаній на міжнародному ринку. Однак, ця модель, незважаючи на економічні переваги, стикається зі специфічними викликами, які суттєво впливають на ефективність організації робочого процесу. Ключові проблеми, такі як значна різниця в часових поясах, необхідність підтримки високого рівня прозорості для іноземного замовника та складність об'єктивного моніторингу продуктивності віддалених команд, часто призводять до затримок, перевищення бюджету та зниження якості кінцевого продукту, особливо при розробці аналогів існуючих рішень, де швидкість виходу на ринок є вирішальною. Саме тому актуальність даного дослідження зумовлена об’єктивною потребою в розробці та впровадженні інструментарію, здатного системно збирати, аналізувати та візуалізувати дані про робочі завдання, трансформуючи їх у дієві рекомендації для оптимізації. Створення такої системи дозволить мінімізувати суб’єктивізм в оцінці роботи, покращити прогнозування термінів виконання та забезпечити синхронізацію між українською командою та іноземним замовником.

Метою створення системи моніторингу й оптимізації робочих завдань є підвищення ефективності організації робочого процесу при розробці програмного забезпечення та веб-додатків для іноземних замовників у форматі офшорної моделі. Система моніторингу й оптимізації робочих завдань — це програмна система класу task-manager, яка містить типовий функціонал для взаємодії між учасниками команд, але при цьому має низку специфічних можливостей, адаптованих під вимоги конкретного замовника.

Однією з особливостей системи є нетривіальна архітектура, побудована на принципі опосередкованого множинного успадкування. Йдеться про те, що класи запозичують методи та властивості один у одного не через пряме

багаторівневе успадкування, а через комбінацію базового класу та об'єктів, які створюються в похідних класах для доступу до необхідної логіки. Такий підхід дозволив суттєво пришвидшити роботу додатка і зменшити навантаження як на сервер, так і на СКБД.

Загалом система складається з трьох основних підсистем:

- модуль для менеджерів і адміністраторів, який забезпечує контроль роботи розробників та комунікацію з клієнтом;
- підсистема для програмістів, де фіксуються відпрацьовані години, відображаються призначені завдання та їхній стан;
- система керування базами даних.

Розподіл завдань між членами команди здійснюється з урахуванням їх компетенцій та досвіду, а також впливає на формування заробітної плати. Система моніторингу й оптимізації робочих завдань веде облік стану кожного активного проєкту та дозволяє відображати основні метрики його виконання. У рамках системи реалізовано:

- TO DO-лист, де зібрані всі актуальні задачі й проєкти, кожен з яких має власну гілку для обговорення;
- інфографічну сторінку, що відображає рейтинги розробників, загальну статистику по підрозділах, статус виконання завдань та витрачений час;
- механізм обчислення заробітної плати, який залежить від відпрацьованих годин, персональних рейтингів співробітників і результатів роботи підрозділів;
- репозиторій проєктів, що містить як поточні, так і завершені роботи з різним рівнем доступу для внутрішнього користування.

Підсистема зберігання даних базується на PostgreSQL — цю СКБД було обрано з огляду на її стабільність, широкий набір можливостей та відкриту ліцензію.

Проєкт системи моніторингу й оптимізації робочих завдань розроблено з використанням фреймворку Yii2, що працює на мові PHP. Окрім збору та зберігання інформації, система забезпечує відображення даних у вигляді

таблиць і графіків у режимі реального часу, формування різноманітних звітів, а також інструменти для архівації та захисту інформації.

1 ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ ТА ФОРМУЛЮВАННЯ ВИМОГ ДО СИСТЕМ УПРАВЛІННЯ ПРОЄКТАМИ ТА ПЕРСОНАЛОМ ДЛЯ КОМАНДИ РОЗРОБНИКІВ

1.1 Проблематика, яку покликані вирішити системи управління проєктами та персоналом для команди розробників

Стрімкий розвиток IT-індустрії та широке застосування моделі офшорного аутсорсингу обумовлюють зростання складності управління проєктами. Впровадження систем управління завданнями (таск-менеджменту) є прямим наслідком об'єктивної потреби в систематизації та оптимізації робочого процесу. Проблематика, яку покликані вирішити такі системи, охоплює чотири ключові аспекти, критичні для забезпечення ефективності.

По-перше, однією з фундаментальних проблем є децентралізація та відсутність прозорості в інформаційному обміні. Використання несистематизованих засобів комунікації (електронна пошта, месенджери) призводить до фрагментації даних, втрати контексту завдань та унеможливлює швидке отримання актуального статусу роботи. Система таск-менеджменту слугує єдиним джерелом істини (Single Source of Truth), централізуючи завдання, документацію та історію обговорень, чим забезпечує синхронну видимість прогресу для всіх зацікавлених сторін.

По-друге, існує проблема неефективного розподілу та обліку ресурсів. Без об’єктивного інструментарію менеджери проєктів зіштовхуються зі значними труднощами в моніторингу реального завантаження персоналу та точному вимірюванні трудомісткості робіт. Це призводить до дисбалансу навантаження та, як наслідок, до уповільнення загального циклу розробки (Cycle Time). Критичною функцією таск-менеджерів у цьому контексті є Time Tracking (облік часу), який є основою для точного білінгу в офшорній моделі та джерелом даних для аналізу продуктивності.

По-третє, сучасна розробка вимагає чіткого управління життєвим циклом завдання (Task Lifecycle) та залежностями. Складні проєкти включають

послідовність взаємозалежних етапів (розробка, код-рев'ю, тестування). Нездатність адекватно керувати цими залежностями спричиняє простої та Bottlenecks (вузькі місця). Системи управління завданнями формалізують робочий процес (Workflow), дозволяючи менеджерам проєкту візуалізувати й контролювати проходження завдання кожною стадією, забезпечуючи тим самим передбачуваність термінів виконання.

Таким чином, системи таск-менеджменту трансформуються з простого інструменту для створення списків у комплексний аналітичний центр, який перетворює неструктуровану діяльність у вимірюваний, керований та оптимізований процес. Це є прямим обґрунтуванням необхідності розробки спеціалізованої системи, орієнтованої на підвищення ефективності саме в умовах офшорного аутсорсингу.

1.2 Огляд комерційних систем управління проєктами

Для всебічного обґрунтування технічної доцільності розробки власної системи, у цьому підрозділі буде проведено аналіз існуючих аналогів (комерційних систем управління проєктами). Цей огляд дозволить ідентифікувати ключові переваги ринкових рішень та, що є найважливішим, виявити їхні функціональні та економічні обмеження, які не задовольняють специфічні потреби офшорної моделі роботи невеликої команди.

1.2.1 Atlassian Jira

Jira Software [1] є спеціалізованою інформаційною системою, розробленою компанією Atlassian, що займає провідні позиції у сфері управління проєктами та завданнями. Цей інструмент був спеціально спроєктований для команд, які займаються розробкою програмного забезпечення (ПЗ), вирізняючись високою гнучкістю та адаптивністю. Завдяки цим характеристикам, Jira може бути налаштована під різні методології управління, включаючи Scrum, Kanban або гібридні підходи (рис.1.1).

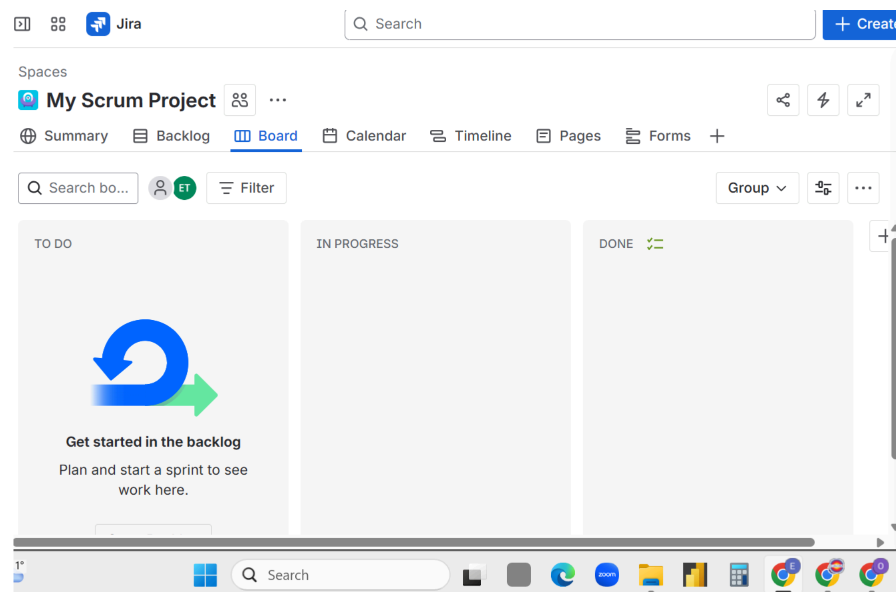


Рисунок 1.1 – Atlassian Jira

Jira Software надає широкий спектр функцій, спрямованих на ефективну організацію роботи, відстеження прогресу, координацію та досягнення результатів у проєктах розробки програмного забезпечення. Розглянемо ключовий функціонал Jira Software.

Управління життєвим циклом завдання. Дозволяє створювати завдання, організовуючи їх у вигляді беклога або спринтів. Надає можливість визначати майлстоуни (ключові етапи проєкту) та встановлювати пріоритети. Кожне завдання має опис, може містити коментарі, вкладення і теги, забезпечуючи

повну прозорість. Чітке відстеження прогресу виконання через налаштовувані статуси (наприклад, "Відкрита", "У процесі", "Завершена").

Планування та координація ресурсів. Система надає інструменти для планування, встановлення термінів, розподілу завдань і ресурсів.Дозволяє відстежувати залежності між завданнями, що є важливим для зниження ризиків та забезпечення своєчасного виконання зобов'язань.

Моніторинг прогресу та звітність. Пропонує інструменти для візуалізації поточного стану проєкту, зокрема через використання діаграм Гантта, дошок Kanban або графіків Burndown/Velocity. Це забезпечує команді та керівництву чітке уявлення про поточний стан виконання завдань і спрощує формування прозорої звітності.

Співпраця та комунікація. Jira полегшує співпрацю, дозволяючи учасникам обговорювати завдання безпосередньо в системі, обмінюватися файлами та відстежувати згадки. Це сприяє обміну ідеями та оперативному вирішенню проблем.

Розширення та інтеграції. Система підтримує численні інтеграції з іншими інструментами розробки (такими як Bitbucket, Confluence, GitHub), дозволяючи працювати в єдиній екосистемі. Архітектура підтримує розширення та плагіни, що дає змогу налаштувати функціональність під індивідуальні потреби команди.

Jira розроблена на мові програмування Java і є частиною екосистеми Atlassian, що гарантує її надійність, постійну підтримку та активну спільноту користувачів. У підсумку, Jira є потужним і гнучким інструментом, який використовується для ефективного управління проєктами розробки ПЗ.

Таким чином, Jira є потужним менеджером завдань, але не є оптимальною системою оптимізації та моніторингу для малих команд в офшорній моделі через свою високу складність, непрямі фінансові витрати на критичну аналітику та недостатню орієнтацію на білінгові потреби та прозорість для клієнта.

1.2.2 Microsoft Azure DevOps

Microsoft Azure DevOps (ADO) [2] являє собою комплексний набір інструментів, що охоплює весь життєвий цикл розробки програмного забезпечення, від планування та керування вимогами до автоматизованого розгортання та моніторингу. Система особливо популярна серед великих підприємств та команд, які використовують технологічний стек Microsoft (рис. 1.2).

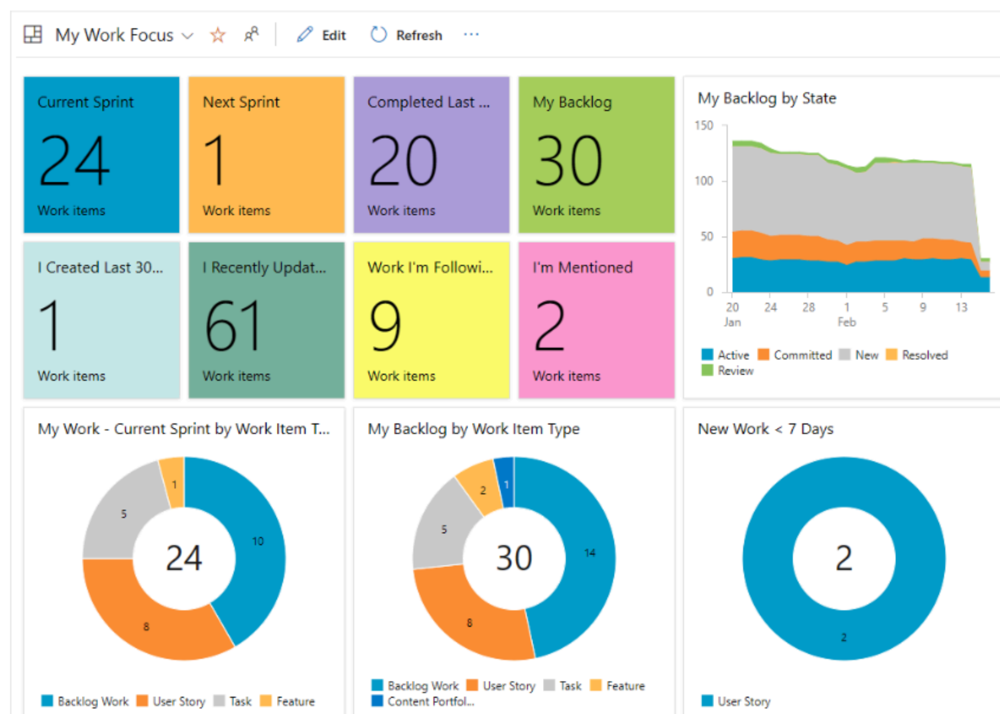


Рисунок 1.2 – Microsoft Azure DevOps

Основне призначення та функціональність. Azure DevOps поєднує п'ять ключових модулів, які забезпечують єдину платформу для роботи: Azure Boards (управління завданнями), Azure Repos (керування версіями), Azure Pipelines (CI/CD), Azure Test Plans (тестування) та Azure Artifacts (керування пакетами).

