НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних систем, мереж та кібербезпеки

__________________ Касаткін Д.Ю., к.пед.н., доц.

(підпис) (ПІБ, вчене звання і ступінь)

«___»____________2025 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

На тему: «Вставити свою назву теми згідно наказу»

Спеціальність 123 «Комп’ютерна інженерія»

Гарант освітньої програми / Нікітенко Є.В. /

к.фіз.-мат.н., доц.

(підпис)

Керівник дипломного проекту:______________________ / Шкарупило В.В. /

(підпис) (ПІБ)

Виконав:____ / Юденко О.А./

(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
_______________ / Касаткін Д.Ю., к.п.н., доц. /
підпис ПІБ, вчене звання і ступінь
«_» _____________20___р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ СТУДЕНТУ

Юденка Олександра Анатолійовича
(прізвище, ім'я, по батькові)
Спеціальність (напрям підготовки): комп'ютерна інженерія
Тема кваліфікаційної бакалаврської роботи: « Розроблення комп'ютерної системи автоматизації розподілених програмних застосунків »

затверджена наказом ректора НУБіП України від "___" _________2024 р. №___________
Термін подання завершеної роботи на кафедру ______________
Вихідні дані до кваліфікаційної бакалаврської роботи проектування та розробка комп'ютерної системи автоматизації розподілених програмних застосунків

Перелік питань, що підлягають розробці:
1. Аналіз технічного завдання
2. Проєктування системи
3. Реалізація системи
4. Реалізація ком'ютерної системи

Перелік графічного матеріалу (за потреби) ______________

Дата видачі завдання "______"____________________2024 р.

Керівник бакалаврської кваліфікаційної роботи ______________ Шкарупило В.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання ______________ Юденко О.А.
(підпис) (прізвище та ініціали)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз технічного завдання	14.02.2025 р - 16.02.2025 р	Виконано
2	Проектування системи	01.03.2025 р - 19.03.2025 р	Виконано
3	Реалізація системи	20.04.2025 р - 15.05.2025 р	Виконано
5	Оформлення пояснювальної записки	11.05.2025 р - 27.05.2025 р	Виконано

Студент __________ **О.А. Юденко**

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____________ **В.В. Шкарупило**

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 56 сторінок, 19 лістингів, 11 джерел

Автоматизація тестування, розподілені застосунки, Jenkins, CI/CD, пайплайн, Docker, Python, програмне забезпечення, контроль якості.

Розроблення комп'ютерної системи автоматизації процесу тестування розподілених програмних застосунків

Об’єкт аналізу – Розроблення комп'ютерної системи автоматизації процесу тестування розподілених програмних застосунків.

Мета роботи – Розроблення комп'ютерної системи автоматизації процесу тестування розподілених програмних застосунків.

Проект складається з чотирьох розділів.

Проєкт складається з чотирьох розділів.

У першому розділі проаналізовано вимоги до системи, здійснено огляд сучасних підходів до тестування розподілених застосунків, а також вивчено існуючі інструменти автоматизації.

Другий розділ присвячено вибору архітектури, мови програмування, середовища виконання, а також обґрунтуванню вибору інструментів автоматизації, таких як Jenkins та Docker.

У третьому розділі представлено реалізацію Jenkins-пайплайну для автоматичного запуску тестів, приклади налаштування середовища та програмного коду.

Четвертий розділ містить опис практичного розгортання проєкту, створення сервісу на Python, приклади тестів та їх інтеграцію в процес CI/CD.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ 6
ВСТУП 7
1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ 10
1.1 Вимоги до комп'ютерної системи автоматизації тестування 10
1.2 Огляд сучасних підходів і інструментів для тестування розподілених застосунків 12
1.3 Аналіз типових архітектур розподілених застосунків 14
1.4 Постановка задачі розроблення 16
2.ПРОЄКТУВАННЯ СИСТЕМИ 18
2.1.1 Вибір моделі тестування розподілених застосунків 21
2.1.2 Мережева структура і середовище виконання тестів 23
2.1.3 Обґрунтування вибору інструменту Jenkins 25
2.2 Програмна частина системи 27
2.2.2 Розроблення тестових сценаріїв для розподілених застосунків 28
2.2.3 Інтеграція Jenkins із Python-сценаріями 30
3. Реалізація системи 33
3.1 Побудова Jenkins-пайплайну 33
3.1.1 Створення тестів і структура каталогу 35
3.1.2 Інтеграція Jenkins із Git-репозиторієм 37
3.1.3 Налаштування середовища виконання Python-скриптів у Jenkins 38
3.1.4 Генерація та публікація звітів про виконання тестів у Jenkins 39
3.2 Тестування застосунку у середовищі Jenkins 42
3.2.1 Розгортання існуючого проєкту в Jenkins 42
4. Реалізація комп'ютерної системи 45
4.1 Створення репозиторію та структури 45
4.2 Реалізація розподіленого застосунку 46
4.3 Розробка автоматизованих тестів для застосунку 48
4.4 Налаштування Jenkins для автоматизації тестування 50
4.5 Аналіз результатів тестування 53
Висновок 55
Список використаних джерел 56

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

CI/CD - Continuous Integration / Continuous Delivery — Безперервна інтеграція / доставка

БД - База даних

PyPI - Python Package Index

VS Code - Visual Studio Code

SPA - Single Page Application

ELK - Elasticsearch, Logstash, Kibana (стек для логування і моніторингу)

ПЗ - Програмне забезпечення

БКР - Бакалаврська кваліфікаційна робота

YAML - Yet Another Markup Language

XML - eXtensible Markup Language

UI - User Interface — Інтерфейс користувача

REST - Representational State Transfer

SPA - Single Page Application

VENV - Virtual Environment (віртуальне середовище Python)

PaaS - Platform as a Service

Git - Розподілена система контролю версій

ВСТУП

Сучасні інформаційні технології швидко розвиваються, охоплюючи все більше сфер людської діяльності. Зокрема, дедалі більшого поширення набувають розподілені програмні застосунки, що функціонують у розподіленому середовищі, взаємодіючи через комп'ютерні мережі. До таких систем належать хмарні сервіси, мікросервісні архітектури, вебзастосунки, мобільні клієнти з серверною частиною тощо. Їхні переваги – масштабованість, надійність, гнучкість та висока продуктивність – забезпечують вирішення складних задач у режимі реального часу. Водночас складність розподілених систем створює значні виклики у процесі їх тестування.

Однією з основних проблем у розробці розподілених застосунків є забезпечення якісного, всебічного та ефективного тестування. Це зумовлено наявністю великої кількості взаємодіючих компонентів, які можуть працювати на різних фізичних або віртуальних машинах, використовувати різні протоколи передачі даних та функціонувати в умовах змінного навантаження або непередбачуваних мережевих затримок. Ручне тестування в таких умовах виявляється недостатнім або надто затратним за часом і ресурсами. Тому актуальною задачею є автоматизація процесу тестування, яка дозволяє не лише знизити трудовитрати, а й підвищити достовірність, повторюваність та повноту перевірок.

Сьогодні існує чимало інструментів і підходів до автоматизованого тестування, однак більшість із них зосереджені на класичних клієнт-серверних або монолітних архітектурах. Для розподілених застосунків виникає потреба у гнучких рішеннях, здатних імітувати паралельну взаємодію компонентів,

враховувати враховувати мережеву топологію, обмеження пропускної здатності, помилки передачі тощо. Крім того, важливо мати можливість централізованого контролю за сценаріями тестування, ведення журналів, збору статистики та аналізу результатів у реальному часі. Враховуючи ці потреби, виникає необхідність у створенні спеціалізованих систем, які дозволяють автоматизувати повний цикл тестування розподілених програмних продуктів.

