Харківський національний університет радіоелектроніки
(повне найменування вищого навчального закладу)

Факультет *інфокомунікацій*
(повна назва)

Кафедра *інформаційно-мережної інженерії*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

на тему *Дослідження шляхів перевірки виконання практичних завдань в локальному оточенні користувача. Розробка архітектури системи*

Виконав: студент *2* курсу, групи *ІМІм-22-1*
напряму підготовки
172 "Телекомунікації і радіотехніка"
(шифр і назва напряму підготовки)
Тіщенко В. В.
(прізвище та ініціали)
Керівник *Костромицький А.І.*
(прізвище та ініціали)

Допускається до захисту

Зав. кафедри _____ *Безрук В.М.*
(підпис) (прізвище, ініціали)

Харків - 2024 рік

Не містить відомостей, заборонених до відкритого публікування

Студент _____

Керівник_____

Харківський національний університет радіоелектроніки
(повне найменування вищого навчального закладу)

Факультет *інфокомунікацій*

Кафедра *інформаційно-мережної інженерії*

Рівень вищої освіти *другий (магістерський)*

Напрям підготовки *172 «Телекомунікації і радіотехніка»*
(шифр і назва)

ЗАТВЕРДЖУЮ

Зав.кафедри _____
(Підпис)

"____" _____ 2023 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Тіщенку Вадиму Вікторовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження шляхів перевірки виконання практичних завдань в локальному оточенні користувача. Розробка архітектури системи*

затверджені наказом ВНЗ від "*23*" *жовтня* 2023 року № *1233 Ст.*

2. Строк подання студентом роботи *10 січня 2024 року*

3. Вихідні дані до роботи *Дослідити шляхи перевірки виконання практичного завдання в локальному оточенні користувача. Розробити архітектуру системи*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Вступ

1.Огляд рішення

2. Загальна архітектура додатку

3. Поглиблення в деталі

4. Реалізація

5. Практичне застосування

6. Використання стороннього інструменту

Висновки

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

Слайди у форматі Power Point (назва та мета роботи, актуальність, система перевірки навичок, типи сховищ, концепції завдань, висновки)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Основна частина	*доц. Костромицький А.І.*	21.10.23	

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Ознайомлення із завданням. Уточнення ТЗ.*	*21.10.23*	
2	*Підбір літератури за темою роботи.*	*23.10-29.10.23*	
3	*Виконання розділу 1*	*30.10-04.11.23*	
4	*Виконання розділу 2*	*05.11-09.11.23*	
5	*Виконання розділу 3*	*10.11-14.11.23*	
6	*Виконання розділу 4*	*15.11-19.11.23*	
7	*Виконання розділу 5*	*20.11-31.12.23*	
8	*Виконання розділу 6*	*01.01.24-04.01.24*	
9	*Оформлення презентаційного матеріалу,*	*05.01.24-10.01.24*	
	підготовка до захисту у ЕК		

Дата видачі завдання *21* *жовтня* *2023 р.*

Студент _____ *Тіщенко В. В.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Костромицький А.І.*
(підпис) (прізвище та ініціали

РЕФЕРАТ

Пояснювальна записка: 87 с., 94 рис., 6 джерел

Об'єкт роботи – система автоматичної перевірки виконання практичних завдань в локальному серидовищі учня.

Мета роботи – дослідження та розробка архітектури системи для автоматичної перевірки виконання практичних завдань, виконаних студентом в локальному серидовищі.

Досліджено шляхи автоматизації перевірок практичних завдань. Розроблена система, що дозволяє студенту виконувати практичні завданні на віртуальній машині в локальному серидовищі, що запобігає засмічуванню його основної робочої машини.

PYTHON, GOLANG, MOODLE, LMS, LTI, СИСТЕМА АВТОМАТИЧНОЇ ПЕРЕВІРКИ

ABSTRACT

Explanatory note: 87 p., 94 fig., 6 sources.

Object of the work – an automatic assessment system for practical tasks performed in a student's local environment.

Objective of the work – research and development of the system architecture for the automatic assessment of practical tasks performed by a student in a local environment.

Various ways of automating the assessment of practical tasks have been explored. A system has been developed that allows students to complete practical tasks on a virtual machine in their local environment, preventing cluttering of their primary work machine.

PYTHON, GOLANG, MOODLE, LMS, LTI, AUTOMATIC TESTING SYSTEM

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

HTTP (Hyper-Text Transfer Protocol) – протокол передачі гіпер-тексту;

TCP (Transmission Control Protocol) – протокол керування передачею даних;

UDP (User Datagram Protocol) – протокол користувацьких датаграм;

LMS (Learning Management System) – система керування навчанням;

LTI (Learning Tools Interoperability) – специфікація освітніх технологій;

JSON (JavaScript Object Notation) – текстовий формат обміну даними;

JWT (JSON Web Token) – відкритий стандарт для створення токенів;

JWK (JSON Web Key) – ключ, що використовується для перевірки JWT токену;

ORM (Object-Relation Mapping) – технологія, що дозволяє пов’язати сутності бази даних, з сутностями коду;

WSGI (Web Server Gateway Interface) – стандарт взаємодії між Python-застосунком на сервері, та самим веб-сервером, на приклад Nginx;

HTML (Hyper-Text Markup Language) – мова гіпертекстової розмітки;

CSS (Cascade Style Sheet) – каскадні таблиці стилів, використовуються, щоб стилізувати базову розмітку мови HTML;

JS (JavaScript) – мова програмування, що дозволяє зробити HTML сторінку більш динамічною, та не тільки;

OIDC (OpenID Connect) – протокол аутентифікації користувача сторонніми сервісами;

ВСТУП

Мета кваліфікаційної роботи – автоматизація перевірки виконання практичних завдань, у локальному середовищу учня.

Сучасний світ стикається з різноманітними викликами та обставинами, які вимагають найкращих підходів до організації та забезпечення навчання та навчальних процесів. Серед цих викликів важко переоцінити вплив воєнних конфліктів і карантинних заходів, які виникають в умовах глобальних криз та криз в сфері охорони здоров'я. У таких умовах стає надзвичайно важливою роль Learning Management System (LMS) чи українською систем керування навчанням (СКН).

Системи керування навчанням визначаються як інформаційні технології, що сприяють організації, адмініструванню та поширенню навчального матеріалу. На перший погляд, вони можуть здатися винятковою інструментальною платформою для навчання. Проте, сучасний світ свідчить про те, що LMS володіють величезною важливістю, особливо в періоди коли очне навчання є чимось неможливим.

1 ОГЛЯД РІШЕННЯ

1.1 Що таке LMS?

Система керування навчанням – це програмний продукт для адміністрування навчальних курсів в рамках дистанційного навчання.

Свою історію СКН беруть з другої половини XX століття в академічному секторі. Причиною того, що СКН беруть свій початок саме в університетах – велика кількість студентів, що несуть за собою велику кількість навантаження на навчальні відділи, що можна було б запобігти автоматизацією перевірок завдань студентів.

На даний момент СКН несуть невід’ємна частину нашого життя, допомагаючи постійно дізнаватись щось нове, та опановувати все нові навички, які допоможуть нам як в роботі, так і в повсякденному житті.

1.2 Переваги та недоліки СКН

Переваги СКН виходять з концепції електронного навчання, та його відмінностей від традиційного:

1) Свобода доступу – учень може навчатись практично з будь-якого місця, в вільний для нього час;

2) Зменшення витрат на навчання – учень не несе витрат на методичні посібники, та навчальну літературу. Також не потрібно утримувати навчальний заклад з усіма його витратами;

3) Гнучкість – процес навчання можна підлаштовувати як під учнів, так і під педагогів;

4) Можливість розвитку в ногу з часом – простіше підтримувати актуальну інформацію для учнів;

5) Потенційно рівні умови навчання – якість навчання незалежне від якості викладання в конкретній школі, чи вузі;

6) Можливість об’єктивного оцінювання знань – є можливість виставляти чіткі критерії для оцінювання;

Недоліками СКН можна назвати наступні пункти:

1) Відсутність контакту учителя та учня.

2) Для впровадження СКН потрібні ресурси у вигляді серверів, а також їх постійна підтримка.

3) Занижується роль вмінь педагогів.

1.3 Актуальність

На даний момент часу, СКН являється дуже актуальною темою, враховуючи світові умови. Глобальна пандемія вимусила переходити на дистанційне навчання, вчитись більше працювати самостійно за допомогою різного роду СКН. Відкритисть багатьох систем дає змогу спільноті доопрацьовувати найвний інструментарій чи розроблювати новий та забезпечувати взаємодію відкритими протоколами.

Враховуючи відсутність подібних систем перевірки завдань в ХНУРЕ (принайми автор під час навчального процесу не стикався з такими системами), вважаю що концепт, який потрібно розробити відповідно технічного завдання на цю роботу, має значні перспективи щодо впровадження.

2 ЗАГАЛЬНА АРХІТЕКТУРА ДОДАТКУ

2.1 Архітектура для додатку

На поточний час є 2 основних підходи які використовуються в побудові додатків: моноліт, та мікросервіси (рис. 2.1).

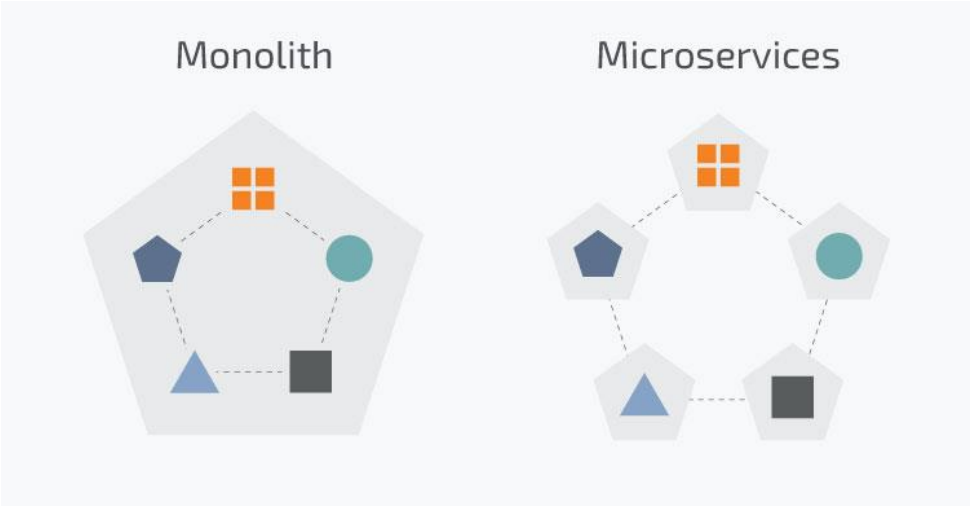

Рисунок 2.1 – Демонстрація моноліту та мікросервісу

Моноліт – це те від чого наразі намагаються відходити, але хто знає куди поверне курс побудови веб-додатків. Основна ідея полягає в тому, що все необхідне для функціонуванню додатку лежить поряд, і розгорнути в одному середовищі, має спільний сервер, спільну базу даних, та спільний сховок файлів. Плюсами такого підходу можна вважати:

1) Консистенстність, узгодженність. Коли застосунок суцільний, немає потреби турбуватись про узгодженість взаємодії між його компонентами. Всі єлементи цього додатку є частиною одного цілого.

2) Легке розгортання. Оскільки все що необхідно для повноцінної роботи додатку знаходиться в одному місці, це легко можна розгорнути.

Але у такого підходу є й свої мінуси:

1) Великі об’єми кодової бази, не легкий шлях розподілення. У монолітних систем важко розмежувати один функціонал від іншого, і часто виходить каша з кодової бази;

2) Проблеми масштабування. Через те, що весь функціонал не від’ємний одне від одного – при зростанні навантаження доводиться масштабувати горизонтально, тобто додавати фізичні ресурси для роботи додатку.

На противагу моноліту – є мікросервісна архітектура. До цього підходу прийшли після моноліту, і в деяких моментах мікросервісний підхід розв’язує руки, але це не панацея від кожної болячки. Із переваг можна виділити наступні пункти:

1) Легкість, та мінімалістичність кожного сервісу. Для окремого функціоналу створюється свій незалежний сервіс, котрий буде відповідальний за обмежений об’єм роботи;

2) Легкість у входженні нових людей. Оскільки мікросервіси несуть в собі мало бізнес-логіки, точніше меншу частину, а ніж вся логіка продукту, новим людям простіше поринути в аспекти роботи конкретного функціоналу;

3) Масштабування. На противагу моноліту, мікросервіси масштабуються за рахунок технології балансування навантаження, що дозволяє підняти більшу кількість сервісів, на котру є пік навантаження, для розподілення цього навантаження (рис 2.2).

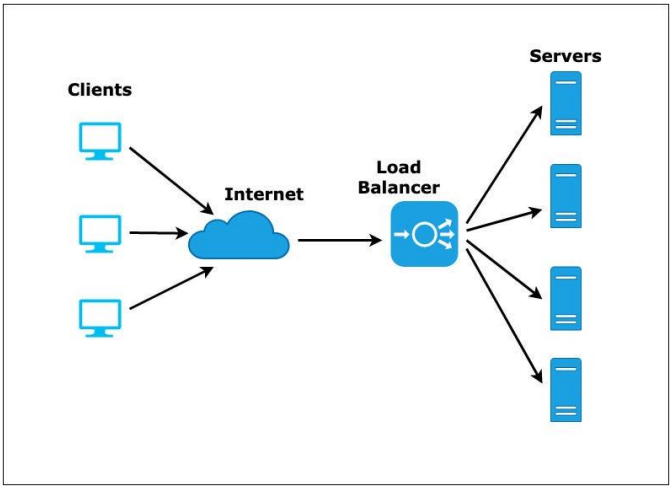

Рисунок 2.2 – Приклад роботи балансувальника навантаження

Також на основі цих двох архітектур є змішана. Змішана архітектура поєднує в собі мікросервіси, та монолітні сервіси. Інколи виходить, що моноліт не можна розділити через різні причини, це може бути або складність, або не доцільність цих змін.

2.2 Проєктування проєкту на основі змішаної архітектури

Чому саме змішана архітектура.

Проєкт побудовано на основі відкритої LMS, що являє собою вже повноцінний продукт готовий до використання, в даному випадку Moodle. Готове рішення було обране задля економії часу та ресурсів.

На схемі (рис. 2.3) видно основні компоненти проєкту.

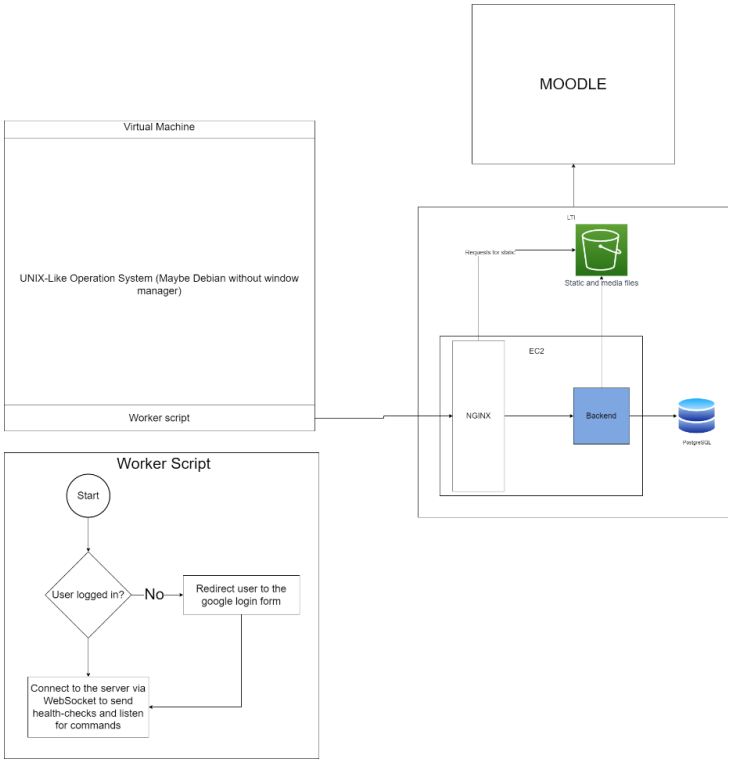

Рисунок 2.3 – Архітектура проєкту

Один з них являє собою монолітну LMS, Moodle. Цей компонент відповідальний за створення завдань, викладання матеріалу для лекцій, та ще багато чого, що надається системами керуванням навчанням.

Другий компонент системи – застосунок, котрий буде виконувати роль автоматизованої перевірки завдань. Цей компонент є окремим сервісом, бо впровадити цей механізм у вже існуючу систему з велетенською кодовою базовою завдання не тривіальне.

Третім компонентом системи являється клієнт. Це локальне середовище учня, на котрому розгорнуто віртуальну машину, з спеціально написаним скриптом, що підтримує синхронізацію між локальним середовищем, а сервісом перевірок домашнього завдання.

Четвертий компонент є продовженням третього. Скрипт котрий несе в собі функціонал синхронізації локального середовища учня, та сервісом перевірок.

3 ПОГЛИБЛЕННЯ В ДЕТАЛІ

3.1 LMS

Рисунок 3.1 – Тлумачення поняття LMS

Основою всього проєкту є LMS. Вона дає основний функціонал, довкола якого буде базуватись система автоматизації перевірки практичних завдань.

LMS дає функціонал створення курсів, завдань, ресурсів для навчання, а також управлінням цього всього.

Вже існує достатньо таких систем у відкритому доступі, найпопулярніші з них є Moodle та Open edX. Вибір припав саме на Moodle, і на це є декілька причин:

1) Moodle проста у використанні, та розгортанні – для роботи цієї системи лише потрібно підняти сервер, котрий вміє виконувати PHP, та СУБД MySQL, і система готова для роботи;

2) Широке користувацьке коло – якщо вилізе якась проблема, скоріш за все з нею вже хтось зтикався, і знайти рішення для неї не буде чимось складним;

3) Великий інструментарій – під капотом вже є все, щоб налаштувати Moodle під себе, в даному випадку OAuth2, та LTI;

4) Вже знайомий інтерфейс – оскільки dl.nure.ua вже використовує дану СКН, інтерфейс виглядає досить знайомим та не відштовхуючим.