Управління завданнями (Azure Boards): підтримує методології Agile (Scrum, Kanban) та надає інструменти для визначення робочих елементів (Work Items), їх пріоритезації, відстеження та візуалізації на дошках.

Інтегрований пайплайн розробки: Ключовою перевагою є тісна інтеграція з CI/CD-пайплайнами (Azure Pipelines), що дозволяє автоматизувати збірку, тестування та розгортання коду безпосередньо з системи управління завданнями.

До сильних сторін Azure DevOps (ADO) відносяться такі.

Наскрізна інтеграція життєвого циклу: ADO пропонує безшовну інтеграцію між завданнями, кодом, тестами та розгортанням. Це створює високий рівень консистентності даних та спрощує відстеження зв'язку між вимогами та кінцевим продуктом.

Потужні засоби звітності: Система має вбудовані, розвинені засоби звітності, які дозволяють створювати детальні діаграми та дашборди для внутрішнього моніторингу прогресу, якості коду та швидкості виконання.

Екосистема Microsoft: Максимальна ефективність досягається при роботі з технологіями Microsoft (Azure Cloud, .NET), забезпечуючи високу продуктивність та стабільність.

У контексті невеликої офшорної команди, що потребує спеціалізованої системи моніторингу та оптимізації, Azure DevOps має суттєві недоліки.

Складність впровадження та налаштування: ADO є корпоративним рішенням з високим порогом входу. Його налаштування та адміністрування вимагає значних знань і часу, що є непродуктивним для малих та середніх команд.

Фокус на внутрішніх процесах: система є більше орієнтованою на внутрішню команду розробки та технічне керівництво, а не на зовнішню прозорість для клієнта. Надання клієнту простого та зрозумілого доступу до ключових метрик білінгу та прогресу є ускладненим через багатошаровість інтерфейсу.

Неоптимальна адаптація до змішаних стеків: Хоча ADO підтримує не-Microsoft технології, його архітектура та глибина інтеграції можуть бути менш ефективними, ніж у спеціалізованих інструментів, якщо команда працює переважно з відкритим (Open Source) стеком.

Спеціалізована аналітика: Подібно до Jira, хоча ADO має потужні звіти, створення вузькоспеціалізованих метрик оптимізації (як-от індекс навантаження або фінансова звітність для білінгу в індивідуальному форматі замовника) часто потребує ручного доопрацювання або використання складних інструментів Power BI, що виходить за рамки простих потреб для малих та середніх команд.

Таким чином, Azure DevOps є надмірно складним і громіздким для завдань для малих та середніх команд, а його фокус на повному життєвому циклі розробки не відповідає необхідності створення висококсфокусованої, легкої системи моніторингу та прозорості.

1.2.3 Вузькоспеціалізовані системи Тайм-Трекінгу

В арсеналі інструментів управління робочими процесами існує окремий клас вузькоспеціалізованих систем, призначених виключно для обліку робочого часу (Time Tracking), прикладами яких є Clockify [3] та Trackabi [4](рис.3). Ці рішення набули широкого застосування в бізнес-моделях, де оплата праці або білінг клієнтів здійснюється на погодинній основі, зокрема, у сфері аутсорсингу послуг.

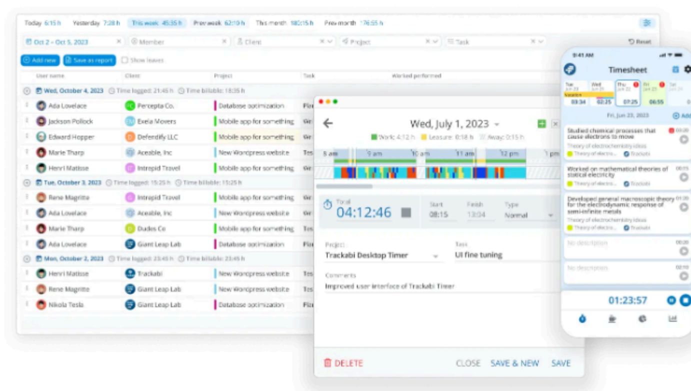


Рисунок 1.3 – Trackabi

Основна функціональна роль таких систем полягає у забезпеченні високої точності фіксації затрат часу на виконання конкретних завдань чи проєктів. Це досягається завдяки інтуїтивно зрозумілому інтерфейсу, можливості запуску/зупинки таймера в один клік, а також автоматичному збору даних про активність користувача. Як наслідок, ці платформи є надзвичайно ефективними для формування фінансової звітності та інвойсингу. Завдяки наявності API, вони легко інтегруються з іншими корпоративними системами, виконуючи функцію точного джерела даних про робочі години.

Незважаючи на високу точність обліку, системи тайм-трекінгу мають значні обмеження в контексті управління та оптимізації процесів. Їхній функціонал є пасивним і реактивним, оскільки вони фіксують лише факт витраченого часу, але не керують самим робочим процесом.

Відсутність управління процесами. Ці інструменти не містять модулів для планування, визначення залежностей між завданнями, налаштування складних робочих потоків (Workflow) або розподілу навантаження.

Обмеженість аналітики. Аналітичні можливості здебільшого зводяться до фінансового виміру. Системи не здатні самостійно ідентифікувати неефективність процесу, наприклад, визначити середній час проходження завдання (Cycle Time), виявити затримки, спричинені очікуванням, або визначити конкретні "вузькі місця" (Bottlenecks) у команді.

Отже, хоча вузькоспеціалізовані системи обліку часу є незамінними для фінансової прозорості, вони є недостатніми для комплексного моніторингу та активної оптимізації продуктивності робочого колективу. Для досягнення цілей ефективності необхідно інтегрувати функціонал точного обліку часу з розширеним аналітичним та управлінським інструментарієм.

1.2.4 Noko Time Tracking

Система Noko Time Tracking [5] є вузькоспеціалізованим інструментом для обліку робочого часу, який вирізняється на ринку завдяки своєму фокусу на простоті використання, високій точності обліку та генерації фінансових звітів (рис. 1.4).

Noko Time Tracking Review

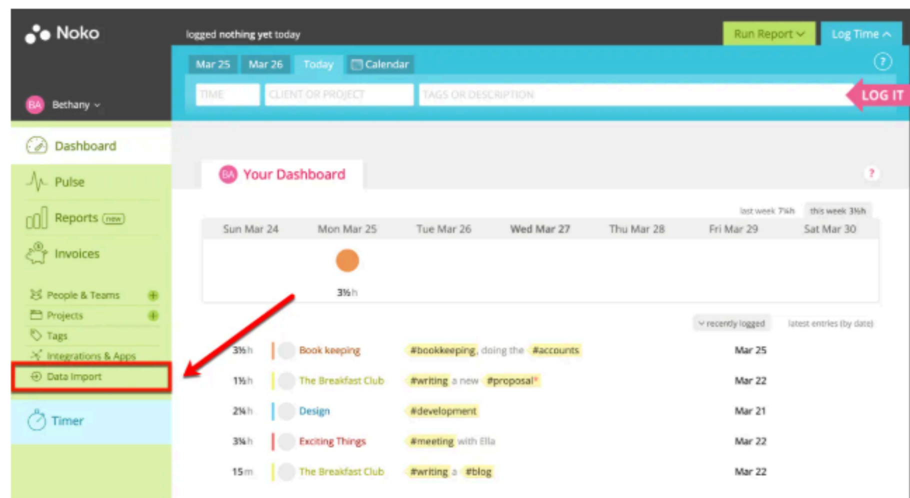


Рисунок 1.4 – Noko Time Tracking

Хоча продукт був перейменований, його основна функціональна філософія залишається незмінною: максимальна ефективність обліку часу за мінімальних зусиль користувача. Розглянемо функціонал цієї системи.

Спрощений Облік Часу та Класифікація через Тегування: Noko мінімізує рутинність введення даних. Замість складних багаторівневих описів, система активно використовує хештеги (#дизайн, #листування) для швидкої категоризації витраченого часу. Це дозволяє системі автоматично класифікувати робочі активності та формувати статистику.

Візуалізація та звітність для білінгу. Система надає потужні, але прості у сприйнятті звіти. На головному дашборді одразу відображаються діаграми розподілу часу за днями, що забезпечує оперативний контроль. Це критично важливо для офшорної моделі, оскільки Noko дозволяє автоматично генерувати білінгові звіти (Invoicing Reports), необхідні для виставлення рахунків іноземним замовникам.

Висока Користувацька Зручність. Система спроєктована з акцентом на мінімалізм, що забезпечує низький поріг входу та зменшує час, який команда витрачає на адміністрування обліку.

Незважаючи на високу ефективність у трекінгу, Noko (Freckle) не може виконувати функцію повноцінної системи моніторингу й оптимізації через свою вузьку спеціалізацію:

Noko є пасивним інструментом обліку, який не містить функціоналу для активного управління процесами. У ньому відсутні модулі для планування завдань, розподілу ресурсів, керування Workflow або визначення залежностей між завданнями.

Система обмежується фінансовим обліком та звітами про використання часу. Вона не проводить аналіз ефективності процесу розробки — не ідентифікує Bottlenecks (вузькі місця), не розраховує ключові метрики Cycle Time (час проходження завдання) або коефіцієнт навантаження команди.

Система не має безкоштовного тарифу, а платна підписка навіть для мінімальної кількості користувачів становить постійні операційні витрати, що є недоцільним для малих та середніх команд, які прагнуть до раціоналізації витрат.

У підсумку, Noko є чудовим джерелом точних даних про час, але не є інструментом оптимізації. Розроблювана система має інтегрувати його ключову перевагу (простий, точний облік часу) з розширеним управлінським та аналітичним функціоналом.

1.3 Висновки по першому розділу

Хоча на ринку представлено широкий спектр потужних комерційних систем управління проєктами, їхнє використання для малих та середніх команд, які працюють за моделлю офшорного аутсорсингу, часто є неоправданим з точки зору вартості, складності та адаптивності.

Доцільність розробки власної системи для малих та середніх команд обґрунтовується такими ключовими факторами.

1. Економічна ефективність та ліцензійна політика (Cost Efficiency and Licensing). Більшість комерційних рішень (наприклад, Jira, Asana) пропонують ліцензії, які хоч і мають безкоштовний рівень, але часто обмежують критичний функціонал (наприклад, розширену аналітику чи інтеграції). Перехід на платний тарифний план може бути нерентабельним для малих та середніх команд, особливо з урахуванням необхідності придбання додаткових плагінів для аналітики.

Власна розробка усуває постійні ліцензійні витрати та дозволяє інвестувати ресурси лише один раз у створення цільового функціоналу.

2. Сфокусованість та уникнення надмірності (Focus and Avoiding Feature Bloat). Великі системи спроєктовані для тисяч користувачів та складних корпоративних процесів. Невелика команда використовує лише 10–20% доступного функціоналу. Зайві опції ускладнюють інтерфейс, збільшують час навчання та знижують продуктивність замість її підвищення.

Власна система може бути спроєктована мінімалістично, включаючи лише ті інструменти, які критично важливі для офшорної моделі: точний Time Tracking, Dashboard для клієнта та базове управління завданнями. Це робить систему швидшою та інтуїтивно зрозумілою для невеликого колективу.

3. Специфічна орієнтація на метрики офшорної розробки (Tailored KPI Integration). Стандартні системи пропонують загальні метрики. Адаптація їх під специфічні потреби офшорної моделі — розрахунок "Індексу Вузьких Місць" (Bottleneck Index), аналіз часу проходження завдання (Cycle Time) з

урахуванням часових поясів або автоматизоване формування білінгових звітів за конкретним форматом замовника — вимагає складного кастомізованого програмування або дорогих плагінів.

Розробка власної системи дозволяє вбудувати необхідну аналітику безпосередньо в її ядро.

4. Повна інтеграція та контроль (Full Control and Seamless Integration). Невеликі команди часто використовують унікальний або гібридний технологічний стек. Інтеграція комерційних систем із цими інструментами може бути обмежена або потребувати додаткових витрат.