Актуальність теми дипломної роботи полягає саме в тому, що вона орієнтована на створення комп'ютерної системи, яка забезпечує автоматизацію тестування розподілених застосунків із врахуванням вищезазначених викликів. Така система має забезпечувати: зручне визначення сценаріїв тестування, запуск багатьох екземплярів клієнтів або служб, моделювання типових ситуацій роботи мережі, логування помилок, збір метрик продуктивності тощо.

Мета даної роботи – розробити комп'ютерну систему для автоматизації процесу тестування розподілених програмних застосунків, яка дає змогу скоротити час перевірки, покращити якість продукту, зменшити ризики помилок у продакшн-середовищі та підвищити загальну ефективність процесу розробки.

Для досягнення мети передбачається вирішити такі завдання:

- проаналізувати існуючі підходи та інструменти автоматизованого тестування розподілених систем;
- визначити основні вимоги до функціоналу майбутньої системи;
- спроєктувати архітектуру програмного засобу;
- реалізувати функціональні модулі системи;
- провести експериментальне тестування розробленого рішення;
- оцінити ефективність запропонованого підходу.

Об'єктом дослідження є процес автоматизації тестування розподілених програмних застосунків.
Предметом дослідження є методи, засоби та архітектурні рішення, які забезпечують реалізацію такої автоматизації.

Практичне значення роботи полягає у створенні програмного засобу, який може бути використаний у середовищі розробки для перевірки надійності, стабільності та продуктивності розподілених застосунків, зокрема на етапі інтеграційного та навантажувального тестування.

Таким чином, тема дипломної роботи є актуальною, має практичне спрямування та сприяє розвитку методів забезпечення якості програмного забезпечення в умовах зростаючої складності сучасних розподілених систем.

1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Вимоги до комп'ютерної системи автоматизації тестування

Розроблення комп'ютерної системи автоматизації процесу тестування розподілених застосунків потребує чіткого визначення функціональних і нефункціональних вимог, які забезпечать ефективність, масштабованість та надійність системи. Розподілені програмні застосунки характеризуються складною архітектурою, взаємодією між численними компонентами, що виконуються на різних вузлах мережі, та високими вимогами до стабільності комунікації між цими компонентами. Тому система автоматизації тестування повинна підтримувати багатопоточність, паралельне виконання тестів та забезпечувати відстеження результатів у реальному часі.

Функціональні вимоги:

- Підтримка розподіленого середовища тестування з можливістю запуску тестів на декількох вузлах одночасно;
- Автоматичне виявлення помилок, збоїв і відхилень у роботі застосунків;
- Створення, збереження та повторне використання наборів тестів;
- Інтеграція з CI/CD системами (наприклад, Jenkins, GitLab CI) для автоматичного запуску тестів після змін у коді;
- Можливість підключення до зовнішніх API для перевірки коректності обміну даними;
- Генерація звітів про результати тестування з детальною інформацією про успішні та провалені тести;
- Сповіщення користувача про завершення тестування та виявлені помилки через e-mail або месенджери (наприклад, Telegram).

Нефункціональні вимоги:

- Надійність: система має забезпечувати безперебійне виконання тестів навіть при нестабільному мережевому з'єднанні;
- Масштабованість: можливість розширення на більшу кількість вузлів без зміни архітектури;
- Продуктивність: система має швидко виконувати тести при великій кількості запитів і підключень;
- Зручність використання: користувацький інтерфейс має бути інтуїтивно зрозумілим для тестувальника або розробника;
- Безпека: тестова інформація та результати мають зберігатись захищено, особливо у випадках роботи з конфіденційними даними;
- Портативність: можливість запуску системи у різних середовищах (Windows, Linux, Docker-контейнери тощо).

Таким чином, коректне формулювання вимог є основою для побудови якісного технічного завдання та подальшого проектування системи, здатної забезпечити ефективне тестування сучасних розподілених програмних продуктів.

1.2 Огляд сучасних підходів і інструментів для тестування розподілених застосунків

Тестування розподілених застосунків — це складний процес, оскільки такі системи складаються з багатьох взаємодіючих компонентів, які можуть працювати на різних фізичних або віртуальних машинах. Основна мета тестування — перевірити узгодженість, стабільність, продуктивність і коректність взаємодії між компонентами у різних сценаріях навантаження та відмов. Існує декілька підходів до організації тестування таких систем, кожен з яких вирішує певні задачі.

Серед основних підходів до тестування виділяють:

- Інтеграційне тестування — перевірка взаємодії між окремими сервісами або модулями, що входять до складу розподіленої системи.
- Системне тестування — перевірка функціональності всієї системи в цілому.
- Тестування продуктивності (performance testing) — імітація навантаження для оцінки, як система поводиться при великій кількості запитів або користувачів.
- Тестування на відмовостійкість (fault tolerance testing) — перевірка здатності системи продовжити роботу при відмові одного або кількох її компонентів.
- Тестування з використанням моків і емуляторів — дозволяє протестувати компоненти окремо, імітуючи поведінку суміжних частин.

Для реалізації цих підходів використовуються різноманітні інструменти, серед яких найбільш поширені:

- Postman — зручний інструмент для функціонального тестування REST API, дозволяє створювати сценарії, перевіряти відповіді та обробляти

помилки.

- JMeter — інструмент для навантажувального тестування, дозволяє моделювати тисячі одночасних користувачів та вимірювати продуктивність сервісів.
- Selenium — платформа для автоматизації браузерного тестування, що може використовуватись для кінцевого тестування веб-компонентів розподілених систем.
- Docker + TestContainers — забезпечують ізоляцію середовища для виконання тестів, дозволяють розгорнути окремі сервіси у контейнерах.
- Cypress — сучасний фреймворк для енд-ту-енд тестування інтерфейсів, особливо популярний для SPA-застосунків.
- K6 — інструмент з відкритим кодом для написання тестів навантаження на JavaScript, добре підходить для масштабних розподілених сервісів.
- Allure — фреймворк для генерації звітів про результати автоматичних тестів з підтримкою візуалізації.

Крім того, у великих проєктах активно використовуються CI/CD-системи (наприклад, Jenkins, GitLab CI, GitHub Actions), що дозволяють автоматизувати процес запуску тестів після кожної зміни в коді, забезпечуючи оперативне виявлення помилок.

Таким чином, сучасні інструменти забезпечують широкий спектр можливостей для всебічного тестування розподілених застосунків. Їх правильне поєднання дозволяє досягнути високої надійності та якості програмного забезпечення у складних середовищах.

1.3 Аналіз типових архітектур розподілених застосунків

Розподілені застосунки є складними програмними системами, які виконуються на кількох фізичних або віртуальних вузлах і взаємодіють між собою за допомогою мережевих протоколів. Вони широко використовуються у веб-сервісах, хмарних платформах, мікросервісних середовищах, корпоративних інформаційних системах тощо. Надійне та ефективне тестування таких застосунків потребує розуміння їхньої архітектури.

Монолітна vs Розподілена архітектура

Монолітна архітектура передбачає об’єднання всіх компонентів застосунку в один єдиний процес, що ускладнює масштабування та тестування окремих частин системи. Натомість розподілена архітектура розбиває застосунок на незалежні модулі або сервіси, кожен з яких може розгортатися, оновлюватися та тестуватися окремо. Це дає змогу досягати кращої гнучкості, надійності та масштабованості системи.

Типові архітектурні підходи:

1. Клієнт-серверна архітектура

Один із найпростіших і найпоширеніших варіантів. Клієнт (наприклад, браузер або мобільний застосунок) надсилає запити до сервера, який обробляє їх і повертає відповідь. Така архітектура є зручною для функціонального тестування API та веб-інтерфейсів.

2. Тришарова архітектура (Presentation – Logic – Data) Складається з трьох основних шарів: інтерфейс користувача, бізнес-логіка і база даних. Це дозволяє чітко відокремлювати логіку і зосереджуватись на тестуванні кожного шару окремо — зокрема, за допомогою юніт-тестів, інтеграційних тестів та тестування бази даних.