Одним з ключових елементів СКН – є можливість авторизації за допомогою пошти. Це дозволить створити наскрізний ключ, який буде чітко ідентифікувати людину всередині цілої системи – це унікальна поштова адреса користувача.

3.2 Інструмент для перевірки завдань

Для виконання головної мети – автоматизація перевірки домашніх завдань, потрібен якийсь інструмент, котрий буде вміти це робити, і тут є два підходи, у вирішення цієї задачі.

Перший підхід – це розширення функціоналу самої LMS, але в нього є фундаментальні недоліки. Щоб це зробити доведеться залізти в величезну кодову базу, при цьому нічого не зламавши. У цього підходу є плюс – це рішення проблеми в лоб. Не потрібно значи нічого більше, окрім основної мови на чому написано LMS.

Другий підхід трошки складніше, але більш правильний з точки зору реалізації – це створення окремого додатку котрий буде перевіряти завдання, підтримувати зв’язок з машиною учня, та ініціювати перевірки виконання завдань. Цей підхід є більш складним, тому що, треба додатково дослідити роботу такого стандарту як LTI, котрий був придуманий саме для таких цілей.

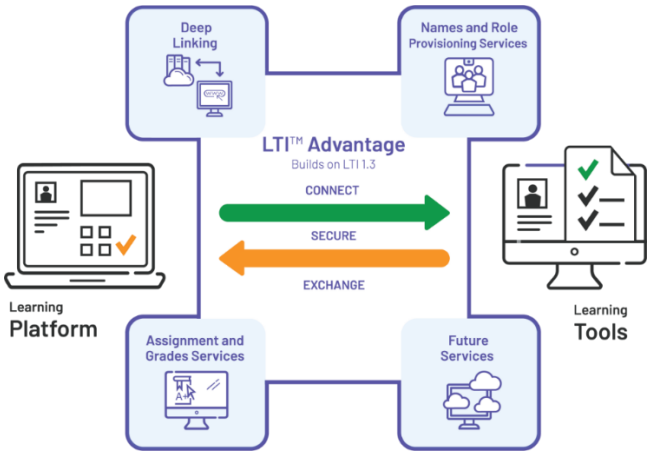

Рисунок 3.2 – Візуалізація стандарту LTI

LTI (Learning Tools Interoperability) – це стандарт для інтеграції різних навчальних технологій або додатків з СКН. Цей стандарт дозволяє інтегрувати різні додатки до основної СКН, що спрощує процес реалізації функціонал тим, що знімає необхідність додавати модулі до СКН, і розбиває новий функціонал на окремі маленькі сервіси, що відповідальні кожен за свій шматочок. Із плюсів цього стандарту можна відмітити:

1) Безпека і аутентифікація – стандарт надає доступ лише аутентифікованим користувачам;

2) Одноразовий вхід – користувачам не потрібно знову вводити свої дані до облікового запису, оскільки застосунок дістає цю інформацію з СКН;

3) Обмін даними – LTI може синхронізувати інформацію з СКН, що допомагає покращити зв’язок між різними компонентами СКН.

У цього стандарту є й деякі недоліки:

1) Складність розробки – реалізація додатків до СКН не є досить популярною темою, то ж знайти багато корисної інформації, чи прикладів використання не досить проста задача;

2) Обмеження стандарту – будь який застосунок написаний під СКН по стандарту LTI буде ним же й обмежений, для правльної роботи додатку обидві сторони мають притримуватись визначеного стандарту.

Для написання серверної сторони додатку, котрий автоматизує перевірку виконання завдань учнем, було обрано мову програмування Python. Ця мова є досить простою в використанні, а що головне у вивчанні. Під неї вже реалізовано досить велику кількість бібліотек, що включає в себе стандарт LTI з прикладами використання.

3.3 Локальне середовище учня

Одною з проблем дистанційного навчання – є захламлення основної робочої машини, і не досить просте її очищення. Взяти до прикладу веб-сервер Denver, його досить легко встановити, а ось щоб почитити компьютер

від нього, потрібно вже зробити декілька фокусів. Тобто мета – це ізольоване середовище, або ж пісочниця, де можна виконувати практичні завдання.

Для вирішення цієї проблеми є декілька підходів. Віртуалізація, та контейнеризація.

Віртуалізація – це створення віртуального середовище, за допомогою технології Hypervisor, що дозволяє одночасно тримати дві запущені операційні системи одночасно, розподіляючи ресурси між ними.

Контейнеризація, технологія яка імпонує більше юнікс подібним системам, а якщо бути точнішим, то операційним системам на базі ядра лінукс, яка використовує функціонал простору імен, для ізоляції одного середовища від іншого, і ось тут ще варто додати, що контейнеризація не працює повноцінно на системах, що не мають цієї функції, і тому для роботи контейнерів використовуються віртуальні машини, що є основним недоліком технології контейнеризації, але не останнім.

Для віртуальної машини було обрано операційну систему на базі ядра лінукс – Debian без desktop середовища, що означає, що в користуванні студента буде лише термінал, в якому йому буде потрібно працювати. Не досить зручно, але це дозволить віртуальній машині займати менше ресурсів, що буде плюсом для студентів, у котрих не дуже потужне залізо.

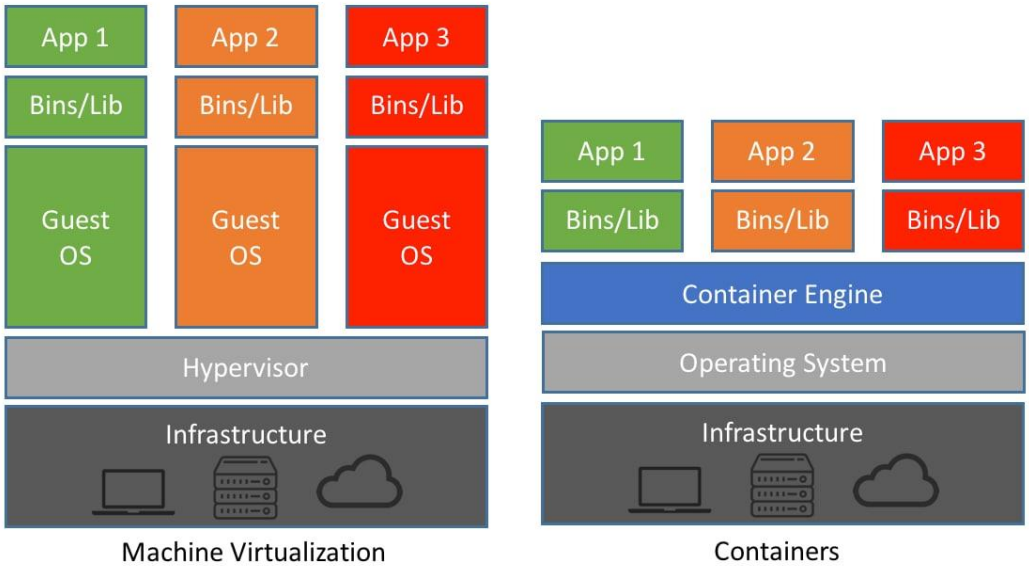

Рисунок 3.3 – Концептуальна різниця віртуалізації і контейнеризації

3.4 Клієнтський рушій

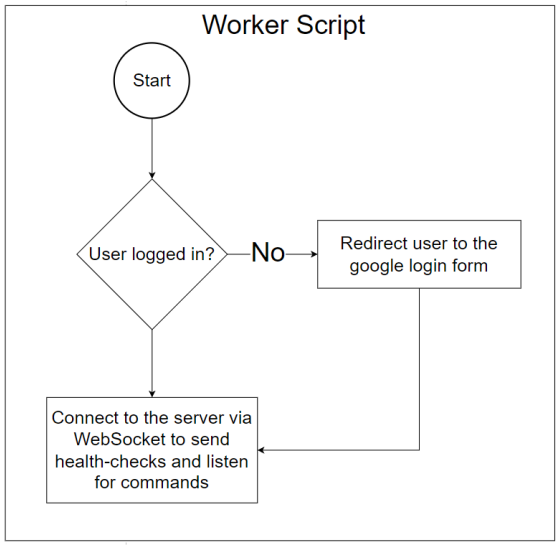

Рисунок 3.4 – Базова схема роботи клієнтського рушія

Для написання клієнтського рушія, котрий буде тримати на собі роль містка між сервером та клієнтом, було обрано мову програмування Golang, і на це є свої причини. Одна з них – це знову ж таки широке коло користувачів, на більшість проблем вже є своє рішення, що облегшує розробку. Друге, що є більш важливим – це компілюєма мова програмування, тобто код котрий буде написаний на ній, в кінцевому результаті переведеться на байткод, котрий буде займати менше місця, і буде швидко виконуватись. Третє, най важливіше – це можливість компілювати під різні платформи, з статичним лінкуванням. Тобто якщо у програми є якісь залежності, вони будуть запаковані в виконуваний файл, і не буде потреби доставляти щось на машину користувача, буде повністю готова програма котру потрібно буде просто запустити, що спрощує роботу.

Звісно для клієнтського рушія є альтернативи, котрі також можна запускати на різних системах, але є важливі недоліки, на приклад для Java або C# потрібно додатково ставити віртуальну машину для запуску, та вони

займають досить багато ресурсів. Python – для нього довелось би пожертвувати гігабайтами пам’яті для його залежностей. Гарною альтернативою могла би стати така мова програмування як Rust, але нажаль вона досить складна для того, щоб її правильно використовувати.

3.5 Канал зв’язку між клієнтом та сервером

Основним каналом зв’язку між клієнтом та сервером був обраний протокол на базі HTTP – WebSockets.

WebSocket – це повнодуплексний протокол зв’язку, що дозволяє підтримувати комунікацію між клієнтом та сервером в реальному часі.

Цей протокол був обраний не просто так. Можна було б обрати TCP або UDP протоколи, але є свої нюанси.

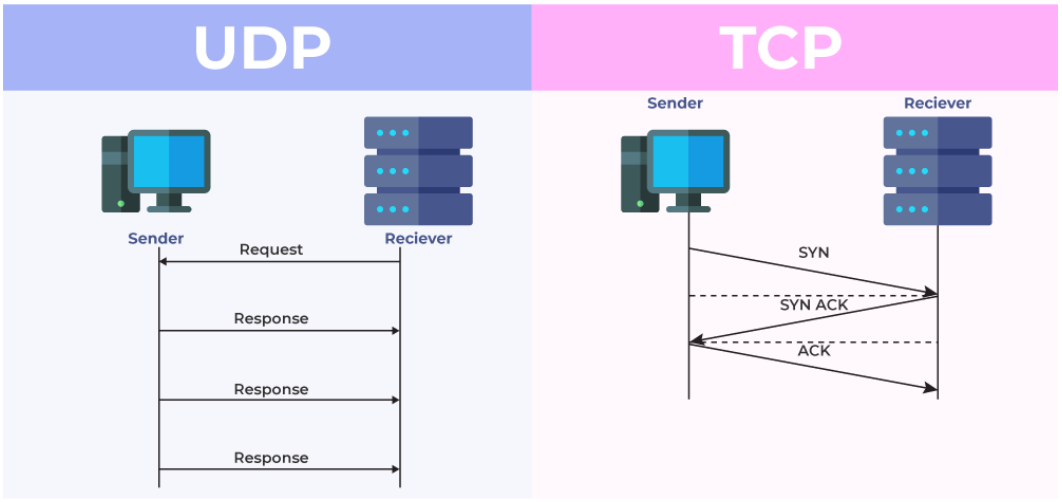

Рисунок 3.5 – Ілюстрація взаємодії TCP та UDP протоколів

UDP протокол – це низькорівневий протокол комунікації, який нехтує гарантією доставки пакетів. Цей протокол в більшості використовується в іграх, в відео-аудіо зв’язку, де втрата декількох пакетів буде відчутна, але не смертельна.

TCP протокол – це протокол коммунікації, який влаштований складніше за UDP, та працює повільніше. Але, цей протокол гарантує доставку пакетів.

Також можна було б розглянути HTTP протокол. Першочергово цей протокол був зроблений задля пересилання гіпер-тексту, але з еволюцією інформаційних технологій, тей протокол також почали використовувати просто як пересилку даних. Але у HTTP є один значущий недолік, зв'язок тримається лише на один запит-відповідь (рис. 3.6).

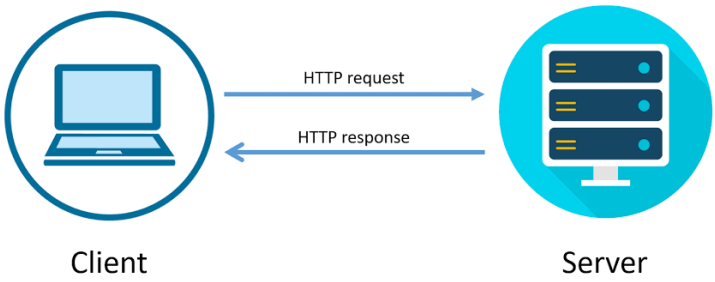

Рисунок 3.6 – Ілюстрація роботи HTTP протоколу

Тобто отримувати оновлення від серверу в реальному часі не є можливим, оскільки з'єднання закривається після того, як сервер надав відповідь. Звісно можна використати підхід, котрий називається Polling, і суть котрого заключається у тому, що клієнт періодично виконує HTTP запити на сервер, з ціллю дізнатись, чи з'явилась на сервері якась нова інформація для обробки.

Цю проблему з закриттям з'єднання вирішує протокол WebSocket. WebSocket базується на основі протоколу HTTP, але з деякими відмінностями. WebSocket – це дуплексний протокол, котрий дозволяє обмінюватись повідомленнями між сервером, та клієнтом в реальному часі, у вигляді текстового повідомлення. Цей протокол працює наступним чином (рис.3.7), клієнт робить на сервер звичайний запит з проханням змінити протокол спілкування з HTTP на WebSocket, після чого сервер відповідає з

кодом 101, що означає переключення протоколо, і після цього, вже є думплексний зв’язок для обміну повідомленням.

Оскільки WebSocket простіше у використанні, не змушує вигадувати якісь протоколи спілкування, а дозволяє використовувати текстові мови розмітки, такі як JSON, це робить його ідеальним протоколом для вибору.

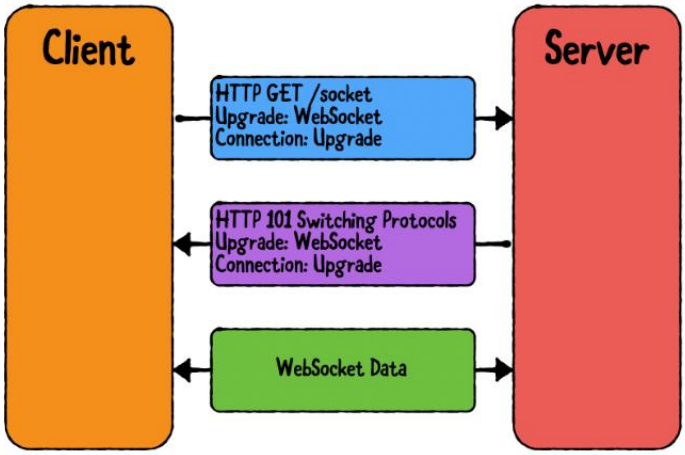

Рисунок 3.7 – Ілюстрація роботи протоколу WebSocket

4 РЕАЛІЗАЦІЯ

4.1 Реалізація серверу для протоколу LTI

Для реалізації протоколу LTI було обрано мову програмування Python. Дану мову було обрано не просто так, а за наступними критеріями:

1) Проста в розумінні, та написанні. Для швидкої реалізації прототипу потрібна мова, котра не змушує реалізовувати складні абстракції, а дає волю діям;

2) Наявність готової реалізації протоколу LTI, що пришвидшить розробку прототипу;

3) Досвід використання.

Перший етап – потрібно зрозуміти які залежності будуть використані, для побудови застосунку. Як веб-сервер – була обрана бібліотека flask, котра дозволяє реалізовувати синхронний веб-сервер. Поверх flask-у, вже будуть реалізовані всі ендпоінти, щоб задовільнити протоколу LTI.

Як залежності було обрано наступні бібліотеки:

1) Flask – бібліотека для реалізації веб-серверу;

2) Flask-Caching – бібліотека до Flask, котра розширює можливості кешування;

3) Flask-Session2 – бібліотека котра розширює стандартний функціонал сесій бібліотеки Flask;

4) Flask-SQLAlchemy – ORM для Flask;

5) Flask-Migrate – бібліотека для міграції моделей SQLAlchemy в схему бази даних, щоб потім мігрувати ці схеми в новому оточенні;

6) flask-sock – бібліотека, що реалізує протокол WebSocket

7) werkzeug – бібліотека котра розширює функціонал WSGI-застосункців.

8) PyLTI1p3 – реалізація протоколу LTI версії 1.3.

