МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Побудова багаторівневого веб-застосування на платформі Amazon Web Services (AWS)

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки та інформаційні технології» - 122

Керівник курсової роботи
к.т.н., ст. викладач
Черкасов Д. І.

_____
(підпис)
"___" _____ 2020 р.
Виконав студент 4 курсу
Галенок Д. О.
"___" _____ 2020 р.

Київ 2020

ЗМІСТ

Анотація

У даній роботі розглядається використання Amazon Web Services (AWS) для побудови багаторівневого веб-застосування. Аналізуються переваги та недоліки моделі хмарних обчислень та різні підходи розгортання веб-застосувань, здійснюється порівняльний аналіз архітектурних шаблонів багаторівневих застосувань. У якості прикладу застосування розглядається створення веб-месенджера. Детально описується процес розробки одного із компонентів розробленого веб-застосування.

ВСТУП

В останні роки хмарні обчислення набувають все більшої популярності завдяки надійності, гнучкості та ефективності. Цей підхід дозволяє знизити витрати для підтримки інфраструктури навколо веб-застосування, швидко масштабувати веб-застосування за потреби, ефективно використовувати та контролювати обчислювальні ресурси. Також, хмарна модель не вимагає від розробника знань апаратної частини обчислювальних ресурсів, а навпаки дозволяє із мінімальними зусиллями адмініструвати їх. Однією із найбільших переваг є те, що користувачі послуг хмарних платформ не несуть витрати на покупку та обслуговування власного серверного обладнання.

Низка провайдерів надають хмарні обчислювальні ресурси, сховища даних, інструменти для розгортання та адміністрування програмного забезпечення, тощо. Найвідомішими хмарними платформами на сьогодні є: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, IBM Cloud Computing. Часто, провайдери хмарних послуг надають низку додаткових сервісів для ефективного управління інфраструктурою.

Дана робота має на меті проаналізувати переваги та недоліки використання хмарних обчислень (а саме платформи AWS) у порівнянні із більш традиційними підходами розгортки багаторівневих веб-застосувань. Розглянути багаторівневу (N-tier) архітектуру застосунків при використанні із платформами хмарних обчислень. Практичну частину цієї роботу присвячено побудові багаторівневого веб-месенджера на платформі AWS.

РОЗДІЛ 1. Аналіз існуючих рішень

Використання хмарної інфраструктури вирішує низку проблем з якими зіткаються організації та компанії при розробці програмних продуктів. Однією із найбільших переваг хмарної архітектури є відсутність потреби здійснювати витрати на придбання, налаштування та підтримку фізичного обладнання. Проте, використання хмарної інфраструктури потребує від розробників чіткого бачення системи на етапі проектування. На початку життєвого циклу розробки будь-якого застосування проектується архітектура, яка згодом може виявитись непридатною до використання із певним архітектурним рішенням. Таким чином, при розробці архітектури особливу увагу потрібно приділити які переваги та недоліки вона має з інфраструктурного боку. Наприклад, 1-рівневе застосування не дає змогу повністю використати можливості хмарної архітектури та є більш придатним для розгортки на виділеному фізичному сервері. З іншого боку, багаторівневі застосування здатні ефективно використовувати ресурси хмарних обчислень та масштабувати певний рівень за потреби.

1.1 Вибір архітектури веб-застосування

Архітектура застосування визначає схему взаємодії між різними компонентами системи. Зручним є розділення архітектури на логічні рівні. Фактично, відбувається фізичне розділення на системи на певні компоненти. Наприклад, такими компонентами можуть бути сервер бази даних, бекенд сервер, користувацький інтерфейс, сервер кешування, messaging сервер, тощо. В залежності від кількості рівнів, застосунки поділяються на 1-рівневі, 2-рівневі, 3-рівнені та багаторівнені. Кожен із цих підходів має свої переваги та недоліки пов’язані із складністю розробки, розгортки та використання.

	Складність розробки	Рівень безпеки	Ефективне масштабування	Швидке оновлення
1-рівневе	Низька	Високий	Ні	Ні
2-рівневе	Низька	Низький	Частково	Ні
3-рівневе	Відносно висока	Високий	Так	Так
Багаторівневе	Висока	Високий	Так	Так

Таблиця 1 - Порівняльна характеристика архітектурних підходів

1.1.1 1-рівневі застосування

1-рівневе застосування - це застосування у якому сервер бізнес-логіки, база даних та користувацький розташовані на одній машині.