Мікросервісна архітектура

Передбачає розділення застосунку на набір дрібних, незалежних сервісів, що взаємодіють через API. Кожен сервіс має власну логіку, базу даних і цикл розгортання. Тестування такої архітектури вимагає не лише перевірки окремих сервісів, а й забезпечення узгодженості між ними, перевірки API, стійкості до збоїв і продуктивності при навантаженні.

Серверless-архітектура (Function as a Service)
Користувач створює окремі функції, які запускаються на вимогу в хмарному середовищі (AWS Lambda, Azure Functions тощо). Це потребує особливих підходів до тестування, зокрема емуляції середовища виконання, контролю тригерів та моніторингу викликів функцій.

Peer-to-Peer (P2P) архітектура

У такій системі всі вузли є рівноправними й можуть виступати як клієнтами, так і серверами. Це складна для тестування модель, оскільки потрібно забезпечити узгодженість даних між вузлами, безпеку передачі, відстеження стану та контроль над дублюванням інформації.

Загальні виклики при тестуванні розподілених архітектур:

- Різнотипні середовища та сервіси;
- Затримки в мережі або втрати пакетів;
- Проблеми сумісності між сервісами;
- Взаємна залежність компонентів та складність їхньої ізоляції;
- Динамічне масштабування систем.

1.4 Постановка задачі розроблення

Метою даної роботи є розроблення комп'ютерної системи, що дозволяє автоматизувати процес тестування розподілених програмних застосунків. Це передбачає створення такого програмного середовища, яке забезпечить виявлення помилок у функціонуванні окремих компонентів розподіленої системи, перевірку взаємодії між ними, а також аналіз поведінки при високому навантаженні, затримках у мережі чи часткових відмовах.

Розподілені застосунки, на відміну від монолітних, функціонують у середовищі з динамічною топологією, де важливо враховувати фактори, пов'язані з передачею даних, синхронізацією станів, відмовостійкістю та масштабованістю. Вручну протестувати всі сценарії, особливо в умовах зміни конфігурацій або оновлень компонентів, практично неможливо. Тому постає потреба в автоматизованій системі, яка може швидко та ефективно перевіряти різні аспекти роботи застосунку без втручання людини.

У рамках роботи ставляться наступні основні задачі:

- проаналізувати існуючі підходи та засоби автоматизованого тестування розподілених систем;
- визначити вимоги до майбутньої системи (функціональні та нефункціональні);
- спроєктувати архітектуру системи, яка підтримуватиме масштабування, модульність та розширюваність;
- реалізувати основні компоненти системи: модуль запуску тестів, засоби логування, аналізу результатів і формування звітів;
- забезпечити взаємодію з CI/CD-платформами для автоматичного запуску тестів після внесення змін у код;
- протестувати створену систему на прикладах реальних розподілених застосунків або емуляторах;

- оцінити ефективність системи та зробити висновки щодо подальшого розвитку.

Таким чином, результатом розробки має стати програмна система, яка здатна автоматично виконувати комплексне тестування розподіленого застосунку, включаючи тестування функціональності, навантаження та відмовостійкості, а також формувати звітність і підтримувати інтеграцію з сучасними засобами розробки.

2.ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Архітектура комп’ютерної системи автоматизації тестування

Архітектура комп’ютерної системи автоматизації тестування розподілених застосунків повинна відповідати сучасним вимогам до надійності, масштабованості та адаптивності. Така система має забезпечувати виконання автоматизованих тестів, моніторинг, зберігання результатів тестування, інтеграцію з іншими сервісами та зручний інтерфейс для користувача.

Система будується за модульним принципом, що дозволяє розділити її на незалежні функціональні компоненти, кожен з яких виконує окреме завдання. Це полегшує обслуговування, масштабування та оновлення програмного комплексу.

Основні компоненти архітектури:

1. Клієнтський інтерфейс (Front-end):

- Призначений для взаємодії користувача з системою.
- Дозволяє запускати тести, переглядати звіти, налаштовувати параметри тестування.
- Може бути реалізований у вигляді веб-додатку з використанням HTML/CSS/JavaScript або фреймворків React/Vue.

2. Сервер додатка (Back-end):

- Обробляє запити з клієнтського інтерфейсу, координує роботу інших компонентів.
- Реалізує бізнес-логіку системи.
- Забезпечує взаємодію з базою даних, тестовими середовищами та зовнішніми API.

- Може бути реалізований на Python (FastAPI, Flask), Node.js або Java Spring Boot.

3. Модуль тестування:

- Безпосередньо відповідає за запуск тестових сценаріїв.
- Підтримує різні типи тестів: юніт-тести, інтеграційні, навантажувальні, енд-ту-енд.
- Дозволяє запуск тестів у розподіленому середовищі (через Docker, Kubernetes або хмарні сервіси).

4. Менеджер конфігурацій та тестових сценаріїв:

- Забезпечує зберігання, структурування та версіонування тестів.
- Дозволяє підключення до зовнішніх репозиторіїв (GitHub, GitLab).
- Підтримує параметризацію тестів.

5. База даних:

- Містить результати тестувань, журнали подій, інформацію про користувачів і конфігурації.
- Вибір між реляційною (PostgreSQL, MySQL) або нереляційною (MongoDB) СУБД залежить від вимог до структури даних.

6. Система логування та моніторингу:

- Здійснює збір логів усіх компонентів.
- Дозволяє виявляти помилки та стежити за стабільністю системи.
- Може використовуватись стек ELK (Elasticsearch, Logstash, Kibana) або Prometheus + Grafana.

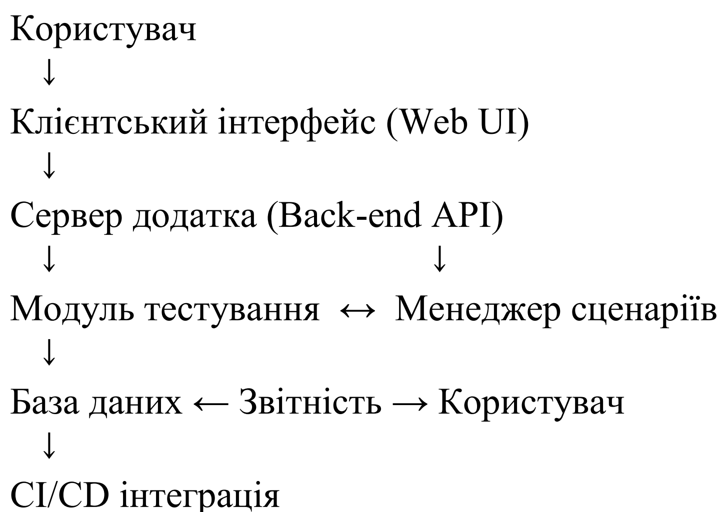
7. Модуль генерації звітів:

- Автоматично формує звіти за результатами тестів у зручному вигляді (PDF, HTML, JSON).
- Може включати візуалізацію результатів: графіки, таблиці, діаграми.

8 .CI/CD-інтеграція:

- Підключення до систем безперервної інтеграції/розгортання (GitHub Actions, Jenkins, GitLab CI).
- Автоматичний запуск тестів після кожного оновлення коду.

Схема взаємодії компонентів



Архітектурний стиль

Для реалізації системи доцільно обрати модульну мікросервісну архітектуру, що дозволяє:

- оновлювати окремі компоненти без зупинки всієї системи;
- запускати окремі сервіси на різних вузлах у розподіленому середовищі;
- підвищити надійність і масштабованість.

2.1.1 Вибір моделі тестування розподілених застосунків

Розподілені застосунки мають складну багаторівневу структуру, що включає кілька взаємопов'язаних компонентів, які можуть функціонувати на різних вузлах мережі. Тестування таких систем вимагає спеціального підходу, що враховує розподіленість, асинхронну взаємодію, можливість часткових відмов, змінну пропускну здатність каналів зв'язку тощо. Тому вибір моделі тестування є критично важливим етапом у проєктуванні системи автоматизації.

