МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)
18 грудня 2023р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Інформаційна технологія проєктування інформаційної системи аграрного підприємства»
здобувачки групи ІН.м-23 Лазоренко Євгенії Сергіївни

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Євгенія ЛАЗОРЕНКО
(підпис)

Керівник,
в.о. завідувача кафедри,
кандидат технічних наук, доцент Ігор ШЕЛЕХОВ _____
(підпис)

Суми – 2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика» здобувачки групи ІН.м-23 Лазоренко Євгенії Сергіївни

1. Тема роботи *Інформаційна технологія проєктування інформаційної системи аграрного підприємства*

затверджую наказом по інституту від «06» грудня 2023 року № 1412-VI

2. Термін здачі здобувачем вищої закінченого роботи_ 18 грудня 2023 року_____

3. Вхідні дані до роботи _____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Огляд технологій, що застосовуються для інформаційних систем аграрних підприємств; 2) Постановка задачі та формування завдань дослідження; 3) Опис архітектури майбутнього застосунку; 4) Розробка додатку; 5) Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	*Огляд технологій для розробки інформаційних систем у агробізнесі*		
2.	*Постановка задачі та завдання дослідження*		
3.	*Архітектура та алгоритми інформаційної системи аграрного підприємства*		
4.	*Розробка інформаційної системи для аграрного підприємства*		
5.	*Оформлення пояснювальної записки до дипломної роботи*		

Здобувач вищої освіти _____ (підпис)

Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 50 стор., 14 рис., 1 додаток, 20 літературних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки присвячена розв'язанню важливої практичної задачі, розробка інформаційної системи орієнтованої на управління та контроль полями.

Об'єкт дослідження — процес проєктування інформаційних систем

Мета роботи — розробка інформаційної системи для аграрного підприємства з використанням Python, Flask та SQLAlchemy, орієнтованої на ефективне управління полями, моніторинг врожайності та оптимізацію логістичних процесів.

Предмет дослідження. Предметом дослідження є розробка та впровадження інформаційної технології, спрямованої на забезпечення ефективної роботи аграрного підприємства.

Результати — проведено детальний аналіз сучасних методів та інструментів для розробки інформаційних систем у сфері аграрного бізнесу. Були вивчені наявні програмні рішення, що використовуються для управління аграрними підприємствами, а також аналізовано підходи до оптимізації аграрних процесів. На основі проведеного аналізу було розроблено власний алгоритм для ефективного управління полями, моніторингу врожайності та логістики.

FLASK, SQLALCHEMY, PYDANTIC, АГРОПРОМИСЛОВІСТЬ, АРХІТЕКТУРА

ЗМІСТ

ВСТУП

У сучасному світі, де інформаційні технології стрімко розвиваються та трансформують багато аспектів нашого життя, важко переоцінити їх вплив на розвиток різноманітних галузей економіки. Однією з таких галузей, що активно впроваджує інноваційні технології, є аграрний сектор. Використання сучасних інформаційних систем в аграрному бізнесі відкриває нові можливості для підвищення продуктивності, управління ресурсами та оптимізації виробничих процесів.

У цій дипломній роботі ми зосереджуємося на розробці інформаційної технології, яка дозволить створювати спеціалізовані інформаційні системи для аграрних підприємств. Метою нашої роботи є не лише аналіз поточного стану ІТ-розвитку в сфері селянського господарства, а й виявлення основних викликів та розробка передових практик, якими можна задовольнити специфічні потреби цього сектору.

Для досягнення поставленої мети ми проведемо всебічний аналіз сучасних тенденцій та технологічних розв'язків, що застосовуються у сфері аграрного бізнесу. Особлива увага буде приділена вивчення специфічних вимог, які ставляться до інформаційних систем у цьому галузі. Це охоплює такі аспекти, як управління земельними ресурсами, прогнозування врожайності, ефективне використання водних та інших природних ресурсів, логістика та управління ланцюгами постачання, а також інтеграція з смарт-технологіями для автоматизації та збору даних.

Крім того, у роботі буде висвітлено важливість впровадження інформаційних технологій для вирішення глобальних викликів, з якими стикається аграрний сектор, включаючи зміни клімату, зростаючі вимоги до екологічності виробництва та необхідність забезпечення продовольчої безпеки.

Результати цієї роботи матимуть важливе практичне значення, оскільки вони дозволять аграрним підприємствам ефективніше управляти своєю діяльністю, оптимізувати витрати та підвищити загальну продуктивність. Також,

вони стануть внеском у розвиток наукової думки у сфері інформаційних технологій для агробізнесу та послужать основою для подальших наукових досліджень.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження актуальності проблеми

Аграрний сектор є одним з фундаментальних і стратегічних напрямків економіки України. Він відіграє важливу роль не тільки у забезпеченні продовольчої безпеки країни, але й у формуванні експортного потенціалу, оскільки Україна є одним зі світових лідерів у виробництві та експорті зернових культур та інших аграрних продуктів.

Хоча великі агрохолдинги в Україні вже активно використовують сучасні ІТ-рішення, включаючи системи точного землеробства, системи управління врожайністю та інші автоматизовані технології, багато середніх та малих аграрних підприємств все ще зіштовхуються з проблемами впровадження сучасних ІТ-рішень. Це обмежує їхню продуктивність та ефективність.

Існує значний потенціал для оптимізації аграрного сектору через впровадження інформаційних систем. Це може включати покращення управління ресурсами, автоматизацію процесів, підвищення точності прогнозування та використання аналітичних інструментів для прийняття обґрунтованих рішень.

Незважаючи на значний потенціал, існують певні виклики та перешкоди, які ускладнюють швидке впровадження ІТ в аграрний сектор. Це включає в себе обмежений доступ до сучасних технологій у регіонах, недостатність кваліфікованих ІТ-фахівців в аграрному секторі, висока вартість інноваційних рішень та необхідність адаптації глобальних ІТ-рішень до локальних умов.

Існує значна потреба в дослідженні та розробці інформаційних систем, які були б спеціально адаптовані до потреб українських аграрних підприємств. Це включає розробку доступних, масштабованих та ефективних рішень, які можуть бути легко інтегровані в існуючі процеси.

У світлі цих факторів, розробка і впровадження ефективних інформаційних систем для українських аграрних підприємств є не тільки актуальною, але й стратегічно важливою задачею, яка може підвищити продуктивність,

ефективність та конкурентоспроможність українського аграрного сектору на світовому ринку.

1.2 Постановка задачі

В контексті сучасного аграрного сектору України, ключовим завданням є розробка інформаційної системи, яка б оптимізувала управлінські процеси в аграрних підприємствах. Основні принципи створення такої системи повинні включати:

1. **Адаптація до специфіки аграрних підприємств України**. Врахування унікальних аспектів аграрного бізнесу в Україні, включаючи розмір підприємств, особливості землекористування, кліматичні умови та сільськогосподарську практику.
2. **Ефективне управління ресурсами та процесами**. Розробка функціоналу для управління земельними ресурсами, відстеження циклів вирощування, управління запасами та логістикою, а також інтеграції з системами контролю врожайності та погодними станціями.
3. **Інтеграція з існуючими системами та технологіями**. Забезпечення можливості інтеграції розробленої системи з вже існуючими програмами та технологіями, використовуваними на аграрних підприємствах, для забезпечення плавного переходу та використання.

Конкретні завдання, які потрібно вирішити в рамках дипломного проекту, включають:

1. **Аналіз потреб аграрного сектору**. Дослідження основних вимог, які пред'являються до інформаційних систем в аграрному секторі України, зі збором даних від аграрних підприємств.
2. **Розробка концепції системи**. Визначення основних компонентів системи, її архітектури та функціональних можливостей, що відповідають виявленим потребам.

3. **Проектування інтерфейсу та взаємодії з користувачем**. Розробка інтуїтивно зрозумілого інтерфейсу, який враховує специфіку роботи керівників та співробітників аграрних підприємств.
4. **Тестування та адаптація системи**. Проведення пілотного тестування розробленої системи на обраних аграрних підприємствах для оцінки її ефективності та внесення необхідних коректив.

Ця дипломна робота спрямована на розробку інформаційної системи, яка зможе значно покращити ефективність управління в аграрному секторі України, а також забезпечити основу для подальшого розвитку та інтеграції сучасних технологій у цій галузі.

2 ОГЛЯД АРХІТЕКТУРИ ТА ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ

2.1 Вибір архітектури

Для створення інформаційної системи аграрного підприємства, важливо ретельно вибрати архітектуру, яка враховуватиме специфічні потреби аграрного сектору, включаючи гнучкість, масштабованість, надійність та інтеграцію з існуючими системами. Ось кілька архітектурних підходів, які можна розглянути:

1. **Монолітна архітектура**:

   - **Опис**: Це традиційний підхід, де всі компоненти системи (інтерфейс користувача, обробка запитів, доступ до бази даних) інтегровані в один програмний пакет.

   - **Переваги**: Простота розробки та розгортання, відсутність складностей з мікросервісною комунікацією.

   - **Недоліки**: Складно масштабувати, важко вносити зміни без впливу на всю систему.

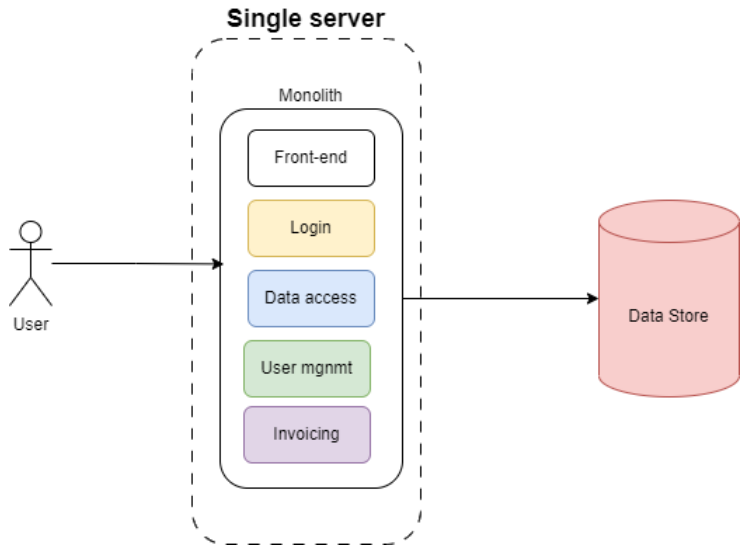

Рисунок 2.1 – Схема монолітної архітектури

2. **Мікросервісна архітектура**:

   - **Опис**: Розробка системи як набору незалежних сервісів, кожен з яких виконує певну функцію і комунікує з іншими через легко використовувані API.
   - **Переваги**: Гнучкість, масштабованість, можливість використання різних технологій для кожного сервісу, полегшення впровадження змін і оновлень.
   - **Недоліки**: Складність управління, потенційні проблеми з продуктивністю через мережеві виклики.