Власна програма забезпечує повний контроль над API та архітектурою. Це гарантує безшовну інтеграцію з Git-репозиторіями, внутрішніми засобами комунікації та іншими інструментами, які вже використовуються командою, забезпечуючи максимальну автоматизацію збору даних.

Таким чином, для малих та середніх команд, що працюють в умовах офшорного аутсорсингу, власна розробка системи є не лише економічно виправданою, але й стратегічно важливою для створення високоспеціалізованого, сфокусованого та максимально адаптованого інструменту для моніторингу й оптимізації продуктивності.

2. ВИБІР ТА ОБҐРУНТУВАННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

Успішна реалізація інформаційної системи вимагає критично обґрунтованого вибору інструментально-технологічного комплексу, який повинен оптимально відповідати як функціональним вимогам проєкту, так і архітектурним принципам надійності, масштабованості та економічної ефективності. Тому в даному розділі буде здійснено вибір та детальний аналіз ключових засобів реалізації: мови програмування високого рівня PHP, реляційної системи управління базами даних PostgreSQL, а також формату обміну даними JSON.

2.1 Мова програмування PHP для серверної частини веб-систем

Обґрунтування вибору технологічної платформи є критичним етапом у проєктуванні інформаційної системи. Для реалізації серверної логіки даної системи обрано мову програмування PHP (Hypertext Preprocessor) [6]. Розвиток PHP розпочався у 1994 році завдяки Расмусу Лердорфу як набір CGI-сценаріїв, призначених для спрощення роботи з персональними веб-сторінками.

Ключові віхи еволюції PHP, які підтверджують її зрілість як корпоративного інструменту. PHP/FI (1995): Перша публічна версія, яка поєднувала PHP-інтерпретатор та функціонал для роботи з базами даних (Form Interpreter). PHP 3 (1997): Повноцінний перезапуск мови, розроблений спільно з Зеевом Сураскі та Енді Гутмансом (засновниками Zend Technologies). Саме тоді мова отримала свою рекурсивну назву (PHP: Hypertext Preprocessor) та набула архітектурної основи, що дозволила їй стати повноцінною мовою програмування. PHP 4 (2000): Впровадження двигуна Zend Engine 1.0, який значно підвищив стабільність та продуктивність. Ця версія закріпила позиції PHP як однієї з ключових технологій у стеку LAMP (Linux, Apache, MySQL,

PHP). PHP 5 (2004): Впровадження об'єктно-орієнтованої парадигми (ООП), що зробило мову привабливою для розробки великих, складних корпоративних систем. PHP 7.x (2015): Революційний етап, пов'язаний із запуском Zend Engine 3. Ця оптимізація забезпечила дво-трикратне підвищення продуктивності та зниження споживання пам'яті, фактично усунувши застарілі уявлення про низьку швидкодію мови. PHP 8.x (2020): Продовження оптимізації та впровадження сучасних функцій (JIT-компіляція, іменовані аргументи), що підтверджує постійну актуальність та інноваційність платформи.

Подальша еволюція тісно пов'язана з ідеологією Open Source, що забезпечило швидкий розвиток, активну підтримку світової спільноти та інтеграцію з іншими ключовими відкритими технологіями, такими як Linux та Apache.

Вибір PHP для реалізації серверної частини системи моніторингу й оптимізації ґрунтується на її архітектурних особливостях та функціональній зрілості, що відповідають вимогам до сучасних веб-додатків.

Архітектурна специфіка та масштабованість. PHP є серверною, мультиплатформенною мовою сценаріїв, що виконує ключову функцію — генерацію динамічного вмісту.

Модель виконання. PHP-код виконується на веб-сервері (наприклад, Apache або Nginx) за допомогою спеціалізованого інтерпретатора (Zend Engine). Це забезпечує поділ відповідальності між клієнтською частиною (HTML, CSS, JavaScript) та серверною логікою, що є основою архітектури клієнт-сервер.

Інтеграція в HTML. Здатність PHP-скриптів інтегруватися безпосередньо в HTML-код спрощує розробку шаблонів та швидку генерацію вихідних сторінок, що є вигідним для систем, які часто оновлюють візуалізацію даних (дашборди, звіти).

Горизонтальна масштабованість. Завдяки своїй shared-nothing архітектурі (кожний запит обробляється незалежно), PHP чудово масштабується горизонтально. Це означає, що для збільшення пропускної здатності системи достатньо додати більше веб-серверів, що є критично важливим для

забезпечення стабільної роботи системи моніторингу під зростаючим навантаженням.

Ефективність та продуктивність виконання. Сучасні версії PHP повністю розвінчали міф про її низьку швидкість.

Zend Engine та Оптимізації. Із впровадженням Zend Engine 3 (PHP 7.x) було досягнуто значного збільшення продуктивності та зниження споживання оперативної пам'яті. Це забезпечує високу швидкість обробки запитів (Time to First Byte) і конкурентну швидкість виконання коду.

JIT-компіляція (PHP 8.x). Введення механізму Just-In-Time (JIT) компіляції дозволяє компілювати PHP-код у машинні інструкції безпосередньо під час виконання. Це особливо вигідно для завдань, що потребують інтенсивних обчислень, як-от аналітична обробка даних та формування складних звітів, які є ключовими для системи оптимізації.

Простота розробки та екосистема. Низький поріг входу та швидкість розробки (Rapid Development). Відносно простий синтаксис і чітка структура мови дозволяють розробникам швидко опановувати її та прискорювати процес кодування. Це є важливим фактором для ефективного управління ресурсами в рамках дипломного проєкту.

Зрілість фреймворків. Наявність потужних, зрілих фреймворків, таких як Laravel та Symfony, забезпечує готову архітектуру (наприклад, MVC — Model-View-Controller), безпеку, ORM (Object-Relational Mapping) для роботи з базами даних та механізми маршрутизації. Це дозволяє зосередитися на бізнес-логіці системи моніторингу, а не на інфраструктурних завданнях.

Підтримка Баз Даних та інтеграція. Широка підтримка СУБД. PHP має вбудовану підтримку для роботи з більшістю популярних реляційних та NoSQL баз даних, що забезпечує гнучкість при виборі сховища даних для метрик і завдань.

Розширюваність. Завдяки активній спільноті та відкритому коду, PHP має величезну кількість готових бібліотек і розширень, які спрощують інтеграцію з зовнішніми сервісами (наприклад, Git-репозиторіями, API

тайм-трекерів, корпоративними месенджерами), що є критично важливим для системи моніторингу.

Враховуючи ці фактори, PHP пропонує ідеальне поєднання гнучкості, продуктивності, економічної ефективності та інструментальної підтримки, що робить його оптимальним вибором для реалізації сучасної веб-орієнтованої Системи моніторингу й оптимізації.

2.2 Характеристика та обґрунтування використання формату JSON

У сучасних розподілених веб-системах вибір формату обміну даними є критичним архітектурним рішенням, що впливає на продуктивність, масштабованість та інтероперабельність компонентів. Для реалізації інформаційної взаємодії в розроблюваній системі було обрано формат JSON (JavaScript Object Notation).

JSON — це текстовий формат обміну даними, який базується на підмножині мови програмування JavaScript (стандарт ECMA-262) [7]. Незважаючи на своє походження, JSON є повністю мовно-незалежним форматом (language-agnostic), стандартизованим специфікацією IETF RFC 8259. Він підтримується всіма сучасними мовами програмування (PHP, Python, Java, C#) через нативні бібліотеки серіалізації та десеріалізації.

На відміну від альтернативних форматів (зокрема XML), JSON орієнтований на мінімізацію синтаксичної надлишковості, що робить його оптимальним для передачі даних у мережах з обмеженою пропускною здатністю, що є характерним для сценаріїв офшорної віддаленої роботи.

Структурна Організація Даних. З точки зору структури, JSON базується на двох універсальних структурах даних, які є фундаментальними для комп'ютерних наук:

Невпорядкована множина пар «ключ-значення» (Object). У різних мовах програмування реалізується як об'єкт, запис, структура (struct), словник (dictionary) або хеш-таблиця. Синтаксично описується фігурними дужками { }.

Впорядкований список значень (Array) реалізується як масив, вектор або список. Синтаксично позначається квадратними дужками [ ].

Формат підтримує базові типи даних: рядки (Strings), числа (Numbers), логічні значення (Booleans), null, а також вкладені об'єкти та масиви. Така ієрархічна структура ідеально відповідає об'єктній моделі розроблюваної системи, де проєкти містять списки завдань, а завдання — набори атрибутів.

Використання JSON у проєкті є стратегічним рішенням, яке об'єднує всі рівні архітектури в єдиний технологічний стек:

Транспортний рівень (Клієнт-Сервер). JSON виступає стандартом для асинхронної взаємодії (AJAX). Клієнтський інтерфейс (на базі JavaScript) відправляє запити на сервер і отримує відповіді у форматі JSON. Завдяки тому, що JSON є "рідним" форматом для JavaScript, парсинг даних на клієнті відбувається нативними засобами браузера (рушій V8) з максимальною швидкодією, без накладних витрат, властивих DOM-парсингу XML.

Завдяки вибору СУБД PostgreSQL, JSON використовується не лише для передачі, а й для зберігання даних. Використання типу даних JSONB дозволяє зберігати напівструктуровані дані (динамічні налаштування користувачів, змінні атрибути завдань) безпосередньо в базі даних. Це усуває необхідність у складних процедурах ORM-мапінгу для цих сутностей.

JSON використовується для зберігання конфігураційних файлів проєкту (наприклад, composer.json для PHP-залежностей, package.json для клієнтських модулів), що забезпечує уніфікацію підходів до управління інфраструктурою.

Вибір JSON забезпечує синхронізацію технологічного стеку: JavaScript на клієнті, PHP на сервері та PostgreSQL (JSONB) у сховищі даних. Це дозволяє створити високоефективну систему з низькими затримками (low latency), простою структурою коду та високою адаптивністю до змін у моделі даних.

2.3 База даних PostgreSQL

Реляційні системи управління базами даних (РСУБД) посідають домінуюче місце в архітектурі корпоративних систем. Завдяки появі відкритих операційних систем (Linux, FreeBSD) та розвитку вільного ПЗ, ринок отримав доступні та високопродуктивні рішення. У цьому контексті PostgreSQL обирається як найбільш функціонально потужний та архітектурно зрілий представник галузі для реалізації системи моніторингу й оптимізації робочих завдань [8].

Вибір PostgreSQL зумовлений комплексом архітектурних переваг, які критично важливі для специфіки даного проєкту.

1. Гібридна модель даних (JSONB) та гнучкість. PostgreSQL позиціонується як об'єктно-реляційна СУБД (ORDBMS). Вибір цієї системи продиктований необхідністю реалізації гібридної моделі даних, що поєднує строгість реляційних схем із гнучкістю NoSQL-рішень. У контексті системи управління проєктами, де атрибути завдань можуть змінюватися, використання бінарного формату JSONB дозволяє зберігати напівструктуровані дані (динамічні мета-дані, налаштування користувачів) без необхідності постійної зміни схеми бази даних (schema migration). Критичною технічною перевагою тут є механізми GIN-індексації (Generalized Inverted Index). На відміну від аналогів, PostgreSQL дозволяє виконувати пошук по атрибутах всередині JSON-документів з тією ж швидкістю, що і по звичайних стовпцях. Це дозволяє уникнути використання архітектурного анти-патерну EAV (Entity-Attribute-Value), який часто призводить до деградації продуктивності.

2. Транзакційна надійність (ACID) для фінансового модуля. Наявність фінансового модуля (білінг, нарахування заробітної плати) висуває безальтернативні вимоги до надійності. PostgreSQL забезпечує суворе дотримання стандартів ACID (Atomicity, Consistency, Isolation, Durability). Це гарантує цілісність фінансових даних навіть у випадку апаратних збоїв.

Можливість використання транзакційного DDL (зміни структури БД всередині транзакції) додатково підвищує надійність процесу розгортання оновлень.

3. Конкурентний доступ та MVCC

Для забезпечення стабільної роботи багатокористувацької системи використовується механізм багатоверсійного контролю конкурентності (MVCC). Він гарантує, що операції читання не блокують операції запису. Це критично важливо для сценарію, коли менеджери генерують "важкі" аналітичні звіти за місяць — це жодним чином не блокує роботу розробників, які паралельно вносять дані про відпрацьований час (Time Tracking).

4. Аналітичні можливості (OLAP)

Система моніторингу вимагає потужних інструментів для аналітичної обробки даних. Оптимізатор запитів (Query Planner) PostgreSQL ефективно справляється зі складними багатотабличними з'єднаннями (JOINs). Нативна підтримка Common Table Expressions (CTE) та розширених віконних функцій дозволяє перенести складну логіку розрахунку метрик ефективності (KPI) безпосередньо на рівень СУБД, розвантажуючи серверну частину додатку.

5. Розширюваність та бізнес-логіка. PostgreSQL надає потужні засоби для вбудовування бізнес-логіки. Серверні функції та процедури дозволяють інкапсулювати складну логіку (на PL/pgSQL, Python, C).

Тригериа автоматизують контроль цілісності при подіях INSERT/UPDATE/DELETE.