Застосунок починається з імпортування залежностей. В мові програмування Python прийнято імпортувати залежності наступним чином, спочатку йдуть залежності котрі вбудовані в мову, тобто стандартні, потім йдуть залежності, що встановлені, і потім ідуть залежності, котрі реалізовані в самому додатку.

```
import datetime    # Робота з датою
import os          # Взаємодія з операційною системою
import json        # Робота з JSON
import shutil      # Робота з файлами
import time        # Робота з часом
import typing as t # Бібліотека типізації
```

Рисунок 4.1 – Імпорт стандартних бібліотек

```
from flask.json import jsonify # Спрощення роботи з json на рівні веб-застосунку
from flask import (
    Flask,           # Об'єкт застосунку
    render_template, # Відмальовка html шаблонів
    send_file,       # Відправка файлів клієнту
    request,         # Доступ до об'єкту запиту
)
from flask_sock import (
    Sock,            # Абстракція над WebSocket
    Server,          # Об'єкт підключення клієнту до серверу
    ConnectionClosed, # Помилка, що символізує, що канал зв'язку закрито
)
from flask_sqlalchemy import SQLAlchemy # Об'єкт ORM
from flask_migrate import Migrate      # Міграції
from flask_caching import Cache        # Кешування
```

Рисунок 4.2 – Імпорт залежностей, що відносяться до веб-серверу

```
# Помилка котра означає що доступ до ресурсу заборонено
from werkzeug.exceptions import Forbidden
# Функція що перетворює ім'я файлу на "безпечне"
from werkzeug.utils import secure_filename
```

Рисунок 4.3 – Імпорт залежностей, що відносяться до допоміжних

```
from pylti1p3.contrib.flask import (
    FlaskOIDCLogin,      # Аутентифікація OpenID Connect
    FlaskMessageLaunch,  # Об'єкт котрий тримає в собі інформацію про запуск
    FlaskRequest,        # Абстракція над запитом Flask-у
    FlaskCacheDataStorage # Доступ до сховища кешу
)
from pylti1p3.grade import Grade                # Доступ до оцінок
from pylti1p3.lineitem import LineItem          # Абстракція над сутностями LSM
from pylti1p3.tool_config import ToolConfJsonFile # Конфігурація LTI з JSON файлу
from pylti1p3.registration import Registration  # Допоміжні інструменти в області Реєстрації
```

Рисунок 4.4 – Імпорт залежностей PyLTI1P3

Після того, як всі залежності імпортовано, потрібно створити головний об’єкт застосунку. Цей об’єкт відповідає за все, що відбувається в самому застосунку, створення ендпоінтів, запуск серверу, тощо.

```
app = Flask(
    "lti-imim-22-1",              # Ім'я застосунку
    template_folder="templates",  # Дирикторія де знаходяться шаблони
    static_folder="static",       # Дирикторія де статичні файли
)
```

Рисунок 4.5 – Створення об'єкту застосунку

Після створення об’єкту додатку його потрібно відконфігурувати.

```
app.config.from_mapping({
    "DEBUG": True,
    "ENV": "development",
    "CACHE_TYPE": "FileSystemCache",
    "CACHE_DIR": make_temp_dir("cache"),
    "CACHE_DEFAULT_TIMEOUT": 600,
    "SECRET_KEY": os.getenv("SECRET_KEY") or "default",
    "SESSION_TYPE": "filesystem",
    "SESSION_FILE_DIR": make_temp_dir("session"),
    "SOCK_SERVER_OPTIONS": {
        "ping_interval": 25
    },
    "MEDIA_ROOT": "/var/lti/media",
    "SQLALCHEMY_DATABASE_URI": "sqlite:///" + os.path.join('/var/lti', 'database.db'),
    "SQLALCHEMY_TRACK_MODIFICATIONS": False
})
```

Рисунок 4.6 – Конфігурація застосунку.

В конфігураційному файлі можна побачити наступні пункти

1) *DEBUG* – Параметр, що відповідає за поводження серверу, в дебаг моді, сервер віддає додаткову інформацію про помилки, котрі трапились в процесі його роботи;

2) *ENV* – Параметр щоб відрізняти оточення, на приклад, в *development* оточенні може бути додатковий функціонал, котрий дозволить спростити етап розробки;

3) *CACHE_TYPE* – тип кешу, або ж, де він буде зберігатись, в пам’яті застосунку, в якомусь сховищі (на приклад, база даних Redis), або ж у файловій системі, як зображено на рисунку 4.6;

4) *CACHE_DIR* – оскільки тип кешу було обрано *FileSystemCache*, тобто кеш у файловій системі, потрібно вказати, де саме в файловій системі буде зберігатись кеш, тобто вказати конкретну папку;

5) *CACHE_DEFAULT_TIMEOUT* – час в секундах, за який кеш буде вважатись не валідним, тобто застарівшим для використання;

6) *SECRET_KEY* – секретний ключ, що використовується у симетричному шифруванні сесій;

7) *SESSION_TYPE* – те ж саме, що й *CACHE_TYPE*, але по відношенню до сесій;

8) *SESSION_FILE_DIRE* – те ж саме, що й *CACHE_DIR*, але по відношенню до сесій;

9) *SOCK_SERVER_OPTIONS* – конфігурація серверу WebSocket;

10) pign_interval – інтервал з яким WebSocket сервер буде простукувати клієнт, щоб перевірити, чи живе з'єднання;

11) *SQLALCHEMY_DATABASE_URI* – ідентифікатор ресурсу бази даних;

12) *SQLALCHEMY_TRACK_MODIFICATIONS* – відслідковування зміни схеми бази даних.

Після того, як застосунок відконфігуровано, потрібно створити додаткові сутності, котрі будуть відповідати за аспекти кешування, веб-сокетів, бази даних, та міграцій

```
65  sock = Sock(app)
66  db = SQLAlchemy(app)
67  cache = Cache(app)
68  migrate = Migrate(app, db)
```

Рисунок 4.7 – Створення додаткових сутностей

Також потрібно створити сутності бази даних, для збереження завдань, створених вчителем. Всього є дві сутності, сутність завдання, та сутність тесту. Ці дві сутності пов'язані між собою, як один до багатьох, тобто одне завдання може мати під собою багато тестів.

```python
class Task(db.Model):
    __tablename__ = 'task'

    id = db.Column(db.Integer, primary_key=True)
    description = db.Column(db.String(255), nullable=False)
    tests = db.relationship('Test', backref='task', passive_deletes=True)


You, 2 months ago | 1 author (You)
class Test(db.Model):
    __tablename__ = 'test'

    id = db.Column(db.Integer, primary_key=True)
    description = db.Column(db.String(255), nullable=False)
    task_id = db.Column(db.Integer, db.ForeignKey('task.id'), nullable=False)
    file_name = db.Column(db.String(255), nullable=False)
```

Рисунок 4.8 – Створення сутностей завдання, та тестів

Коли всі сутності створені, та застосунок відконфігурований, можна зайнятись створенням ендпоінтів, котрі будуть використовуватись.

```python
@app.route("/jwks/", methods=["GET"])
def get_jwks():
    tool_conf = ToolConfJsonFile(get_lti_config_path())
    return jsonify(tool_conf.get_jwks())
```

Рисунок 4.9 – Реалізація енпоінту, що поверне публічні JWK

Ендпоінт jwks потрібен для того, щоб LMS могла верифікувати токен, з котрим прийде юзер, вони потрібні для того, щоб LMS була впевнена тому, що токен точно був виписаний системою LTI.

```python
@app.route("/login/", methods=["GET", "POST"])
def login():
    tool_conf = ToolConfJsonFile(get_lti_config_path())
    launch_data_storage = get_launch_data_storage()

    flask_request = FlaskRequest()
    target_link_uri = flask_request.get_param("target_link_uri")
    if not target_link_uri:
        raise Exception("Missing 'target_link_uri' param")

    oidc_login = FlaskOIDCLogin(flask_request, tool_conf, launch_data_storage=launch_data_storage)
    return oidc_login \
        .enable_check_cookies() \
        .redirect(target_link_uri)
```

Рисунок 4.10 – Ендпоінт логіну

Ендпоінт логіну, це перша сторінка на котру потрапляє користувач, на цій сторінці він аутентифікується, щоб при подальшому використанні LTI, було відомо, що саме за юзер прийшов, і після успішної аутентифікації, користувач буде перенаправлений на основну сторінку LTI.

```python
@app.route("/", methods=["POST"])
def launch():
    tool_conf = ToolConfJsonFile(get_lti_config_path())
    flask_request = FlaskRequest()
    launch_data_storage = get_launch_data_storage()
    message_launch = FlaskMessageLaunch(flask_request, tool_conf, launch_data_storage=launch_data_storage)
    message_launch_data = message_launch.get_launch_data()
    task_id = message_launch_data["https://purl.imsglobal.org/spec/lti/claim/resource_link"]["id"]
    tpl_kwargs = {
        "page_title": PAGE_TITLE,
        "is_deep_link_launch": message_launch.is_deep_link_launch(),
        "launch_data": message_launch.get_launch_data(),
        "launch_id": message_launch.get_launch_id(),
        "email": message_launch_data.get("email", ""),
        "task_id": task_id,
    }

    task = db.session.get(Task, task_id)
    tpl_kwargs.update({
        "task": task
    })

    if task:
        tpl_kwargs.update({
            "tests": task.tests
        })

    if message_launch.check_teacher_access():
        return render_template("teacher.html", **tpl_kwargs)

    return render_template("student.html", **tpl_kwargs)
```

Рисунок 4.11 – Ендпоінт логіну

Ендпонінт логіну – сюди користувач потрапляє після аутентифікації, на цьому етапі відомо, що це за користувач, яка його роль в системі (вчитель, чи студент), до якого завдання відноситься цей запуск (task_id), та на основі цих даних, вирішується, яку сторінку малювати, сторінку вчителя, з можливістю додавання завдання, та тестів, чи сторінку студента, з можливістю перевірки виконання тестів.

Цей ендпоінт використовується вчителем, для завантаження завдання, та тестів, котрі будуть запускатись на віртуальній машині учня. Він працює наступним чином, якщо для цього елементу LMS (завдання, або *task_id*), вже було створені елементи завдання, чи тестів в базі даних – вони видаляються, якщо вже були підвантажені якісь тести, і вони є на сервені, вони також

видаляються, і потім зберігаються нові тести та завдання, що підвантажив вчитель. Такий підхід потрібен, щоб була можливість перезавантаження контенту.

```python
@app.route('/tests', methods=['POST'])
def create_tests():
    form = request.form
    task_id = form['task_id']
    media_root = app.config['MEDIA_ROOT']
    task_dir = os.path.join(media_root, str(task_id))

    db.session.query(Task).filter(Task.id == task_id).delete()
    db.session.query(Test).filter(Test.task_id == task_id).delete()

    task = Task(id=form['task_id'], description=form['description'])
    db.session.add(task)
    if not os.path.exists(task_dir):
        os.mkdir(task_dir)
    else:
        # recreate directory
        shutil.rmtree(task_dir)
        os.mkdir(task_dir)

    for tf in request.files:
        f = request.files[tf]
        filename = secure_filename(f.filename)
        idx = int(tf.split('-')[2])
        db.session.add(Test(description=form[f'test-description-{idx}'], task_id=task.id, file_name=filename))
        f.save(os.path.join(task_dir, filename))

    db.session.commit()
    return jsonify({
        'message': 'ok'
    }), 201
```

Рисунок 4.12 – Ендпоінт створення тестів, та завдання

Ендпоінт оцінювання, з назви зрозуміло, відповідає за виставлення оцінок відповідному користувачу.

На цьому етапі вже є майже всі ендпоінти, котрі потрібні для роботи LTI, лишилось реалізувати логіку взаємодії між студентом, та його віртуальною машиною. Ця логіка будується на основі протоколу WebSocket. Створюються глобальні об’єкти типу *dict***.** Dict – це об’єкт мови Python, що реалізовує структуру даних HashMap, тобто дозволяє робити асоціативний масив, у котрого ключ – це користувач, а значення – це з’єднання з цим користувачем.

```python
@app.route("/api/score/<launch_id>/<earned_score>", methods=["POST"])
def score(launch_id, earned_score):
    tool_conf = ToolConfJsonFile(get_lti_config_path())
    flask_request = FlaskRequest()
    launch_data_storage = get_launch_data_storage()
    message_launch = FlaskMessageLaunch.from_cache(launch_id, flask_request, tool_conf,
                                                   launch_data_storage=launch_data_storage)

    resource_link_id = message_launch.get_launch_data() \
        .get("https://purl.imsglobal.org/spec/lti/claim/resource_link", {}).get("id")

    if not message_launch.has_ags():
        raise Forbidden("Don't have grades!")

    sub = message_launch.get_launch_data().get("sub")
    timestamp = datetime.datetime.utcnow().isoformat() + "Z"
    earned_score = int(earned_score)

    grades = message_launch.get_ags()
    sc = Grade()
    sc.set_score_given(earned_score) \
        .set_score_maximum(100) \
        .set_timestamp(timestamp) \
        .set_activity_progress("Completed") \
        .set_grading_progress("FullyGraded") \
        .set_user_id(sub)

    sc_line_item = LineItem()
    sc_line_item.set_tag("score") \
        .set_score_maximum(100) \
        .set_label("Score")
    if resource_link_id:
        sc_line_item.set_resource_id(resource_link_id)

    result = grades.put_grade(sc, sc_line_item)

    return jsonify({"success": True, "result": result.get("body")})
```

Рисунок 4.13 – Ендпоінт оцінювання

```python
clients: dict[str, Server] = {}
servers: dict[str, Server] = {}
```

Рисунок 4.14 – Створення об’єктів котрі зберігають в собі WebSocket з’єднання

Створено саме два об’єкти, щоб відрізняти, це підключений користувач, чи його віртуальна машина. Щоб наповнити ці об’єкти, потрібно реалізувати ще два додаткових ендпоінти, котрі будуть реалізовувати логіку підключення до серверу через протокол WebSocket.

```python
@sock.route("/ws/server/<email>")
def server_sock(ws: Server, email: str):
    global servers

    servers[email] = ws
    send_client(email, {"event": "connected", "args": []})

    while True:
        try:
            ws.send('{"message": "ping"}')
            time.sleep(ws.ping_interval)
        except ConnectionClosed:
            print(f"server connection with {email} closed.")
            del servers[email]
```

Рисунок 4.15 – Ендпоінт для підключення віртуальної машини користувача до сервера

```python
@sock.route("/ws/client/<email>")
def client_sock(ws: Server, email: str):
    global clients

    clients[email] = ws

    if email in servers:
        send_client(email, {"event": "connected", "args": []})

    while True:
        try:
            ws.receive()
        except ConnectionClosed:
            print(f"client connection with {email} closed.")
            del clients[email]
            return
```

Рисунок 4.16 – Ендпоінт підключення користувача (студента) до сервера

Концептуально, ці два ендпоінти майже нічим не відрізняються, обидва додають з’єднання до словників, та обидва опитують з’єднання, та якщо з’єднання закриті, то воно видаляє їх зі словників, але є одне виключення, що коли підключається клієнт, то, якщо його віртуальна машина вже підключена до сервера – клієнт буде про це сповіщений, і навпаки, коли підключається віртуальна машина до сервера, то клієнт буде сповіщений про цю подію. Це

потрібно, щоб розуміти, чи підключена віртуальна машина, і чи можуть бути запущені на ній тести.

Також реалізовані допоміжні функції для роботи за схемою коли є окремі підключення віртуальної машини, та користувача.

```python
def send_client(email: str, event: Event):
    if email not in clients:
        return

    clients[email].send(json.dumps(event))
```

Рисунок 4.17 – Допоміжна функція відправки повідомлення користувачу через WebSocket

```python
def send_server(email: str, data: bytes):
    if email not in servers:
        return

    try:
        servers[email].send(data)
    except ConnectionClosed:
        print(f"server connection with {email} closed.")
        del servers[email]
        return
```

Рисунок 4.18 – Допоміжна функція відправки повідомлення віртуальній машині користувача

```python
def receive_server(email: str) -> bytes:
    if email not in servers:
        return b""

    try:
        return servers[email].receive()
    except ConnectionClosed:
        print(f"server connection with {email} closed.")
        del servers[email]
```

Рисунок 4.19 – Допоміжна функція для отримання даних з віртуальної машини

Тепер сервер, може в реальному часу розподіляти повідомлення між користувачем, та його віртуальною машиною. Це потрібно для того, щоб можна було запускати тести, та одразу отримати відповіді від віртуальної

машини, про їх статус, а також відслідковувати, чи доступна віртуальна машина для запуску тестів.

Коли є механізм комунікації між користувачем, та його віртуальною машиною, потрібно зробити механізм запуску тестів.

```
241 @app.route("/tests/run/<email>", methods=["POST"])
242 def run_tests(email):
243     global servers
244 
245     if email not in servers:
246         return jsonify({
247             "message": "server is offline"
248         }), 400
249 
250     send_server(email, request.data)
251     data = json.loads(receive_server(email))
252     send_client(email, {"event": "test_result", "args": data})
253 
254     return jsonify(data)
```

Рисунок 4.20 – Ендпоінт для запуску тестів

Ендпоінт для запуску тестів, приймає параметром ім’я тесту, що потрібно запусти, та пошту користувача, що запускає цей тест, оскільки пошта є скрізним ключем між користувачем, та його віртуальною машиною. Суть роботи цього ендпоінт, перевірити, чи є для цього користувача підключення до віртуальною машини, відправити на віртуальну машину тест, що потрібно запустити, віртуальна машина має відповісти результатом тесту, та цей результат передається назад на клієнт. Коли клієнт отримує результат запуску всіх тестів, він передає назад на сервер запит, для оцінки.

Лишилось дати можливість серверу завантажувати собі тести для роботи. Для цього потрібно реалізувати ще один ендпоінт, котрий просто дасть можливість скачувати тести, котрі потім будуть запущені на віртуальній машині.

```
@app.route("/tests/download/<task_id>/<filename>", methods=["GET"])
def download_test(task_id, filename):
    return send_file(os.path.join(
        app.config['MEDIA_ROOT'],
        task_id,
        filename
    ))
```

Рисунок 4.21 – Ендпоінт для скачування тестів

Ендпоінт для скачування тестів потрібен, щоб віртуальна машина могла завантажити собі тести для запуску.

Після цього, всі ендпоінти реалізовано, та застосунок вважається робочим.

4.2 Реалізація клієнтської частини LTI

Сервер потрібен для бізнес-логіки, але клієнту буде не зручно руками робити запити на сервер, та самому зв’язати використання кожного ендпоінту, тому потрібно реалізувати клієнтську частину, що сама зв’яже всі ендпоінти, в один працюючий продукт.

Для відмальовки інтерфейсу використовувався шаблонізатор Jinja2. Шаблонізатори дозволяють в тексті використовувати функції, цикли, тощо, тобто на основі шаблону, генерувати різне відображення основане на вхідних даних.