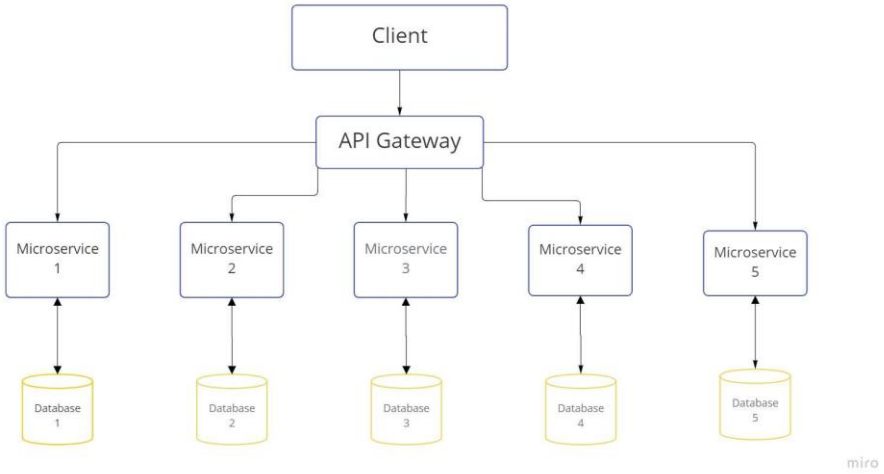

Рисунок 2.2 – схема мікросервісної архітектури

3. **Івент-орієнтована архітектура**:

   - **Опис**: Система реагує на події або зміни стану, ідеально підходить для систем, де потрібно швидко реагувати на різні умови та зміни.
   - **Переваги**: Висока реактивність, гнучкість у відповіді на події, легкість інтеграції з іншими системами.

- **Недоліки**: Може бути складно управляти та моніторити велику кількість подій.

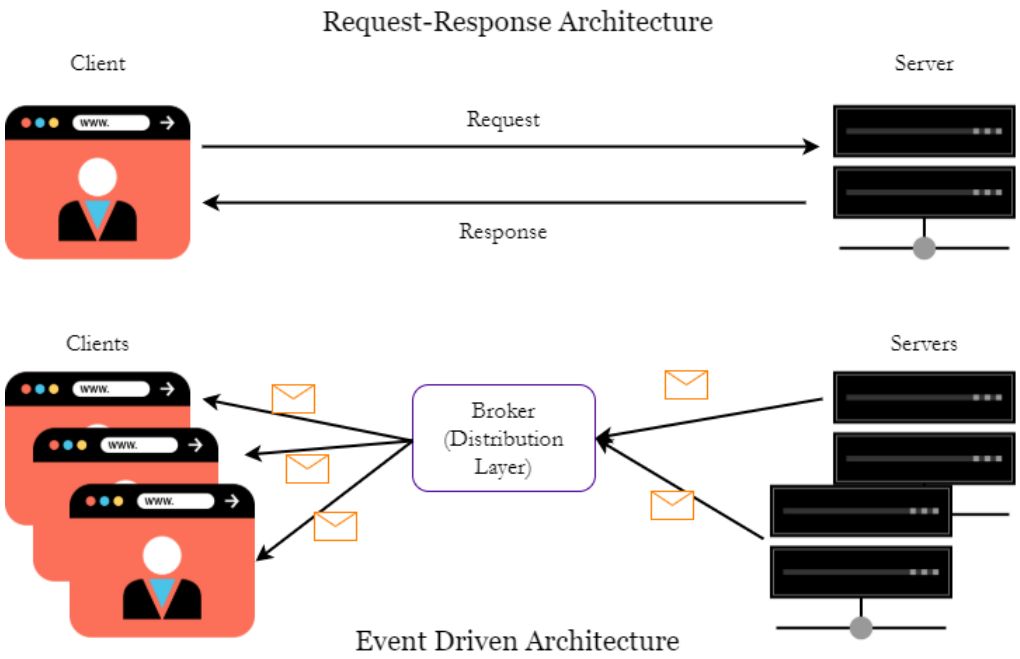

Рисунок 2.3 – івент-орієнтована архітектура

4. **Безсерверна (serverless) архітектура**:

   - **Опис**: Використання хмарних послуг для запуску коду без необхідності управління серверами. Функції виконуються у відповідь на події, такі як запити HTTP або зміни в базі даних.

   - **Переваги**: Мінімізація витрат на інфраструктуру, гнучкість, масштабованість.

   - **Недоліки**: Обмеження залежно від постачальника хмарних послуг, потенційні проблеми з продуктивністю при "холодному старті" функцій.

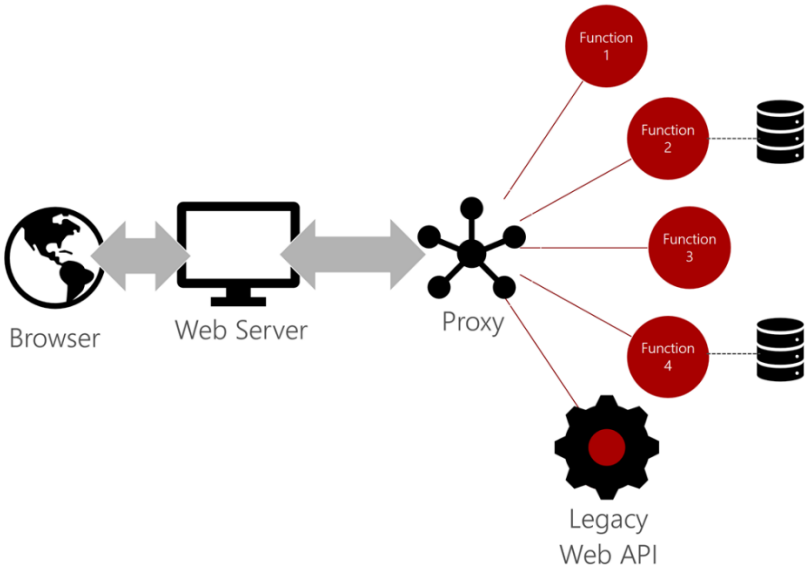

Рисунок 2.4 – схема serverless архітектури

5. **Інтегрована гібридна Архітектура**:

   - **Опис**: Комбінація декількох архітектурних підходів, наприклад, інтеграція мікросервісів з монолітними додатками або використання івент-орієнтованих компонентів у мікросервісній архітектурі.

   - **Переваги**: Гнучкість у виборі найкращих підходів для різних частин системи, можливість поступової міграції та оновлення.

   - **Недоліки**: Складність управління та інтеграції різних компонентів.

Вибір архітектури залежатиме від специфічних вимог до системи, наявного бюджету, технічної експертизи команди та довгострокових цілей проекту. Важливо також врахувати можливість майбутньої інтеграції з іншими системами та технологіями.

Для нашого майбутнього проекту доцільним буде розпочати з використання монолітної архітектури, адже вона є швидкою у розробці та більш-менш можливою у масштабуванні. У подальшому можливо буде інтегрувати мікросервісну або serverless архітектуру.

2.2 Аналітичний огляд мов програмування

Для вирішення задачі, пов’язаної з розробкою інформаційної технології проєктування інформаційної системи аграрного підприємства, потрібно розглянути мови програмування, які можуть бути використанні для досягнення цілей.

Для розробки інформаційної системи аграрного підприємства, вибір мови програмування є ключовим аспектом, який визначає не тільки можливості реалізації, але й гнучкість, продуктивність та масштабованість системи. Розглянемо кілька мов програмування, які можуть бути використані для досягнення цієї цілі:

1. **Python**
    - Переваги: Чудова підтримка бібліотек для різноманітних завдань (від веб-розробки до машинного навчання), читабельний та простий у вивченні синтаксис. Висока продуктивність розробки.
    - Недоліки: Відносно повільніша швидкість виконання порівняно з компільованими мовами. Не завжди ідеально підходить для розробки з високими вимогами до ресурсів.
2. **Java**
    - Переваги: Масштабована, надійна та добре підходить для підприємницьких додатків. Має сильну типізацію та велику екосистему.
    - Недоліки: Може бути важчою для вивчення та розробки порівняно з Python. Не така гнучка для швидких ітерацій розробки.
3. **JavaScript (Node.js для бекенду)**
    - Переваги: Підтримує повноцінну розробку як на клієнтській (фронтенд), так і на серверній стороні (Node.js), що забезпечує уніфікований підхід до розробки.

- Недоліки: Асинхронна природа JavaScript може ускладнити розуміння та управління кодом. Відносно новий у контексті серверної розробки.

4. **C# (.NET Framework)**
    - Переваги: Сильна інтеграція з .NET Framework, що надає потужні інструменти для розробки. Підходить для створення як десктопних, так і веб-додатків.
    - Недоліки: Менш портативна порівняно з Java або Python. Традиційно залежна від Microsoft екосистеми, хоча .NET Core розширив можливості міжплатформеної розробки.
5. **Ruby (Ruby on Rails)**
    - Переваги: Висока продуктивність розробки завдяки "convention over configuration" підходу у Ruby on Rails. Чудово підходить для швидкого прототипування.
    - Недоліки: Може мати проблеми з продуктивністю на великих масштабах. Менш поширена в порівнянні з іншими мовами.
6. **Go (Golang)**
    - Переваги: Висока продуктивність, легка мультипотоковість завдяки горутинам, компільована мова. Підходить для високонавантажених систем та мікросервісів.
    - Недоліки: Менший обсяг бібліотек та інструментів порівняно з Python або Java.

З усіх цих мов, Python здається оптимальним вибором для розробки інформаційної системи аграрного підприємства, особливо з огляду на його широку підтримку в областях, які є ключовими для такого проекту (наприклад, обробка даних, інтеграція з науковими бібліотеками). Python також забезпечує гнучкість та швидкість розробки, що є важливим для проектів, де потрібно швидко адаптуватися та вносити зміни.

2.3 Аналітичний огляд бібліотек розробки

У попередньому підрозділі ми визначилися з мовою програмування, яка є однією зі складових фундаменту нашої майбутньої інформаційної технології. Ми використовуватимемо мову Python, яка є не лише простою у вивченні та масштабуванні, а також має велику кількість популярних бібліотек, що значно спрощують розробку продуктів.

Розглянемо основні інструменти, які можуть бути корисними для нашої дипломної роботи. Почнемо з веб-фреймворків, що дозволяють спростити розробку веб-застосунків за допомогою мови програмування Python.

1. **Веб-фреймворки:**
   - 1.1. **Django.** Це високорівневий веб-фреймворк, який дозволяє швидко розробляти безпечні та обслуговувані веб-додатки. Django містить багато вбудованих функцій для управління користувачами, базами даних, формами та іншими типовими веб-завданнями, що робить його відмінним вибором для створення повнофункціональних веб-інтерфейсів для інформаційних систем.
   - 1.2. **Django REST Framework.** Розширення для Django, яке спрощує створення RESTful API. Це корисно для створення інтерфейсів, які можуть взаємодіяти з веб-клієнтами або іншими сервісами.
   - 1.3. **Flask.** Легкий веб-фреймворк, який надає більше гнучкості у виборі компонентів для веб-додатку. Flask ідеально підходить для створення легковажних RESTful API, які можуть служити бекендом для веб-інтерфейсів або мобільних додатків.
   - 1.4. **FastAPI.** Даний фреймворк є відносно новим серед перелічених, проте вже складає потужну конкуренцію ветеранам серед фреймворків веб-програмування. Він за своєю сутністю дещо схожий на Flask, проте очікувано має свої відмінності. FastAPI ідеально працює у парі з валідатором даних Pydantic. Тобто, вам не доведеться писати сотні ліній коду, щоб провалідувати чи

правильні дані ми отримали на вхід та чи правильні дані надіслав сервер. Pydantic дозволяє спростити такі процеси до декількох десятків ліній коду, але при цьому забезпечувати той самий та навіть кращий результат.

Також нам варто врахувати, що наша інформаційна система може мати власну базу даних. Ми можемо використовувати звичайні SQL-запити у нашому коді, будувати таблиці за допомогою SQL-скриптів. В певних випадках такий підхід є виправданим, проте ми повинні врахувати що з ним може бути два варіанти: або розробнику доведеться постійно відволікатися та перемикатися з синтаксису мови програмування на SQL, або для цього потрібно буде залучати додаткових розробників з експертизою у базах даних.