Для проєктованої системи доцільно застосовувати комбіновану модель тестування, що включає кілька рівнів:

1. **Юніт-тестування (Unit Testing)**

Забезпечує перевірку окремих модулів або функцій застосунку на коректність логіки. Це найнижчий рівень тестування, який легко автоматизується. Дає змогу виявляти помилки на ранньому етапі розробки.

2. **Інтеграційне тестування (Integration Testing)**

Перевіряє взаємодію між окремими модулями або сервісами, що працюють разом. У розподілених системах особливу увагу приділяють:

- взаємодії API між мікросервісами;
- обміну даними через брокери повідомлень (наприклад, RabbitMQ, Kafka);
- узгодженості даних між базами.

3. **Системне тестування (System Testing)**

Оцінює поведінку всієї системи як єдиного цілого. Перевіряються основні бізнес-сценарії, стабільність, безпека, обробка помилок, граничні умови.

4. **Енд-ту-енд тестування (End-to-End Testing)**

Виконується з точки зору кінцевого користувача — імітує повний цикл взаємодії з системою через інтерфейс. Включає тестування веб-інтерфейсу, API, бази даних. Забезпечує найвищу впевненість у працездатності системи в реальних умовах.

5. **Навантажувальне та стрес-тестування (Load & Stress Testing)**

Дає змогу перевірити, як система поводиться під великим навантаженням, при пікових запитах або в умовах обмежених ресурсів. У розподілених застосунках це особливо важливо для виявлення "вузьких місць".

6. **Тестування на стійкість (Resilience Testing)**

Моделює збої в мережі, відмову окремих компонентів, втрату з'єднання тощо, з метою перевірити, як система відновлює працездатність і зберігає дані.

2.1.2 Мережева структура і середовище виконання тестів

Тестування розподілених застосунків неможливе без належно організованого мережевого середовища, яке імітує або відображає реальні умови функціонування системи. Відповідна мережева структура дозволяє оцінити поведінку компонентів у розподіленому середовищі, протестувати їхню взаємодію, затримки, пропускну здатність, відмовостійкість тощо.

Мережева структура системи автоматизації тестування

Проєктована система передбачає використання типового середовища для розподілених обчислень, яке включає:

- Клієнтські машини (тестові вузли) – пристрої, які запускають розподілені модулі застосунку та/або автоматизовані тести;
- Сервер контролю тестування – центральний компонент системи, що координує процеси запуску тестів, збору результатів, моніторингу та звітності;
- Сервер бази даних – зберігає результати тестів, журнали подій, конфігураційні параметри;
- CI/CD-сервер – відповідальний за автоматичний запуск тестів при змінах у репозиторії проєкту;
- Зовнішні інтерфейси/API – інтеграція з іншими сервісами, хмарними платформами або емуляторами.

Мережа побудована на основі TCP/IP-протоколів, із застосуванням VPN або внутрішньої підмережі, що забезпечує захист переданих даних. Можливе також використання контейнеризації (Docker) і віртуалізації (наприклад, за допомогою VMware або VirtualBox) для ізоляції тестових середовищ.

Середовище виконання тестів

Для виконання тестів обрано сучасне та гнучке середовище, що відповідає вимогам масштабованості, автоматизації та зручності:

- Операційна система: Ubuntu Server або інші дистрибутиви Linux, що легко автоматизуються і підтримують Docker/Kubernetes.
- Контейнеризація: використання Docker дозволяє швидко розгортати окремі частини застосунку і тестового середовища в ізольованих контейнерах. Це дає змогу створити однакові умови для всіх тестів незалежно від апаратного забезпечення.
- Система оркестрації: Kubernetes (або його спрощені версії, як-от Minikube чи k3s) використовується для управління розгортанням контейнерів і балансуванням навантаження.
- Системи CI/CD: інтеграція з Jenkins, GitHub Actions або GitLab CI дозволяє автоматизувати процес запуску тестів при кожному оновленні коду.
- Тестові фреймворки:
- для юніт-тестів — PyTest, JUnit, Mocha тощо;
- для інтеграційного і системного тестування — Postman/Newman, TestNG, REST Assured;
- для енд-ту-енд тестування — Selenium, Cypress або Playwright;
- для навантажувального тестування — JMeter, Locust, Gatling.

Імітація розподіленого середовища

Оскільки справжня розподіленість важлива для достовірності тестів, система повинна підтримувати:

- запуск тестів на різних вузлах (реальні або віртуальні машини);
- затримки мережі (імітація за допомогою tc/netem);
- відмови сервісів (штучна деградація, симуляція збоїв).

2.1.3 Обґрунтування вибору інструменту Jenkins

У процесі розроблення комп’ютерної системи автоматизації тестування розподілених застосунків ключовим завданням є забезпечення безперервного та надійного виконання тестів після кожної зміни в коді. Це передбачає інтеграцію з системами контролю версій, запуск тестів у середовищі, наближеному до продуктивного, зберігання результатів, сповіщення розробників та аналіз виконання. У цьому контексті Jenkins є оптимальним вибором як система безперервної інтеграції та доставки (CI/CD).

Jenkins — це популярний інструмент з відкритим кодом, що надає широкі можливості автоматизації процесів побудови, тестування та деплою програмного забезпечення. Його архітектура дозволяє гнучко налаштовувати пайплайни, інтегрувати зовнішні інструменти, запускати процеси на віддалених вузлах, а також масштабувати систему відповідно до зростаючих вимог.

Переваги Jenkins у проєктованій системі:

1. Автоматизація тестування

Jenkins дозволяє автоматизовано запускати юніт-, інтеграційні та енд-ту-енд тести щоразу, коли відбуваються зміни в репозиторії застосунку. Це підвищує оперативність виявлення помилок та дозволяє реагувати на них ще на етапі розробки.

2. Інтеграція з Git

Jenkins підтримує інтеграцію з GitHub, GitLab та іншими системами контролю версій, що дозволяє запускати пайплайни після кожного коміту або pull-запиту.

Гнучка конфігурація через Pipeline

Використання Jenkins Pipeline (написаних мовою Groovy або через декларативний синтаксис) дозволяє створити складні сценарії тестування, враховуючи специфіку розподілених застосунків.

Підтримка Docker і розподіленого Pipeline

Jenkins легко інтегрується з Docker, що дозволяє запускати тести в ізольованих контейнерах, імітуючи роботу в реальному середовищі. Крім того, Jenkins Master може керувати кількома агентами (worker nodes), розподіляючи навантаження між ними.

Масштабованість та спільнота

Завдяки великій кількості плагінів Jenkins підтримує роботу з такими інструментами, як Postman (через Newman), JMeter, Allure, Slack, Email тощо. Це дає змогу розширити функціональність системи без потреби створення власних рішень.

Звіти та аналітика

Jenkins надає зручний веб-інтерфейс для перегляду статусів виконання тестів, логів, історії змін та метрик. Це спрощує аналіз результатів тестування та підвищує прозорість процесів.

Jenkins дозволяє організувати повністю автоматизовану систему тестування розподілених застосунків, яка інтегрується в сучасний DevOps-процес. Його використання забезпечує оперативне виявлення помилок, зменшує людський фактор, дозволяє швидко реагувати на зміни та сприяє створенню надійного програмного забезпечення.

2.2 Програмна частина системи

2.2.1 Мова програмування Python / JavaScript — вибір середовища

У рамках реалізації комп'ютерної системи автоматизації процесу тестування розподілених застосунків обрана мова програмування Python. Вибір цієї мови обумовлений низкою технічних і практичних переваг, що дозволяють ефективно реалізувати необхідну функціональність:

- простота та виразність синтаксису;
- велика кількість бібліотек для тестування (pytest, unittest), роботи з мережею (requests), автоматизації (subprocess, os, docker-py);
- підтримка API-клієнтів для взаємодії з Jenkins, Git, Docker;
- широке ком'юніті та документація.