Для задоволення потреб LTI, коли є дві сторінки, сторінки вчителя, та сторінки учня, достатньо створити три шаблони, перший шаблон – це шаблон вчителя, другий учня, та третій – це шаблон котрий містить у собі те що спільне у обох шаблонах. Загальну структуру шаблонів можна побачити на рисунку 4.22

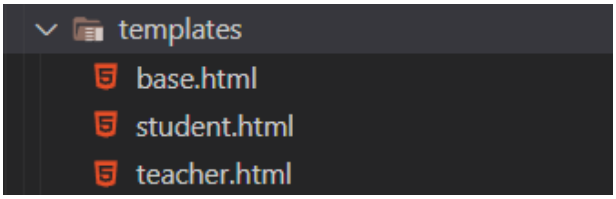

Рисунок 4.22 – Структура шаблонів

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ page_title }}</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/bootstrap.css') }}">
    <script type="text/javascript" defer src="{{ url_for('static', filename='js/bootstrap.js') }}"
            charset="utf-8"></script>
    {% block head %}
    {% endblock %}
</head>
<body>
{% block content %}
{% endblock %}
</body>
</html>
```

Рисунок 4.23 – Вміст базового шаблону

Базовий шаблон містить у собі посилання на CSS та JS файли, тобто на статику, котра загальна як для учня, так і для вчителя, і надає можливість додавати якісь компоненти в окремих шаблонах, такі як додаткові стилі, скрипти, чи вставити свій контент в блок тіла.

Сторінка вчителя складається з форми завантаження тестів, та деяких додаткових скриптів.

```
{% extends 'base.html' %}

{% block head %}
    <script>
        const taskId = +"{{ task_id }}";
        let dataIndex = +"{{ tests|length }}";
    </script>
    <script type="text/javascript" defer src="{{ url_for('static', filename='js/test add.js') }}"
            charset="utf-8"></script>
    <script type="text/javascript" defer src="{{ url_for('static', filename='js/test submit.js') }}"
            charset="utf-8"></script>
{% endblock %}
```

Рисунок 4.24 – Додавання скриптів на сторінку вчителя

На сторінці вчителя усього два скрипти, скрипт *test_add.js* (див. Додаток Б) потрібен просто для того, щоб зробити додавання тестів гарним, та динамічним, а *test_submit.js* (див. Додаток В) потрібен для того, щоб скомпанувати всю інформацію, що вніс вчитель, та відправити на сервер.

```
{% block content %}
    <h1>Teacher {{ email }}</h1>
    <form id="tests-form" class="d-flex justify-content-center align-items-center flex-column">
        <div class="mb-3">
            <label class="form-label" for="description">Task Description:</label>
            <textarea class="form-control" id="description" name="description" cols="50" rows="10">{{ task.description }}</textarea>
        </div>
        <div class="mb-3">
            <h5>Add tests:</h5>

            <div id="tests">
                {% for test in tests %}
                    <div class="d-flex flex-row justify-content-between mb-3" id="test-{{ loop.index0 }}">
                        <input required type="text" name="test-description-{{ loop.index0 }}" class="form-control mx-2"
                               placeholder="Test description" value="{{ test.description }}"/>
                        <input required type="file" name="test-file-{{ loop.index0 }}" id="test-file-{{ loop.index0 }}"
                               class="form-control mx-2"/>
                        <button class="btn btn-danger" onclick="handleRemoveClick(event)">-</button>
                    </div>
                {% endfor %}
                {% with tests_len = tests|length %}
                    <div class="d-flex flex-row justify-content-between mb-3" id="test-{{ tests_len }}">
                        <input required type="text" name="test-description-{{ tests_len }}" class="form-control mx-2"
                               placeholder="Test description"/>
                        <input required type="file" name="test-file-{{ tests_len }}" id="test-file-{{ tests_len }}"
                               class="form-control mx-2"/>
                        <button id="add-btn" class="btn btn-success">+</button>
                    </div>
                {% endwith %}
            </div>
        </div>
        <input type="submit" id="submit" class="btn btn-success mb-4" value="Submit">
        <div id="upload-result"></div>
    </form>
{% endblock %}
```

Рисунок 4.25 – Блок контенту для сторінки вчителя

Блок контенту підмальовує форму вводу, всі додані тести, та дає можливість редагувати вже створені тести.

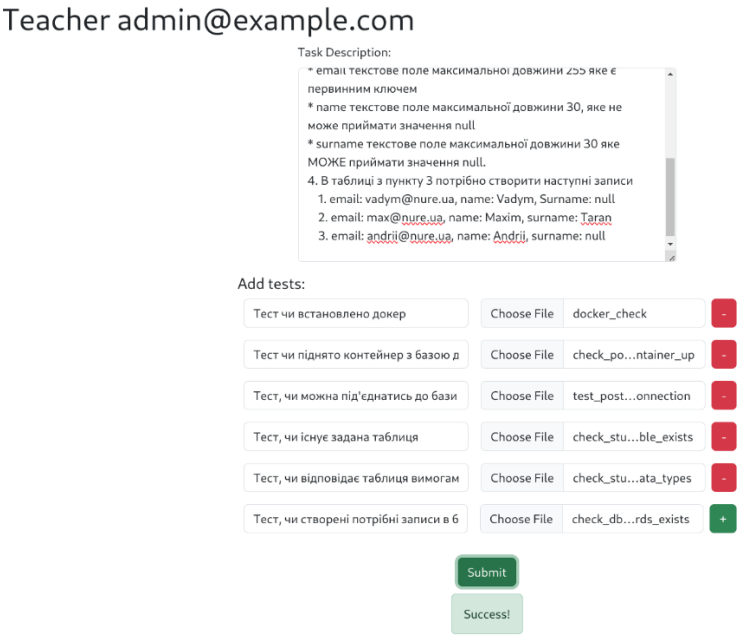

Рисунок 4.26 – Кінцевий вигляд сторінки вчителя

Кнопка плюсика дозволяє розширити кількість тестів, кнопка мінуса, дозволяє видалити тест, саме за це відповідає скрипт *test_add.js*.

```
{% extends 'base.html' %}
{% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
    <script>
        const taskId = "{{ task_id }}";
        const email = "{{ email }}";
        const launchID = "{{ launch_id }}";
    </script>
    <script type="text/javascript" defer src="{{ url_for('static', filename='js/socket.js') }}"
            charset="utf-8"></script>
    {% if task %}
        <script type="text/javascript" defer src="{{ url_for('static', filename='js/test_runner.js') }}"></script>
    {% endif %}
{% endblock %}
{% block content %}
    <div class="d-flex flex-column justify-content-center align-items-center">
        <h1>Test for: {{ email }}</h1>
        <h2>Server state: <span style="color: red" id="server-state">off</span></h2>
        {% if task %}
            <pre>{{ task.description }}</pre>
            <h3>Tests</h3>
            <ul id="tests">
                {% for test in tests %}
                    <li id="test-{{ test.file_name }}" data-test-name="{{ test.file_name }}">
                        <span class="test-desc">{{ test.description }}</span>
                        <div class="result d-flex"></div>
                    </li>
                {% endfor %}
            </ul>
            <button id="btn-run" class="btn btn-success">Run tests</button>
        {% else %}
            <h3>There is no task yet :(</h3>
        {% endif %}
    </div>
{% endblock %}      Vadym Tishchenko, 3 months ago * Add lti
```

Рисунок 4.27 – Реалізація сторінки студента

На сторінці студента просто відмальовуються тести, що додав вчитель, статус його віртуальної машини (вона увімкнена, чи вимкнена), та кнопка, що відповідає за запуск тестів. Також підключено додаткові скрипти *test_runner.js* (див. Додаток Д), та *socket.js*, що відповідальний за комунікацію з сервером через WebSocket (див Додаток Г).

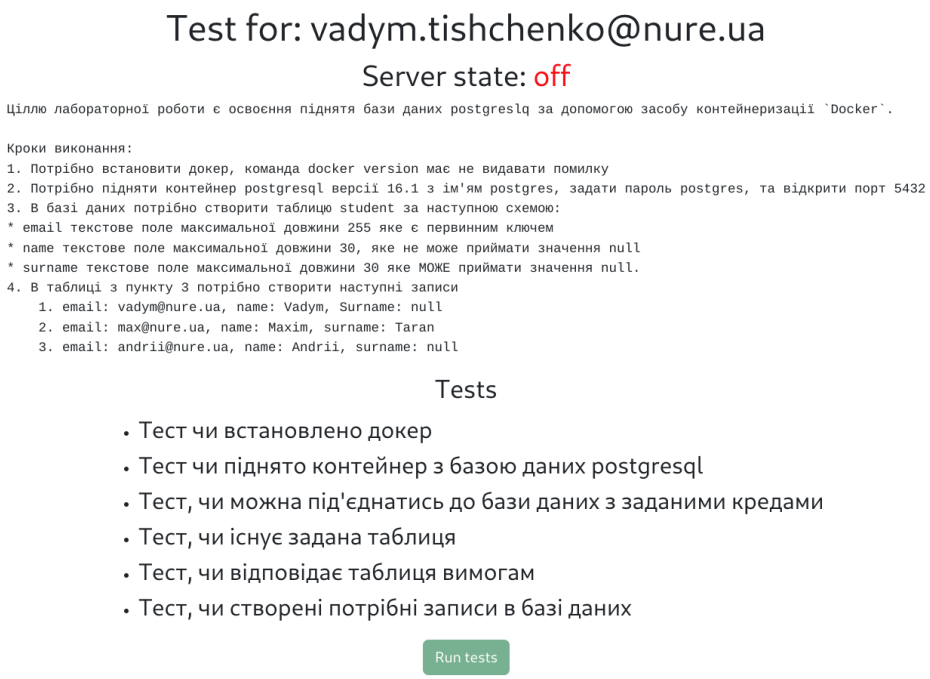

Рисунок 4.27 – Сторінка студента

4.3 Реалізація рушія на стороні віртуальної машини студента

Сам рушій реалізовано на мові програмування Go, сама по собі мова дуже повна функціоналом, що потрібен для виконання завдання, але потрібно встановити додаткову залежність для WebSocket.

```
1  module github.com/qerdcv/masters-project/worker
2
3  go 1.20
4
   Check for upgrades | Upgrade transitive dependencies | Upgrade direct dependencies
5  require github.com/gorilla/websocket v1.5.0
6
```

Рисунок 4.28 – Встановлення залежності для WebSocket

Всі інші аспекти можна реалізувати без проблем інструментами самої мови.

Перше, що потрібно зробити – це зробити функціонал для конфігурування рушія, оскільки він буде запускатись на різних машинах, та у різних серидовищах.

```
22  var (
23      userEmail,
24      ltiHost,
25      testsDir,
26      env string
27  )
28
29  func init() {
30      userEmail = os.Getenv("USER_EMAIL")
31      ltiHost = os.Getenv("LTI_HOST")
32      testsDir = os.Getenv("TESTS_DIR")
33      env = os.Getenv("ENV")
34  }
```

Рисунок 4.29 – Конфігурування рушія

Процес конфігурації досить простий, всього потрібно знати 4 параметри, email користувача, щоб була можливість ідентифікувати машину цього користувача на сервері. *ltiHost* – інформація про хост на якому піднято LTI, в локальному оточенні – це буде ip-адреса локального хоста (127.0.0.1), а ось в продакшн оточенні, це вже буде днс-ім'я серверу, на котрому розгорнуте LTI. *testsDir* – папка куди будуть завантажені тести, сама папка буде всередині */tmp,* щоб не накопичувати сміття на віртуальній машині користувача. Env – оточення в котрому запущено рушій, від того, в якому оточенні запущено рушій залежать деякі аспекти його роботи.

Також реалізовані деякі допоміжні функції.

```
36  func logErr(msg string) {
37      log.Println("ERROR:", msg)
38  }
```

Рисунок 4.30 – Реалізація функції для логування помилок

```
40  func fatal(msg string) {
41      logErr(msg)
42      os.Exit(1)
43  }
```

Рисунок 4.31 – Реалізація функції для логування помилки, та виходу з програми з кодом 1 (помилка)

Старт програми відбувається в функції main

```
148 func main() {
149     scheme := "ws"
150     if env == "prod" {
151         scheme = "wss"
152     }
153
154     u := url.URL{Scheme: scheme, Host: ltiHost, Path: "/ws/server/" + userEmail}
155     c, _, err := websocket.DefaultDialer.Dial(u.String(), nil)
156     if err != nil {
157         fatal(err.Error())
158     }
```

Рисунок 4.32 – Старт програми

При старті програма під’єднується до LTI через WebSocket за допомогою ідентифікатору (email користувача). Та додатково змінює протокол з ws на wss якщо це продакшн оточення.

Після під’єднання до серверу, запускається 2 фонових процеси, один оброблює повідомлення з серверу, другий опитує сервер, чи з’єднання живе.

```
160     ctx, cancel := context.WithCancel(context.Background())
161     interrupt := make(chan os.Signal, 1)
162     signal.Notify(interrupt, os.Interrupt, syscall.SIGINT, syscall.SIGTERM)
163
164     go proceedMessages(ctx, c)
165     go func(ctx context.Context) {
166         t := time.NewTicker(5 * time.Second)
167         defer t.Stop()
168
169         for {
170             select {
171             case <-ctx.Done():
172                 return
173             case <-t.C:
174                 if err = c.WriteMessage(websocket.PingMessage, nil); err != nil {
175                     logErr(err.Error())
176                     return
177                 }
178             }
179         }
180     }(ctx)
```

Рисунок 4.33 – Запуск фонових процесів

Після запуску фонових процесів, програма слухає системні події на елемент завершення програми, та коли ця подія приходить, програма посилає на сервер подію закриття з’єднання, закриває з’єднання, та виходить з програми, цей процес називається *grateful shutdown*. Тобто програма не просто закривається, а додатково повідомляє усіх учасників взаємодії про те, що комунікація завершена.

```
182     <-interrupt
183
184     if err = c.WriteMessage(
185         websocket.CloseMessage,
186         websocket.FormatCloseMessage(websocket.CloseNormalClosure, "")); err != nil {
187         log.Println("write close:", err)
188         return
189     }
190
191     if err = c.Close(); err != nil {
192         logErr("close connection")
193     }
194
195     cancel()
196     log.Println("Exit..")
197 }
198
```

Рисунок 4.34 – Реалізація *grateful shutdown*

Після запуску фонових процесів, програма просто чекає, поки не прийде подія на її вимкнення, але в фоні відбувається вся бізнес-логіка, а особливо в горутині *proceedMessages*. Ця горутина відповідальна за обробку подій з серверу.

```
50  func proceedMessages(ctx context.Context, c *websocket.Conn) {
51      for {
52          select {
53          case <-ctx.Done():
54              return
55          default:
```

Рисунок 4.35 – Реалізація обробки подій, слухання контексту

Обробка повідомлення складається з нескінченного циклу, котрий зупиниться, або коли станеться якась помилка, і програма вийде з кодом 1, або програма отримає подію завершення, та сама зупинить цей цикл (<-*ctx.Done()*). Якщо програма не отримує повідомлення про завершення – вона намагається прочитати дані з веб-сокету, оскільки формат даних на серверу, та клієнту (віртуальній машині) узгоджений, його потрібно підігнати під об’єкти, котрі можна використовувати всередині мови, бо саме повідомлення – це набір байтів, в котрому находиться JSON строка.

```
56  _, message, err := c.ReadMessage()
57  if err != nil {
58      logErr(err.Error())
59      return
60  }
61
62  var test testRequest
63  if err = json.Unmarshal(message, &test); err != nil {
64      logErr(err.Error())
65      continue
66  }
```

Рисунок 4.36 – Вичитка байтів з веб-сокету, та приведеня їх до структури

```
45  type testRequest struct {
46      TaskID string `json:"task_id"`
47      Test   string `json:"test"`
48  }
```

Рисунок 4.37 – Структура повідомлення з веб-сокету

Після отримання запиту на віртуальну машину, йде валідація, чи передано ідентифікатор завдання, для котрого запускається тест, якщо ні – то просто читається наступне повідомлення, бо не відомо, що саме за тест потрібно запустити, якщо так, то ініціалізується запуск тесту.

```
68  if test.TaskID == "" {
69      continue
70  }
71
72  result, err := runTest(test.TaskID, test.Test)
73  if err != nil {
74      logErr(err.Error())
75      continue
76  }
```

Рисунок 4.37 – Валідація повідомлення, та запуск тесту

```
100 func runTest(taskID string, test string) (testResult, error) {
101     scheme := "http"
102     if env == "prod" {
103         scheme = "https"
104     }
```

Рисунок 4.38 – Ініціалізація запуску тесту

Перший етап запуску тесту – це вибір протоколу спілкування, *http* чи *https*.

Після того, як протокол спілкування обрано, формується посилання на ресурс, scheme + host + urn, і по цьому посилання програма йде на сервер, щоб завантажити тест.

```
106     u := url.URL{Scheme: scheme, Host: ltiHost, Path: "/tests/download/" + taskID + "/" + test}
107     resp, err := http.Get(u.String())
108     if err != nil {
109         return testResult{}, fmt.Errorf("http get: %w", err)
110     }
111
112     defer resp.Body.Close()
```

Рисунок 4.39 – Формування посилання на ресурс, та завантаження тесту

Щоб завантажити тест, потрібно спочатку створити файл, в котрий від буде завантажений.

```
114     testPath := path.Join(testsDir, fmt.Sprintf("%s_%s", test, taskID))
115     f, err := os.Create(testPath)
116     if err != nil {
117         return testResult{}, fmt.Errorf("create test: %w", err)
118     }
```

Рисунок 4.40 – Створення файлу, куди буде завантажено тест

Після створення файлів, за допомогою інструменту мови (io.Copy) байти з відповіді серверу, копіюються до файлу, котрий був створений, та файл закривається, щоб байти котрі лишились в буфері дописались до файлу.

```
120     if _, err = io.Copy(f, resp.Body); err != nil {
121         return testResult{}, fmt.Errorf("write test: %w", err)
122     }
123
124     if err = f.Close(); err != nil {
125         return testResult{}, fmt.Errorf("close file: %w", err)
126     }
```

Рисунок 4.41 – Копіювання байтів з відповіді серверу, до файлу

Після того, як тест було завантажено, він встановлюється на видалення після завершення тесту, і змінюється модифікація файлу на запускаємий, тобто такий файл, котрий можна запустити.

```
128     defer os.Remove(testPath)
129
130     if err = os.Chmod(testPath, fs.ModePerm); err != nil {
131         return testResult{}, fmt.Errorf("change test mode: %w", err)
132     }
```

Рисунок 4.42 – Видалення файлу після тесту, та зміна модифікатора файлу на *executable*

Коли є готовий тест для запуску, потрібно сформувати результат запуску, та запустити сам тест, якщо тест виконався з помилкою, то статус тесту помічається як failed і в поле Error записується причина того, чому тест впав, якщо тест пройшов без помилок – то success.