Тому пропонується використовувати так звані ORM (Object-Relational Mapping) інструменти. За допомогою них ми можемо покрити такі випадки:

А) побудова таблиць у базі даних за допомогою імплементації моделей бази даних;

Б) вибудова зв’язків між таблицями та специфікація різного роду поведінки (наприклад, поведінка у випадку видалення тієї чи іншої сутності з БД);

В) міграція бази даних, а саме у випадках коли або змінюються таблиці та колонки в них, або ж ми переходимо на іншу СУБД;

Г) різного роду запити, а саме на додавання даних, отримання даних за певними критеріями, оновлення даних або ж їх видалення з БД

Саме ці 4 пункти та навіть більше покриває ORM, а найголовніше – ми не відходимо від синтаксису мови програмування.

Тож розглянемо основні ORM-інструменти, які існують в контексті мови програмування Python.

2. ORM-бібліотеки:

2.1. SQLAlchemy. Дана бібліотека є ветераном серед усіх ORM. Вона повністю підтримує більшість популярних СУБД та їх особливості, а для використання їх у парі з іншими СУБД достатньо всього лише

встановити так званий «драйвер». Наприклад, якщо ми хочемо використовувати SQLAlchemy у парі з PostgreSQL, достатньо буде лише встановити бібліотеку psycopg2. Активно використовується з більшістю веб-фреймворків, окрім Django.

2.2. Django ORM. Дана бібліотека йде у парі з веб-фреймворком Django, що вкотре підтверджує твердження «Django – швейцарський ніж серед веб-фреймворків». Підтримуються популярні СУБД разом з їх особливостями, є можливість отримання чистих SQL-скриптів для різних випадків де це потрібно шляхом певної конфігурації запитів або моделей.

2.3. GINO. Дана ORM є полегшеною версією SQLAlchemy. Її особливість в тому, що вона є асинхронною, на відміну від старих версій SQLAlchemy (примітка: нові версії SQLAlchemy також мають можливість використання асинхронного підходу). Її полегшеність полягає у тому, що в середньому 3-5 ліній в оригінальній його бібліотеці зводяться до 1-2 ліній, але функціонал від цього не змінюється. Ідеально підходить для розробки легких застосунків, де не вимагаються складні операції з БД.

2.4. SQLModel. Бібліотека від розробника FastAPI. Її перевагою від усіх інших є те, що вона у собі поєднує як ORM, що працює із базою даних, так і функціонал валідації даних, який є притаманним бібліотеці Pydantic. Грубо кажучи, це симбіоз SQLAlchemy та Pydantic. Дана ORM досі на початковій стадії розробки, тому вона не рекомендується для масштабних застосунків, так як перехід на неї може викликати безліч запитань та проблем.

В подальшому також може бути потреба у складних розрахунках та візуалізації їх результатів, а також у візуалізації вже існуючих даних. В цьому випадку Python також надає широкий вибір інструментів, через що спеціалісти у сфері Data Science переважно використовують саме цю мову.

Розглянемо основні бібліотеки для складних розрахунків та візуалізації даних.

3. Складні розрахунки та візуалізація даних:

3.1. Pandas. Високоефективна бібліотека Python, що забезпечує розгорнутий набір інструментів для обробки та аналізу даних. Ця бібліотека користується значною популярністю і широко використовується у сферах аналізу даних та науки про дані. Pandas спрощує процес введення даних з різних типів файлів, включаючи CSV, Excel, JSON, SQL та інші формати. Крім того, вона надає зручність для збереження оброблених даних назад у ці ж формати.

3.2. **NumPy.** Фундаментальна бібліотека для наукових обчислень у Python. Вона надає підтримку для великих, багатовимірних масивів та матриць, разом з великим набором високорівневих математичних функцій для оперування з цими масивами. Бібліотека оптимізована для швидких обчислень, особливо для великих масивів даних, завдяки внутрішньому представленню даних у вигляді однорідних, низькорівневих структур. Включає функції для базових математичних операцій, статистичного аналізу, лінійної алгебри, трансформацій Фур'є, і багато інших.

3.3. Matplotlib. Бібліотека для створення статичних, анімованих та інтерактивних візуалізацій у Python. Вона є однією з найбільш використовуваних та впізнаваних бібліотек для графічного представлення даних. Вона надає детальний контроль над елементами графіків, дозволяючи налаштувати майже кожен аспект візуалізації – від маркерів ліній до шрифтів та розмітки.

Matplotlib підтримує створення інтерактивних графіків, які можна використовувати у веб-додатках та інтерфейсах. Ідеально інтегрується з NumPy, дозволяючи легко використовувати масиви NumPy для створення графіків.

3.4. Seaborn. Високорівнева бібліотека візуалізації даних для Python, що побудована на основі Matplotlib. Вона надає більш елегантний та зрозумілий інтерфейс для створення статистичних графіків. Seaborn автоматично застосовує стилі, які роблять графіки більш привабливими та читабельними. Бібліотека спрощує процес візуалізації складних типів даних із різноманітними варіантами графіків. Seaborn ідеально інтегрується з Pandas, роблячи візуалізацію даних з DataFrame зручною та ефективною. Бібліотека забезпечує інструменти для детального статистичного аналізу та візуалізації даних.

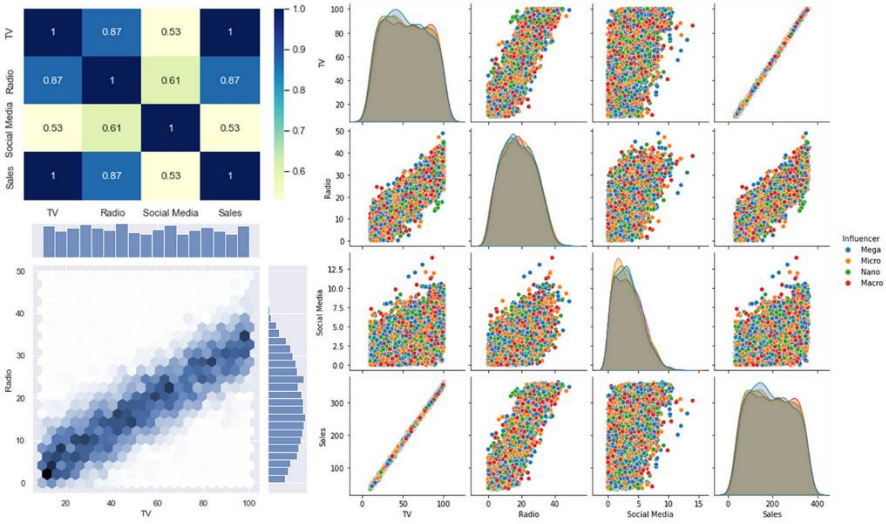

Рисунок 2.5 – візуалізація даних за допомогою Seaborn

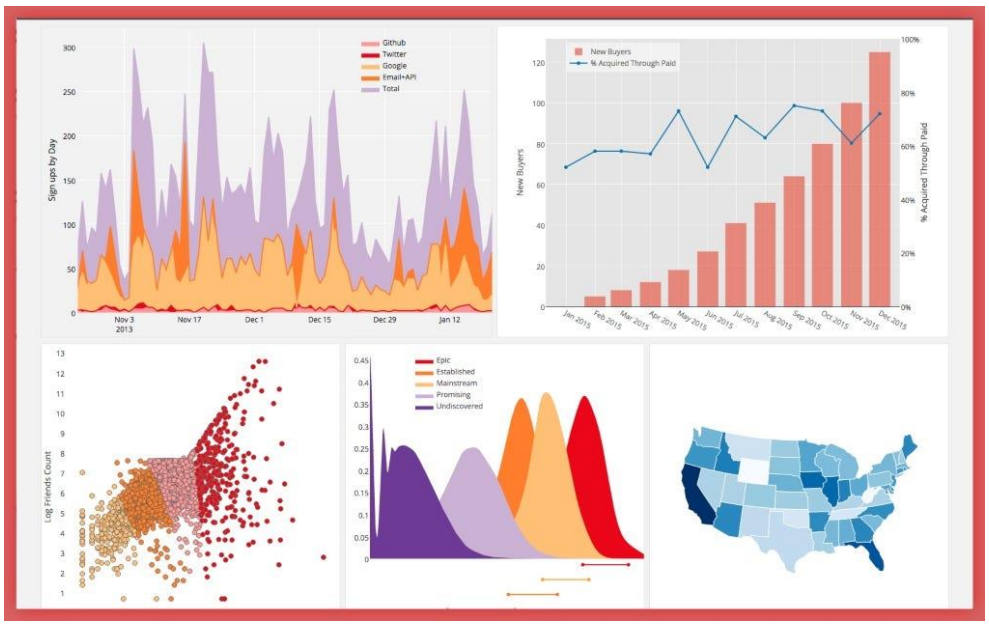

Рисунок 2.6 – візуалізація даних за допомогою Matplotlib

Якщо ми матимемо у нашій ІС велику БД, складні запити до неї, а також масштабні розрахунки та їх візуалізацію – наш застосунок може не впоратися з таким великим навантаженням. Проте така проблема нівелюється, якщо ми спробуємо застосувати певні інструменти, що дозволять асинхронізувати певні процеси і таким чином не конфліктувати один з одним.

Розглянемо певні з таких інструментів, які ми можемо використовувати у Python.

4. Інструменти для зменшення навантаження на наш застосунок:

4.1. Celery. Потужний і гнучкий інструмент для асинхронного виконання задач у Python. Ця бібліотека ідеально підходить для обробки задач у фоновому режимі, дозволяючи додаткам Python зосередитися на обробці запитів в реальному часі. Celery підтримує різні системи черг повідомлень для передачі задач, з RabbitMQ і Redis як найпопулярнішими варіантами. Вона також дозволяє легко інтегрувати задачі з веб-додатками, створеними за допомогою популярних фреймворків, таких як Django або Flask. Однією з ключових особливостей Celery є її масштабованість. Це означає, що вона може обробляти як невелике навантаження з

декількома задачами, так і великі системи з тисячами одночасних задач. Бібліотека також надає вдосконалені можливості для моніторингу та управління задачами, забезпечуючи високий рівень контролю та видимості процесу виконання задач.

4.2. RabbitMQ. Є одним з найбільш використовуваних брокерів повідомлень відкритого коду, що відіграє ключову роль у розподіленій обробці даних та асинхронній комунікації в різноманітних програмних рішеннях. Як брокер повідомлень, RabbitMQ забезпечує надійний механізм для обміну повідомленнями між компонентами розподіленої системи, дозволяючи легко масштабувати додатки та покращувати їхню продуктивність. Однією з важливих переваг RabbitMQ є його висока продуктивність і простота масштабування, що дозволяє ефективно обробляти великі обсяги повідомлень, а також забезпечує гнучкість для розширення системи зі зростанням вимог. Крім того, RabbitMQ включає в себе інструменти моніторингу та управління, які надають цінний огляд стану системи та її продуктивності.

Ці інструменти Python дозволяють вирішувати широкий спектр завдань, пов'язаних з розробкою інформаційних систем: від наявності клієнта та серверу до обробки даних і їх візуалізації.

Для розробки першопочаткової версії ми використаємо веб-фреймворк Flask та ORM SQLAlchemy, адже це є найпопулярнішою зв’язкою, що не дивно: ці два інструменти у поєднанні дають високоефективний застосунок.

У подальшому для складних розрахунків, візуалізації даних та нівелювання їх навантаження на застосунок, можна використати Pandas/Numpy, Matplotlib/Seaborn, Celery та RabbitMQ.