Середовище розробки

Для написання тестових сценаріїв та допоміжних скриптів використовується середовище PyCharm Community Edition, оскільки воно підтримує:

- інтеграцію з Git;
- зручну роботу з віртуальними середовищами (venv, pipenv);
- запуск юніт-тестів із GUI;
- автодоповнення та дебагер.

Також для легких скриптів використовується VS Code, особливо при роботі в контейнерах Docker.

Структура проєкту

```
automation/
├── jenkins/
│   └── Jenkinsfile        # Опис пайплайну
├── tests/
│   ├── test_api.py        # Тестування REST API
│   └── test_db.py         # Тестування БД
├── utils/
│   └── deploy.py          # Скрипт автоматичного деплою
├── requirements.txt       # Залежності
└── run_tests.py           # Основний запуск тестів
```

Використання мови Python дозволяє створити зрозумілу, гнучку та масштабовану систему автоматизованого тестування. Завдяки бібліотекам для роботи з API, тестуванням, інструментами автоматизації та хорошою інтеграцією з Jenkins — Python став логічним вибором у реалізації даного проєкту.

2.2.2 Розроблення тестових сценаріїв для розподілених застосунків

У процесі тестування розподілених застосунків надзвичайно важливо перевірити не лише правильність обчислень, але й взаємодію між окремими сервісами, відповідність мережевих відповідей очікуваним результатам, коректну обробку збоїв та відмовостійкість.

У даному проєкті для автоматизованого тестування розроблені сценарії, які охоплюють:

- тестування REST API запитів;
- перевірку доступності сервісів;
- перевірку консистентності даних при роботі з кількома вузлами;
- тестування затримок та таймаутів;
- поведінку при збої одного з мікросервісів (за допомогою Docker)

Розглянемо декілька прикладів:

Приклад 1: Тест доступності сервісу

Лічтинг 2.2.1

```
# tests/test_service_availability.py
import requests
def test_main_service_up():
    url = "http://main-service:5000/health"
    response = requests.get(url)
    assert response.status_code == 200
    assert response.json().get("status") == "ok"
```

Приклад 2: Тест відповідності даних між сервісами

Лістинг № 2.2.2

```
# tests/test_data_consistency.py
import requests

def test_data_consistency_between_services():
    user_service = requests.get("http://user-
service:5001/user/1").json()
    billing_service = requests.get("http://billing-
service:5002/user/1").json()

    assert user_service["id"] == billing_service["userId"]
    assert user_service["active"] == True
```

Приклад 3: Тест затримки та повторного запиту

Лічтинг № 2.2.3

```
# tests/test_timeout_handling.py
import requests
import time

def test_slow_service_retry():
    start = time.time()
    response = requests.get("http://slow-
service:5003/data", timeout=10)
    end = time.time()
```

```
assert response.status_code == 200
    assert (end - start) < 10  # Перевірка, що запит не завис
```

Переваги такого підходу:

- тести легко масштабуються на нові вузли;
- вони є незалежними та можуть запускатися паралельно;
- можна емулювати неполадки (відсутність відповіді, таймаути, некоректні дані) для покриття граничних ситуацій;
- їх легко інтегрувати в CI/CD пайплайн через Jenkins.

Розроблені тестові сценарії дозволяють автоматично перевіряти як загальну працездатність розподіленої системи, так і коректність логіки міжвузлової взаємодії. У поєднанні з Jenkins вони забезпечують основу для впровадження безперервного тестування (Continuous Testing) у рамках розроблення та підтримки застосунку.

2.2.3 Інтеграція Jenkins із Python-сценаріями

Для реалізації повноцінного процесу автоматизації тестування розподілених застосунків потрібно забезпечити ефективну інтеграцію між системою Jenkins та тестовими скриптами, написаними мовою Python. Завдяки гнучкій архітектурі Jenkins та можливостям запуску shell-команд, така інтеграція є досить простою і зручною в налаштуванні.

Загальний принцип роботи

- Jenkins слідкує за змінами в репозиторії з тестами.
- Після коміту або за розкладом він ініціює пайплайн.
- У рамках пайплайну виконується встановлення залежностей (через pip).
- Далі запускаються Python-тести (pytest) з генерацією звітів.
- Результати тестів збираються та виводяться в інтерфейсі Jenkins.

Приклад Jenkinsfile

Лістинг № 2.2.4

```
pipeline {
    agent any

    environment {
        // Назва віртуального середовища Python
        VENV_DIR = "venv"
    }

    stages {
        stage('Отримання коду') {
            steps {
                // Клонування репозиторію з тестами
                git 'https://github.com/user/distributed-app-tests.git'
            }
        }

        stage('Налаштування середовища') {
            steps {
                // Створення віртуального середовища
                sh 'python3 -m venv $VENV_DIR'
                // Оновлення pip
                sh './$VENV_DIR/bin/pip install --upgrade pip'
                // Встановлення залежностей з requirements.txt
                sh './$VENV_DIR/bin/pip install -r requirements.txt'
            }
        }

        stage('Запуск тестів') {
            steps {
                // Запуск тестів з pytest і формування звіту в JUnit-формат
```

```
sh './$VENV_DIR/bin/pytest tests/ --junitxml=results.xml'
                }
            }

            stage('Публікація результатів') {
                steps {
                    // Вивід результатів у вигляді звіту в
Jenkins
                    junit 'results.xml'
                }
            }
        }

        post {
            always {
                // Додаткові дії після виконання всіх стадій
                echo 'Тестування завершено. Очищення
середовища.'
            }
        }
    }
```

3. Реалізація системи

3.1 Побудова Jenkins-пайплайну

Побудова Jenkins-пайплайну є ключовим етапом у процесі автоматизації тестування розподілених програмних застосунків. Jenkins забезпечує автоматичне виконання різних етапів CI/CD (безперервної інтеграції та доставки), включаючи отримання коду, підготовку середовища, запуск тестів і формування звітів. Для Python-проєкту пайплайн будується з урахуванням специфіки створення віртуального середовища, встановлення залежностей і запуску тестів засобами pytest.

У межах дипломної роботи використовується declarative pipeline, оскільки він є більш структурованим та рекомендованим для більшості Jenkins-проєктів. Усі налаштування виконуються у файлі Jenkinsfile, який зберігається у кореневій директорії Git-репозиторію.

Пайплайн включає такі основні етапи:

- клонування репозиторію з тестовими скриптами;
- створення Python-віртуального середовища;
- встановлення необхідних залежностей із файлу requirements.txt;
- запуск тестів за допомогою pytest;
- генерація звітів у форматі JUnit;
- публікація результатів у графічному інтерфейсі Jenkins.

Нижче наведено приклад конфігурації Jenkinsfile для реалізації цього пайплайну:

Лістинг № 3.1

```
pipeline {
    agent any

    environment {
        VENV_DIR = "venv"
    }

    stages {
        stage('Отримання коду') {
            steps {
                git  'https://github.com/user/distributed-app-tests.git'
            }
        }

        stage('Налаштування середовища') {
            steps {
                sh 'python3 -m venv $VENV_DIR'
                sh './$VENV_DIR/bin/pip install --upgrade pip'
                sh './$VENV_DIR/bin/pip install -r requirements.txt'
            }
        }

        stage('Запуск тестів') {
            steps {
                sh './$VENV_DIR/bin/pytest tests/ --junitxml=results.xml'
            }
        }

        stage('Публікація результатів') {
```

```
            steps {
                junit 'results.xml'
            }
        }
    }

    post {
        always {
            echo 'Тестування завершено. Очищення середовища.'
        }
    }
}
```

Цей скрипт дозволяє автоматично перевіряти якість розподілених застосунків після кожного оновлення коду. В результаті використання Jenkins досягається висока надійність, повторюваність і зручність тестування, що особливо важливо в умовах розподіленої архітектури.