```
134     result := testResult{
135         Name: test,
136     }
137
138     if out, execErr := exec.Command(testPath).CombinedOutput(); execErr != nil {
139         result.Result.Status = "failed"
140         result.Result.Error = fmt.Sprintf("%s: %s", string(out), execErr.Error())
141         return result, nil
142     }
143
144     result.Result.Status = "success"      Vadym Tishchenko, 3 months ago • Add lti
```

Рисунок 4.43 – Формування результату запуску та запуск самого тесту

Після запуску тесту, сформований результат повертається із функції.

```
144     result.Result.Status = "success"
145     return result, nil
146 }
```

Рисунок 4.44 – Повернення результату запуску тесту

```
type testResult struct {
    Name   string `json:"name"`
    Result struct {
        Status string `json:"status"`
        Error  string `json:"error,omitempty"`
    } `json:"result"`
}
```

Рисунок 4.45 – Структура результату тесту

Після того, як отримано результат запуску тесту – він перегоняється назад в JSON та відправляється назад на сервер.

```
        result, err := runTest(test.TaskID, test.Test)
        if err != nil {
            logErr(err.Error())
            continue
        }

        b, err := json.Marshal(result)
        if err != nil {
            logErr(err.Error())
            continue
        }

        if err = c.WriteMessage(websocket.TextMessage, b); err != nil {
            logErr(err.Error())
            continue
        }
```

Рисунок 4.46 – Запаковка результату тесту в JSON та відправка на сервер.

Після того, як результат запуску повернувся на сервер – сервер його опрацьовує, та повертає результат користувачу, що ініціював запуск тесту.

5 ПРАКТИЧНЕ ЗАСТОСУВАННЯ

5.1 Підняття лмс в локальному оточення

Як було сказано раніше, за СКН було взяти відкриту реалізацію під ім’ям Moodle.

Для того щоб його розгорнути локально, достатньо базових навичок роботи з докером. Сама СКН написана на мові програмування PHP, і потребує як залежність базу даних MySQL. Тобто підняття мудл можна почати з підняття бази даних.

```yaml
version: '3'

services:
  db:
    image: mysql:8.1.0
    restart: always
    environment:
      MYSQL_DATABASE: 'moodle'
      MYSQL_USER: 'moodle'
      MYSQL_PASSWORD: 'moodle'
      MYSQL_ROOT_PASSWORD: 'toor'
    ports:
      - '3306:3306'
    expose:
      - '3306'
    volumes:
      - db:/var/lib/mysql
    networks:
      - moodle
```

Рисунок 5.1 – Конфігурація бази даних

На рисунку 5.1 можна побачити конфігурацію бази даних, котра включає в себе MySQL версії 8.1.0, конфігурація користувача бази даних, портів котрі будуть доступні з локальної машини поза мережі докеру,

волюмів для збереження даних бази, і мережі, щоб об’єднати базу даних з мудлом, і щоб вони один одного бачили.