2.4 Аналітичний огляд інструментів

Обравши бібліотеки, які ми використовуватимемо для розробки нашої системи, важливо також проаналізувати супутні інструменти, як наприклад СУБД, інструменти для фронтенд розробки. Для розробки інформаційної системи аграрного підприємства, важливо вибрати відповідні інструменти, які дозволять ефективно реалізувати всі аспекти системи.

Почнемо з розгляду СУБД, адже ми маємо справу з підприємством, де важливо зберігати дані, оброблювати їх, а їх масиви можуть налічувати більше ніж тисячі записів.

1. **Бази даних:**
   - 1.1. **PostgreSQL.** Потужна система управління реляційними базами даних (СУРБД) з відкритим вихідним кодом. Вона відома своєю міцністю, надійністю, гнучкістю та підтримкою широкого спектру складних даних, від простих запитів до великих транзакцій. PostgreSQL підтримує розширений SQL стандарт, включаючи складні запити, зовнішні ключі, відкладені транзакції, в'юз (views), тригери та збережені процедури. Крім стандартних типів даних, таких як текст, числа та дати, PostgreSQL підтримує географічні об'єкти (PostGIS), JSON/XML, масиви та інші складні типи. PostgreSQL може бути використаний на більшості операційних систем і сумісний з багатьма мовами програмування та інструментами, що робить його універсальним рішенням для розробників.
   - 1.2. **MongoDB.** Високоефективна, не реляційна (NoSQL) база даних, що зберігає дані у вигляді гнучких JSON-подібних документів. Її головна особливість – це здатність легко обробляти великі обсяги різноманітних даних і швидко адаптуватися до змінних вимог до

структури даних. MongoDB пропонує масштабованість і гнучкість з можливістю індексації і запитів, що робить її популярним вибором для великих додатків та додатків, які вимагають швидкої обробки великої кількості даних.

Також наша система потребуватиме зручний інтерфейс, адже у випадку постійної взаємодії з даними, без зручного інтерфейсу підприємство потребуватиме або більш досвідчених фахівців, або більше часу для того щоб вже існуючі фахівці опанували існуючі шляхи взаємодії з системою.

Розглянемо певні фронт-енд фреймворки, які широко використовуються у розробці.

2. **Фронтенд розробка:**

2.1. React.js. Декларативна, ефективна та гнучка JavaScript бібліотека для побудови інтерфейсів користувача. Розроблена Facebook, вона дозволяє розробникам створювати великі веб-додатки, які можуть змінювати дані, не перезавантажуючи сторінку. Основною перевагою React є те, що він фокусується на компонентному підході до розробки UI, що дозволяє розробникам розділяти інтерфейс на незалежні, повторно використовувані частини, що полегшує управління станом та властивостями компонентів. Однією з ключових особливостей React є віртуальний DOM, який оптимізує оновлення інтерфейсу користувача, підвищуючи продуктивність та роблячи додатки швидшими та більш відгуковими. React також інтегрується з іншими бібліотеками та фреймворками, що робить його відмінним вибором для розробки складних веб-додатків.

2.2. Vue.js. Інтуїтивно зрозуміла, гнучка та легка у вивченні JavaScript-бібліотека, створена для побудови інтерфейсів користувача та односторінкових додатків (SPA). Ця бібліотека поєднує реактивність та компонентний підхід до розробки, що робить її зручною для створення інтерактивних веб-інтерфейсів. Однією з ключових

особливостей Vue.js є його легка інтеграція з існуючими проектами, що робить його відмінним вибором для покращення вже наявних веб-сторінок. Vue.js пропонує детальний контроль над декларативним рендерингом та реактивними даними, а також забезпечує систему для управління станом додатку та роутингу. Vue.js вирізняється своєю легкістю, гнучкістю та зручністю, що робить його популярним серед розробників всіх рівнів. Він також має активну спільноту та широкий спектр плагінів та додаткових інструментів, що забезпечують широкі можливості для розширення функціоналу.

При масштабуванні нашого підприємства, є сенс у більшій автоматизації певних задач та розрахунків. Для цього ми можемо інтегрувати штучний інтелект, який буде навчено під конкретні задачі, а також у подальшому він матиме здатність до самонавчання.

Розглянемо основні інструменти, пов’язані з машинним навчанням та штучним інтелектом.

3. **Машинне навчання та штучний інтелект:**
    - 3.1. **Scikit-learn.** Це відкрита бібліотека для мови програмування Python, призначена для машинного навчання. Вона надає прості та ефективні інструменти для аналізу даних та моделювання. Бібліотека включає велику кількість алгоритмів для класифікації, регресії, кластеризації, зниження розмірності даних та вибору моделі. Scikit-learn забезпечує консистентний інтерфейс для всіх моделей, що спрощує їх використання та порівняння. Ця бібліотека тісно інтегрована з NumPy і SciPy, двома іншими популярними бібліотеками для наукових обчислень у Python. Scikit-learn надає засоби для попередньої обробки даних, включаючи нормалізацію, масштабування та кодування категорійних змінних. Бібліотека має розширену документацію та велику кількість прикладів, що

полегшує навчання та впровадження алгоритмів у практичні проекти.

3.2. **TensorFlow.** Цю бібліотеку, створену командою Google, відрізняє її відкритий код і гнучкість для створення машинно-навчальних моделей. Підходить для різних типів машинного навчання, від нейронних мереж до складних алгоритмів. Ідеально підходить для обробки великих обсягів даних завдяки підтримці розподілених систем.

3.3. **PyTorch.** Розробка Facebook, PyTorch славиться своєю швидкістю та адаптивністю. Дозволяє легко модифікувати графи нейронних мереж на льоту. Містить функції для автоматичного диференціювання та інших процесів глибокого навчання. Підтримується активною спільнотою, що забезпечує широкий спектр додаткових бібліотек і інструментів.

Для першопочаткового продукту буде обрано PostgreSQL у якості СУБД. У подальшому, для складних розрахунків можна використовувати MongoDB, адже він дозволяє зберігати дані у неструктурованому форматі та витримувати їх постійне оновлення. Для розробки інтерфейсу у подальшому можна обрати Vue.js, так як він часто використовується у розробці фронтенд-частини підприємства. Також у подальшому, для автоматизації певних задач та розрахунків, можна використовувати один з трьох перелічених інструментів для машинного навчання та штучного інтелекту.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Проєктування бази даних

Кожен подібний проєкт повинен мати базу даних для того, щоб зберДля створення схеми бази даних інформаційної системи аграрного підприємства, ми повинні спочатку визначити основні сутності та їхні взаємозв'язки, що відповідають цілям вашого диплому. Ось загальний опис сутностей та їхніх атрибутів для цієї бази даних:

1. **Поля (Fields)**:
    - ID поля (PK, int, unique)
    - Назва поля (varchar(25))
    - Площа поля (int32)
    - Тип ґрунту (varchar(25))
    - Географічні координати (varchar(40))
2. **Культури (Cultures)**:
    - ID культури (PK, int, unique)
    - Назва культури (varchar(25))
    - Вимоги до ґрунту (varchar(255))
    - Вимоги до клімату (varchar(255))
3. **Посівний План (CropPlan)**:
    - ID плану (PK, int, unique)
    - ID поля (FK (Fields))
    - ID культури (FK (Cultures))
    - Планова дата посіву (DateTime)
    - Планова кількість насіння (int32)
4. **Роботи з Догляду (CareWork)**:
    - ID роботи (PK, int, unique)
    - Тип роботи (varchar(25))
    - Дата виконання (DateTime)
    - Задіяні ресурси (varchar(50))

5. **Врожай (Harvests)**:
    - ID врожаю (PK, int, unique)
    - ID поля (FK (Fields))
    - ID культури (FK (Culture))
    - Дата збору врожаю (DateTime)
    - Обсяг врожаю (int32)
6. **Погодні Дані (WeatherData)**:
    - ID запису (PK, int, unique)
    - Дата (DateTime)
    - Температура (int32)
    - Вологість (int32)
    - Опади (int32)
7. **Ресурси (Resources)**:
    - ID ресурсу (PK, int, unique)
    - Назва ресурсу (varchar(25))
    - Тип ресурсу (varchar(25))
    - Кількість в наявності (int32)
    - Одиниця виміру (varchar(15))
8. **Працівники (Employee)**:
    - ID працівника (PK, int, unique)
    - Ім'я (varchar(25))
    - Посада (varchar(25))
    - Відділ (varchar(25))
    - Контактні дані (varchar(25))

В цьому списку ми можемо помітити такі помітки, як PK та FK. З PK (Primary Key) все зрозуміло – це унікальний ідентифікатор кожного запису в таблиці. А от FK (Foreign Key) – це ідентифікатор, який є посиланням на конкретний об’єкт в інший таблиці. Він дозволяє встановлювати відношення між

таблицями, на яких ми зараз і зупинимося. В нашій базі даних будуть такі відношення:

- **Поля - посівний план**. Кожне поле може мати кілька посівних планів.
- **Культури - посівний план**. Кожна культура може бути включена в декілька посівних планів.
- **Поля - роботи з догляду**. Кожне поле може мати кілька запланованих робіт з догляду.
- **Поля – врожай**. Кожне поле дає врожай, що відображається в окремих записах.
- **Погодні дані – поля**. Погодні умови асоційовані з конкретними полями.
- **Ресурси - роботи з догляду**. Ресурси використовуються для виконання робіт з догляду.
- **Працівники - роботи з догляду**. Працівники відповідають за виконання робіт з догляду.

Для візуального представлення цієї моделі рекомендується створити ER-діаграму, використовуючи спеціалізоване програмне забезпечення. Така діаграма допоможе у визначенні ключових зв'язків та оптимізації структури бази даних.

На рисунку 3.1 представлено базову ER-діаграму нашого майбутнього проєкту.

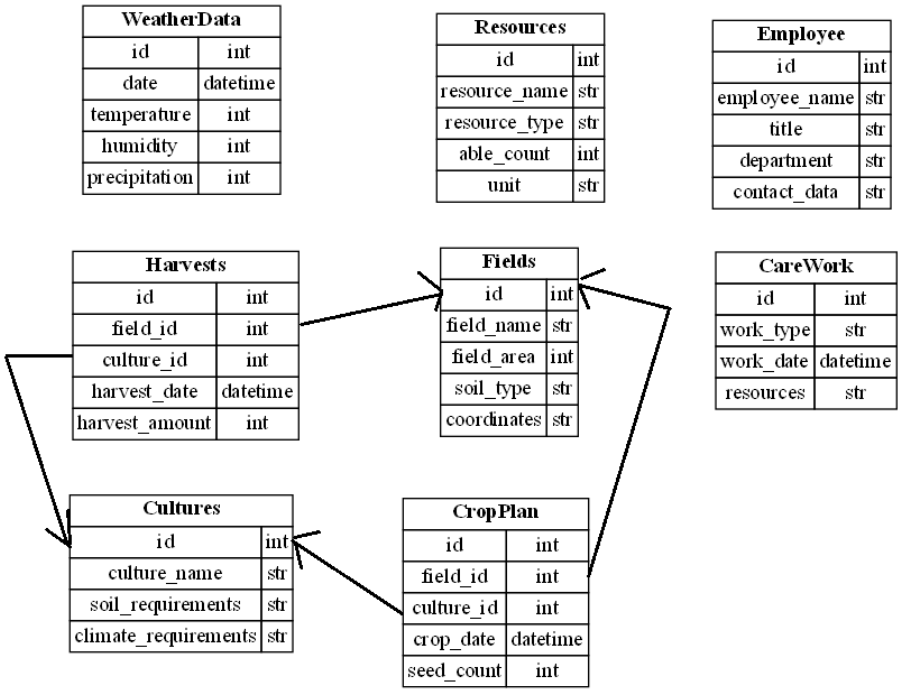

Рисунок 3.1 – базова ER-діаграма нашого проєкту

На цій діаграмі ми можемо помітити взаємовідносини між врожаями, полями, культурами та планами посіву. Всі інші таблиці у базі даних не впливають одна на одну та не залежать від даних одна-одної.