Рисунок 1.1- Візуалізація 1-рівневого застосування

1-рівневі застосування володіють низкою переваг, проте це архітектурне рішення є непридатним до використання у веб-застосуваннях, адже усі компоненти програмної системи мусять бути на одній фізичній машині. Ключовими перевагами є: відсутність мережевих запитів до ресурсів системи, швидкий доступ до даних, відносно гарний рівень безпеки. Прикладами 1-рівневих застосувань є настільки додатки (desktop applications). Одним із найбільших недоліків даних застосувань є складність внесення змін після того як застосунок було доставлено. Наприклад, якщо користувач встановив певне програмне забезпечення, яке містить помилку, він мусить власноруч завантажити та встановити оновлення. Цей недолік змушує розробників дуже ретельно тестувати, адже будь-який помилковий код який було доставлено буде складно виправити.

1.1.2 2-рівневі застосування

2-рівнева архітектура включає в себе клієнт-серверну взаємодію. Зазвичай, клієнтський рівень містить користувацький інтерфейс, а сервер фактично виступає у ролі сервера бази даних.

Рисунок 1.2 - Візуалізація 2-рівневого застосування

Таке розділення рівнів дозволяє розмістити різні рівні на різні машини (віртуальні або фізичні). Безумовно, розміщення бізнес-логіки на клієнтській стороні створює потенційні загрози безпеці. Користувач матиме змогу здійснити reverse engineering та отримати доступ до критичних аспектів системи. Проте, даний підхід має й низку позитивних аспектів. Розміщення користувацького інтерфейсу та бізнес-логіки на одному рівні дозволяє зменшити кількість мережевих запитів до бази даних. В свою чергу, це дозволяє зменшити витрати на підтримку серверної інфраструктури.

1.1.3 3-рівнені застосування

3-рівнева архітектура є однією з найбільш використовуваних у веб-застосуваннях. Вона строго розподіляє застосунок на 3 рівні: користувацький інтерфейс, бізнес-логіка застосунку та база даних.

Рисунок 1.3 - Візуалізація 3-рівневого застосування

Даний підхід вирішує більшість недоліків які присутні у 1-рівневій та 2-рівневій архітектурі. 3-рівневі застосунки добре масштабуються, дозволяють потворно використовувати певні компоненти, спрощують процес оновлення системи, зменшують ризики безпеки. Проте, у зв'язку з тим, що комунікація між рівнями відбувається завдяки мережі, постає задача безпечної передачі даних. Також, варто зазначити що загалом складність цієї системи є більшою, внаслідок потреби коректно організовувати комунікацію між компонентами.

1.1.4 Багаторівневі застосування

Багаторівнева архітектура (або N-tier architecture) є розширення 3-рівневої архітектури. Даний підхід наслідує принцип Single Responsibility, тобто кожен рівень повинен мати свою задачу і виконувати її добре. Прикладами додаткових рівнів які можуть використовувати багаторівневі застосування є: компоненти кешування, messaging сервіси, балансувальники навантаження, пошукові сервіси, тощо. Застосування із

багаторівневою архітектурою є найбільш придатними для ефективного управління ресурсами та масштабування за потребою. Цей архітектурний підхід дозволяє масштабувати окремі рівні (компоненти) в залежності від наявного навантаження. Загалом, така архітектура підвищує гнучкість системи та спрощує адміністрування.

1.2 Розміщення на власних ресурсах

Розміщення застосування на власних ресурсах є одним із найстаріших підходів у доставці веб-застосувань. Проте, цей підхід є широковживаним навіть зараз, здебільше у великих корпораціях, які володіють великими (часто legacy) програмними продуктами. Раніше, кожна з таких корпорацій будувала та константно оновлювала власну інфраструктуру. У свою чергу, будь-які нові програмні продукти які розроблялися орієнтувалися на вже вибудовану інфраструктуру. Використання власних ресурсів має низку переваг: можливість повністю контролювати інфраструктуру, підвищений рівень захищеності даних. Організація, яка володіє певними обчислювальними ресурсами має також можливість оновити інфраструктуру (обладнання), щоб максимально задовольнити свої потреби. Величезною перевагою є зберігання даних на власних серверах, адже таким чином, компанія повністю контролює усе що відбувається із цими даними. Також, варто зазначити, що існують певні сфери у яких компанії зобов'язані зберігати дані користувачів на власних серверах. Проте, також існують певні недоліки у такого підходу.

По-перше, вартість покупки та налаштування фізичного обладнання є досить високою. На початковому етапі створення будь-якого веб-застосунку організація повинна оцінити приблизне навантаження на систему, щоб придбати відповідне обладнання. Також, у компанії повинні бути фахівці, які б підтримували інфраструктуру, що також підвищує витрати. Для невеликих організацій та стартапів цей недолік є часто ключовим у виборі способу розміщення їхнього веб-застосування.