```
35  moodle:
36    image: 'webdevops/php-apache:8.2-alpine'
37    environment:
38      - MOODLE_DBHOST=db
39      - MOODLE_DBPORT=3306
40      - MOODLE_DBUSER=moodle
41      - MOODLE_DBPASS=moodle
42      - MOODLE_DBNAME=moodle
43    ports:
44      - '80:80'
45      - '443:443'
46    networks:
47      - moodle
48    volumes:
49      - type: bind
50        source: /home/qerdcv/own_projects/php/moodle_data
51        target: /moodledata/moodledata
52      - type: bind
53        source: /home/qerdcv/own_projects/php/moodle
54        target: /app
55    depends_on:
56      - db
```

Рисунок 5.2 – Конфігурація запуску Moodle

На рисунку 5.2 можна побачити базову конфігурацію Moodle. Вона включає в себе налаштування змінних оточення, а саме конфігурація параметрів доступу до бази даних, відкриті порти для доступу поза докером, мережу, котра включає в себе базу даних, щоб мудл та база могли спілкуватись всередині мережі докеру, та волюми, щоб зберегти файли мудлу. Також додатково додане поле «*depends_on*», котре позначає, що старт мудлу залежить від бази даних, тобто поки база даних не буде готова приймати нові з’єднання, мудл не підіймиться. Важливо відмітити, що підіймається не сам мудл, а веб-сервер, котрий вміє виконувати PHP, в даному випадку Apache, і зможе віддавати файли на запит користувача.

І також можна побачити конфігурацію волюмів, та мережі докеру, рисунок 5.3.

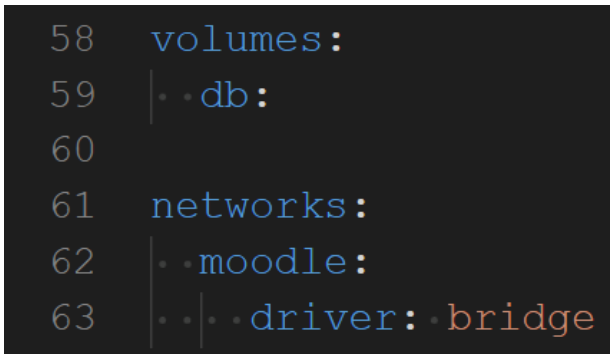

Рисунок 5.3 – Налаштування волюмів бази, та мережі

Після цих простих маніпуляцій, можна запускати Moodle (рисунок 5.4).

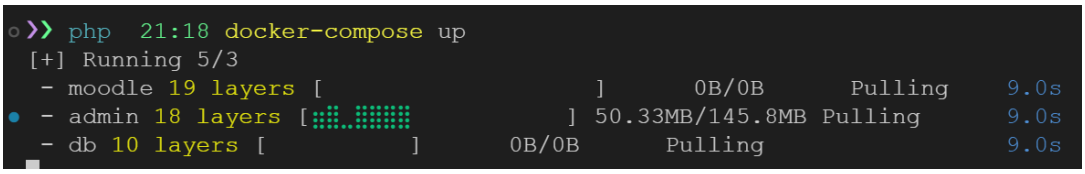

Рисунок 5.4 – Процес запуску Moodle

Після того, як все запуститься (рисунок 5.5), можна рухатись далі.

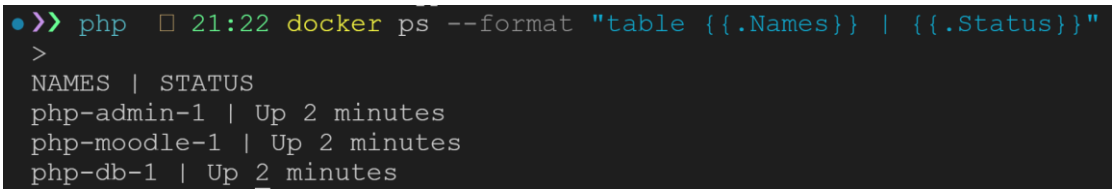

Рисунок 5.5 – Статуси запущених контейнерів

5.2 Конфігурація Moodle

Після запуску, Moodle зустрічає сторінкою конфігурації (рисунок 5.6).

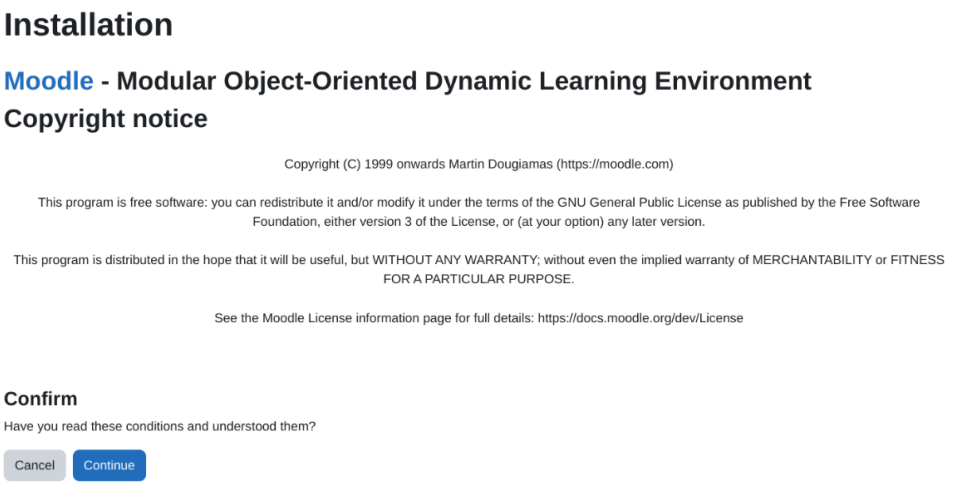

Рисунок 5.6 – Стартова сторінка Moodle

На даній сторінці, мудл пропонує ознайомитись з тим, що за продукт являється мудл, та копірайтом.

Після прочитання умов, та погодження з ними, мудл почне встановлення залежностей, тобто того, без чого його робота буде не можлива (рисунок 5.7).

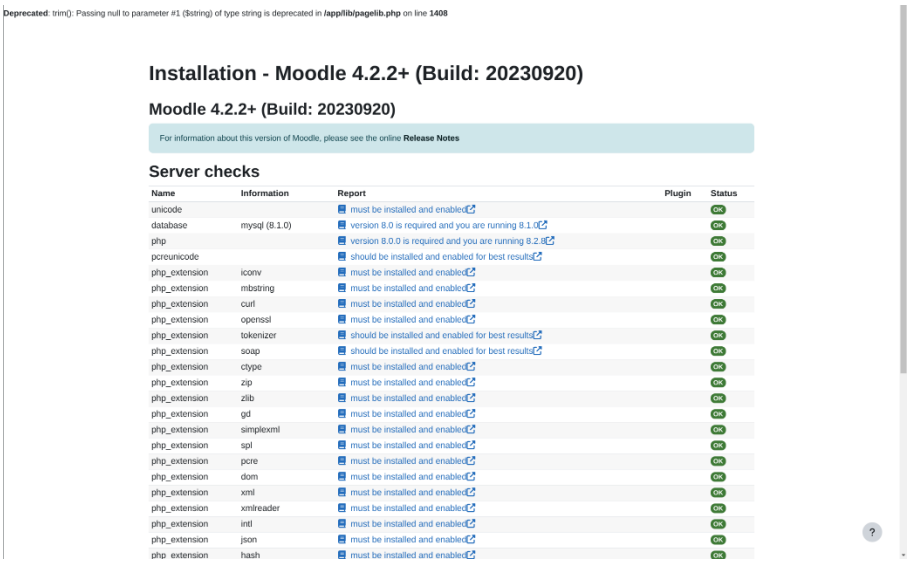

Рисунок 5.7 – Встановлення залежностей Moodle

Після встановлення залежностей мудл попросить створити користувача-адміністратора (рисунок 5.8).

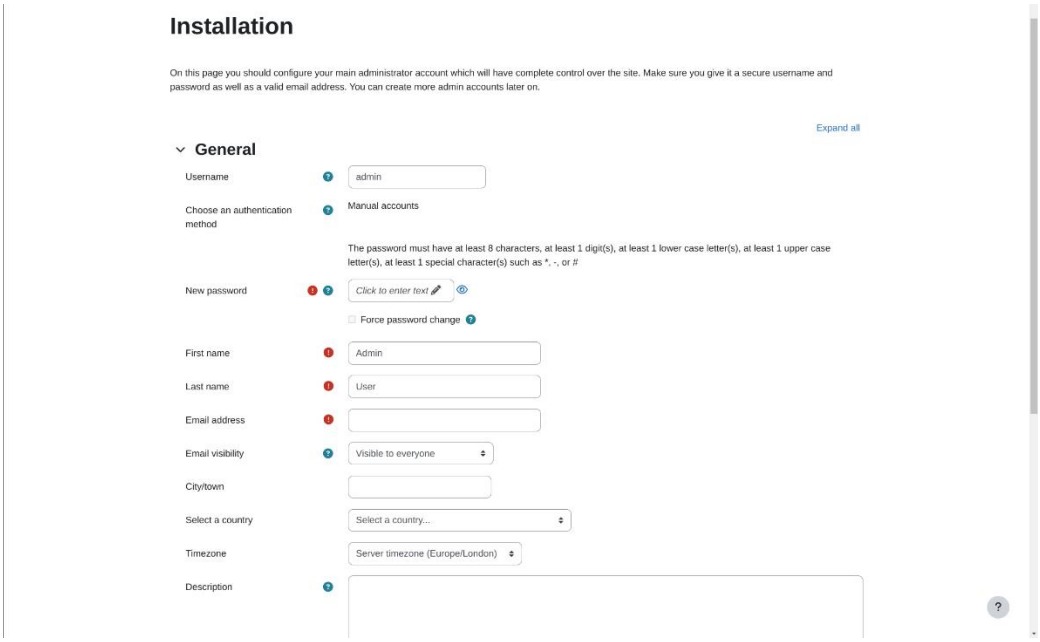

Рисунок 5.8 – Створення адміністративного користувача

Після всіх налаштувань, нарешті з’явиться головна, та вже знайома головна сторінка мудлу (рисунок 5.9).

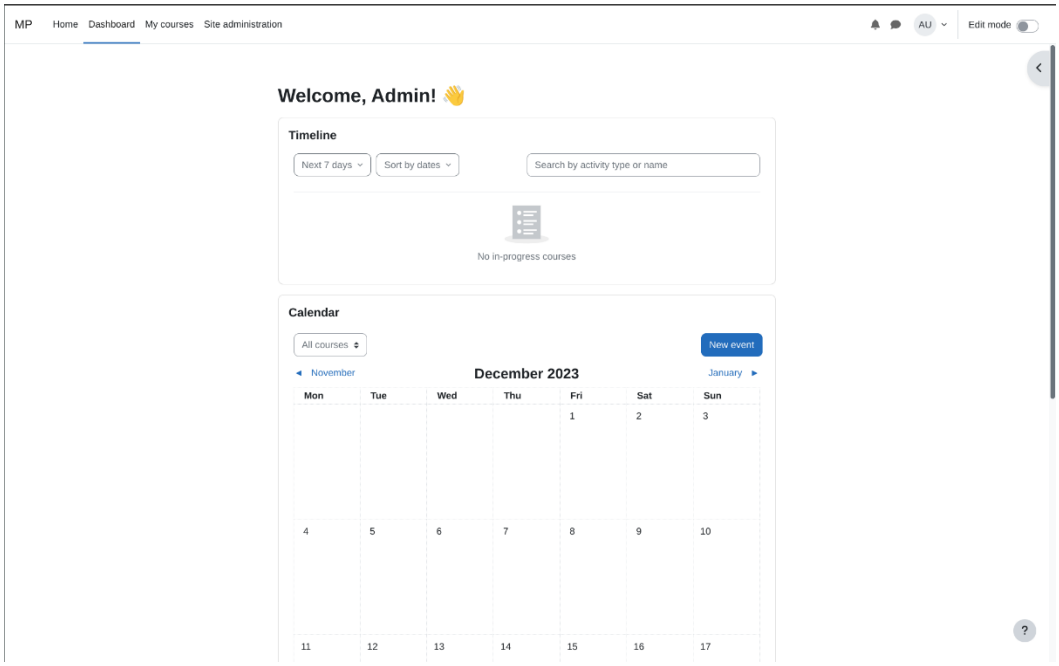

Рисунок 5.9 – Говона сторінка Moodle

Після базового налаштування, потрібно створити тестовий курс, на котрому будуть проводитись експерименти.

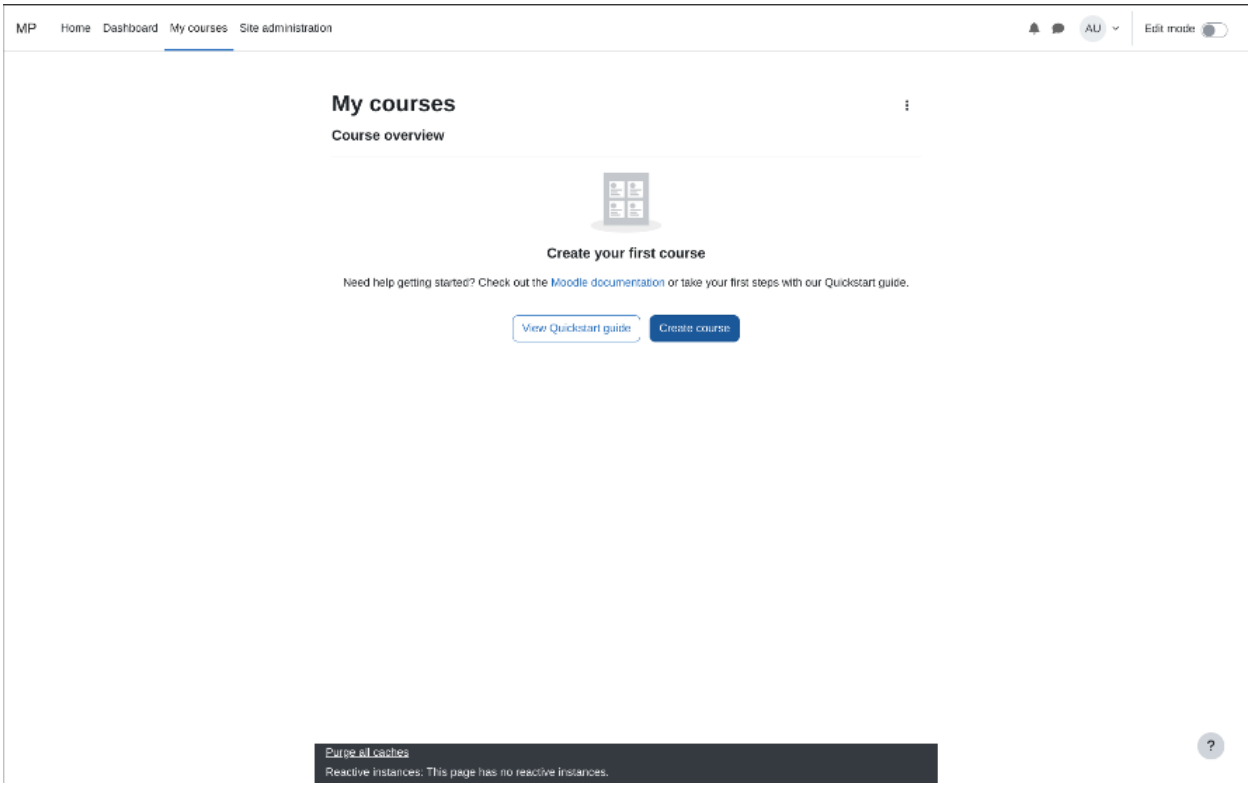

Рисунок 5.10 – Створення тестового курсу

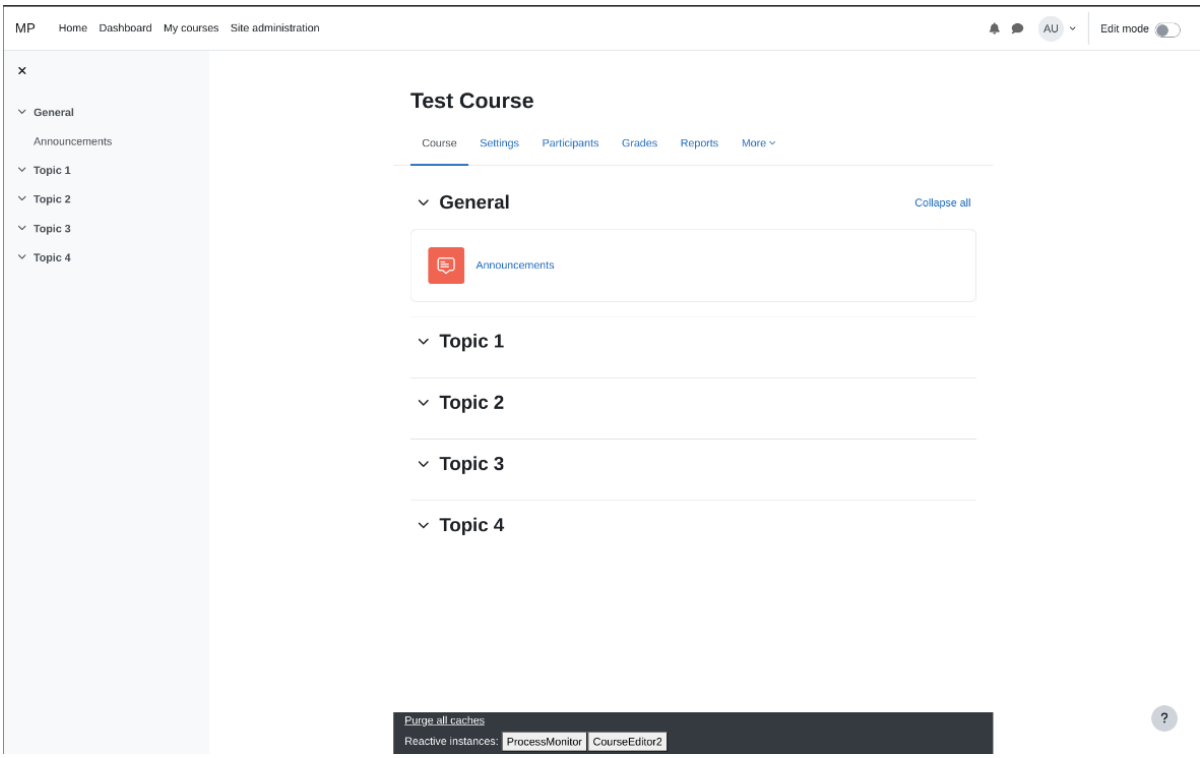

Рисунок 5.11 – Сторінка новостворенного курсу

5.3 Налаштування OIDC

OIDC, або OpenID Connect – це стандарт, розроблений для обміну інформацією про юзера між аутентифікаційними сервісами, або простими словами, OIDC надає можливість людині, котра зареєстрована в Google, аутентифікуватись на іншому ресурсі за допомогою даних, котрі віддасть Google. Для задач котрі вирішуються, потрібна можливіть аутентифікації в СКН за допомогою облікового запису на домені nure.ua.

OIDC побудовано на основі іншого стандарту авторизаціх OAuth2, і працює за схожим принципом, який зображено на рисунке 5.12.

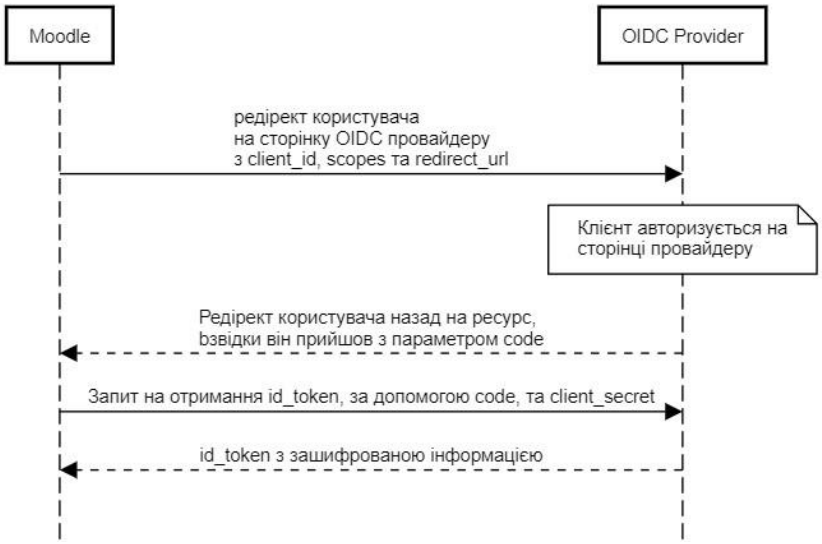

Рисунок 5.12 – Схема роботи OIDC

В чому сенс, на першому етапі OIDC провайдер видає своєму клієнту, в даному випадку Moodle, пару значень, *client_id*, та *client_secret*. За допомогою *client_id*, клієнт може звернутись до OIDC провайдеру з запитом на отримання даних, а з *client_secret* клієнт може власне отримати запрошені дані.

Першим етапом в OIDC є редірект користувача на сторінку OIDC провайдера. При редіректі вказується *client_id*, щоб OIDC провайдер зрозумів, хто до нього прийшов, *scopes*, щоб провайдер зрозумів, який набір даних від нього хочуть, та *redirect_url*, щоб провайдер знав куди перенаправляти користувача, після того, як аутентифікація пройде успішно.

Коли користувач опиняється на сторінці OIDC провайдеру, він проходить на ньому авторизацію. Після успішної авторизації, провайдер перенаправляє користувача назад на клієнт за *redirect_url* котру надав клієнт, і додає в запит code, це код по якому можна звернутись до OIDC провайдеру за даними.

Після того, як користувач потрапляє на сторінку клієнту, клієнт ходить на OIDC провайдер щоб отримати *id_token*, в запиті вказуючи code котрий клієнт отримав при перенаправлені користувача, та *client_secret*, котрий клієнт отримав на OIDC провайдері.

Завдяки тому, що це доволі поширений стандарт, його реалізації вже можна найти в готовому вигляді, і таке саме вже є в мудлі.

Щоб налаштувати аутентифікацію через сторонній ресурс, потрібно спочатку перейти на сторінку *Site administration -> Server -> OAuth2 Services* (рисунок 5.13).

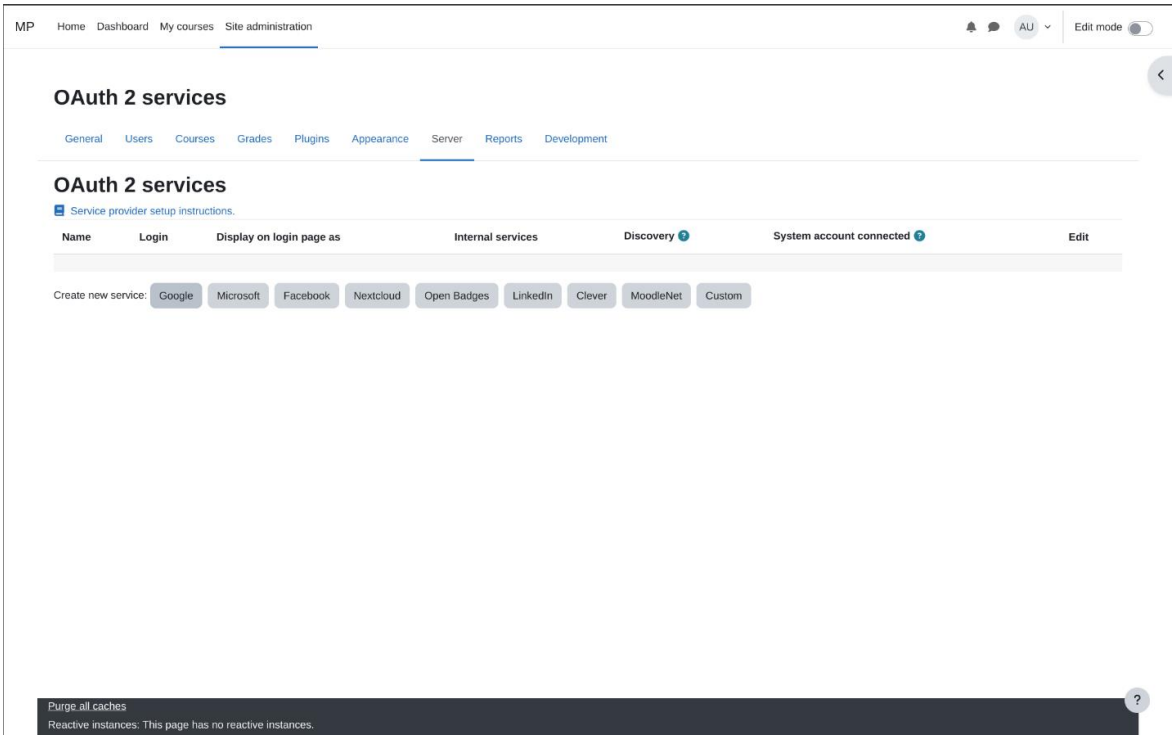

Рисунок 5.13 – Сторінка налаштування аутентифікації через сторонній сервіс

На цій сторінці потрібно обрати Google.

Після чого перейти на сторінку гугла, та отримати *client_id*, та *client_secret.*

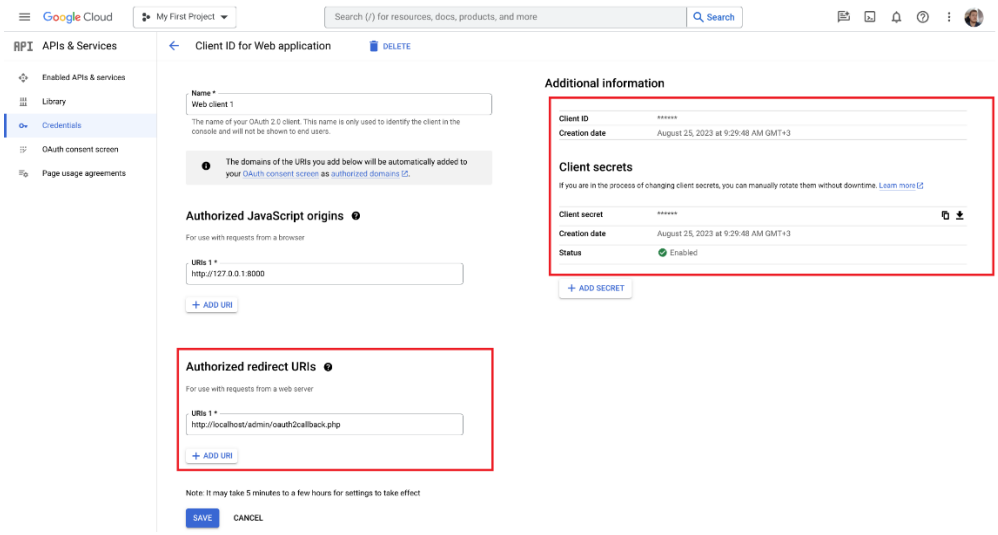

Рисунок 5.14 – Отримання *client_id* та *client_secret*

Також на сторінці гугла, додатково вказується дозволені посилання на переадресацію користувача, для більшої безпеки.

Отримані дані вводяться на сторінці мудлу, та додатково прописується, з якого домену дозволені користувачі (рисунок 5.15).

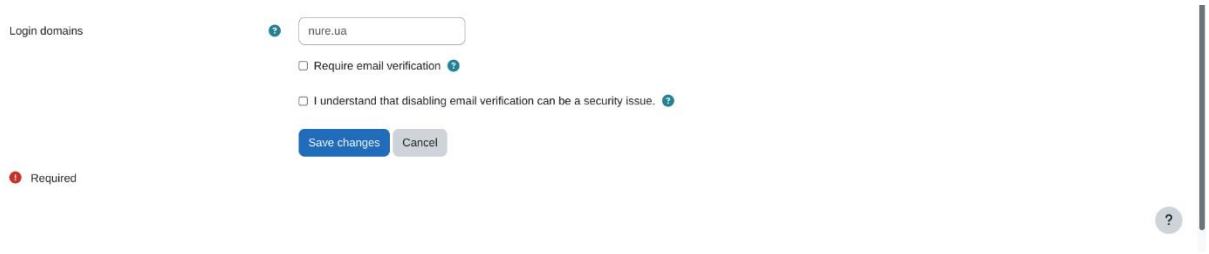

Рисунок 5.15 – Налаштування користувачів тільки з домену nure.ua

Після налаштування OAuth2 сервісу, його потрібно увімкнути в Moodle, та поставити, як пріоритетним.

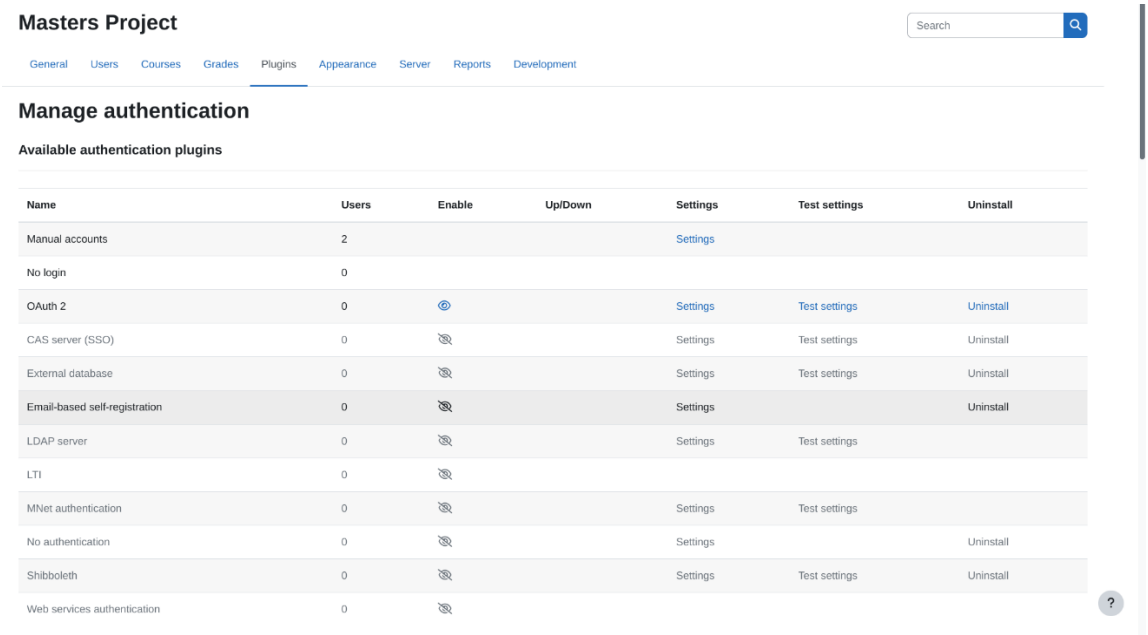

Рисунок 5.16 – Увімкнення плагіну аутентифікації OAuth2

Після цих маніпуляцій, при аутентифікації в системі, з’явиться можливість використати Google (рисунок 5.17).

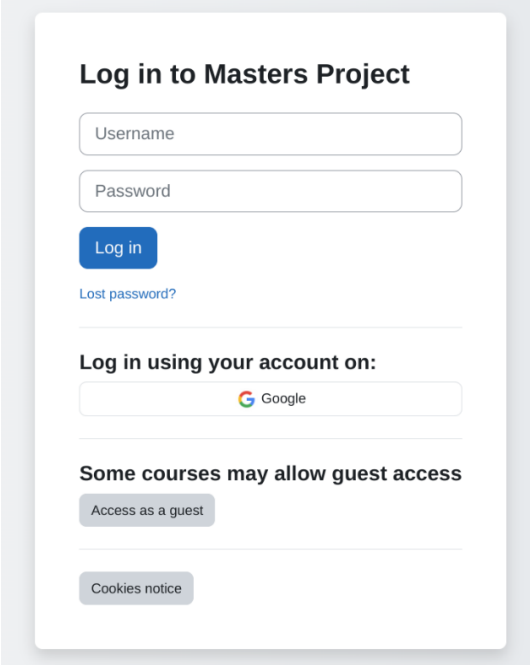

Рисунок 5.17 – Відображення кнопки соціальної мережі для аутентифікації

Перевірка роботи OAuth2 з корпоративної пошти nure.ua (рисунок 5.18).

Рисунок 5.18 – Успішна аутентифікація з корпоративної пошти nure.ua

Перевірка того, що аутентифікуватись можна тільки за допомогою корпоративної пошти nure.ua (рисунок 5.19).

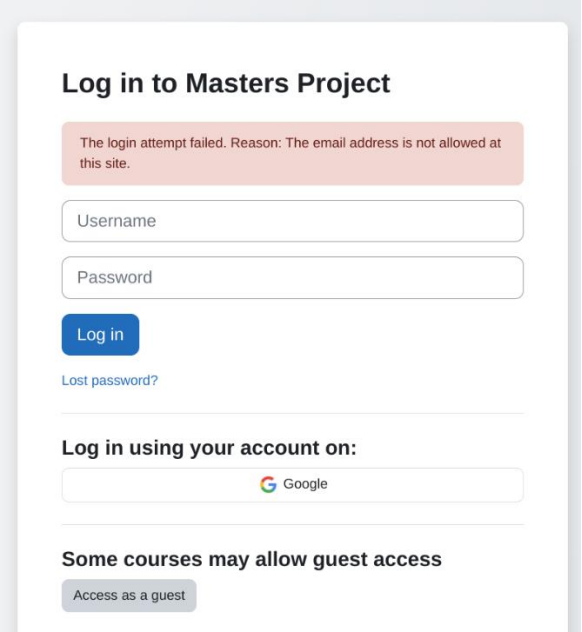

Рисунок 5.19 – Помилка аутентифікації з не корпоративної пошти

5.4 Додавання LTI до курсу

На сторінці курсу в режимі редагування додається новий ресурс (рисунок 5.20).

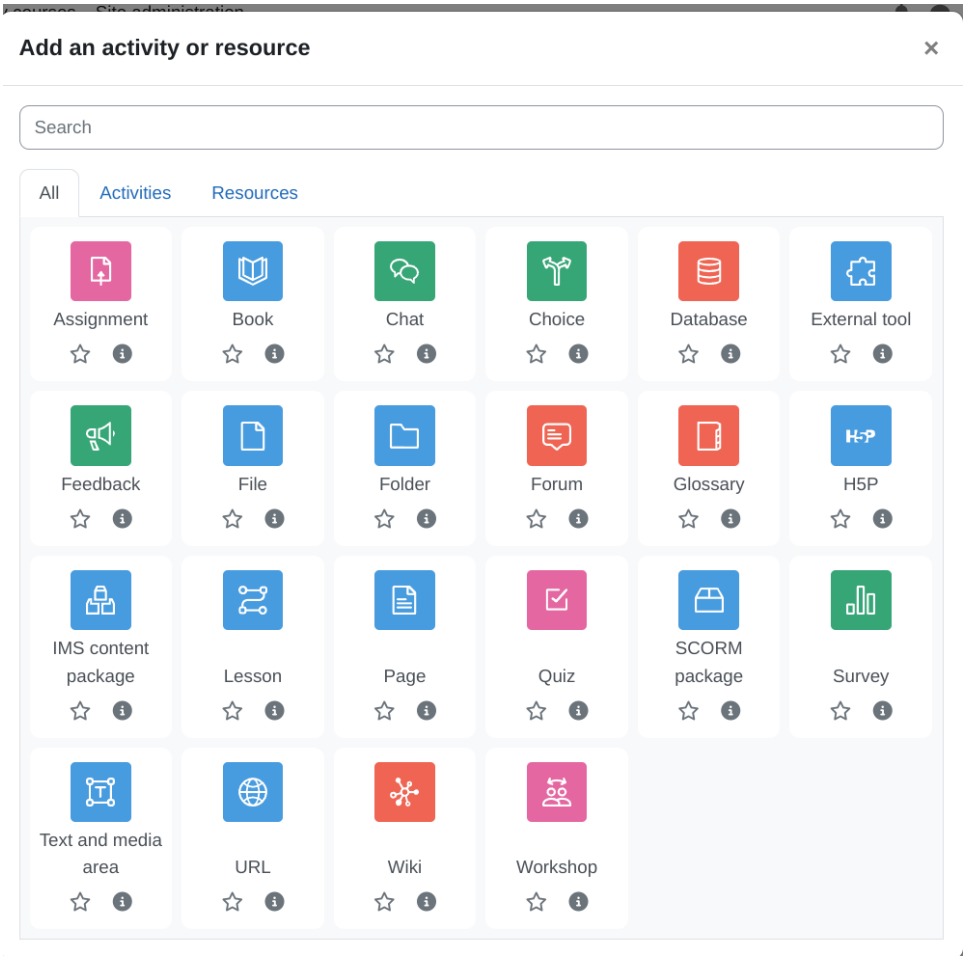

Рисунок 5.20 – Додавання нового ресурсу

В меню додавання ресурсу потрібно обрати «External tool», або «Зовнішній інструмент».

У вікні редагування зовнішнього інструменту, потрібно провести конфігурацію (рисунок 5.21).

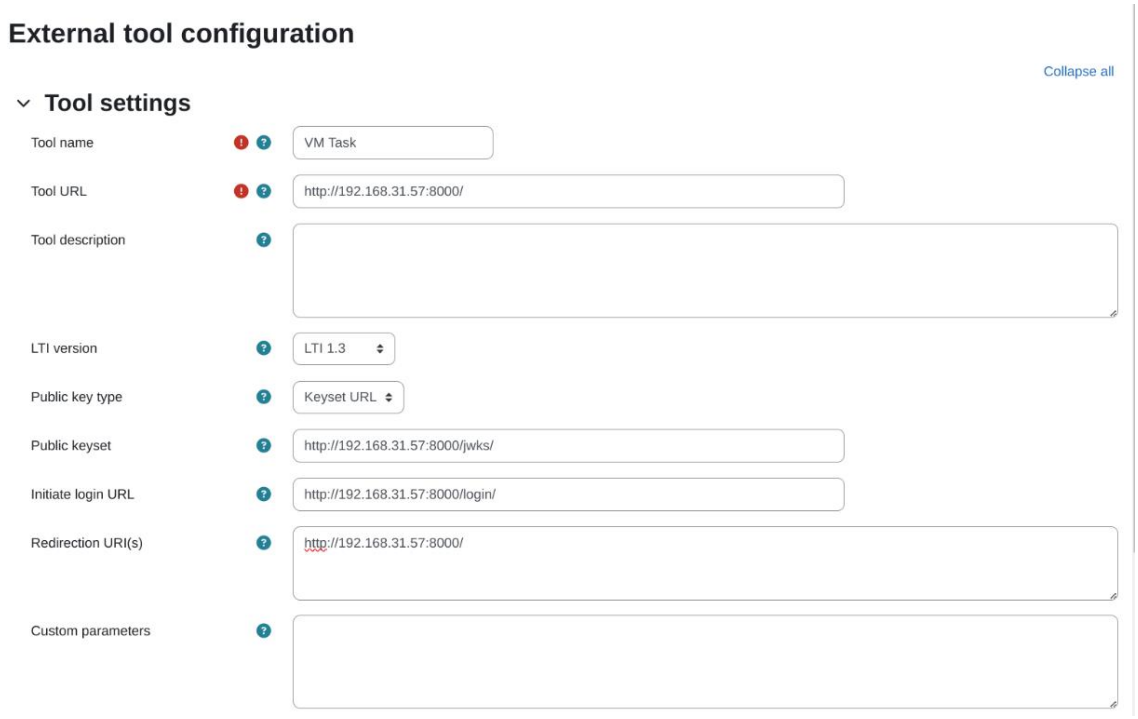

Рисунок 5.21 – Конфігурація зовнішнього інструменту

У вікні конфігурації задається ім’я інстументу, посилання на нього, верція протоколу, що використовується, метод, по якому беруться публічні ключі, якщо метод вказаний «Keyset URL», то додатково потрібно вказати посилання, де можна взяти публічні ключі, посилання логіну, та куди буде перенапралений користувач після логіну.

Також інстументу потрібно надати можливість взяти роль користувача (це студент, чи вчитель), та надати можливість проставляти та отримувати оцінки (рисунок 5.22).

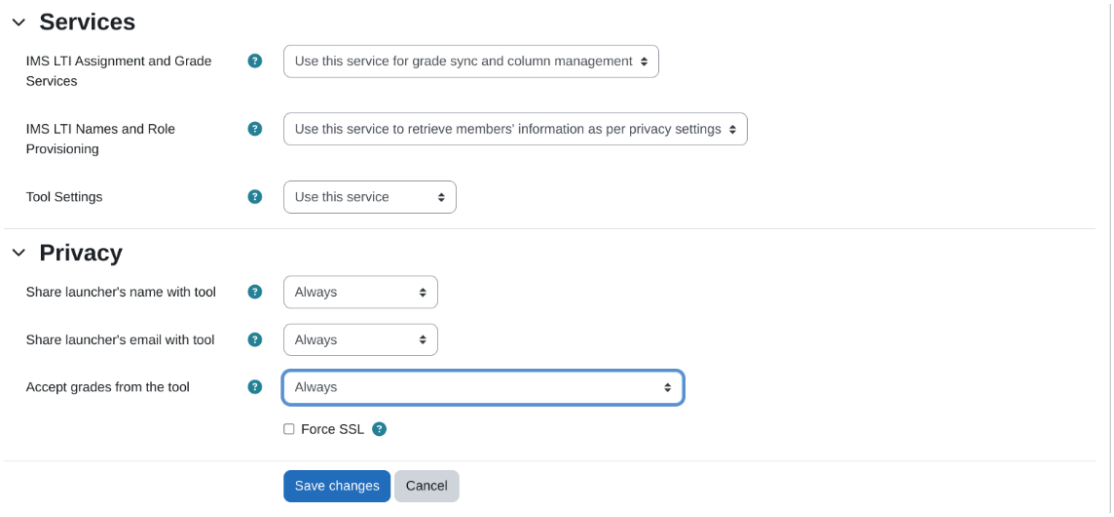

Рисунок 5.22 – Налаштування доступів інструменту

Після конфігурації стороннього інструменту, він з’явиться в полі преконфігурованих інструментів, та його можна буде обрати (рисунок 5.23).

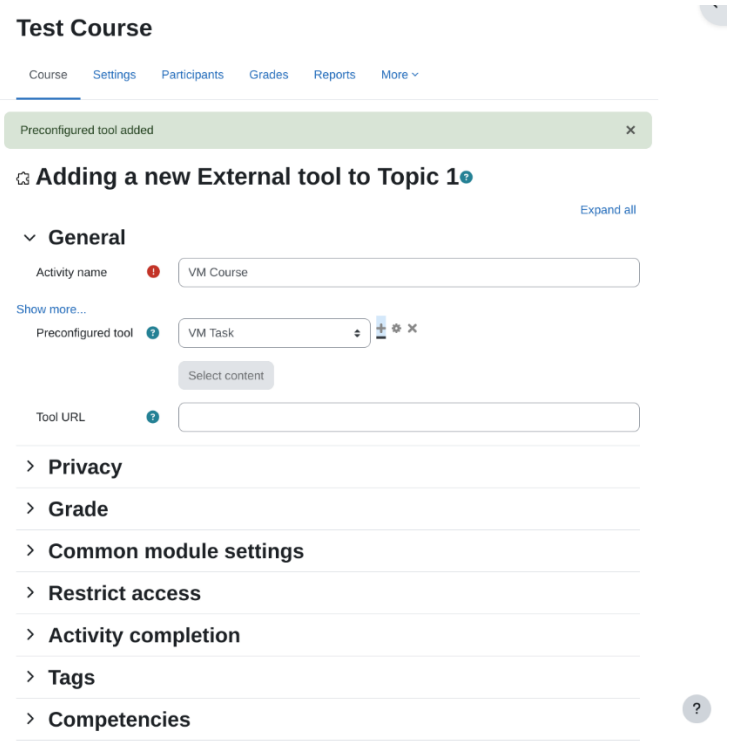

Рисунок 5.23 – Додавання пре-конфігурованого інструменту до нового ресурсу курсу

Після чого, локальне оточення вважається налаштованим.

6 ВИКОРИСТАННЯ СТОРОННЬОГО ІНСТРУМЕНТУ

6.1 Використання інструменту вчителем

Інструмент логічно поділяється на дві частини, те що бачить вчитель, та те, що бачить студент.

У вчителя є змога задати контент сторінки, прописати завдання, та підвантажити тести, котрі будуть запускатись на стороні студенту (рисунок 6.1).

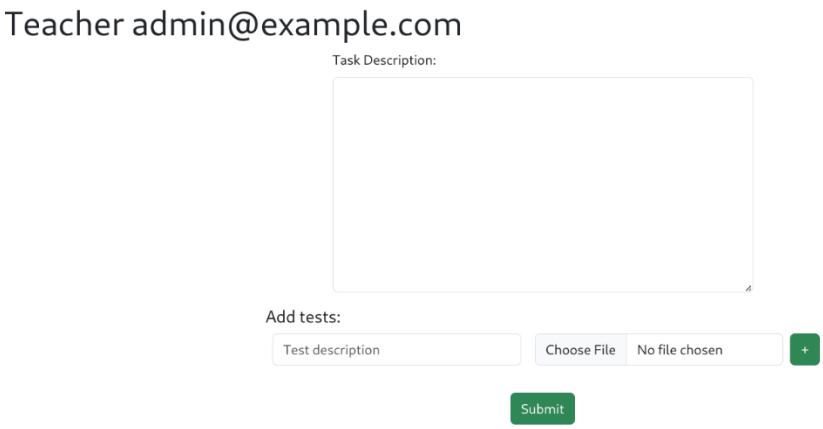

Рисунок 6.1 – Вигляд сторінки вчиталя

По елементам сторінки:

1) Task Description – це поле слугує для прописування умов завдання, в першому прототипі – це просто текс, котрий описує те, що потрібно зробити.

2) Add tests – ця секція слугує для того, щоб додавати конкретні тести, по результатам котрих буде оцінюватись студент, сам тест складається з 2-х елементів, з опису тесту (що саме цей тест тестує), та з самого тесту (статично зібраний бінарний файл під архітектуру linux amd64), котрий буде запущений на віртуальній машині студенту.

Заповнену сторінку вчителя можна побачити на рисунку 6.2.

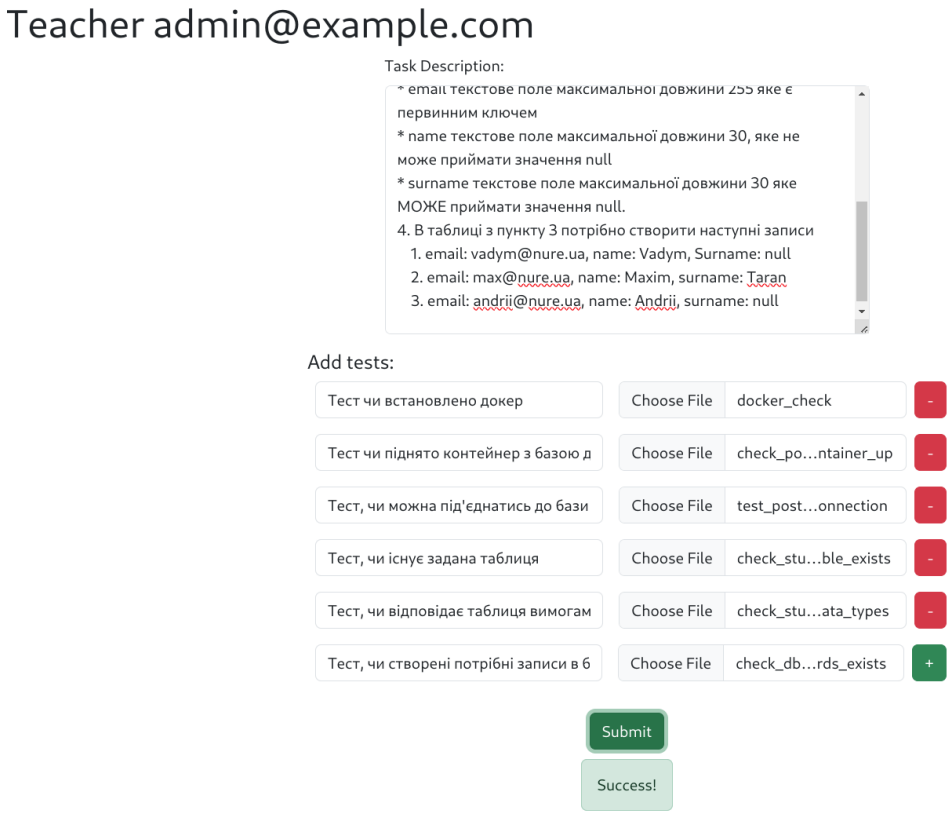

Рисунок 6.2 – Заповнена сторінка вчителя

6.2 Використання інструменту студентом

На сторінці студента відображається текст завдання, та тести, що будуть запущені на віртуальній машині студента.

Якщо детальніше розібрати елементи сторінки, то там є статус віртуальної машини студента, вона увімкнена, чи вимкнена, опис завдання, що потрібно зробити, секція тестів, котрі будуть перевірятись на віртуальній машині студента, та кнопка запуску, котра активна, тільки якщо студент запустив свою віртуальну машину (рисунок 6.3)

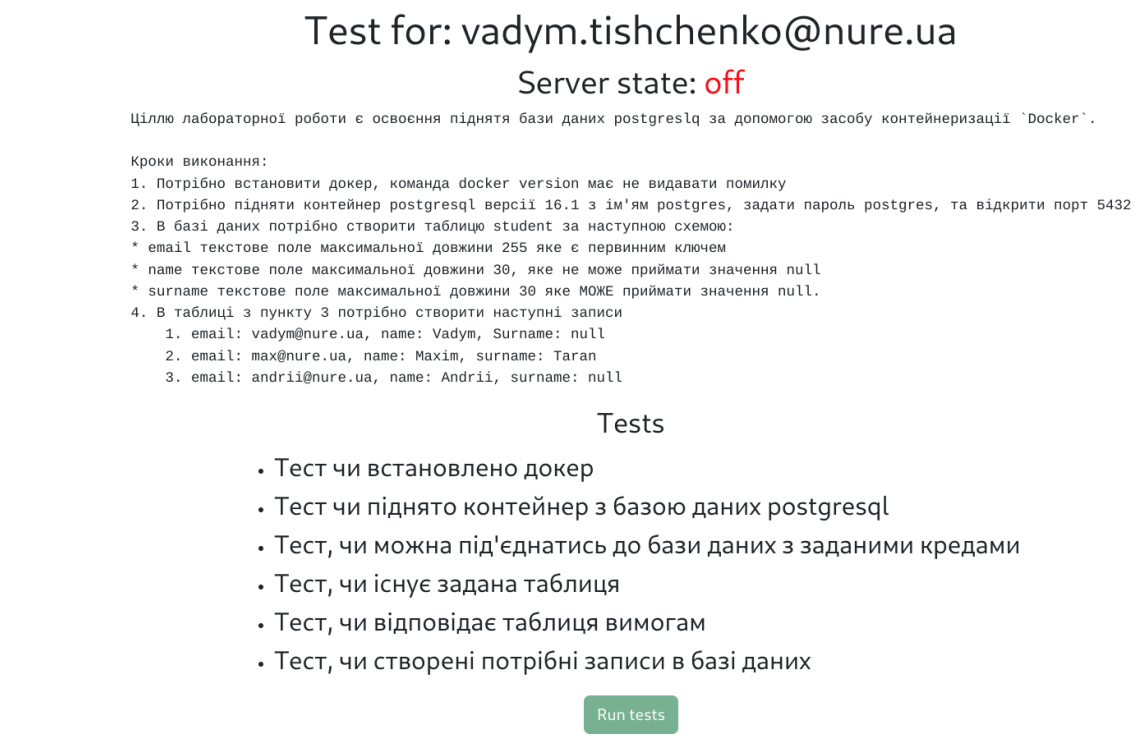

Рисунок 6.3 – Сторінка студента

Після запуску віртуальної машини студента, статус серверу стане зеленим, та кнопка запуску тестів розблокується.

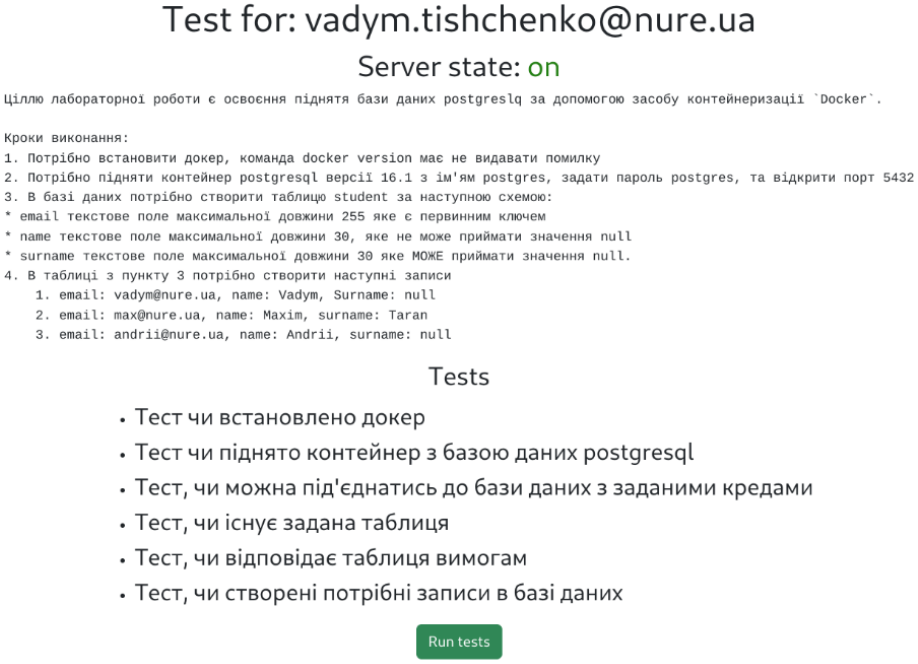

Рисунок 6.4 – Сторінка студента, у котрого запущена віртуальна машина

Якщо спробувати запустити тести, без виконаних завдань – то вони просто попадають (рисунок 6.5).

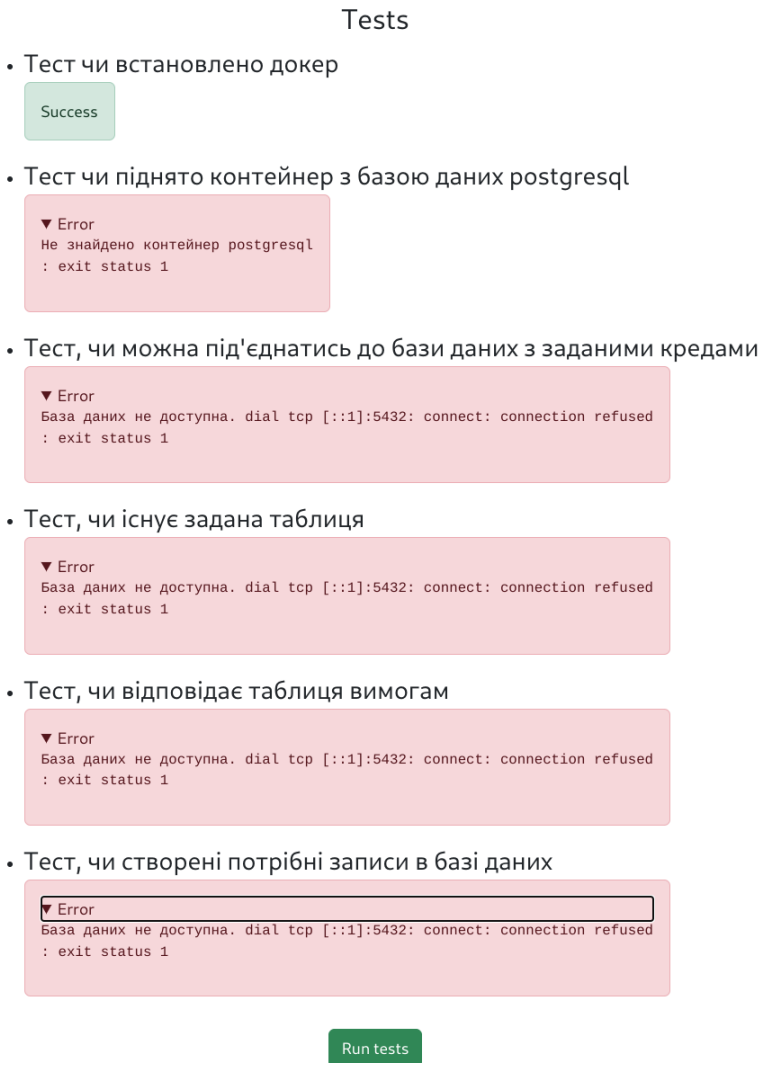

Рисунок 6.5 – Результат запуску тестів, без виконання завдання

В кожній помилці розписано, чому саме тест провалився, в даному випадку, всі тести залежать від наявності бази даних, і тому починаючи з того, що база не піднята – всі тести просто впали.

Якщо виконати перший пункт завдання (рисунок 6.6), і тим самим задовільнити 2 тести, то результат буде іншим (рисунок 6.7).