В якості СУБД планується використовувати PostgreSQL, адже вона є однією з найпоширеніших та найзручніших СУБД у використанні, а також славиться своєю надійністю. Проте, для тестування буде достатньо використовувати SQLite – ту саму відому «однофайлову» СУБД, яка не вимагає глибоких знань у налаштуванні БД.

Розробивши структуру нашої майбутньої БД, можемо перейти до аналізу та розробки use-cases.

3.2 Аналіз та розробка use-cases

Нам важливо визначити потенційні ролі та функціонал у нашій майбутній роботі, адже за їх відсутності неможливо зрозуміти, яким чином наш продукт зможе використовуватися у майбутньому.

Отже, почнемо з ролей.

1. **Адміністратор Системи (System Administrator)**
    - Повноваження:
        - Встановлення та налаштування системи
        - Створення та управління користувацькими рахунками
        - Налаштування прав доступу для всіх користувачів
        - Моніторинг та оптимізація роботи системи
        - Резервне копіювання та відновлення даних
        - Впровадження заходів безпеки та шифрування даних
2. **Агроном (Agronomist)**
    - Повноваження:
        - Аналіз стану ґрунтів та кліматичних умов
        - Розробка плану посівної кампанії
        - Вибір та ротація культур
        - Розрахунок потреби в добривах та ЗЗР (засобах захисту рослин)
        - Моніторинг врожайності та розробка рекомендацій для підвищення ефективності
3. **Менеджер Поля (Field Manager)**
    - Повноваження:
        - Контроль за виконанням плану робіт на полях
        - Координація роботи персоналу на полях
        - Ведення обліку використання ресурсів
        - Відправлення звітів про стан полів вищому керівництву
        - Реагування на надзвичайні ситуації та адаптація планів

4. **Бухгалтер (Accountant)**
    - Повноваження:
        - Ведення фінансового обліку
        - Планування та контроль бюджету
        - Формування звітності для податкових органів
        - Аудит та фінансовий контроль
        - Виплата зарплати та інших витрат
5. **Логіст (Logistician)**
    - Повноваження:
        - Планування та оптимізація логістичних потоків
        - Моніторинг транспортних засобів
        - Управління запасами та складським обліком
        - Взаємодія з постачальниками та покупцями
        - Контроль за якістю та термінами доставки

Ось такі ролі можуть бути у нашому майбутньому проєкті. Проте, ролей та їх повноважень недостатньо, щоб мати повне розуміння потенціалу нашого продукту. Тому нам потрібно перейти до визначення потенційних use-cases нашої майбутньої системи. Такими use-cases можуть бути наступні сценарії:

1. **Управління полями**
    - **Основний сценарій**: Менеджер поля використовує систему для моніторингу стану полів, вводить дані про виконані агротехнічні заходи, реєструє потребу в ресурсах для певних полів, отримує звіти про стан рослинності та ґрунту.
2. **Планування посівної кампанії**
    - **Основний сценарій**: Агроном аналізує дані про ґрунти та клімат, вибирає оптимальні культури для посіву, визначає строки та обсяги посіву, планує необхідність в ресурсах та добривах.
3. **Фінансове планування та контроль**

- **Основний сценарій**: Бухгалтер використовує систему для ведення фінансового обліку, контролює витрати та доходи, формує звіти, відстежує платежі та проводить аналіз фінансових показників.

4. **Логістика та управління запасами**
    - **Основний сценарій**: Логіст планує поставки ресурсів на аграрне підприємство, контролює запаси на складі, організовує доставку продукції до клієнтів, відстежує та оптимізує маршрути.

Ми визначили потенційні сценарії нашого майбутнього проєкту, тому можемо перейти до програмної реалізації.

3.3 Програмна реалізація

Для створення нашого проєкту, як вже було визначено раніше, ми використовуватимемо такі інструменти:

1. Мова программування Python (не нижче версії 3.8).
2. Веб-фреймворк Flask .
3. ORM SQLAlchemy (для взаємодії нашої бекенд-частини з базою даних).
4. СУБД SQLite.
5. Бібліотека валідації даних Pydantic.

Почнемо з мови програмування Python. Перш ніж її використовувати для розробки, нам потрібно встановити її інтерпретатор.

Щоб встановити Python, нам потрібно слідувати наступним етапам та крокам:

Етап 1: Завантаження Python

1. Переходимо на офіційний сайт Python python.org.
2. На головній сторінці ми знаходимо посилання для завантаження. Ми можемо завантажити найновішу версію Python, проте в даному випадку ми використаємо версію 3.8, адже ця версія є перевіреною роками, актуальною, а також це може позитивно вплинути у випадку подальшого залучення розробників до масштабування проєкту.

Етап 2: Встановлення Python

1. Після завантаження ми відкриваємо інсталятор.

2. У вікні інсталятора важливо обрати опцію "Add Python to PATH" перед тим, як натиснути "Install Now". Це дозволить нам запускати Python за допомогою командного рядка.

Рисунок 3.2 – вікно інсталятора Python

Етап 3: Перевірка Встановлення

1. Для перевірки встановлення Python, відкриваємо командний рядок (можна натиснути **Win+R**, ввести **cmd** та натиснути **Enter**).
2. У командному рядку введіть **python --version** і натисніть **Enter**. Якщо Python встановлено правильно, ви побачите повідомлення з версією Python.
3. Щоб перевірити, чи встановлений PIP (менеджер пакетів Python), введіть **pip --version**.

Етап 4: Оновлення PIP

PIP є інструментом Python, що дозволяє завантажувати та інсталювати усі відомі бібліотеки або фреймворки даної мови програмування. Для коректної роботи, рекомендується оновити PIP до останньої версії. Виконуємо в командному рядку команди:

bashCopy code

python -m pip install --upgrade pip

Після цього наш Python готовий до використання.

Також варто зауважити, що:

- Для MacOS та Linux процес встановлення трохи інший, але загальні принципи схожі.
- Встановлення додаткових пакетів Python можна здійснювати за допомогою команди **pip install package_name**.
- Для роботи з більш складними проектами рекомендується використовувати віртуальне середовище (наприклад, venv).

Перед тим, як ми перейдемо до написання перших рядків коду, нам потрібно встановити потрібні інструменти – Flask, SQLAlchemy та Pydantic. Все це робиться за допомогою команди pip install %library_name% (наприклад, pip install Flask), тому детальніше на цьому кроці ми зупинятися не будемо.

Після того, як ми активували віртуальне середовище та встановили потрібні бібліотеки, ми можемо розпочати роботу. Перш за все, створюємо основний файл main.py, директорію app та в ній - два основні python-файли, а саме models та routes. Потребу в цих файлах буде пояснено пізніше.

Наш Main.py виглядатиме наступним чином:

```
from flask import Flask
from app.models import db

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///your_database.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db.init_app(app)

with app.app_context():
    db.create_all()

if __name__ == '__main__':
    app.run(debug=True)
```

В ньому імпортуються потрібні фреймворки для коректної роботи, а також файл models з директорії app. Після цього ініціалізується наш застосунок, до

якого додаються конфігурації з розташуванням нашої БД та тим, чи відслідковувати зміни, які були внесені прямо під час роботи застосунку.

Після цього, ініціалізується з'єднання з базою даних нашого застосунку, а також контекстний менеджер, який створить базу даних у випадку, якщо вона відсутня.

Далі, класичні рядки, які вказують на запуск нашого застосунку, якщо він знаходиться у правильній робочій директорії.

Тепер перейдемо до файлу models.py. В даному файлі у нас будуть визначені моделі нашої бази даних, за допомогою яких наш застосунок матиме дві можливості:

1. Розуміти, які таблиці, рядки в них, типи даних має наша БД.
2. Створити нову БД у випадку, якщо її не існує або додати нові таблиці до вже існуючої.

Наш файл models.py виглядатиме приблизно так:

```
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime

db = SQLAlchemy()

class Fields(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    field_name = db.Column(db.String(25))
    field_area = db.Column(db.Integer)
    soil_type = db.Column(db.String(25))
    coordinates = db.Column(db.String(40))
```

Як і у випадку з main.py, імпортуються потрібні пакети, а також ініціалізується наша ORM. Після цього, у вигляді класів, які наслідуються від db.Model, ми створюємо моделі наших таблиць, які будуть присутні у нашій БД. Наприклад, клас Fields, який містить в собі певні колонки. Кожна з цих колонок ініціалізована у вигляді змінної, значення якої – це db.Column з певними аргументами. Частіше за все, це тип даних колонки, проте також можуть вказуватися такі аргументи, як primary_key, unique тощо.

Якщо казати про ініціалізацію foreign_key, то це виглядатиме таким чином:

```
class CropPlan(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    field_id = db.Column(db.Integer, db.ForeignKey('fields.id'))
    culture_id = db.Column(db.Integer, db.ForeignKey('cultures.id'))
```

Тобто, ми вказуємо тип даних, а також в якості аргументу додаємо db.ForeignKey та в якості аргументу для нього додаємо назву таблиці та колонку, з якої ми хочемо отримати значення.

Тепер перейдемо до файлу routes. В цьому файлі у нас будуть так звані endpoints, які власне займаються тим, щоб прийняти запит, обробити його та повернути відповідь.

Приклад одного з таких ендпоїнтів представлено нижче:

```
from flask import request, jsonify
from app.models import db, Fields


@app.route('/fields', methods=['POST'])
def create_field():
    data = request.json
    new_field = Fields(field_name=data['field_name'], ...)
    db.session.add(new_field)
    db.session.commit()
    return jsonify({"message": "Field created"}), 201
```

Кожен ендпоїнт містить декоратор. У Python це особливість, яка дозволяє модифікувати роботу нашої функції, не модифікуючи безпосередньо її «тіло». В даному випадку декоратор app.route реєструє наш ендпоїнт в полі зору застосунку, тобто застосунок заздалегідь знатиме, які «маршрути» він має. В якості аргументів декоратору виступає постфікс нашого посилання (наприклад, за посиланням localhost:8000/fields виконуватиметься функція create_field()), а також методи запитів, що прийматиме наш ендпоїнт. Так як наш ендпоїнт відповідатиме за створення поля у базі даних, а це пряма зміна даних на сервері, то тут підійде метод POST.

Тепер розберемо безпосередньо наш ендпоїнт. Він приймає запит, конвертує його у зрозумілий для застосунку формат і після цього готує запит до БД, щоб створити нове поле. За створення нового запису в БД у нас відповідатимуть рядки db.session.add (операція створення нового запису) та

db.session.commit() (операція підтвердження потрібних змін). Якщо операція буде вдалою, наш застосунок поверне повідомлення ‘Field created’ та статус код 200.

Проте, можуть бути помилки у передачі даних. Однією з найпоширеніших є відсутність певних обов’язкових полів або їх неправильний тип в тілі запиту. Тут в гру вступає Pydantic. Це потужна бібліотека, яка відповідає за валідацію даних, які відправляються застосунком або ж отримуються застосунком в запиті. За допомогою цієї бібліотеки ви можете описати, який тип даних ви очікуєте від користувача. Тобто якщо в тілі запиту він має field_name зі значенням 322, а ви в Pydantic вказали що очікуєте рядок (str), то буде висвітлена помилка, яка вкаже на неправильність отриманих даних. Це корисно тим, що користувач розумітиме в чому його помилка замість того, щоб постійно отримувати статус код 500 (Internal Server Error) без пояснення причини.