Підтримка часткових та функціональних індексів дозволяє тонко налаштовувати продуктивність.

6. Ліцензійна чистота. Вибір PostgreSQL мінімізує юридичні ризики в контексті офшорної розробки. Система поширюється під ліберальною PostgreSQL License (аналог MIT/BSD), яка надає повну свободу використання без ризиків, пов'язаних з вірусними ефектами ліцензії GPL (характерної для MySQL) або корпоративною політикою власників. Це забезпечує ліцензійну чистоту кінцевого продукту, що передається замовнику.

7. Механізми успадкування та секціонування (Partitioning). Важливою архітектурною особливістю PostgreSQL як об'єктно-реляційної системи є

підтримка успадкування таблиць (Table Inheritance). Цей функціонал дозволяє створювати ієрархії, де дочірні таблиці автоматично переймають структуру (схему) та обмеження батьківської таблиці.

2.4 Наслідування в PostgreSQL

Таблиці можуть успадковувати характеристики та набори від інших таблиць (батьківських). При цьому дані, додані в породжену таблицю, автоматично будуть брати участь (якщо це не зазначено окремо) в запитах до батьківської таблиці [9].

Функціонал успадкування (Table Inheritance) у PostgreSQL дозволяє:

- успадковувати дочірнім таблицям (child tables) структуру (стовпці, обмеження) від батьківської таблиці (parent table);
- автоматично включати дані з дочірніх таблиць при виконанні запитів до батьківської таблиці (якщо не використовується ключове слово ONLY).

Успадкування таблиць у PostgreSQL має важливі архітектурні обмеження, які відрізняють його від класичного ООП-успадкування:

- відсутність індексування через успадкування: індекси, визначені на батьківській таблиці, не успадковуються автоматично дочірніми таблицями. Індекси на кожній дочірній таблиці повинні створюватися окремо;
- обмеження унікальності (Unique Constraints): унікальні обмеження (UNIQUE) та обмеження первинного ключа (PRIMARY KEY), визначені на батьківській таблиці, не застосовуються до даних у всіх дочірніх таблицях разом. Це означає, що значення, яке є унікальним у батьківській таблиці, може дублюватися в різних дочірніх таблицях;
- обмеження FOREIGN KEY: Обмеження зовнішнього ключа (FOREIGN KEY) можуть посилатися на батьківську таблицю, але лише

батьківська таблиця може бути посиланням для інших таблиць (дочірні таблиці не можуть бути посиланнями для зовнішніх ключів).

Незважаючи на ці обмеження, функціонал успадкування залишається достатнім для практичного використання у специфічних випадках.

Розділення даних (Partitioning): успадкування часто використовується як основа для ручного або автоматизованого (починаючи з PostgreSQL 10) секціонування таблиць (Declarative Partitioning), де дочірні таблиці зберігають дані за певними критеріями (наприклад, датою).

Логічне об'єднання: для забезпечення логічного об'єднання даних, які мають схожу структуру, але розділені за семантикою.

Висновки по другому розділу

В результаті аналізу функціональних вимог та архітектурних потреб системи моніторингу й оптимізації робочих завдань було здійснено обґрунтований вибір програмних засобів реалізації, які формують технологічний фундамент проєкту.

1. Обґрунтування вибору мови програмування

Вибір PHP (Hypertext Preprocessor) як мови реалізації серверної логіки є оптимальним через її високу мультиплатформенність, зрілу екосистему фреймворків (зокрема, Yii2) та конкурентну продуктивність у сучасних версіях (PHP 7.x, 8.x). PHP забезпечує необхідну швидкість розробки та масштабованість, що є критичним для веб-орієнтованих корпоративних систем.

2. Обґрунтування вибору СУБД

Система управління базами даних PostgreSQL була обрана як технологічна основа проєкту завдяки своїм архітектурним перевагам як об'єктно-реляційна СУБД (ORDBMS). Її надійна підтримка транзакційної цілісності (ACID) та використання механізму багатоверсійності (MVCC) є

фундаментальною вимогою для забезпечення коректності фінансового та облікового контуру (Time Tracking, розрахунок заробітної плати) без блокування роботи користувачів.

Додатковим аргументом на користь PostgreSQL стала підтримка гібридної моделі даних (JSONB) та механізмів секціонування, що забезпечує системі необхідну гнучкість для роботи з динамічними атрибутами завдань та потенціал для масштабування при зростанні обсягів історичних даних.

3. Обґрунтування вибору формату обміну даними

В результаті аналізу було визначено, що формат JSON є оптимальним вибором для реалізації інформаційного обміну в системі. Його компактність, висока швидкість парсингу та нативна підтримка веб-технологіями дозволяють досягти необхідних показників продуктивності та забезпечити зручність подальшого супроводу програмного коду.

Обраний технологічний стек, що об'єднує PHP, PostgreSQL та формат JSON, є архітектурно узгодженим та технічно збалансованим рішенням. Така конфігурація повною мірою задовольняє висунуті вимоги щодо транзакційної надійності, гнучкості обробки даних та масштабованості, створюючи міцний фундамент для успішної реалізації та довготривалої експлуатації системи..

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ ТА ПЕРСОНАЛОМ ДЛЯ КОМАНДИ РОЗРОБНИКІВ

3.1 Системний аналіз операційних процесів управління проєктами та персоналом

Для забезпечення архітектурної валідності розроблюваної інформаційної системи управління проєктами та персоналом проведемо системний аналіз ключових компонентів предметної області: робочих процесів, інформаційних потоків, документообігу та ієрархії задач. Аналіз сфокусовано на специфіці команд розробників, що працюють в умовах офшорного аутсорсингу, де критично важливими є прозорість, точність обліку та мінімізація часових затримок.

Аналіз робочих процесів (Business Process Analysis) [10].

Робочі процеси поділяються на чотири основні фази, що відображають життєвий цикл проєкту та супутні адміністративні функції.

Управління конфігурацією проєкту (Project Configuration). Процес ініціюється Адміністратором або Менеджером і включає створення нових проєктів, визначення бюджетних лімітів, встановлення фінансових ставок та формування командного складу.

Виконання та облік завдань (Execution and Time Logging): Основний робочий процес, де Виконавець здійснює декомпозицію задачі, фіксує витрачений робочий час (*Time Entry*) та оновлює статус задачі. Цей процес є критичним джерелом первинних метрик.

Моніторинг та аналіз (Monitoring and Control). Процес, що виконується Системою та Менеджером, включає збір агрегованих метрик, розрахунок ключових показників діяльності (КПД), зокрема $Cycle\ Time$ та ідентифікацію "вузьких місць" (*bottlenecks*) у робочому потоці.

Фінансовий та звітний облік (Fiscal Accounting). Процес, відповідальний за генерацію білінгових звітів для клієнтів (на основі затвердженого Time Tracking) та автоматичний розрахунок нарахування заробітної плати персоналу.

Аналіз ієрархії задач (Work Breakdown Structure, WBS)

Для забезпечення гранульованого контролю та точного обліку WBS має бути структурована на чотири рівні.

Рівень 0 (Агрегаційний): Обліковий запис Клієнта/Контракт.

Рівень 1 (Управлінський): Проєкт — основна одиниця бюджетування.

Рівень 2 (Функціональний): Фаза/Епік — великі функціональні або архітектурні блоки.

Рівень 3 (Операційний): Задача (*Task*) — мінімальна одиниця роботи, що має статус та відповідального.

Рівень 4 (Атомарний): Облік Часу (*Time Entry*) — атомарна транзакція, прив'язана до конкретного виконавця, задачі та тривалості.

Аналіз інформаційних потоків визначає джерела та приймачі даних, які інформаційна система управління проєктами та персоналом має обробляти представлено в таблиці 3.1.

Інформаційна система управління проєктами та персоналом повинна забезпечувати надійне керування структурованим документообігом (фінансові звіти) з гарантіями транзакційної цілісності та ефективну підтримку неструктурованого контенту (База Знань) з механізмами версіонування.

Проведений системний аналіз підтверджує, що інформаційна система управління проєктами та персоналом повинна функціонувати як інтегрована платформа, яка забезпечує двоконтурну взаємодію (Адміністративний та Користувацький контури) і здатна обробляти як високоточні фінансові транзакції, так і гнучкий інформаційний контент.

Таблиця 3.1 – Аналіз інформаційних потоків

Потік	**Джерело**	**Призначення**	**Характер**	**Функціональна Роль**
Обліковий Вхід	Виконавець	СУБД (PostgreSQL)	Структурований (JSON Object)	Формування первинних транзакцій (Time Entry).
Керуючий Вхід	Менеджер/ Клієнт	СУБД (PostgreSQL)	Структурований (JSON)	Створення та оновлення сутностей (Проєкти, Задачі, Користувачі).
Білінговий Вихід	Система	Клієнт	Структурований (JSON/PDF)	Забезпечення фінансової прозорості та експорт рахунків-фактур.
Аналітичний Вихід	Система	Менеджер (Dashboard)	Агрегований (JSON для візуалізації)	Передача розрахованих метрик (KPI) для побудови графіків та підтримки прийняття рішень.
Документаційний	Користувач	База Знань	Напівструктурований (Markdown/Text)	Зберігання, індексація та версіонування технічної документації.

3.2 Обґрунтування архітектури інформаційної системи управління проєктами та персоналом

Архітектура розробленої системи базується на інтеграції сучасних підходів до обробки даних, притаманних корпоративним інформаційним системам (Enterprise Information Systems). Ключові функціональні домени системи зводяться до маніпуляції як структурованими даними (завдання, звіти часу), так і неструктурованим контентом (документація, база знань).

Для реалізації цих вимог було обрано архітектурний патерн MVC (Model-View-Controller), реалізований у фреймворку Yii2, що дозволяє ефективно вирішувати наступні завдання:

1. Централізація та управління структурованим контентом Ядро системи — це набір сутностей, що потребують суворого контролю цілісності:

Управління завданнями (Task Management): Кожне робоче завдання, його опис, статус, пріоритет та коментарі розглядаються як об'єкти з чіткою структурою. Використання ORM (Object-Relational Mapping) у Yii2 дозволяє зручно маніпулювати цими даними.

Управління проєктами та обліковими записами: Проєкти, клієнти та профілі розробників є об'єктами даних, якими необхідно централізовано керувати через адміністративну панель, що забезпечує єдину точку входу для менеджменту.

2. Управління неструктурованим контентом та знаннями Ефективна команда розробників потребує швидкого доступу до інформації:

База знань (Knowledge Base): Система включає модуль для зберігання технічної документації та інструкцій. Це зменшує час на онбординг нових співробітників та пошук рішень.

Контроль версій та історія змін: Архітектура системи передбачає логування дій користувачів (Audit Trail), що є критично важливим для відстеження змін у статусах завдань та списанні часу.

3. Розмежування відповідальності та інтерфейс Використання патерну MVC забезпечує ключовий принцип відокремлення бізнес-логіки від її представлення:

Модульність та шаблонізація: Використання механізму View (представлень) та віджетів у Yii2 дозволило створити адаптивний інтерфейс, зручний як для десктопного використання, так і для мобільних пристроїв.

Авторизація та Ролі (RBAC): Система використовує вбудований у фреймворк компонент RBAC (Role-Based Access Control). Це забезпечує надійне розмежування доступу між адміністративним контуром (менеджери, адміністратори) та користувацьким контуром (розробники), захищаючи критичні дані від несанкціонованих змін.

Обраний підхід дозволяє уніфікувати управління даними, забезпечити масштабованість системи та високий рівень користувацької ергономіки. На відміну від коробочних CMS-рішень, кастомна розробка на базі фреймворку забезпечує вищу швидкодію, гнучкість у налаштуванні специфічних бізнес-процесів офшорного аутсорсингу та відсутність надлишкового коду.

3.3 Розроблення архітектури програмного застосунку

Перш ніж переходити до опису конкретних елементів системи, варто коротко пояснити принцип шарової архітектури, яка дуже вдало накладається на структуру, що використовується у фреймворку Yii [11]. Немає великого сенсу прив’язуватися до певної версії Yii — важливо зрозуміти саму ідею.

Шарова архітектура передбачає, що вся система складається з кількох послідовних рівнів (шарів), кожен з яких виконує свою окрему функцію. Характерна особливість такого підходу — те, що шар не знає, як влаштовані шари, що знаходяться над ним, і не працює з їх внутрішніми деталями. Взаємодія відбувається тільки з нижчим шаром і лише через узгоджені

інтерфейси. Завдяки цьому кожен рівень може реалізовувати власну модель даних і бути відокремленим від інших.

Комунікація між шарами зводиться до передачі параметрів і отримання відповідей. Використання глобальних даних між шарами — неприпустимо, бо це руйнує логіку розділення відповідальності.