```
→  php docker run -d --name postgres -e POSTGRES_PASSWORD=postgres  -p 5432:543
2 postgres:16.1
dcd041678fe3b31d819b3bbe098eafeb261eb3ffa557130d7062ef342fd1ade4
→  php docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  CREAT
ED          STATUS         PORTS
                                   NAMES
dcd041678fe3   postgres:16.1                        "docker-entrypoint.s…"   3 sec
onds ago    Up 3 seconds   0.0.0.0:5432->5432/tcp, :::5432->5432/tcp
                                   postgres
```

Рисунок 6.6 – Запуск бази дани postgresql версії 16.1 з назвою контейнера postgres

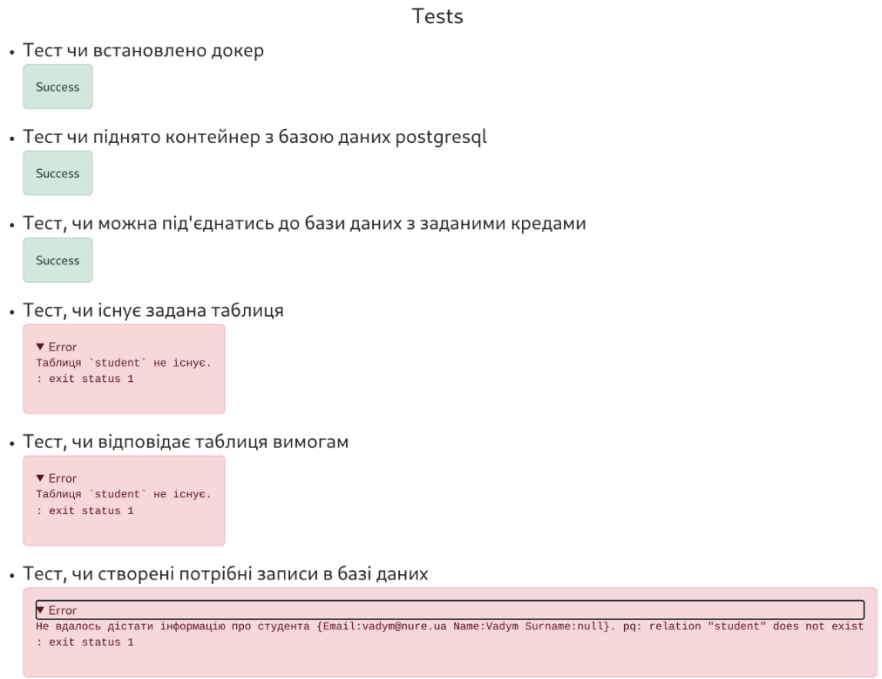

Рисунок 6.7 – Результат виконання тестів, після запуску бази даних

Можна побачити, що тепер 3 теста успішних. 1-й тест перевірив на наявність такого інструменту контейнеризації як докер, другий тест перевірив, що піднято потрібний контейнер з базою даних, і третій тест перевірив, що до бази даних можна під’єднатись. Інші тести впали, з деталізованою помилкою, чому вони впали. В тесті 4 йде перевірка, чи створена потрібна таблиця в базі даних, в тесті 5 перевіряється, чи відповідає створена таблиця в тесті 4 заданим вимогам, і в пункті 6 перевіряється

наповненість цієї таблиці, оскільки таблиці не існує, всі залежні від цієї таблиці тести впали.

Після створення потрібної таблиці (рисунок 6.8), не проходить лише один тест, тест котрий перевіряє наповненість (рисунок 6.9), бо таблиця поки що пуста.