По-друге, відсутня можливість швидкого масштабування за потреби. У випадку швидкого зростання навантаження на систему та недостатньої кількості апаратних ресурсів може виявитись, що система не здатна обробити усі запити достатньо швидко, а у найгіршому випадку система

може навіть впасти. Одним із рішень цієї проблеми є збільшення апаратних ресурсів власних серверів. Проте, для оновлення та розширення апаратних ресурсів потрібен певний час на придбання, встановлення та налаштування. Також, досить часто трапляються випадки коли спостерігається підвищене навантаження на систему у певний час (наприклад ввечері), а у інші періоди часу навантаження є мінімальним. Таким чином, апаратні ресурси використовуються неефективно, проте їхня наявність є критично важливою задля коректного функціонування системи у так звані піки навантаження.

По-третє, використання власної інфраструктури не гарантує повної надійності роботи системи. Оновлення ресурсів часто вимагає певного часу простою системи, що може бути неприйнятним для певних сфер застосування. Також, інколи можуть виникати мережеві перебої, або певні природні катастрофи, які можуть вивести систему з ладу.

Загалом, використання власних ресурсів є традиційним способом розміщення веб-застосунків. Проте, цей спосіб є непростим у підтримці, ресурсозатратним та складно масштабується. Найчастіше використовується при чіткому розумінні рівня навантаженості на систему і/або потребі підвищеної захищеності даних.

1.3 Розміщення на орендованих ресурсах

Розміщення на орендованих ресурсах є непоганою альтернативою використанню власних ресурсів. Насамперед, завдяки відсутності великих витрат на початковому етапі розробки застосування. Часто, такий підхід використовується, якщо є потреба у розміщенні застосування на певний невеликий період часу.

У порівнянні із використанням власних ресурсів даний підхід має декілька недоліки: менша гнучкість у виборі апаратних ресурсів, менший рівень захищеності даних. Проте, варто зазначити декілька ключових переваг використання орендованих ресурсів:

По-перше, такий підхід суттєво спрощує процес розгортання веб-застосувань. Зазвичай, провайдер обчислювальних ресурсів дає можливість користувачам обрати кількість оперативної пам’яті, постійної пам’яті та кількість процесорів (або їх різновид). Також, наразі більшість провайдерів дають можливість обрати операційну систему та певні можливості віртуалізації для більш ефективного використання ресурсів.

По-друге, існує можливість швидкого масштабування. Фактично, провайдери обчислювальних ресурсів надають віртуальну інфраструктуру, яка може бути розширена майже миттєво. Найбільш зручним це є для веб-застосувань із нерівномірним навантаженням та у випадках коли навантаження на систему складно прогнозувати.

По-третє, більшість провайдерів гарантують високий рівень надійності роботи системи та захищеності даних. Також, певні провайдери пропонують резервні екземпляри інфраструктури. У випадку збою в роботі, трафік може бути перенаправленим на інший сервер без помітної деградації роботи веб-застосування. На додачу, більшість провайдерів надають можливість робити резервні копії даних.

1.4 Розміщення на хмарній платформі

Зазвичай, хмарні послуги надаються за трьома моделями: IaaS (Infrastructure as a service), PaaS (Platform as a service), SaaS (Software as a service). Інколи також виділяють додаткову модель DaaS (Data storage as a service), яка фактично є особливим випадком IaaS.

Модель IaaS надає доступ до обчислювальних ресурсів, наприклад: сервери, сховища даних, мережні ресурси. Ці ресурси можуть бути масштабованими за потреби, та часто підтримують віртуалізацію. Прикладами IaaS є AWS EC2, Google Compute Engine, DigitalOcean, тощо.

Модель PaaS надає користувачам доступ до хмарної платформи, в якому вони можуть розроблювати, адмініструвати та доставляти застосування. Така модель спрощує розміщення веб-застосувань, адже зникає необхідність адмініструвати власну або хмарну інфраструктуру. Прикладами PaaS є AWS Elastic Beanstalk, Google App Engine, Heroku, тощо.

Модель SaaS надає користувачам доступ до хмарних програмних продуктів. Зазвичай, ці програмні продукти існують у вигляді веб-застосувань. Користувачі не встановлюють ніяке програмне забезпечення для взаємодії із застосуваннями. Прикладами SaaS є Google Apps, Dropbox, Slack.

1.4.1 Розміщення із використанням AWS EC2, S3, RDS

Amazon Elastic Compute Cloud або EC2 - це гнучкий сервіс, який дозволяє розгортати віртуальні сервери. Фактично, ці віртуальні сервери підтримують весь функціонал яким володіють фізичні сервери. Сервіс EC2 надає можливість обрати операційну систему за вподобанням. Наразі підтримуються найпопулярніші дистрибутиви Linux та Windows Server. Для вибору операційної системи використовується AMI (Amazon Machine

Instance). AMI містить інформацію про конфігурацію системи та додаткове програмне забезпечення яке буде встановлено. Окрім вибору операційної системи, AWS EC2 надає можливість обрати кількість обчислювальних ресурсів (CPU, RAM, ємність постійної пам'яті). За потреби, кількість цих ресурсів може бути зміненою після створення екземпляру EC2. Також, AWS дозволяє обрати тип екземпляру EC2. Кожен із цих типів оптимізований під використання для вирішення певних задач. Таких типів є багато, проте вони розділені на види: загального призначення, для задач із обчисленнями, для задач із обробкою великих масивів у пам'яті, для задач які потребують апаратного прискорення та для задач які потребують швидкого доступу до читання та запису на локальний диск.