Якщо не брати до уваги мережеві та серверні аспекти, які вже самі по собі є «шаром», то сам застосунок можна умовно поділити на три основні рівні (рис. 3.1):

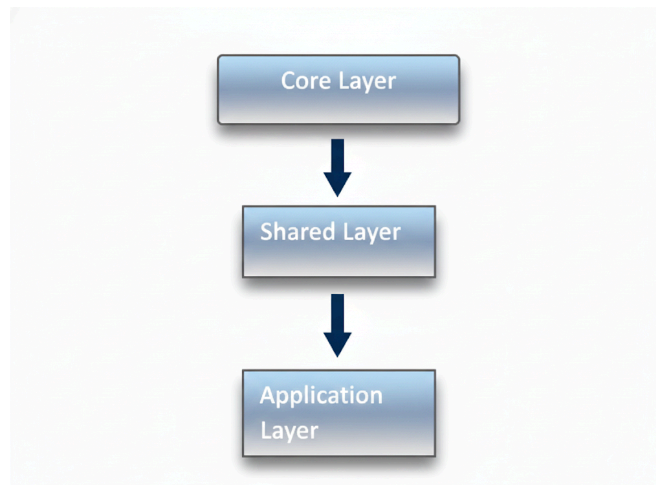


Рисунок 3.1– Шари технічної бази

Core layer — шар ядра. Тут розміщені низько рівневі бібліотеки і фреймворки. Наприклад, Yii.

Shared layer — шар модулів які реалізують певний функціонал системи і можуть бути використані в майбутньому в інших проєктах.

Application layer — шар застосунку з їх специфічними модулями.

Важливим моментом є те, що модулі повинні мати можливість легко переїхати із рівня application layer в shared layer. Пояснюється це досить просто: треба думати про майбутнє а може потім зявиться подібний продукт і йому знадобляться подібні модулі тоді не треба буде переписувати код двічі. І потім, той факт що модуль можна легко перемістити до shared layer свідчить що у його нема залежностей(dependency), а це завжди добре. Розглянемо файлову організацію в Application layer з врахуванням специфікації yii (рис. 3.2).

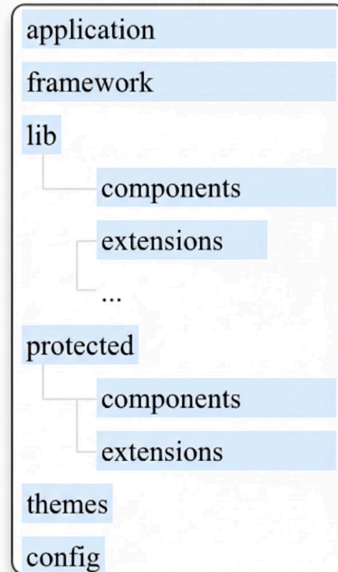


Рисунок 3.2 – Файлова організація Yii застосунку

Типовий Yii застосунок має вигляд, що представлений на рисунку 7. Файл index.php є вхідною точкою, на неї направляється домен, і там створюється екземпляр WebApplication (є ще ConsoleApplication). В тому ж файлі підключаються конфігураційні файли, які в свою чергу підтягують компоненти із sharedlayer. Від того файла старує і роутинг. Всі дії можуть стартувати 35 тільки від контролерів, а якщо вдаватись у деталі то від конкретних action того контролера. Контролери оперують вхідними даними, працюють з віджетами, моделями даних і готую для користувачів html вивід або іншого роду відповіді json, xml чи, те що треба.

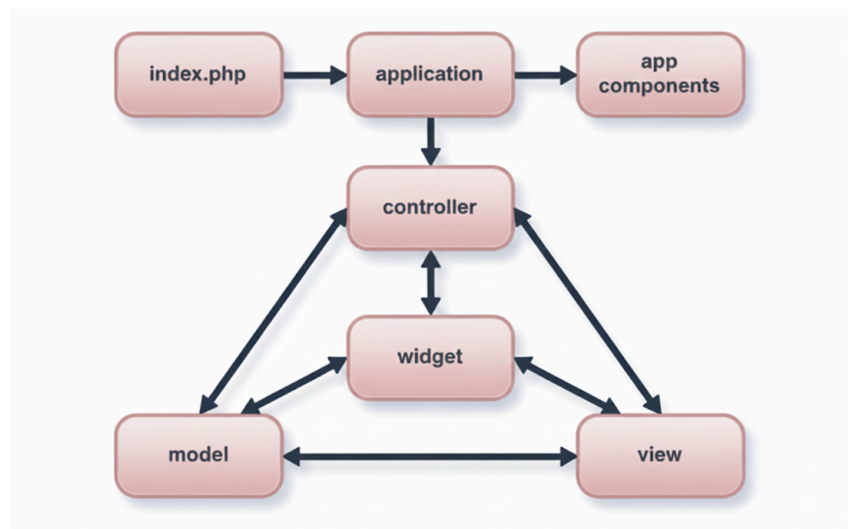


Рисунок 3.3 – Будова типового Yii застосунку

Моделі в Yii повністю абстрагують джерело даних. Розробник може виконувати CRUD-операції чи валідацію незалежно від того, яка СУБД використовується.

Віджет — це невеликий автономний блок із власною логікою та шаблоном. Найпростіший приклад — слайдер фотографій чи форма авторизації.

Обробка вхідного HTTP-запиту у веб-додатку реалізується згідно з принципами централізованої диспетчеризації та архітектурного шаблону MVC (рис. 3.4).

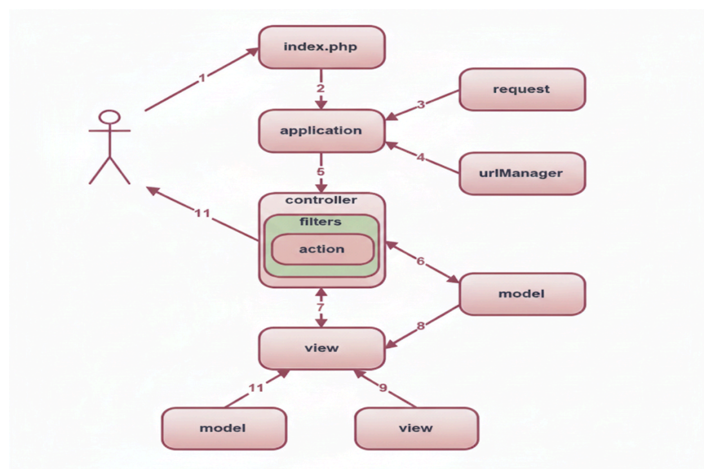


Рисунок 3.4 – Сценарій використання застосунку

Цей процес складається з низки послідовних етапів:

1. Ініціалізація Запиту (Request Initiation): Користувацький клієнт (браузер) генерує HTTP-запит до визначеного URL (наприклад, http://www.example.com/index.php?r=post/show&id=1).

2. Перехоплення Веб-Сервером: Веб-сервер (наприклад, Apache або Nginx) спрямовує запит до вхідної точки застосунку — файлу index.php (Bootstrap File).

3. Створення Екземпляра Застосунку: У файлі index.php відбувається ініціалізація та конфігурація основного екземпляра застосунку (Application Instance).

4. Збір та Передача Даних: Застосунок агрегує всі параметри запиту (URI, метод, заголовки) та інкапсулює їх у спеціалізований системний компонент Request.

5. Маршрутизація (Routing): Компонент urlManager (Маршрутизатор) аналізує дані запиту та визначає відповідний контролер (PostController) та дію (*Action*, наприклад, actionShow), відповідальну за обробку даного URL.

6. Виконання Контролера: Система створює екземпляр визначеного контролера та викликає його цільовий метод (*Action*).

7. Отримання Моделі (Model Retrieval): Метод *Action* взаємодіє з Моделлю для вибірки необхідних даних з бази даних (наприклад, об'єкт Post з ідентифікатором ID = 1).

8. Передача Даних до Подання (View Rendering): Отримані Моделлю дані передаються до компонента Подання (*View*) для формування вихідного інтерфейсу.

9. Генерація Додаткових Компонентів: Подання може ініціювати виконання додаткових модулів або віджетів (Widgets) для вбудовування допоміжних функціональних елементів.

10. Фінальна Компоновка: Сформований HTML-код інтегрується в основний шаблон сторінки (*Layout*), який містить загальну структуру, навігацію та стилі.

11. Формування Відповіді: Сформована та скомпільована HTTP-відповідь повертається через Веб-сервер клієнту.

Цей послідовний процес забезпечує чітке дотримання архітектурних принципів та гарантує, що логіка обробки запитів є стабільною, передбачуваною та легко підтримуваною.

Такий підхід показує, наскільки гнучким є Yii і скільки завдань можна вирішити в межах одного фреймворка.

3.4 Розробка БД PostgreSQL

У архітектурі розроблюваного інтернет-додатку Системи моніторингу й оптимізації ключова роль відводиться системі управління базами даних (СУБД). Для забезпечення цілісності та надійності даних було обрано PostgreSQL.

Вибір PostgreSQL як основної СУБД є обґрунтованим через її високу здатність до обробки транзакцій (Transactional Processing). Ця функціональна властивість є необхідною передумовою для коректного ведення фінансового та облікового контуру системи, який включає моніторинг загального бюджету проєктів та індивідуальних балансів робочого часу співробітників.

Транзакція в контексті СУБД являє собою логічно неподільну послідовність операцій над даними, яка виконується як єдине ціле. Ключові гарантії, що надаються транзакційним механізмом, описуються акронімом ACID (Atomicity, Consistency, Isolation, Durability):

Атомарність (Atomicity): гарантує, що транзакція або виконується повністю (commit), або не виконується взагалі (abort/rollback). У разі виникнення помилки чи збою, СУБД автоматично скасовує всі проміжні зміни, повертаючи базу даних до попереднього консистентного стану.

Узгодженість (Consistency): Забезпечує перехід бази даних з одного коректного стану в інший, дотримуючись усіх визначених обмежень (constraints).

Ізольованість (Isolation): гарантує, що паралельні транзакції не впливають на результати одна одної.

Надійність (Durability): гарантує, що після успішного завершення (фіксації) транзакції, її зміни залишаються постійними, незалежно від подальших збоїв системи.

У сучасних СУБД застосовуються різні архітектурні підходи до реалізації транзакцій. Розглянемо можливі варіанти.

Послідовні (Serial): стандартні транзакції, які виконуються одна за одною, забезпечуючи максимальну ізоляцію.

Паралельні (Concurrent): транзакції, що виконуються одночасно. PostgreSQL використовує механізми MVCC (Multi-Version Concurrency Control)

для управління паралельним доступом без блокування читання, що значно підвищує продуктивність у багатокористувацьких системах.

Розподілені (Distributed): транзакції, що оперують даними у кількох незалежних СУБД. Вони вимагають складних протоколів синхронізації, зокрема двофазного підтвердження (Two-Phase Commit, 2PC), для забезпечення атомарності по всьому кластеру.

Автономні Підтранзакції (Autonomous Subtransactions): деякі системи підтримують вкладені транзакції, які функціонують незалежно в межах "батьківської" транзакції, дозволяючи фіксувати чи скасовувати окремі частини операції.

Вибір PostgreSQL, який відомий своєю надійною реалізацією ACID та використанням MVCC, є оптимальним для забезпечення високої цілісності даних у фінансовому модулі розроблюваної системи.Транзакційні системи зазвичай орієнтуються на принципи ACID (Atomicity, Consistency, Isolation, Durability) — набір вимог, сформований ще наприкінці 1970-х Джимом Греєм. Ці властивості гарантують стабільну та передбачувану роботу з даними.

Атомарність означає, що жодна транзакція не може бути збережена частково. Або всі дії всередині транзакції виконуються успішно, або не виконується жодна. Для цього і використовується механізм відкату, який відновлює БД до попереднього стану, якщо щось пішло не так.

Узгодженість — одна з найскладніших характеристик ACID. Система має залишатися коректною як до початку, так і після завершення транзакції. Узгодженість часто плутають з цілісністю даних, але це не одне й те саме. Цілісність — це про технічні обмеження (типи, зв’язки між таблицями), а узгодженість — про логіку роботи предметної області. Наприклад, у фінансових операціях сума списання має дорівнювати сумі зарахування. Це бізнес-правило, яке не можна забезпечити одними тільки обмеженнями БД — його реалізує програмний код.

При цьому в процесі виконання транзакції система може тимчасово бути в неузгодженому стані. Це нормально, доки інші транзакції не можуть бачити цю «проміжну» невідповідність — це вже забезпечує ізоляція.

Ізольованість означає, що паралельні транзакції не повинні заважати одна одній та впливати на результати. Залежно від обраного рівня ізоляції можуть бути дозволені чи заборонені певні типи конкурентних аномалій (наприклад, Phantom Read або Non-Repeatable Read). На рівні Repeatable Read можливість впливу паралельних транзакцій уже частково допускається.

Надійність (Durability) гарантує, що якщо транзакцію завершено й система підтвердила її успішність, то зміни не зникнуть навіть у випадку збою обладнання чи відключення електроенергії.

З урахуванням цих вимог була сформована структура таблиць, яка забезпечує стабільну роботу додатку. Схеми таблиць наведено на рис. 3.5.