```
postgres=# create table student (email VARCHAR(255) PRIMARY KEY,
postgres(# name VARCHAR(30) NOT NULL,
postgres(# surname VARCHAR(30) NULL);
CREATE TABLE
```

Рисунок 6.8 – Створення таблиці, що відповідає вимогам

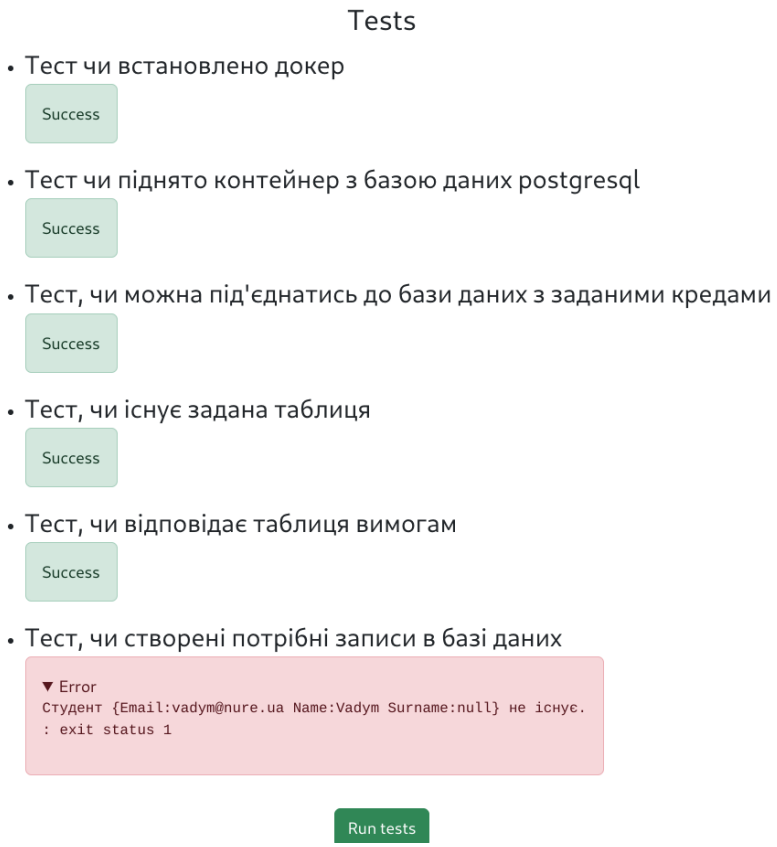

Рисунок 6.9 – Запуск тестів після створення таблиці, що відповідає вимогам

Щоб задовільнити останній тест, потрібно наповнити створену таблицю даними (рисунок 6.10), після чого, тест має бути успішним (рисунок 6.11).

```
postgres=# insert into student (email, name, surname) values ('vadym@nure.ua',
 'Vadym', null),
('max@nure.ua', 'Maxim', 'Taran'),
('andrii@nure.ua', 'Andrii', null);
INSERT 0 3
```

Рисунок 6.10 – Наповнення таблиці даними

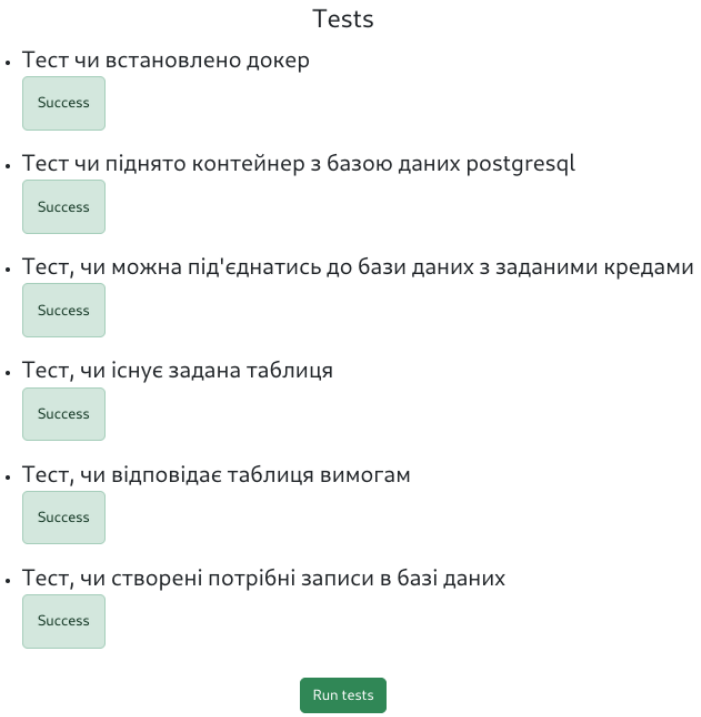

Рисунок 6.11 – Успішно пройдені тести

Сторонній інструмент має функціонал оцінювання, тобто після кожного запуску він проставляє оцінку відповідну до того, скільки тестів було виконано. Сама оцінка розраховується за формулою 6.1

$$mark = \frac{Failed}{Total} * 100$$

(6.1)

Де оцінка дорівнює кількості провалених тестів поділених на кількість всих тестів, домножених на 100. Таким чином, оцінка виходить чесною, якщо

студент виконає 60% завдань, то в нього буде задовільна оцінка, якщо студент виконає всі завдання, то в нього буде відмінно, або 100%.

Після того, як всі тести пройдені, на сторінці курсу, можна побачити, що оцінка виставилась, і функціонал відпрацював (рисунок 6.12).

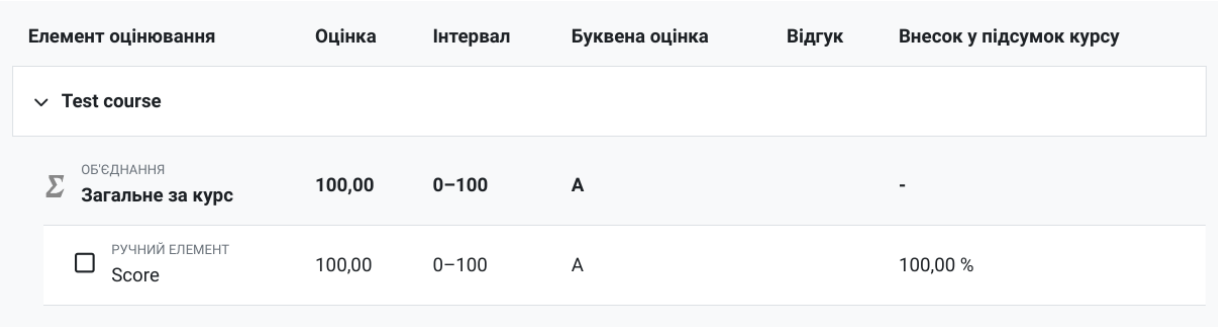

Рисунок 6.12 – Автоматично-виставлена оцінка за завдання

ВИСНОВКИ

У ході виконання кваліфікаційної роботи була розроблена архітектура системи, яка базується на використанні системи керування навчанням Moodle та інтеграції її з додатком, що використовує Learning Tools Interoperability (LTI) для комунікації між системами. Ця архітектура дозволяє здійснювати автоматизовану перевірку практичних завдань в локальному віртуальному середовищі користувача.

Подібна автоматизація може допомогти студентам не захламляти свої комп’ютери, при цьому працювати локально виконуючи в повному обсязі практичні завдання.

Подібна архітектура є лише базовим прикладом, котрий можна розвинути до чогось більшого, що в теорії зніме навантаження з викладачів, та зробить навчання студентів більш легким в плані самостійного навчання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Документація до LMS Moodle [Електронний ресурс] Режим доступу до ресурсу: https://docs.moodle.org/403/en/Main_page

2. Специфікація LTI [Електронний ресурс] Режим доступу до ресурсу: https://www.imsglobal.org/spec/lti/v1p3/impl/

3. Communication protocol [Електронний ресурс] Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Communication_protocol

4. Virtualization VS Containerization [Електронний ресурс] Режим доступу до ресурсу: https://www.burwood.com/blog-archive/containerization-vs-virtualization

5. Microservices VS Monolith [Електронний ресурс] Режим доступу до ресурсу: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

5. How OpenID Connect Works [Електронний ресурс] Режим доступу до ресурсу: https://openid.net/developers/how-connect-works/

6. Таран М. В. Дослідження шляхів автоматизації перевірки виконання практичних завдань в локальному віртуальному оточенні користувача / М. В. Таран, В. В. Тіщенко, А. І. Костромицький. // Міжнародна науково-технічна конференція «Інформаційно-комунікаційні технології та кібербезпека (ІКТК-2023)» Харків, 7 – 8 грудня 2023.