3.1.1 Створення тестів і структура каталогу

На етапі створення Jenkins-пайплайну важливим є не лише його конфігурація, але й підготовка проєкту, який цей пайплайн обслуговуватиме. Структура каталогу тестового проєкту повинна бути логічною та зручною для масштабування. Крім того, тести мають бути написані у спосіб, що дозволяє легко запускати їх автоматично у рамках пайплайну Jenkins.

У межах проєкту для тестування розподілених застосунків використовуються тести, написані на мові програмування Python з використанням фреймворку pytest. Основна увага приділяється тестуванню REST API застосунку, яке зазвичай є критичним компонентом у розподіленій архітектурі.

Структура каталогу проєкту

distributed-app-tests/

├── Jenkinsfile

```
├── requirements.txt
├── tests/
│   ├── test_healthcheck.py
│   ├── test_api_status.py
│   └── __init__.py
```

- Jenkinsfile – опис етапів автоматичного виконання (побудова, тестування, публікація результатів).
- requirements.txt – перелік бібліотек, необхідних для запуску тестів.
- tests/ – директорія з усіма тестовими модулями.
- init__.py – дозволяє Python розпізнавати папку як модуль.

Приклад тесту:

Лістинг № 3.2

```
import requests

def test_healthcheck():
    response =
requests.get("http://localhost:5000/api/health")
    assert response.status_code == 200
```

Цей тест виконує базову перевірку доступності API сервісу. Якщо застосунок успішно працює, запит повинен повернути HTTP статус 200 OK. У випадку невдачі пайплайн відобразить тест як провалений.

Генерація звіту у форматі, що підтримується Jenkins:

Лістинг № 3.3

```
pytest tests/ --junitxml=results.xml
```

Таким чином, створення правильної структури тестового проєкту та написання якісних тестів на Python дозволяє повністю автоматизувати перевірку працездатності розподіленого застосунку за допомогою Jenkins.

3.1.2 Інтеграція Jenkins із Git-репозиторієм

Для повноцінної автоматизації процесу тестування розподілених застосунків важливо налаштувати безперервну інтеграцію Jenkins із системою керування версіями — зокрема, з Git. Це дозволяє кожного разу після оновлення коду (наприклад, пушу в репозиторій) автоматично запускати перевірку тестів, що забезпечує оперативне виявлення помилок.

Підключення до репозиторію Git

У рамках реалізації даного проєкту використовується GitHub як віддалене сховище. У Jenkins підключення до Git реалізується за допомогою плагіна Git plugin, який зазвичай входить до базового пакету.

У пайплайні описано клонування репозиторію:

Лістинг № 3.4

```
stage('Clone repository') {
    steps {
        git 'https://github.com/user/distributed-app-tests.git'
    }
}
```

Цей блок у Jenkinsfile виконує команду клонування репозиторію перед подальшими діями.

Автоматичний запуск після комітів

Щоб Jenkins запускав перевірку автоматично після кожного коміту, потрібно:

1. Налаштувати GitHub Webhook на адресу відповідну адресу
2. У Job-налаштуваннях Jenkins активувати опцію:

"Build when a change is pushed to GitHub"

Приклад конфігурації в Jenkins:

- Проєкт типу: *Pipeline*.
- SCM: *Git*.
- Jenkinsfile: зберігається в корені репозиторію.
- Trigger: *GitHub hook trigger for GITScm polling*.

Таким чином, інтеграція Jenkins із Git-репозиторієм забезпечує перехід до безперервної інтеграції (CI), де кожна зміна в коді автоматично проходить перевірку на коректність, що значно підвищує якість та стабільність розподілених застосунків.

3.1.3 Налаштування середовища виконання Python-скриптів у Jenkins

Оскільки тести автоматизації у проєкті реалізовані мовою Python, необхідно забезпечити правильне налаштування середовища для їх виконання на машині Jenkins. Створення ізольованого віртуального середовища гарантує, що залежності будуть керованими, а тести — відтворюваними на будь-якій машині.

Створення віртуального середовища

На етапі пайплайну виконується створення окремого Python-середовища за допомогою стандартного модуля venv. Це дозволяє уникнути конфліктів між різними проєктами на одній CI-машині.

Фрагмент з Jenkinsfile:

Лістинг № 3.1.1

```
stage('Set up Python environment') {
    steps {
        sh 'python3 -m venv venv'
        sh './venv/bin/pip install --upgrade pip'
        sh './venv/bin/pip install -r requirements.txt'
    }
}
```

python3 -m venv venv — створює віртуальне середовище в каталозі venv.

pip install -r requirements.txt — встановлює всі бібліотеки, необхідні для тестування (наприклад, pytest, requests).

requirements.txt – файл, що містить перелік залежностей які мають бути встановлені.

Його слід розмістити в корені, щоб Jenkins мав доступ до нього при створені середовища.

Застосування середовища

Після встановлення залежностей, тести запускаються у межах створеного середовища:

```
stage('Run tests') {
    steps {
        sh './venv/bin/pytest tests/ --junitxml=results.xml'
    }
}
```

Таким чином, незалежно від операційної системи або конфігурації Jenkins-агента, скрипти виконуються у точно визначених умовах, що забезпечує стабільність і передбачуваність виконання.

Вимоги до Jenkins-агента

Щоб виконання Python-скриптів пройшло успішно, Jenkins-агент повинен мати:

- Встановлений Python 3.x;
- Доступ до bash (або shell-оточення);
- Доступ до Internet (для завантаження пакетів з PyPI, якщо не кешуються локально).

Завдяки правильно налаштованому середовищу виконання Jenkins-пайплайн працює стабільно, забезпечуючи ізольованість та контрольованість процесу тестування розподілених застосунків.

3.1.4 Генерація та публікація звітів про виконання тестів у Jenkins

Завершальним етапом автоматизованого тестування є формування та публікація звітів про результати запуску тестів. Це дозволяє розробникам і тестувальникам швидко отримувати зворотний зв'язок щодо якості коду. Jenkins має вбудовані інструменти для зчитування результатів тестування та формування звітів у зручному форматі.

Формат звітів: JUnit XML

Для інтеграції з Jenkins, результати тестів у Python (з використанням

pytest) мають бути збережені у форматі JUnit XML. Це стандартний формат, який підтримується багатьма CI/CD системами.

Запуск тестів у пайплайні з генерацією звіту:

Лістинг № 3.1.2

```
stage('Run tests') {
    steps {
        sh './venv/bin/pytest tests/ --junitxml=results.xml'
    }
}
```

Після виконання тестів утворюється файл results.xml, який містить структуровану інформацію про:

- кількість тестів;
- які з них пройдено успішно;
- які завершилися помилкою чи були пропущені.

Публікація звіту в Jenkins

Щоб Jenkins зчитував файл звіту, використовується спеціальний блок junit:

Лістинг 3.1.3

```
stage('Publish results') {
    steps {
        junit 'results.xml'
    }
}
```

Якщо тести не проходять, Jenkins автоматично позначає збірку як нестабільну або помилкову, залежно від налаштувань.

Відображення результатів

Після запуску пайплайну в інтерфейсі Jenkins у розділі Test Result з'являється детальна статистика:

- графік проходження тестів у часі;
- список упавших або нестабільних тестів;
- час виконання кожного тесту.

Це дозволяє:

- відстежувати регресії;
- вчасно реагувати на зміни в поведінці системи;
- бачити загальну картину стабільності коду.

Завдяки автоматичній генерації та публікації звітів у пайплайні, розробники отримують оперативний зворотний зв'язок, що значно підвищує якість розробки та спрощує процес підтримки розподілених застосунків.