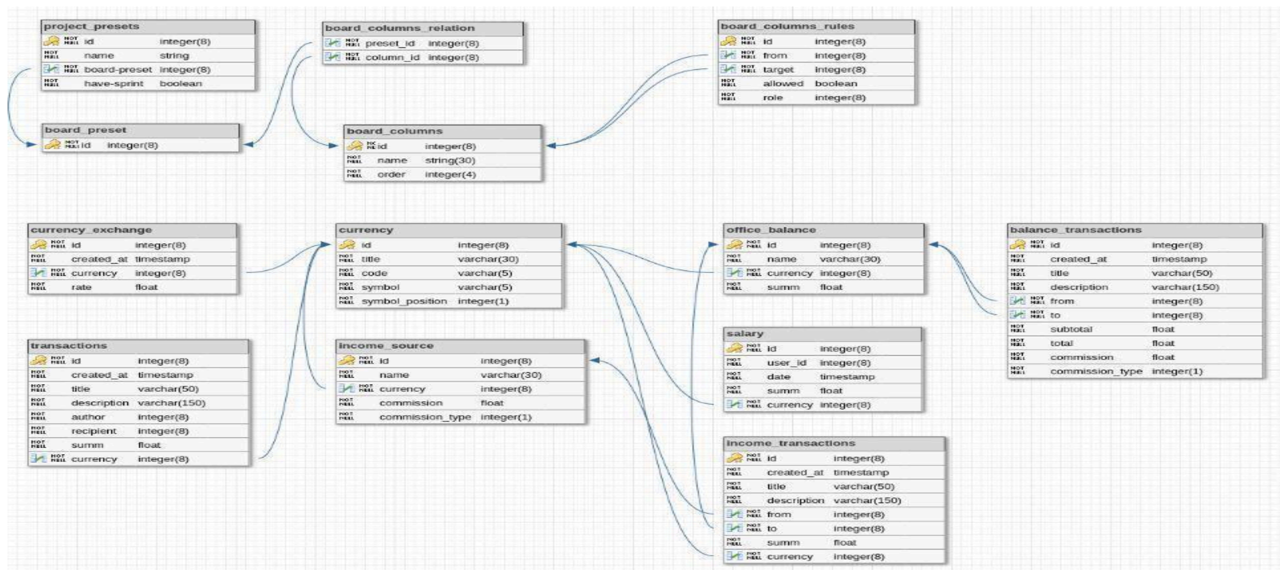


Рисунок 3.5 – Схема таблиць

3.5 Програмна реалізація веб застосунку

У контексті системного проєктування, архітектура розроблюваного веб-додатку доцільно структурована на дві функціонально незалежні складові (контури). Такий підхід базується на фундаментальному принципі розмежування відповідальності (Separation of Concerns), що є наріжним каменем сучасної інженерії програмного забезпечення. Цей принцип, вперше сформульований Е. В. Дейкстрою у 1974 році, вимагає розділення системи на окремі, непересічні модулі, кожен з яких відповідає за певну сферу функціональності [12]. В обраній архітектурі це забезпечує кілька ключових переваг.

Полегшення масштабування: незалежні контури можуть бути масштабовані асиметрично, відповідно до реального навантаження, оптимізуючи використання обчислювальних ресурсів.

Підвищення супроводу та модифікації (Maintainability): зміни, що вносяться в логіку одного контуру (наприклад, оновлення інтерфейсу користувача), мінімально впливають на стабільність іншого (наприклад, налаштування адміністративних процесів).

Чітке розмежування доступу та безпеки: завдяки розділенню можна застосовувати різні політики автентифікації та авторизації, суворо контролюючи доступ до даних і функціоналу адміністративного рівня.

Таким чином, структурна побудова системи на базі незалежних функціональних контурів оптимізує технічне обслуговування та підвищує адаптивність системи до майбутніх функціональних розширень [13].

1. Адміністративний Контур (Адміністративна Панель)

Цей контур являє собою закриту частину системи, призначену для управління та конфігурації усіх ресурсів. Він виконує роль централізованого хабу, де реалізується: контроль над даними (CRUD-операції); налаштування системних параметрів і робочих процесів (Workflow); моніторинг ключових показників ефективності (KPI) та генерація аналітичної звітності; керування

правами доступу користувачів; адміністративний контур призначений для керівного складу, менеджерів проєктів та системних адміністраторів, які відповідають за функціонування та оптимізацію робочих процесів.

2. Користувацький Контур (Клієнтський Інтерфейс)

Цей контур є публічною або частково закритою частиною, з якою безпосередньо взаємодіють кінцеві користувачі (розробники, тестувальники) під час виконання своїх робочих завдань. Його основне призначення: взаємодія із завданнями (перегляд, зміна статусу); фіксація часу (Time Tracking); перегляд індивідуального прогресу.

Такий архітектурний поділ, що базується на моделі Back-End/Front-End та принципі розподілу прав доступу, дозволяє чітко розмежувати функціональність, підвищити безпеку даних та значно спростити розвиток та підтримку окремих модулів системи.

3.6 Адміністративний контур

Адміністративний контур (Admin Panel) функціонує як авторизований центр управління життєвим циклом даних та системними операціями. Його архітектурна роль полягає у забезпеченні централізованого контролю та конфігурації усіх критичних функціональних доменів системи.

Основними доменами управління є такі.

Управління ресурсами та обліковими записами (Resource and Account Management).

Реалізація повного спектру операцій CRUD (Create, Read, Update, Delete) над сутностями клієнтів та користувачів-співробітників.

Конфігурація та підтримка реєстру проєктів, включаючи встановлення параметрів, бюджетних обмежень та термінів.

Диспетчеризація робочих процесів (Workflow Dispatching):

Контроль над виконанням завдань, моніторинг їхнього статусу, перепризначення відповідальних осіб.

Цей домен є необхідним для превентивного виявлення "вузьких місць" (bottlenecks) та оперативного втручання в робочий потік.

Фінансовий та обліковий моніторинг (Fiscal and Accounting Oversight):

Модуль призначений для контролю білінгових даних, ведення обліку загального фінансового балансу проєктів та виконання операцій з розрахунку заробітної плати персоналу на основі інтегрованих показників обліку робочого часу (Time Tracking).

Механізми забезпечення безпеки та доступу [14].

Доступ до адміністративного контуру підлягає вимогам інформаційної безпеки та політиці авторизації [15].

Принцип мінімізації привілеїв: доступ надається виключно автентифікованим користувачам, які мають відповідні адміністративні привілеї та ролі, визначені в системі управління доступом.

Логічна ізоляція: контур є логічно ізольованим від клієнтської частини системи, що реалізується через спеціалізований маршрутний префікс (наприклад, /admin/).

Цілісність даних: таке обмеження доступу є фундаментальним для забезпечення конфіденційності, цілісності фінансових даних та запобігання несанкціонованим операціям у критично важливих системних параметрах.

У адміністративному інтерфейсі реалізовано такі сторінки:

1. головна сторінка (рис 3.6), з якої починається робота адміністратора;

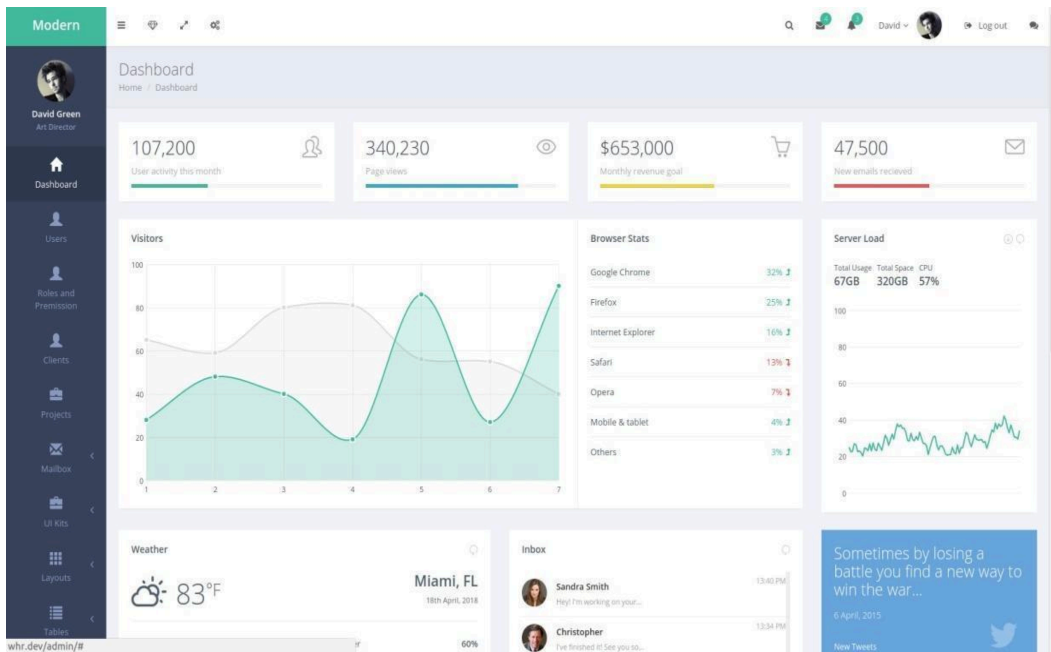


Рисунок 3.6 – Головна сторінка

2. сторінка зі списком прав для різних ролей (рис. 3.7), де можна переглянути, хто й що має право робити в системі;

Рисунок 3.7 – Ролі та права

3. Список користувачів (рис. 3.8) — таблиця з усіма зареєстрованими співробітниками;

Рисунок 3.8 – Список користувачів

4. Сторінка клієнтів (рис. 3.9);

Рисунок 3.9 – Список клієнтів

5. сторінка проєктів (рис. 3.10);

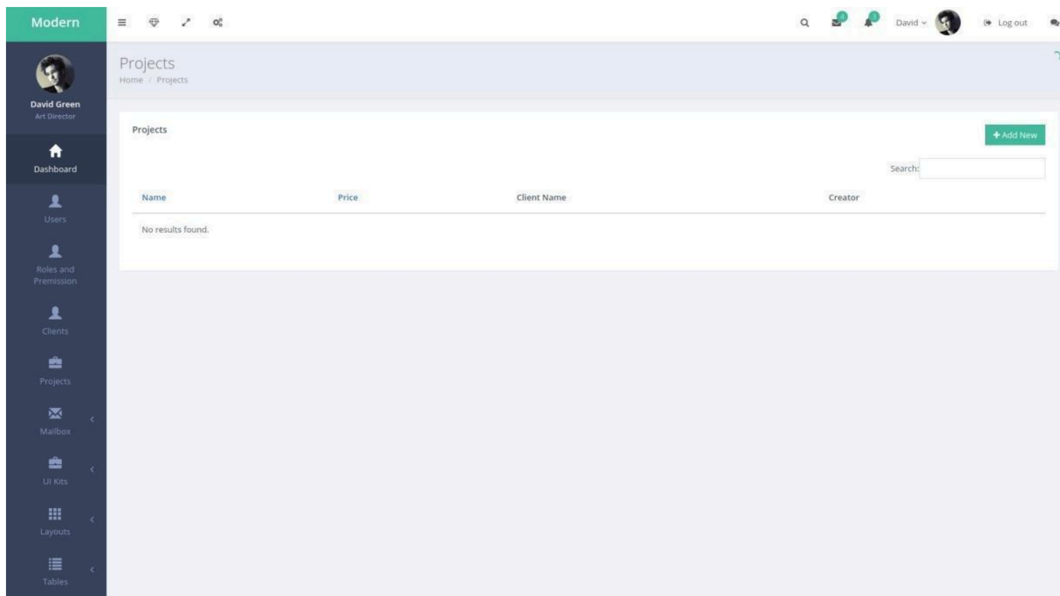


Рисунок 3.10 – Список проєктів

6. сторінка з створенням звіту по кожному завданню;
7. сторінка керування фінансами;
8. Календар (рис. 3.11) — зручний інструмент для перегляду дедлайнів та важливих подій;

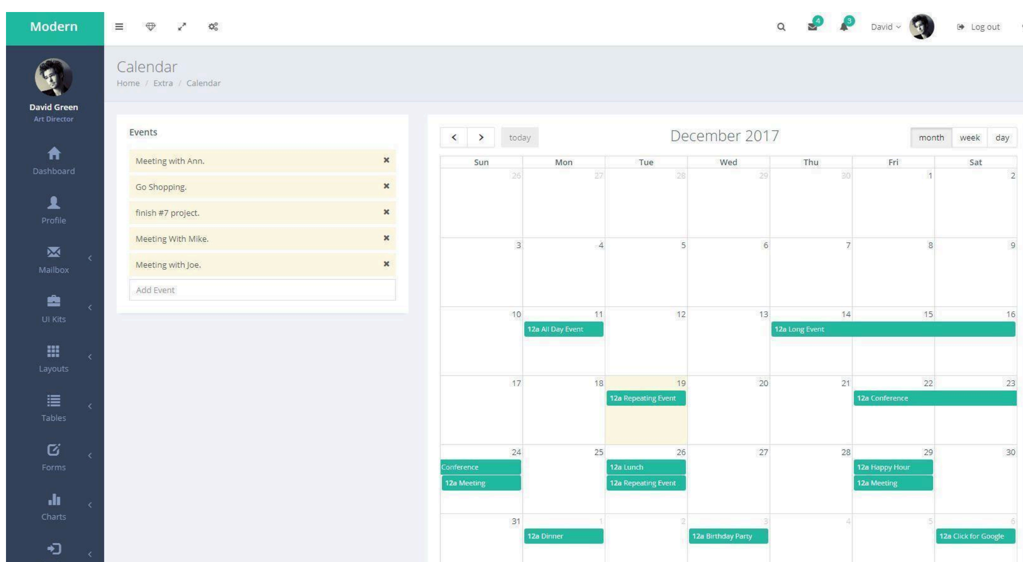


Рисунок 3.11 – Список проєктів

3.7 Користувацький контур

Користувацький контур (Клієнтський інтерфейс) є фронтальним модулем системи, спроєктованим для безпосередньої взаємодії з виконавцями робочих завдань (розробниками, тестувальниками, іншими фахівцями). Його ключова мета — забезпечити швидкий і прозорий доступ співробітників до їхньої індивідуальної робочої статистики та операційних даних.