Ось приклад таких Pydantic-моделей:

```
from pydantic import BaseModel, Field
from datetime import datetime
from typing import Optional

class Fields(BaseModel):
    id: int
    field_name: str = Field(max_length=25)
    field_area: int
    soil_type: str = Field(max_length=25)
    coordinates: str = Field(max_length=40)

class Cultures(BaseModel):
    id: int
    culture_name: str = Field(max_length=25)
    soil_requirements: str = Field(max_length=255)
    climate_requirements: str = Field(max_length=255)

class CropPlan(BaseModel):
    id: int
    field_id: int
    culture_id: int
    crop_date: datetime
    seed_count: int
```

В цьому прикладі ми можемо помітити моделі валідації для полів (Fields), культур (Cultures) та планів посіву (CropPlan). Тобто, ми вказуємо, які типи

даних у колонок в тілах запиту повинні бути. Також ми вказали, яка максимальна довжина певних колонок у тілах запитів.

Зважаючи на масивність застосунку, першочергово буде розроблено автоматизовані тести. Їх відмінність та переваги у порівнянні з ручним тестуванням полягають в наступному:

1. Написання коду. Для написання автоматизованих тестів, у 90% випадків вимагаються навички програмування як мінімум на базовому рівні. Для більш складних сценаріїв, відповідно, вимагається поглибленні знання у програмуванні.
2. Економія часу та ресурсів. Для того, щоб перевірити, чи правильно потрібна функція оброблює дані та повертає статус-код 200, або ж її поведінку під час неправильної передачі даних, за допомогою ручного тестування залучається декілька людей, які підготовлюють ці дані та мають під рукою інструменти, які демонструють повідомлення від клієнту/серверу. Окрім цього, ручним тестувальникам доведеться витратити час на підготовку звітів. За допомогою автоматизованих тестів, ми можемо через певний час отримати повний журнал протестованих функцій, а саме – які функції було перевірено, який результат вони повернули, якщо негативний, то яку саме помилку було допущено тощо. Для написання таких тестів може бути достатньо одного розробника.
3. Повторне використання. Ці тести може бути розгорнуто у CI/CD (Continuous Integration/Continuous Delivery) середовищі (наприклад, Jenkins (рис.3.3)), що дозволяють або виконувати ці тести в певний інтервал часу, або у випадку можливих змін в нашому застосунку для того, щоб перевірити чи не повпливають ці зміни негативно. Також можливо додати інструмент Allure, який дозволить автоматизовано підготовлювати звіти з графіками, усіма даними тощо (рис.3.4).

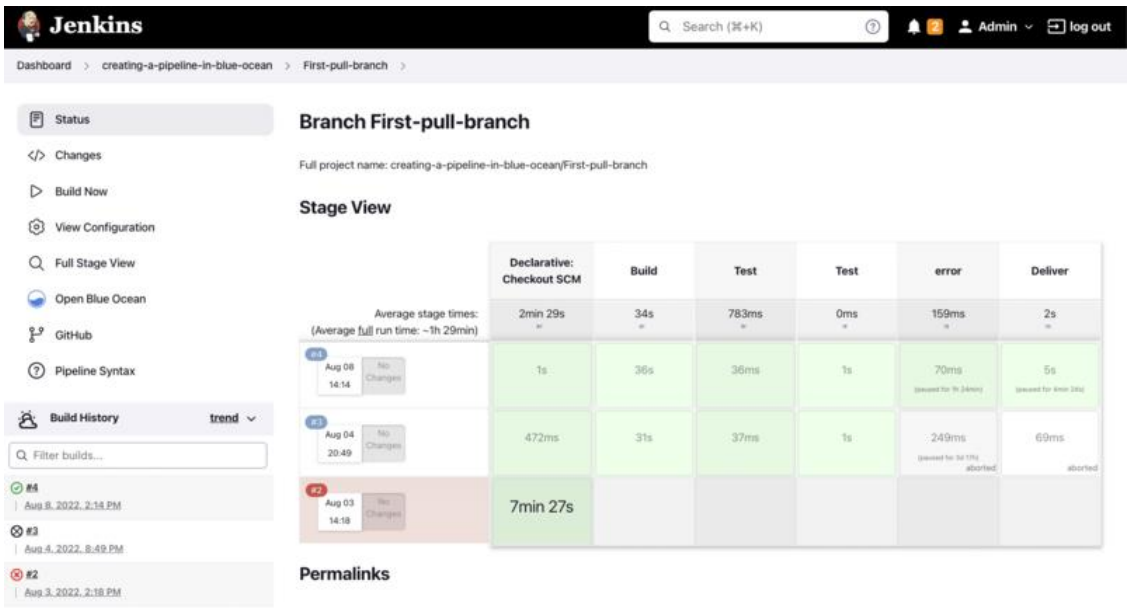

Рисунок 3.3 – CI/CD середовище Jenkins

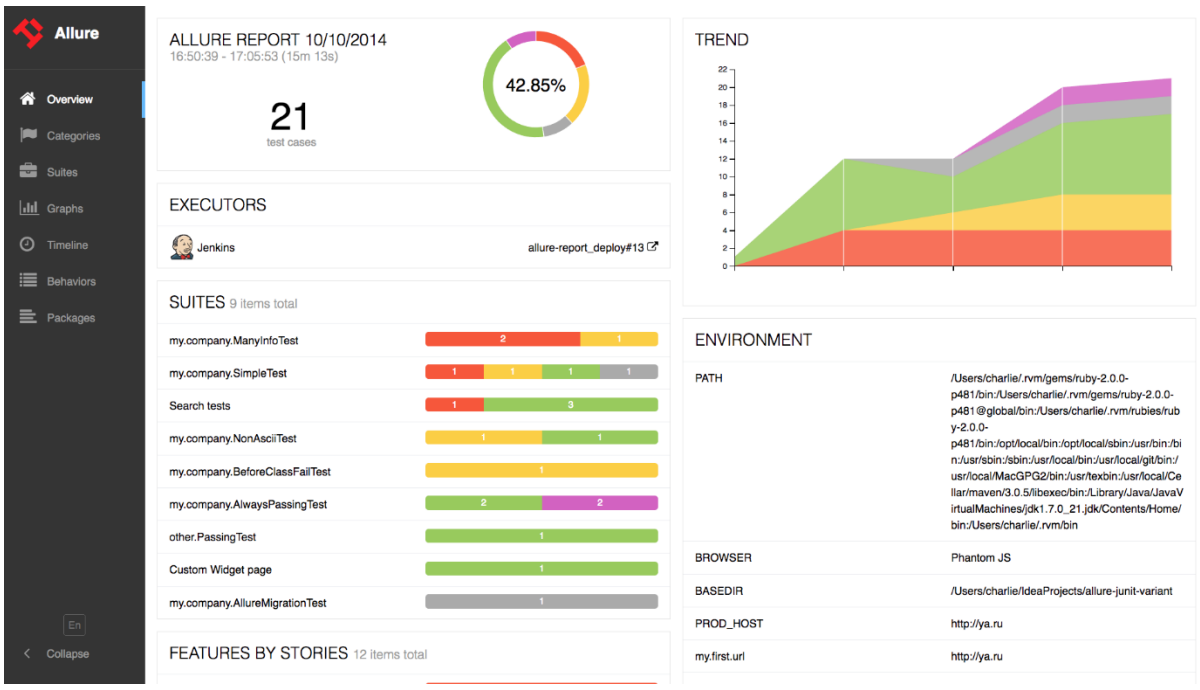

Рисунок 3.4 – приклад звіту, згенерованого за допомогою Allure

Власне, у зв’язку з цими відмінностями та перевагами, буде розроблено автоматизовані тести.

Розпочнемо з написання тестів для ендпоїнтів нашого застосунку. Перед тим, як розпочати, потрібно встановити дві важливі бібліотеки:

1. Pytest – популярна бібліотека, що дозволяє розробляти автоматизовані тести для Python-застосунків.

2. Flask-Testing – бібліотека, яка розроблена спеціально для написання тестів під застосунки, що використовують фреймворк Flask.

Встановлюємо їх традиційним чином, за допомогою команди pip install <library_name>.

Починаємо з імпортування потрібних бібліотек та пакунків. Це виглядатиме таким чином:

```
import pytest
from flask_testing import TestCase
from app import app, db
from models import WeatherData, Resources, Employee
from datetime import datetime
```

В цьому випадку ми імпортуємо pytest, з бібліотеки flask_testing імпортуємо TestCase. Також нам потрібно імпортувати складові нашого застосунку, а саме файл ініціалізації нашого застосунку та моделі БД. І наостанок – імпортуємо бібліотеку datetime, яка буде нам надавати потрібні дату та час.

Після цього, ми створюємо клас TestFlaskRoutes, який наслідується від TestCase, що ми раніше вже імпортували. В цьому застосунку є 3 обов'язкові функції – create_app, setup, tearDown.

```
class TestFlaskRoutes(TestCase):

    def create_app(self):
        app.config['TESTING'] = True
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'
        return app

    def setUp(self):
        db.create_all()


    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

Кожна з цих функцій містить наступне:

1. Create_app. Ініціалізуються конфігурації нашого застосунку, а саме TESTING = True, а посилання на базу даних в нас містить посилання на створену SQLite СУБД в оперативній пам'яті. Після цього, дана функція повертає наш застосунок.

2. setUp. В даній функції у нас створюється екземпляр тестової бази даних. Окрім цього, в ній також можна додати певні змінні, які будуть доступні для кожного з тестів.
3. tearDown. Дана функція виконуватиметься по завершенню усіх тестів. В нашому випадку ми видаляємо сесію з нашою тестовою базою даних і також повністю видаляємо саму тестову базу даних.

Тепер перейдемо до розробки тестів наших так званих «ендпоїнтів». Наприклад, нам потрібно протестувати ендпоїнт, що викликається за запитом з методом POST та за адресою /weather-data. Отже, в цьому тесті нам потрібно буде також підготувати дані, які ми передаватимемо до нашого застосунку. Виглядатиме це наступним чином:

```
# WeatherData POST Test
def test_create_weather_data(self):
    response = self.client.post('/weather-data', json={
        'date': '2023-01-01',
        'temperature': 25,
        'humidity': 60,
        'precipitation': 10
    })
    self.assertEqual(response.status_code, 201)
    self.assertIn('Weather Data created', response.json['message'])
```

Наш тест називається test_create_weather_data. Ми ініціалізуємо змінну response, в якій виконується POST-запит до /weather-data з певним набором даних, що буде конвертовано у JSON. Після цього, ми маємо так звані assert'и, які звіряють отримані дані з очікуваними. В цьому випадку ми очікуємо, що статус-код буде 201, а повідомленням від серверу буде 'Weather Data created'.

Тепер розглянемо приклад тестування GET-запиту. В цьому випадку, щоб отримати певні дані, нам потрібно їх для початку розмістити у нашій БД. Тестова функція виглядатиме наступним чином:

```
# WeatherData GET Test
def test_get_weather_data(self):
    weather_data = WeatherData(
        date=datetime.utcnow(),
        temperature=25,
        humidity=60,
        precipitation=10
```

```
        )
        db.session.add(weather_data)
        db.session.commit()
        response = self.client.get(f'/weather-data/{weather_data.id}')
        self.assertEqual(response.status_code, 200)
```

В ній ми підготували певну WeatherData, додали її до нашої СУБД, а після цього зробили запит до `/weather-data/{weather_data.id}` (в нашому випадку id буде дорівнювати 1). І тут ми також очікуємо статус-код 200.

Тестування PUT (оновлення) та DELETE (видалення) запитів відбувається схожим чином з попередніми функціями – підготовлюється певний набір даних, робиться запит, очікується потрібний статус-код та/або повідомлення від серверу.