3.2 Тестування застосунку у середовищі Jenkins

3.2.1 Розгортання існуючого проєкту в Jenkins

Для забезпечення безперервного тестування розподіленого застосунку, що вже розроблений, першим кроком є його інтеграція з Jenkins. У цьому підрозділі описується підключення існуючого проєкту до Jenkins, налаштування параметрів середовища виконання та запуску пайплайну, який автоматизує процес тестування.

Інтеграція з репозиторієм

Проєкт зберігається у віддаленому Git-репозиторії, наприклад, на GitHub. У Jenkins створюється нова задача типу Pipeline, де у конфігурації вказується шлях до репозиторію.

Структура проєкту

Репозиторій містить наступні основні каталоги:

```
├── src/              # Основний код застосунку
├── tests/            # Модулі з pytest-тестами
├── requirements.txt  # Залежності Python
├── Jenkinsfile       # Сценарій пайплайну
```

Це дозволяє Jenkins легко знайти як код, так і тести для їх подальшого виконання.

Файл Jenkinsfile вже розміщено в корені репозиторію. Він містить усі необхідні етапи: клонування, створення віртуального середовища, встановлення залежностей, запуск тестів і публікацію результатів.

Лістинг 3.2.1

```
pipeline {
    agent any
    environment {
        VENV_DIR = "venv"
    }
    stages {
        stage('Checkout') {
```

```
                    steps {
                        git 'https://github.com/user/distributed-app-tests.git'
      }
                }
                stage('Setup Environment') {
                    steps {
                        sh 'python3 -m venv $VENV_DIR'
                        sh './$VENV_DIR/bin/pip install -r requirements.txt'
                    }
                }
                stage('Run Tests') {
                    steps {
                        sh './$VENV_DIR/bin/pytest tests/ --junitxml=results.xml'
                    }
                }
                stage('Publish Results') {
                    steps {
                        junit 'results.xml'
                    }
                }
            }
        }
```

Результати запуску

Після запуску пайплайну Jenkins:

- завантажує код;
- готує середовище;
- виконує тести;
- формує звіт.

Якщо тестування пройшло успішно, збірка відмічається зеленим статусом (SUCCESS), у разі помилок — відповідно UNSTABLE або FAILURE. Всі результати відображаються в інтерфейсі Jenkins.

3.2.2 Аналіз результатів тестування та обробка помилок

Після успішного запуску Jenkins-пайплайну важливо не лише отримати результати тестування, а й правильно їх інтерпретувати та вживати відповідних заходів у разі виявлення помилок. У цьому підрозділі описується, як Jenkins обробляє результати тестів, які засоби аналізу використовуються, і як організована реакція на виявлені збої.

Генерація звітів

Система автоматично формує звіт у форматі JUnit XML за допомогою команди:

Лістинг 3.2.2

```
./venv/bin/pytest tests/ --junitxml=results.xml
```

Цей файл з результатами підхоплюється Jenkins-ом на етапі Publish Results, де використовується плагін JUnit, який будує наочну візуалізацію проходження тестів:

- загальна кількість тестів;
- кількість успішних/провалених/пропущених;
- час виконання;
- деталі по кожному тесту.

Обробка помилок

У випадку, якщо деякі тести завершуються з помилками або фейлами, Jenkins:

- помічає збірку як UNSTABLE або FAILURE;
- підсвічує неуспішні тести червоним кольором;
- дозволяє перейти до перегляду логів конкретного етапу;
- надсилає повідомлення (опційно — на e-mail, у Telegram чи Slack).

Це дозволяє одразу побачити, що саме пішло не так, і за необхідності виправити код, створити issue або перезапустити тестування.

Автоматичне блокування наступних етапів

У складніших CI/CD-процесах можна додатково налаштувати Jenkins так, щоб у разі неуспішного тестування відбувалась зупинка тесту.

4. Реалізація комп’ютерної системи

4.1 Створення репозиторію та структури

Для початку створено репозиторій у якому зберігатимуться всі компоненти системи — код застосунку, тести, конфігурації Jenkins, Docker-файли, документація.

Для зберігання проєкту використано систему контролю версій Git та платформу GitHub. Це забезпечує:

- централізоване зберігання коду;
- автоматичне оновлення Jenkins-пайплайну після пушу;
- зручне ведення історії змін.

Лістинг 4.1

```
# Ініціалізація репозиторію
git init

# Створення нового репозиторію на GitHub:
# https://github.com/your-username/distributed-app-tests.git

# Підключення remote-репозиторію
git remote add origin https://github.com/your-username/distributed-app-tests.git

# Перший коміт
git add .
git commit -m "Initial project structure"
git push -u origin main
```

Базова структура проєкту

```
distributed-app-tests/
│
├── app/                 # Код застосунку
│   ├── __init__.py
│   └── main.py
│
```

```
├── tests/                # Автоматизовані тести
│   ├── __init__.py
│   └── test_main.py
│
├── requirements.txt      # Залежності Python
├── Dockerfile            # Контейнеризація середовища
├── Jenkinsfile           # Опис пайплайну для Jenkins
├── README.md             # Документація проєкту
└── results.xml           # Результати тестування (генерується)
```

4.2 Реалізація розподіленого застосунку

Буде реалізована простий розподілений застосунок, який складається з клієнтської та серверної частин. Така архітектура дозволяє моделювати типові сценарії взаємодії між вузлами в розподілених системах та забезпечує можливість автоматичного тестування їх взаємодії.

Розподілений застосунок реалізує наступну функціональність:

- клієнт надсилає запит з деякими даними (наприклад, числом);
- сервер приймає запит, обробляє його та надсилає відповідь (наприклад, подвоєне число);
- клієнт отримує відповідь та виводить результат.

Цей підхід демонструє базовий принцип RPC (Remote Procedure Call) у розподілених системах.

Структура застосунку

```
app/
├── __init__.py
├── server.py
└── client.py
```

Почнемо з серверної частини

Лістинг 4.2.1

```
from flask import Flask, request, jsonify

app = Flask(__name__)
```

```
@app.route('/process', methods=['POST'])
def process_data():
    data = request.json
    number = data.get("number", 0)
    result = number * 2
    return jsonify({"result": result})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Цей код створює REST-сервер на базі Flask, який приймає JSON-запити з числом, обробляє їх (множить на 2) і повертає результат.

Клієнтська частина

Лістинг 4.2.2

```
import requests

def send_request(number):
    url = 'http://localhost:5000/process'
    response = requests.post(url, json={'number': number})
    return response.json()

if __name__ == '__main__':
    result = send_request(10)
    print("Отримано результат:", result['result'])
```

Клієнт надсилає запит на сервер і виводить отриману відповідь у консоль.

Цей застосунок є базовою розподіленою взаємодією, яка легко тестується автоматично. У подальшому буде реалізовано тести та налаштовано Jenkins для автоматичного запуску перевірок після кожної зміни коду.

4.3 Розробка автоматизованих тестів для застосунку

Для забезпечення якості та стабільності розподіленого застосунку розробляються автоматизовані модульні тести. Вони дозволяють:

- перевірити правильність обробки даних на сервері;
- виявити помилки під час змін у коді;
- автоматично запускати перевірки у Jenkins.

У цьому проєкті використовуються бібліотеки pytest та requests.

Розглянемо декілька автоматизиваних тестів:

Лістинг 4.3.1

```
def test_process_data_success():
    url = 'http://localhost:5000/process'
    data = {'number': 7}
    response = requests.post(url, json=data)
    assert response.status_code == 200
    result = response.json().get("result")
    assert result == 14
```

Перевіряє чи правельно оброблено число 7, очікується результат 14.

Лістинг 4.3.2