Розглянемо основні функціональні можливості користувацького контуру.

Можливість моніторингу індивідуальної продуктивності.

Статистичний огляд: користувач може оперативно переглядати деталі виконаної роботи в розрізі конкретних проєктів.

Історія діяльності: доступ до повної хронології робочої активності та завершених завдань.

Облік робочого часу та фінансовий моніторинг: облік відпрацьованих годин (Time Tracking), відображення накопиченої кількості відпрацьованих годин, що є основою для білінгу.

Моніторинг балансу: надання актуальної інформації щодо поточного розрахункового балансу заробітної плати, ґрунтуючись на зареєстрованому робочому часі та фінансових ставках.

Взаємодія із завданнями: надання інтерфейсу для перегляду призначених завдань, зміни їхніх статусів та фіксації робочого часу.

На відміну від адміністративного контуру, доступ до користувацької частини здійснюється через основну адресу веб-додатку.

Автентифікація: вхід до контуру вимагає успішної автентифікації (введення логіна та пароля) користувача.

Авторизація: доступ користувача обмежується виключно його власними даними та завданнями, забезпечуючи дотримання принципів конфіденційності та мінімізації привілеїв.

Така архітектурна організація максимізує прозорість для виконавців щодо їхніх результатів та фінансових показників, що є важливим елементом для підтримки мотивації та відповідальності в умовах віддаленої роботи.

У цьому інтерфейсі реалізовані такі сторінки:

1. головна сторінка (рис. 3.12), де відображається коротка інформація по активних проєктах;

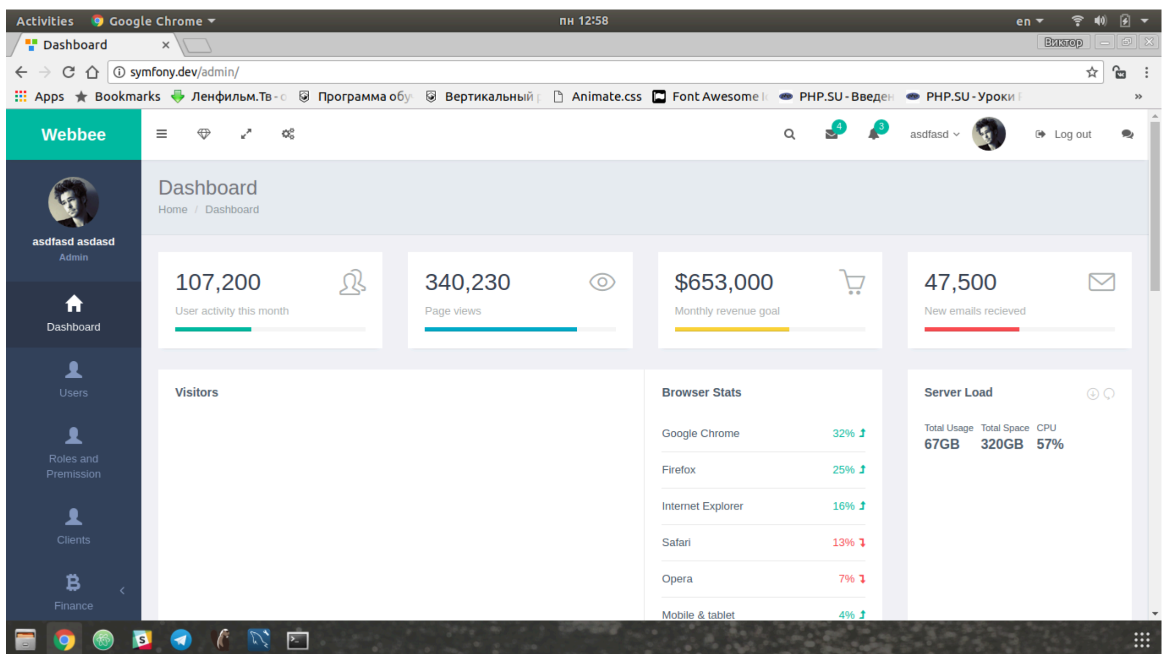


Рисунок 3.12 – Список проєктів

2. сторінка To-Do List;

3. сторінка зі звітом по напрацьованим годинам, нарахованій з.п., та списком проєктів в яких робітник приймав участь;

4. сторінка з календарем для візуалізації термінів здачі проєктів та подій, які можна додати в адмін. панелі (рис. 3.13);

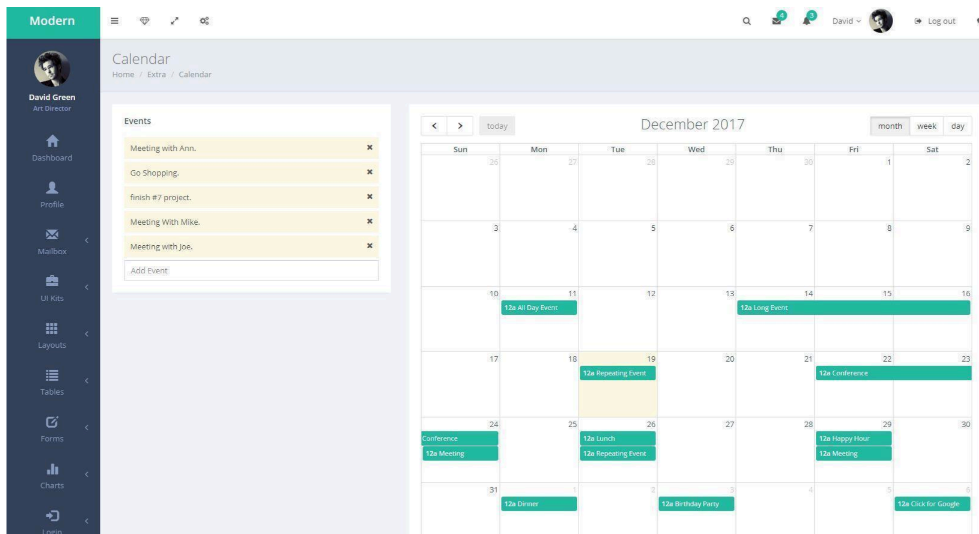


Рисунок 3.13 – Список проєктів

3.8 Тестування застосунку

Завершальний етап розробки Системи моніторингу й оптимізації робочих завдань включав проведення комплексного тестування, метою якого було підтвердження її відповідності технічному завданню (ТЗ) та забезпечення функціональної надійності.

В ході виконаного тестування було підтверджено, що всі ключові функціональні вимоги, визначені на етапі проєктування, реалізовані коректно. Зокрема, успішно пройшли перевірку:

Транзакційна цілісність: тести підтвердили коректність роботи модуля обліку робочого часу (Time Tracking) та фінансових операцій, забезпечуючи дотримання принципів ACID (як того вимагав вибір СУБД PostgreSQL).

Логічна ізоляція контурів: підтверджено надійне розмежування доступу між Адміністративним та Користувацьким контурами системи.

Аналітична функціональність: верифікована коректність алгоритмів обробки даних, що використовуються для розрахунку ключових показників ефективності (KPI) та формування аналітичної звітності, необхідної для оптимізації офшорних процесів.

Були проведені тести працездатності, які охоплювали функціональне, інтеграційне та пользователське тестування:

Функціональна Працездатність: система демонструє стабільне та коректне виконання всіх заявлених операцій, включаючи створення, редагування та моніторинг робочих завдань.

Мультиплатформенність: підтверджено стабільну роботу застосунку на різних операційних системах (Windows, Mac OS X та Linux), що забезпечує гнучкість його впровадження в IT-інфраструктуру замовників.

Користувацька зручність (Usability): інтерфейс відповідає вимогам ергономіки, що мінімізує помилки при введенні даних та спрощує взаємодію користувачів з функціоналом.

Результати комплексного тестування дозволяють зробити висновок, що розроблений програмний продукт повністю відповідає вимогам технічного завдання та є надійною, функціонально завершеною системою.

Висновки по третьому розділу

У даному розділі було виконано ключові етапи проєктування та безпосередньої програмної реалізації системи моніторингу й оптимізації робочих завдань.

Реалізація системи ґрунтувалася на ретельно обґрунтованому технологічному стеку. Вибір фреймворку Yii2 для серверної логіки забезпечив використання архітектурного шаблону MVC (Model-View-Controller), що гарантує модульність, швидкість розробки та високий рівень безпеки.

Структурний поділ системи на Адміністративний та Користувацький контури забезпечив чітке розмежування відповідальності (Separation of Concerns) та дозволив застосувати диференційовані політики безпеки та доступу.

Розроблена система демонструє високий рівень користувацької ергономіки та має інтуїтивно зрозумілий інтерфейс. Це мінімізує час, необхідний для адаптації персоналу, та сприяє коректному використанню ключового функціоналу: Time Tracking та Monitoring.

В рамках завершального етапу роботи було проведено комплексне тестування програмного забезпечення, що охоплювало функціональні та інтеграційні аспекти. Результати тестування підтвердили повну відповідність розробленої системи вимогам, викладеним у технічному завданні, та засвідчили її високу функціональну надійність та працездатність.

Таким чином, цілі проєктування та програмної реалізації були успішно досягнуті, що створює готову до впровадження платформу для підвищення ефективності організації.

ВИСНОВКИ

В результаті виконання дипломної кваліфікаційної роботи магістра, якому передував ґрунтовний системний аналіз необхідних робочих процесів, інформаційних потоків, документообігу та ієрархії задач, було успішно спроєктовано, розроблено програмний застосунок інформаційної системи управління проєктами та персоналом, орієнтований на команду розробників, що функціонують у парадигмі офшорного аутсорсингу.

Мета роботи, що полягала у створенні інструментарію для системного підвищення ефективності та прозорості організації робочого процесу, була досягнута. Розроблений програмний продукт функціонально орієнтований на оптимізацію ключових показників діяльності (КПД) на рівні організації та підвищення індивідуальної продуктивності співробітників за рахунок автоматизованого моніторингу та аналізу робочих процесів.

Реалізація програмного продукту базувалася на використанні сучасного та адаптивного технологічного стеку. Використання архітектурного патерну MVC на базі фреймворку Yii2 для серверної логіки забезпечив швидкість розробки, архітектурну стабільність та високу безпеку системи.

Розроблена система демонструє високий рівень користувацької ергономіки, маючи інтуїтивно зрозумілий та орієнтований на користувача інтерфейс. Це мінімізує час на адаптацію персоналу та сприяє коректному використанню функціоналу Time Tracking та Monitoring.

В рамках завершального етапу роботи було проведено комплексне тестування програмного забезпечення. Результати тестування підтвердили повну відповідність розробленої системи вимогам, викладеним у технічному завданні, та засвідчили її високу працездатність і функціональну надійність.

Створений застосунок є готовим до практичного впровадження в реальні робочі процеси, що забезпечить підвищення прозорості та ефективності управління проєктами, що виконуються за моделлю офшорного аутсорсингу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Jira [Електронний ресурс]. — Режим доступу: https://www.atlassian.com/software/jira (дата звернення: 14.10.2025).
2. Azure DevOps Documentation [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/en-us/azure/devops/?view=azure-devops (дата звернення: 06.11.2025).
3. ClickUp [Електронний ресурс]. — Режим доступу: https://clickup.com/ (дата звернення: 23.09.2025).
4. Trackabi [Електронний ресурс]. — Режим доступу: https://trackabi.com/ (дата звернення: 18.10.2025).
5. NokoTime [Електронний ресурс]. — Режим доступу: https://nokotime.com/ (дата звернення: 02.11.2025).
6. PHP Manual [Електронний ресурс] / The PHP Group. — Режим доступу: https://www.php.net/manual/ (дата звернення: 05.11.2025).
7. Bray T. The JavaScript Object Notation (JSON) Data Interchange Format (RFC 8259) [Електронний ресурс] / T. Bray. — Internet Engineering Task Force (IETF), 2017. — Режим доступу: https://tools.ietf.org/html/rfc8259 (дата звернення: 09.11.2025).
8. Stonebraker M. The design of POSTGRES / M. Stonebraker, L. A. Rowe // Proceedings of the ACM SIGMOD International Conference on Management of Data. — 1986. — P. 340–355.
9. PostgreSQL Documentation. The PostgreSQL Tutorial [Електронний ресурс] / The PostgreSQL Global Development Group. — Режим доступу: https://www.postgresql.org/docs/manual/ (дата звернення: 10.10.2025).
10. Dumas M. Fundamentals of business process management / M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers. — 2nd ed. — Berlin ; Heidelberg : Springer, 2018. — DOI: 10.1007/978-3-662-56509-4.
11. Yii Framework [Електронний ресурс]. — Режим доступу: https://www.yiiframework.com/ (дата звернення: 16.09.2025).