```
    # Resources PUT Test
    def test_update_resource(self):
        # Припустимо, що у нас уже є ресурс з id=1 у базі даних
        response = self.client.put('/resources/1', json={
            'resource_name': 'NewName',
            'resource_type': 'Type',
            'able_count': 100,
            'unit': 'kg'
        })
        self.assertEqual(response.status_code, 200)

    # Employee DELETE Test
    def test_delete_employee(self):
        employee = Employee(
            employee_name='Jane Doe',
            title='Developer',
            department='IT',
            contact_data='jane@example.com'
        )
        db.session.add(employee)
        db.session.commit()
        response = self.client.delete(f'/employees/{employee.id}')
        self.assertEqual(response.status_code, 200)
        self.assertIn('Employee deleted', response.json['message'])
```

Повний код тестів нашого застосунку можна переглянути у додатку 1.

Тепер ми просто запускаємо розроблені тести та отримуємо наступний результат:

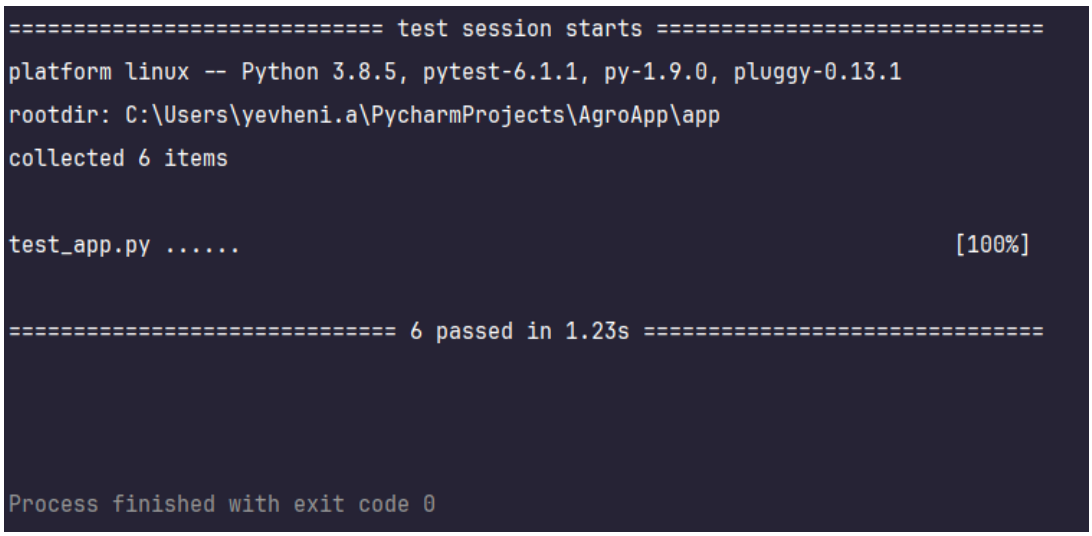

Рисунок 3.5 – результат виконання тестів нашого застосунку

Також нам потрібно розробити тести для нашої бази даних. За допомогою них ми матимемо розуміння, чи є працездатною наша база даних на даний момент та чи буде вона такою у випадку певних значних змін.

Для цього нам буде достатньо бібліотеки pytest та підготовлених SQLAlchemy моделей.

Імпорти виглядатимуть наступним чином:

```
import pytest
from app import create_app, db
from models import Fields, Cultures, CropPlan, CareWork, Harvests, WeatherData,
Resources, Employee
from datetime import datetime
```

Тобто, майже ніякої різниці з тестуванням застосунку, проте деякі пакунки не було імпортовано (наприклад, TestCase з бібліотеки flask_testing).

Одна з основних відмінностей від тестування застосунку – це те, що в цьому випадку ми не матимемо класу. Замість цього, ми використаємо функціональний підхід програмування.

Ми повинні ініціалізувати тестовий клієнт, який дозволить нам під’єднуватися до бази даних та виконувати певні дії. Перед його ініціалізацією ми повинні вказати декоратор pytest.fixture, що відповідає за виконання певних функцій напередодні інших тестів, а також дана функція викликатиметься та використовуватиметься у наших тестах.

```
@pytest.fixture
```

```
def test_client():
    flask_app = create_app('testing')
    with flask_app.test_client() as testing_client:
        with flask_app.app_context():
            db.create_all()
            yield testing_client
        db.drop_all()
```

Приклади таких тестів наведено нижче:

```
def test_create_and_query_field(test_client):
    field = Fields(field_name="Test Field", field_area=100, soil_type="Loam",
coordinates="25.789N, 80.226W")
    db.session.add(field)
    db.session.commit()

    queried_field = Fields.query.first()
    assert queried_field.field_name == "Test Field"

def test_create_and_update_culture(test_client):
    culture = Cultures(culture_name="Wheat", soil_requirements="Loamy",
climate_requirements="Temperate")
    db.session.add(culture)
    db.session.commit()

    culture.soil_requirements = "Sandy"
    db.session.commit()

    updated_culture = Cultures.query.get(culture.id)
    assert updated_culture.soil_requirements == "Sandy"
```

Тобто, ми ініціалізуємо змінну, в якій ініціалізуємо певну модель, додаючи до неї в якості аргументів дані, що будуть додаватися до нашої БД. Після цього, ми намагаємося отримати створений запис у нашій БД та за допомогою assert очікуємо, що певний атрибут з цього запису (наприклад, field_name) буде співпадати з тим, який ми передавали до БД у якості аргументу.

Повний код даних тестів буде представлено у додатку. Запускаємо наші тести та на рисунку 3.6 представлено результат їх виконання.

```
============================= test session starts ==============================
platform linux -- Python 3.8.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\yevheni.a\PycharmProjects\AgroApp\app
collected 11 items

test_models.py ...........                                               [100%]

============================== 11 passed in 12.62s ==============================
```

Рисунок 3.6 – результат виконання тестів моделей БД

Отже, автоматизовані тести було розроблено правильно, а наш застосунок є працездатним.

ВИСНОВКИ

У ході виконання цієї дипломної роботи було проведено глибокий аналіз проблеми створення ефективної інформаційної системи для аграрного підприємства. Основна увага була зосереджена на розгляді сучасних архітектурних підходів, вивченні наявних інструментів розробки, та виборі оптимального стеку технологій для реалізації проекту.

Було визначено, що використання мови програмування Python у поєднанні з легким, але потужним веб-фреймворком Flask, системою управління об'єктно-реляційною базою даних SQLAlchemy, та валідатором Pydantic ефективно вирішує поставлені завдання. Вибір SQLite як системи управління базами даних виявився вдалим рішенням завдяки її легковазі, надійності та простоті інтеграції.

Розроблена інформаційна система демонструє високу гнучкість, простоту масштабування та забезпечує відмінний рівень функціональності. Система покриває всі основні аспекти діяльності аграрного підприємства, включаючи управління полями, планування посівних кампаній, фінансовий та логістичний менеджмент.

Використання Pytest для тестування системи гарантує високу якість коду та надійність роботи інформаційної системи. Тестування було здійснено на різних рівнях: від модульних тестів окремих компонентів до інтеграційних тестів всієї системи.

Результатом роботи є повноцінна інформаційна технологія проєктування, яка не тільки відповідає всім сучасним вимогам до подібних систем, але й має потенціал для подальшого розвитку та вдосконалення. Ця система може стати надійним інструментом у руках аграріїв, сприяючи підвищенню ефективності їх роботи та оптимізації бізнес-процесів.

Таким чином, виконана дипломна робота демонструє успішне застосування сучасних технологій у розробці інформаційних систем і може слугувати добрим прикладом для подальших розробок у цій галузі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Python Official website [Електронний ресурс] – Режим доступу: https://www.python.org/
2. SQLAlchemy 2.0 Documentation [Електронний ресурс] – Режим доступу: https://docs.sqlalchemy.org/en/20/
3. Welcome to PostgreSQL [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/
4. Проектування інформаційних систем. Моделі і методи проектування інформаційних систем [Електронний ресурс] – Режим доступу: https://elearning.sumdu.edu.ua/free_content/lectured:de1c9452f2a161439391120eef364dd8ce4d8e5e/20160217112601/170352/index.html
5. Microservices vs. monolithic architecture. Atlassian [Електронний ресурс] – Режим доступу: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
6. Аграрний бізнес, металурги, IT, хімія та інші: як справи в гігантів української економіки [Електронний ресурс] – Режим доступу: https://www.epravda.com.ua/publications/2022/06/6/687837/
7. Pandas: управління даними проєкту [Електронний ресурс] – Режим доступу: https://medium.com/stinopys/pandas-%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F-%D0%B4%D0%B0%D0%BD%D0%B8%D0%BC%D0%B8-%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83-2-e1cf6cbc3ee9
8. An Introduction to Pydantic: the powerful Data Validation for your REST APIs [Електронний ресурс] – Режим доступу: https://engineering.projectagora.com/an-introduction-to-pydantic-the-powerful-data-validation-for-your-rest-apis-a6edfb46b0e8
9. PostgreSQL vs. Oracle: Let’s compare [Електронний ресурс] – Режим доступу: https://www.datavail.com/blog/postgresql-vs-oracle-lets-compare/
10. Testing a Flask framework with PyTest [Електронний ресурс] – Режим доступу: https://circleci.com/blog/testing-flask-framework-with-pytest/

11. Pydantic: Fast and Pythonic Data Validation for Your Python Applications [Електронний ресурс] – Режим доступу: https://vik-y.medium.com/pydantic-fast-and-pythonic-data-validation-for-your-python-applications-70fe339e4107

12. Software Architecture Patterns: What Are the Types and Which Is the Best One for Your Project [Електронний ресурс] – Режим доступу: https://www.turing.com/blog/software-architecture-patterns-types/

13. Як IT врятує агросектор [Електронний ресурс] – Режим доступу: https://app.agro-online.com/51684/details/

14. Агросектор нашого майбутнього [Електронний ресурс] – Режим доступу: https://www.agroone.info/publication/agrosektor-nashogo-majbutnogo/

15. What is a use case? [Електронний ресурс] – Режим доступу: https://www.wrike.com/blog/what-is-a-use-case/

16. Use Cases [Електронний ресурс] – Режим доступу: https://www.usability.gov/how-to-and-tools/methods/use-cases.html

17. What is an ER diagram? [Електронний ресурс] – Режим доступу: https://www.lucidchart.com/pages/er-diagrams

18. UML для бізнес-моделювання: для чого потрібні діаграми процесів [Електронний ресурс] – Режим доступу: https://evergreens.com.ua/ua/articles/uml-diagrams.html

19. What Is Web Application Development and How Do I Get Started? [Електронний ресурс] – Режим доступу: https://www.upwork.com/resources/what-is-web-application-development

20. How to get started with PostgreSQL [Електронний ресурс] – Режим доступу: https://www.freecodecamp.org/news/how-to-get-started-with-postgresql-9d3bc1dd1b11/

ДОДАТКИ

Додаток А. Вихідний код продукту

Models.py (моделі бази даних)