```
def test_process_data_zero():
    url = 'http://localhost:5000/process'
    data = {'number': 0}
    response = requests.post(url, json=data)
    assert response.status_code == 200
    result = response.json().get("result")
    assert result == 0
```

Перевіряє чи правильно оброблено число 0, тобото якщо число яке було надіслано 0 то і у відровіть має отримати 0.

Лістинг 4.3.3

```
def test_process_data_missing_field():
    url = 'http://localhost:5000/process'
```

```
data = {}  # Поле number відсутнє
    response = requests.post(url, json=data)
    assert response.status_code == 200
    result = response.json().get("result")
    assert result == 0  # Сервер обробляє відсутнє значення
як 0
```

Перевіряє чи правильно обробляється, якщо поле “number” не було надіслано, у такому випадку сервер обробить як 0 і поверне у відповідь 0.

Перейдемо до ручного тестування після запуску сервера

Лістинг 4.3.4

```
pytest tests/
Отримаємо такий результат тесту
================ test session starts ================
collected 3 items

tests/test_main.py ...                              [100%]

================= 3 passed in 0.42s =================
```

Ці тести розроблено з урахуванням базової функціональності застосунку. У подальшому вони інтегруються в Jenkins-пайплайн, що дозволить проводити тестування автоматично після кожної зміни в коді.

4.4 Налаштування Jenkins для автоматизації тестування

З метою забезпечення безперервного та надійного процесу тестування розробленого розподіленого застосунку було реалізовано автоматизацію за допомогою Jenkins — популярного інструмента для CI/CD. Завдяки своїй модульності, сумісності з Python, а також підтримці великої кількості плагінів Jenkins є ефективним рішенням для створення тестового конвеєра, який реагує на зміни в репозиторії.

Встановлення Jenkins та початкове налаштування

На першому етапі Jenkins було встановлено на локальну машину розробника. Встановлення здійснено через пакетний менеджер відповідної операційної системи. Після запуску Jenkins було здійснено початкову конфігурацію, зокрема створення облікового запису адміністратора та встановлення рекомендованих плагінів, таких як:

- Pipeline — для опису сценаріїв тестування в декларативному форматі;
- Git — для інтеграції з Git-репозиторієм проєкту;
- JUnit — для публікації результатів тестування у зручному вигляді;
- Warnings Next Generation — для аналізу якості коду та попереджень.

Створення Jenkins pipeline job

У наступному кроці було створено нову задачу типу Pipeline. Ця задача виконує скрипт, розміщений у файлі Jenkinsfile всередині репозиторію застосунку. Основна структура пайплайну:

1. Клонування репозиторію.
2. Налаштування Python-середовища.
3. Встановлення залежностей.

4. Запуск тестів з використанням pytest.

5. Публікація результатів.

Jenkinsfile

Лістинг 4.4.1

Нижче наведено зміст Jenkinsfile, який реалізує зазначену автоматизацію:

```
pipeline {
    agent any
    environment {
        VENV_DIR = 'venv'
    }

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/user/distributed-app-tests.git'
            }
        }

        stage('Setup Python Environment') {
            steps {
                sh 'python3 -m venv $VENV_DIR'
```

```
        sh './$VENV_DIR/bin/pip install --upgrade pip'
                    sh    './$VENV_DIR/bin/pip    install    -r
requirements.txt'
                }
            }

            stage('Run Tests') {
                steps {
                    sh   './$VENV_DIR/bin/pytest   tests/   --
junitxml=results.xml'
                }
            }

            stage('Publish Results') {
                steps {
                    junit 'results.xml'
                }
            }
        }

        post {
            always {
                echo 'Cleaning up workspace...'
                deleteDir()
            }
        }
    }
```

Після кожного коміту Jenkins автоматично виконує тестування, відображає результати в інтерфейсі та сигналізує про успішне чи помилкове проходження перевірок. Це значно пришвидшує процес розробки, забезпечує оперативний зворотний зв’язок та зменшує ризик інтеграційних помилок.

4.5 Аналіз результатів тестування

Після реалізації автоматизованого пайплайну в Jenkins та запуску серії тестів для розробленого розподіленого застосунку було отримано результати, які детально відображають стан функціональності програмного забезпечення на поточному етапі розробки.

Загальна характеристика результатів

Jenkins автоматично зберігає та візуалізує результати виконаних тестів завдяки плагіну JUnit. У ході тестування було перевірено декілька ключових сценаріїв взаємодії з API застосунку:

- Надсилання валідних запитів з числовим параметром.
- Обробка запиту з нульовим значенням.
- Поведінка системи при відсутності обов'язкового параметра.

Кожен з тестів виконується з використанням бібліотеки pytest, яка генерує XML-звіт (results.xml). Jenkins обробляє цей звіт та формує наступні аналітичні дані:

- Загальна кількість тестів.
- Кількість успішних, помилкових та пропущених тестів.
- Графік змін у результатах з кожним запуском.
- Детальний список помилок або невдач, якщо такі є.

Оцінка стабільності системи

Усі розроблені тести пройшли успішно, що свідчить про правильну реалізацію базової логіки обробки даних. Зокрема, API застосунку коректно повертає результат обчислень та обробляє виняткові ситуації без виникнення критичних помилок. Це підтверджує готовність застосунку до подальшого розширення функціоналу.

Потенційні напрямки покращення

Хоча поточні тести охоплюють базову функціональність, на наступних етапах доцільно розширити тестовий набір, включаючи:

- Перевірку граничних значень (наприклад, дуже великі або від'ємні числа).

- Інтеграційні тести з декількома компонентами.
- Перевірку продуктивності та витривалості під навантаженням.
- Тестування системи у розподіленому середовищі з використанням контейнерів Docker.

Інтеграція тестів у Jenkins-пайплайн значно підвищила надійність розробки. Кожна зміна коду одразу супроводжується запуском тестів, що знижує ризик регресій. Отримані результати свідчать про правильність обраного підходу до побудови автоматизованої системи тестування розподілених застосунків.

Висновок

В рамках даної дипломної роботи була успішно реалізована комп'ютерна система для автоматизації процесу тестування розподілених додатків з використанням інструменту Jenkins та мови програмування Python. Основною метою дослідження було створення ефективного, масштабованого та автоматизованого рішення, що дозволяє спростити та прискорити процес перевірки функціональності програмних компонентів у розподіленому середовищі.

В ході роботи було зроблено наступне:

Сформульовано вимоги до системи автоматизації та обґрунтовано вибір архітектури майбутнього рішення.

Розроблено систему, яка використовує Jenkins для запуску тестів, та реалізовано тестовий приклад REST-додатку на Python з відповідним тестовим пакетом.

Налаштовано процес CI, який автоматично перевіряє коректність змін у коді після кожного внесення змін до репозиторію.

Список використаних джерел

1. Prometheus. Безкоштовні онлайн-курси [Електронний ресурс]. https://prometheus.org.ua
2. Розподілене програмне забезпечення / Вікіпедія [Електронний ресурс]. https://uk.wikipedia.org/wiki/Розподілене_програмне_забезпечення
3. Jenkins Documentation [Електронний ресурс]. https://www.jenkins.io/doc/
4. Docker Documentation [Електронний ресурс]. https://docs.docker.com
5. Pytest Documentation [Електронний ресурс]. https://docs.pytest.org/en/stable/
6. GitHub Docs. CI/CD automation [Електронний ресурс]. https://docs.github.com
7. REST API Tutorial [Електронний ресурс]. https://restfulapi.net
8. FreeCodeCamp. Безкоштовні навчальні матеріали з Jenkins, Docker, Python [Електронний ресурс]. https://www.freecodecamp.org.
9. Stack Overflow. Платформа питань і відповідей для розробників [Електронний ресурс]. https://stackoverflow.com.
10. Dev.to. Спільнота розробників [Електронний ресурс]. https://dev.to.
11. IEEE Xplore Digital Library. Публікації з теми тестування та CI/CD [Електронний ресурс]. https://ieeexplore.ieee.org.