12. Dijkstra E. W. On the role of scientific thought / E. W. Dijkstra // Communications of the ACM. — 1974. — Vol. 17, no. 12. — P. 690–693.

13. Литвин В. В. Системний аналіз та проєктування інформаційних систем / В. В. Литвин, В. С. Бакаєв. — Львів : Новий Світ–2000, 2016.

14. ISO/IEC 27001:2022. Information security, cybersecurity and privacy protection. Information security management systems. Requirements. — Geneva : ISO/IEC, 2022. — 32 p.

15. Saltzer J. H., The protection of information in computer systems / J. H. Saltzer, M. D. Schroeder // Proceedings of the IEEE. — 1975. — Vol. 63, no. 9. — P. 1278–1308.

ДОДАТОК А

ТЕКСТ ПРОГРАМИ

```
<?php
namespace common\models;
use Yii;
use yii\base\NotSupportedException;
use yii\behaviors\TimestampBehavior;
use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

use common\models\UserBalance;
use common\models\UserMeta;
/**
 * User model
 *
 * @property integer $id
 * @property string $username
 * @property string $first_name
 * @property string $last_name
 * @property string $username
 * @property string $password_hash
 * @property string $password_reset_token
 * @property string $email
 * @property string $auth_key
 * @property string $company
 * @property integer $status
 * @property integer $last_login
 * @property integer $created_at
 * @property integer $updated_at
 * @property string $password write-only password
 */
```

```
class User extends ActiveRecord implements IdentityInterface
{
    const STATUS_DELETED = 0;
    const STATUS_ACTIVE = 10;

    public $meta = [];

    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return '{{%user}}';
    }

    /**
     * @inheritdoc
     */
    public function rules()
    {
        return [
            [['first_name', 'last_name', 'email'], 'required'],
            [['first_name', 'last_name', 'email'], 'trim'],
            ['email', 'unique', 'targetAttribute' => 'email', 'targetClass' =>
'\common\models\User', 'when' => function($model){
                return $this->isAttributeChanged('email');
            }],
            ['status', 'default', 'value' => self::STATUS_ACTIVE],
            ['status', 'in', 'range' => [self::STATUS_ACTIVE,
self::STATUS_DELETED]],
            [['salary_type'], 'in', 'range' => ['fixed', 'hourly']]
```

```
    ];
}

/**
 * @inheritdoc
 */
public function behaviors()
{
    return [
        TimestampBehavior::className(),
    ];
}

public function getUser_meta()
{
    return $this->hasMany(UserMeta::className(), ['user_id' => 'id']);
}

public function getDisplay_name()
{
    return $this->first_name . ' ' . $this->last_name;
}

public function getProjects()
{
    return $this->hasMany(Project::className(), ['client_id' => 'id']);
}

/**
 * @inheritDoc
 */
```

```
public function beforeSave($insert) {
   return parent::beforeSave($insert);
}

/**
 * @inheritDoc
 */
public function afterSave($insert, $changedAttributes){
   parent::afterSave($insert, $changedAttributes);

   $this->saveMetaFields($insert);
}

/**
 * Save user meta
 *
   * @param bool $insert If `false`, it means the method is called while
updating a record.
 * @return void
 */
public function saveMetaFields($insert)
{
   $postadata = Yii::$app->request->post('User');
   $meta = $postadata['meta'] ?? [];

   if( !empty($meta) ) {

      if( !$insert ) {
         foreach ($meta as $meta_key => $meta_value) {
            $this->updateMeta($meta_key, $meta_value);
         }
```

```
            } else {
                foreach ($meta as $meta_key => $meta_value) {
                    $this->meta[] = [
                        'user_id' => $this->id,
                        'meta_name' => $meta_key,
                                                    'meta_value'  =>  is_array($meta_value)?
serialize($meta_value) : $meta_value
                    ];
                }

                $userMeta = new UserMeta;


Yii::$app->db->createCommand()->batchInsert(UserMeta::tableName(),     ['user_id',
'meta_name', 'meta_value'], $this->meta)->execute();
            }
        }
    }

    /**
     * @param string $id [User ID]
     */
    public static function findIdentity($id)
    {
        return static::findOne(['id' => $id, 'status' => self::STATUS_ACTIVE]);
    }

    /**
     * @param string $token [description]
     * @param string $type
     */
```

```
public static function findIdentityByAccessToken($token, $type = null)
{
    throw new NotSupportedException('"findIdentityByAccessToken" is not
implemented.');
}

/**
 * Finds user by email
 *
 * @param string $email
 * @return static|null
 */
public static function findByEmail($email)
{
        return static::findOne(['email' => $email, 'status' =>
self::STATUS_ACTIVE]);
}

public static function findByEmailAndRole($email, $role)
{
   return static::find()
      ->select('user.*')
      ->leftJoin('auth_assignment', 'auth_assignment.user_id = user.id')
      ->where(['user.email' => $email])
      ->where(['auth_assignment.item_name' => $role])
      ->with('auth_assignment')
      ->one();
}

/**
 * Finds user by password reset token
```

```
 *
 * @param string $token password reset token
 * @return static|null
 */
public static function findByPasswordResetToken($token)
{
    if (!static::isPasswordResetTokenValid($token)) {
        return null;
    }

    return static::findOne([
        'password_reset_token' => $token,
        'status' => self::STATUS_ACTIVE,
    ]);
}

/**
 * Finds out if password reset token is valid
 *
 * @param string $token password reset token
 * @return bool
 */
public static function isPasswordResetTokenValid($token)
{
    if (empty($token)) {
        return false;
    }

    $timestamp = (int) substr($token, strrpos($token, '_') + 1);
    $expire = Yii::$app->params['user.passwordResetTokenExpire'];
    return $timestamp + $expire >= time();
```

```
}

/**
 * @inheritdoc
 */
public function getId()
{
   return $this->getPrimaryKey();
}

/**
 * @inheritdoc
 */
public function getAuthKey()
{
   return $this->auth_key;
}

/**
 * @param string $authKey
 */
public function validateAuthKey($authKey)
{
   return $this->getAuthKey() === $authKey;
}

/**
 * Validates password
 *
 * @param string $password password to validate
 * @return bool if password provided is valid for current user
```

```
     */
    public function validatePassword($password)
    {
                    return    Yii::$app->security->validatePassword($password,
$this->password_hash);
    }

    /**
     * Generates password hash from password and sets it to the model
     *
     * @param string $password
     */
    public function setPassword($password)
    {
                                        $this->password_hash         =
Yii::$app->security->generatePasswordHash($password);
    }

    /**
     * Generates "remember me" authentication key
     */
    public function generateAuthKey()
    {
        $this->auth_key = Yii::$app->security->generateRandomString();
    }

    /**
     * Generates new password reset token
     */
    public function generatePasswordResetToken()
    {
```

```
$this->password_reset_token =
Yii::$app->security->generateRandomString() . '_' . time();
}

/**
 * Removes password reset token
 */
public function removePasswordResetToken()
{
   $this->password_reset_token = null;
}

public function getMeta($key)
{
   $userMeta = new UserMeta();
   return $userMeta->getMeta($this->id, $key);
}

public function addMeta($key, $value=null)
{
   $userMeta = new UserMeta();
   return $userMeta->addMeta($this->id, $key, $value);
}

public function updateMeta($key, $value=null)
{
   $result = false;
   $meta_field = UserMeta::find()->where(['user_id' => $this->id,
'meta_name' => $key])->one();

   if( !$meta_field ) {
```

```
            $meta = new UserMeta;
            $meta->user_id = $this->id;
            $meta->meta_name = $key;
            $meta->meta_value = is_array($value)? serialize($value) : $value;
            $result = $meta->save();

        } else {

                $meta_field->meta_value = is_array($value)? serialize($value) :
$value;
            $result = $meta_field->update();

        }

        return $result;
    }

    public function deleteMeta($key)
    {
        $userMeta = new UserMeta();
        return $userMeta->getMeta($this->id, $key);
    }

    public function getUserBalance()
    {
        return $this->hasOne(UserBalance::className(), ['user_id' => 'id']);
    }

    public function getCurrency()
    {
```

```
        return $this->hasOne(Currency::className(), ['id' => 'currency']);
    }
}
<?php
namespace backend\controllers;

use Yii;
use common\models\Currency;
use common\models\UserBalance;
use yii\data\Pagination;
use yii\filters\AccessControl;
use yii\data\ActiveDataProvider;
use yii\web\NotFoundHttpException;
use phpDocumentor\Reflection\Types\Integer;
use yii\web\Controller;


use yii\web\Response;
use backend\models\UserForm;
use common\models\User;
use common\models\Options;
use common\models\Exercise;
use common\web\BaseController;
use yii\bootstrap\ActiveForm;


/**
 * Site controller
 */
class UserController extends BaseController
{
```

```
/**
 * @inheritdoc
 */
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'actions' => [
                        'index', 'add', 'edit', 'delete'
                    ],
                    'allow' => true,
                    'roles' => ['admin'],
                ],
            ],
        ]
    ];
}

/**
 * @inheritdoc
 */
public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
    ];
```

```
}

/**
 * Displays homepage.
 *
 * @return string
 */
public function actionIndex()
{
    $current_user = Yii::$app->user;

    $query = User::find()->where(['!=', 'id', $current_user->id]);
    $search = Yii::$app->request->get('s');
    $filter = Yii::$app->request->get('role');

    if( $search ) {
        $search = explode(' ', $search);
        $filter = ['or'];

        foreach( $search as $search_word ) {
            $filter[] = ['like', 'first_name', $search_word];
            $filter[] = ['like', 'last_name', $search_word];
            $filter[] = ['like', 'email', $search_word];
        }

        $query->andFilterWhere( $filter );
    }

    if ( $filter ) {
        $usersId = Yii::$app->authManager->getUserIdsByRole($filter);
        $query->andFilterWhere(['id' => $usersId]);
```

```
        }
        return $this->render('index', [
            'query' => $query,
        ]);
    }

    public function actionAdd()
    {
        $model = new UserForm();
        $userBalanceModel = new UserBalance;
        $balanceCurrency = Currency::find()->indexBy('id')->all();
        $arrCurrencys = [];
        $userBalanceData = [];
        $newUserId = '';
        $postdata = Yii::$app->request->post();

        foreach ($balanceCurrency as $value) {
            $arrCurrencys[$value->id] = $value->code;
        }


        if (Yii::$app->request->isAjax && $model->load($postdata)) {
           Yii::$app->response->format = Response::FORMAT_JSON;
           return ActiveForm::validate($model);
        }


        if ($model->load(Yii::$app->request->post()) && $model->save()){
                    $newUserId = User::find()->indexBy('id')->where(['email' =>
$model->email])->one();
            $userBalanceData = [
```

```
                'user'      => $newUserId->id,
                'currency'  => $model->currency,
            ];
                          if ($userBalanceModel->load($userBalanceData) &&
$userBalanceModel->save()) {
                return $this->redirect('../user');
            }
        }

        return $this->render('add', [
            'model' => $model,
            'arrCurrencys' => $arrCurrencys
        ]);
    }

    public function actionEdit( $id )
    {
        $model = new UserForm();
        $user = $model->findOne(['id' => $id]);
        $balanceCurrency = Currency::find()->indexBy('id')->all();
        $arrCurrencys = [];

        foreach ($balanceCurrency as $value) {
            $arrCurrencys[$value->id] = $value->code;
        }

        $model->id = $user->id;
        $model->first_name = $user->first_name;
        $model->last_name = $user->last_name;
        $model->email = $user->email;
        $model->salary_type = $user->salary_type;
```

```
        $model->currency = $user->currency;
        $model->salary = $user->salary;

        $postdata = Yii::$app->request->post();
        if (\Yii::$app->request->isAjax && $model->load($postdata)) {
          \Yii::$app->response->format = Response::FORMAT_JSON;
          return ActiveForm::validate($model);
        }

        if ($model->load($postdata) && $model->save()){
        }

        return $this->render('add', [
            'model' => $model,
            'user' => $user,
            'arrCurrencys' => $arrCurrencys,
        ]);
    }

    public function actionDelete( $id )
    {
        $model = new UserForm();
        $current_user = Yii::$app->user;
                            if      (Yii::$app->user->can('manage'      .
ucfirst(key(Yii::$app->authManager->getRolesByUser($id))))){
            if (User::find()->where(['!=', 'id', $current_user->id]) !== $id){
                $user = $model->findOne(['id' => $id]);
                $user->delete();
            }
        $this->redirect(['user/index']);
        }
```

```
    }
}
```