```
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime

db = SQLAlchemy()

class Fields(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    field_name = db.Column(db.String(25), nullable=False)
    field_area = db.Column(db.Integer, nullable=False)
    soil_type = db.Column(db.String(25), nullable=False)
    coordinates = db.Column(db.String(40), nullable=False)

class Cultures(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    culture_name = db.Column(db.String(25), nullable=False)
    soil_requirements = db.Column(db.String(255), nullable=False)
    climate_requirements = db.Column(db.String(255), nullable=False)

class CropPlan(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    field_id = db.Column(db.Integer, db.ForeignKey('fields.id'), nullable=False)
    culture_id = db.Column(db.Integer, db.ForeignKey('cultures.id'),
nullable=False)
    crop_date = db.Column(db.DateTime, nullable=False)
    seed_count = db.Column(db.Integer, nullable=False)

class CareWork(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    work_type = db.Column(db.String(25), nullable=False)
    work_date = db.Column(db.DateTime, nullable=False)
    resources = db.Column(db.String(50), nullable=False)

class Harvests(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    field_id = db.Column(db.Integer, db.ForeignKey('fields.id'), nullable=False)
    culture_id = db.Column(db.Integer, db.ForeignKey('cultures.id'),
nullable=False)
    harvest_date = db.Column(db.DateTime, nullable=False)
    harvest_amount = db.Column(db.Integer, nullable=False)

class WeatherData(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    date = db.Column(db.DateTime, nullable=False)
    temperature = db.Column(db.Integer, nullable=False)
    humidity = db.Column(db.Integer, nullable=False)
    precipitation = db.Column(db.Integer, nullable=False)

class Resources(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    resource_name = db.Column(db.String(25), nullable=False)
    resource_type = db.Column(db.String(25), nullable=False)
```

```
    able_count = db.Column(db.Integer, nullable=False)
    unit = db.Column(db.String(15), nullable=False)

class Employee(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    employee_name = db.Column(db.String(25), nullable=False)
    title = db.Column(db.String(25), nullable=False)
    department = db.Column(db.String(25), nullable=False)
    contact_data = db.Column(db.String(25), nullable=False)
```

main.py (файл з ініціалізацією Flask-застосунку)

```
from flask import Flask, request, jsonify
from models import db, Fields, Cultures, CropPlan, CareWork, Harvests,
WeatherData, Resources, Employee

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///your_database.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db.init_app(app)
```

CRUD-операції згідно моделей БД

Fields

```
@app.route('/fields', methods=['POST'])
def create_field():
    data = request.json
    new_field = Fields(
        field_name=data['field_name'],
        field_area=data['field_area'],
        soil_type=data['soil_type'],
        coordinates=data['coordinates']
    )
    db.session.add(new_field)
    db.session.commit()
    return jsonify({"message": "Field created"}), 201
```

Cultures

```
@app.route('/cultures', methods=['GET'])
def get_cultures():
    cultures = Cultures.query.all()
    return jsonify([culture.to_dict() for culture in cultures]), 200
```

CropPlans

```
@app.route('/cultures', methods=['GET'])
def get_cultures():
    cultures = Cultures.query.all()
    return jsonify([culture.to_dict() for culture in cultures]), 200
```

CareWork

```
@app.route('/care-work', methods=['PUT'])
def update_care_work():
    data = request.json
    care_work = CareWork.query.get(data['id'])
    if not care_work:
```

```
        return jsonify({"message": "Care Work not found"}), 404
    care_work.work_type = data['work_type']
    db.session.commit()
    return jsonify({"message": "Care Work updated"}), 200
```

Harvests

```
@app.route('/harvests/<int:id>', methods=['DELETE'])
def delete_harvest(id):
    harvest = Harvests.query.get(id)
    if not harvest:
        return jsonify({"message": "Harvest not found"}), 404
    db.session.delete(harvest)
    db.session.commit()
    return jsonify({"message": "Harvest deleted"}), 200
```

Ендпоїнти

```
@app.route('/weather-data', methods=['POST'])
def create_weather_data():
    data = request.json
    new_weather_data = WeatherData(
        date=data['date'],
        temperature=data['temperature'],
        humidity=data['humidity'],
        precipitation=data['precipitation']
    )
    db.session.add(new_weather_data)
    db.session.commit()
    return jsonify({"message": "Weather Data created"}), 201

@app.route('/weather-data/<int:id>', methods=['GET'])
def get_weather_data(id):
    weather_data = WeatherData.query.get_or_404(id)
    return jsonify(weather_data.to_dict()), 200

@app.route('/resources', methods=['GET'])
def get_resources():
    resources = Resources.query.all()
    return jsonify([resource.to_dict() for resource in resources]), 200

@app.route('/resources/<int:id>', methods=['PUT'])
def update_resource(id):
    data = request.json
    resource = Resources.query.get(id)
    if not resource:
        return jsonify({"message": "Resource not found"}), 404
    resource.resource_name = data['resource_name']
    resource.resource_type = data['resource_type']
    resource.able_count = data['able_count']
    resource.unit = data['unit']
    db.session.commit()
    return jsonify({"message": "Resource updated"}), 200

@app.route('/employees', methods=['POST'])
def create_employee():
    data = request.json
    new_employee = Employee(
```

```
        employee_name=data['employee_name'],
        title=data['title'],
        department=data['department'],
        contact_data=data['contact_data']
    )
    db.session.add(new_employee)
    db.session.commit()
    return jsonify({"message": "Employee created"}), 201

@app.route('/employees/<int:id>', methods=['DELETE'])
def delete_employee(id):
    employee = Employee.query.get(id)
    if not employee:
        return jsonify({"message": "Employee not found"}), 404
    db.session.delete(employee)
    db.session.commit()
    return jsonify({"message": "Employee deleted"}), 200
```

Тести

```
# test_app.py

import pytest
from flask_testing import TestCase
from app import app, db
from models import WeatherData, Resources, Employee
from datetime import datetime

class TestFlaskRoutes(TestCase):

    def create_app(self):
        app.config['TESTING'] = True
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'
        return app

    def setUp(self):
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    # WeatherData POST Test
    def test_create_weather_data(self):
        response = self.client.post('/weather-data', json={
            'date': '2023-01-01',
            'temperature': 25,
            'humidity': 60,
            'precipitation': 10
        })
        self.assertEqual(response.status_code, 201)
        self.assertIn('Weather Data created', response.json['message'])

    # WeatherData GET Test
    def test_get_weather_data(self):
        weather_data = WeatherData(
            date=datetime.utcnow(),
```

```
            temperature=25,
            humidity=60,
            precipitation=10
        )
        db.session.add(weather_data)
        db.session.commit()
        response = self.client.get(f'/weather-data/{weather_data.id}')
        self.assertEqual(response.status_code, 200)

    # Resources GET Test
    def test_get_resources(self):
        response = self.client.get('/resources')
        self.assertEqual(response.status_code, 200)

    # Resources PUT Test
    def test_update_resource(self):
        # Припустимо, що у нас уже є ресурс з id=1 у базі даних
        response = self.client.put('/resources/1', json={
            'resource_name': 'NewName',
            'resource_type': 'Type',
            'able_count': 100,
            'unit': 'kg'
        })
        self.assertEqual(response.status_code, 200)

    # Employee POST Test
    def test_create_employee(self):
        response = self.client.post('/employees', json={
            'employee_name': 'John Doe',
            'title': 'Manager',
            'department': 'HR',
            'contact_data': 'john@example.com'
        })
        self.assertEqual(response.status_code, 201)

    # Employee DELETE Test
    def test_delete_employee(self):
        employee = Employee(
            employee_name='Jane Doe',
            title='Developer',
            department='IT',
            contact_data='jane@example.com'
        )
        db.session.add(employee)
        db.session.commit()
        response = self.client.delete(f'/employees/{employee.id}')
        self.assertEqual(response.status_code, 200)
        self.assertIn('Employee deleted', response.json['message'])

if __name__ == '__main__':
    pytest.main()
```

Тестування БД

```
import pytest
from app import create_app, db
```

```
from models import Fields, Cultures, CropPlan, CareWork, Harvests, WeatherData,
Resources, Employee
from datetime import datetime

@pytest.fixture
def test_client():
    flask_app = create_app('testing')
    with flask_app.test_client() as testing_client:
        with flask_app.app_context():
            db.create_all()
            yield testing_client
        db.drop_all()

def test_create_and_query_field(test_client):
    field = Fields(field_name="Test Field", field_area=100, soil_type="Loam",
coordinates="25.789N, 80.226W")
    db.session.add(field)
    db.session.commit()

    queried_field = Fields.query.first()
    assert queried_field.field_name == "Test Field"

def test_create_and_update_culture(test_client):
    culture = Cultures(culture_name="Wheat", soil_requirements="Loamy",
climate_requirements="Temperate")
    db.session.add(culture)
    db.session.commit()

    culture.soil_requirements = "Sandy"
    db.session.commit()

    updated_culture = Cultures.query.get(culture.id)
    assert updated_culture.soil_requirements == "Sandy"
def test_crop_plan_operations(test_client):
    field = Fields(field_name="Field1", field_area=200, soil_type="Clay",
coordinates="25.770N, 80.214W")
    culture = Cultures(culture_name="Corn", soil_requirements="Rich",
climate_requirements="Warm")
    db.session.add(field)
    db.session.add(culture)
    db.session.commit()

    crop_plan = CropPlan(field_id=field.id, culture_id=culture.id,
crop_date=datetime.utcnow(), seed_count=500)
    db.session.add(crop_plan)
    db.session.commit()

    queried_crop_plan = CropPlan.query.first()
    assert queried_crop_plan.seed_count == 500

# Test for CareWork
def test_care_work_operations(test_client):
    care_work = CareWork(work_type="Irrigation", work_date=datetime.utcnow(),
resources="Water Pump")
    db.session.add(care_work)
    db.session.commit()

    queried_care_work = CareWork.query.first()
```

```
    assert queried_care_work.resources == "Water Pump"

# Test for Harvests
def test_harvest_operations(test_client):
    harvest = Harvests(field_id=1, culture_id=1, harvest_date=datetime.utcnow(),
harvest_amount=300)
    db.session.add(harvest)
    db.session.commit()

    queried_harvest = Harvests.query.first()
    assert queried_harvest.harvest_amount == 300

# Test for WeatherData
def test_weather_data_operations(test_client):
    weather_data = WeatherData(date=datetime.utcnow(), temperature=20,
humidity=50, precipitation=12)
    db.session.add(weather_data)
    db.session.commit()

    queried_weather_data = WeatherData.query.first()
    assert queried_weather_data.humidity == 50

# Test for Resources
def test_resources_operations(test_client):
    resource = Resources(resource_name="Tractor", resource_type="Equipment",
able_count=2, unit="Piece")
    db.session.add(resource)
    db.session.commit()

    resource.able_count = 3
    db.session.commit()

    updated_resource = Resources.query.first()
    assert updated_resource.able_count == 3

# Test for Employee
def test_employee_operations(test_client):
    employee = Employee(employee_name="Alice Smith", title="Agronomist",
department="Agriculture", contact_data="alice@example.com")
    db.session.add(employee)
    db.session.commit()

    queried_employee = Employee.query.first()
    assert queried_employee.title == "Agronomist"

    db.session.delete(queried_employee)
    db.session.commit()

    assert Employee.query.count() == 0


def test_employee_creation_and_deletion(test_client):
    employee = Employee(employee_name="John Doe", title="Developer",
department="IT", contact_data="john@example.com")
    db.session.add(employee)
    db.session.commit()

    assert Employee.query.count() == 1
```

```
db.session.delete(employee)
db.session.commit()

assert Employee.query.count() == 0
```