Amazon Simple Storage Service або S3 - це сервіс, який дозволяє зберігати будь-які дані у хмарі. Це можуть бути статичні файли веб-застосування, фото, відео, текстові файли, тощо. Сервіс S3 не має обмежень у розмірі файлів та надає можливість зберігати дані будь-якого виду. Об'єкти, або документи зберігаються у так званих buckets, кожен із яких прив'язаний до певного регіону. Доступ до цих об'єктів надається через API. Також, якщо об'єкт є публічно доступним, до нього можна звернутися використовуючи HTTP/HTTPS запит.

Amazon Relation Database Service або RDS - це сервіс, який надає можливість розгорнути реляційну базу даних у хмарі. Наразі, підтримуються PostgreSQL, MySQL, MariaDB, Oracle, MSSQL. Даний сервіс спрощує процес адміністрування бази даних, а саме управлінням резервними копіями, масштабування, підтримання високої доступності.

1.4.2 Розміщення із використання AWS ECS

Amazon Elastic Container Service або ECS - це хмарний сервіс для оркестрування контейнерів із підтримкою Docker-контейнерів. ECS

дозволяє розміщувати контейнеризовані застосування на платформі AWS. Цей сервіс дозволяє швидко масштабувати контейнери за потребою. Водночас, він сильно спрощує взаємодію із інфраструктурою. ECS використовується у багатьох випадках, наприклад для розміщення певного застосування за гібридною моделлю, для вирішення задач машинного навчання, для виконання паралельних обчислень, тощо. При використанні із веб-застосуваннями, ECS надає можливість автоматичного масштабування при великих навантаженнях на застосунок.

1.4.3 Розміщення із використанням AWS Lambda

AWS Lambda - досить новий спосіб розміщення веб-застосувань. Цей сервіс базується на принципі безсерверних (serverless) обчислень. Безсерверні обчислення повністю звільняють користувачів від будь-якої взаємодії із інфраструктурою. У дійсності, провайдер послуг безсервервних обчислень динамічно розподіляє ресурси між користувачами. З точки зору розробника, для взаємодії із AWS Lambda потрібно проектувати систему таким чином, щоб можна було виділити певну кількість функцій. Згодом, розробник завантажує цей код на платформу AWS у вигляді так званої Lambda функції. Важливим аспектом є те, що опісля виклику функції, будь-які незбережені дані знищуються. Виклик Lambda функції може здійснюватися різними шляхами: певна подія стороннього AWS сервіса, API запит, сповіщення AWS SNS, тощо. Наразі, AWS Lambda підтримує такі мови програмування: Java, Go, PowerShell, Node.js, C#, Python та Ruby.

РОЗДІЛ 2. Структурна розробка веб-застосування

У якості прикладу, розглядається багаторівневе веб-застосування для обміну миттєвими текстовими повідомленнями та файлами. Ключові вимоги до застосування: розбиття на рівні, підтримка обміну файлами,

системи ідентифікації користувачів, можливість комунікації між двома користувачами, використання сервісів AWS.

Було обрано наступні сервіси AWS для розробки веб-застосування: EC2, S3, RDS, CloudFront, Route53, Application Load Balancer.

Сервіс EC2 виступає у ролі головного бекенд-сервера, який відповідає за бізнес-логіку застосування. Для простоти спілкування із іншими компонентами, даний сервер наслідує архітектуру REST API. Повідомлення передаються у форматі JSON.

Сервіс S3 виконує декілька задач. По-перше, на ньому розміщуються статичні файли для роботи користувацького інтерфейсу. По-друге, зберігаються медіа-файли, якими обмінюються користувачі.

Сервіс RDS виступає у ролі реляційної бази даних в якій зберігаються дані про користувачів, а також їх повідомлення.

Сервіс CloudFront використовується для доставки об’єктів користувацького інтерфейсу, та фактично являється системою CDN.

Сервіс Route53 дозволяє зареєструвати доменне ім’я, та пов’язати цей домен (або сабдомен) із певним ресурсом AWS.

Сервіс Application Load Balancer розподіляє запити між екземплярами сервісу EC2 за певним алгоритмом, наприклад round robin.

2.1 Структурна схема веб-застосування

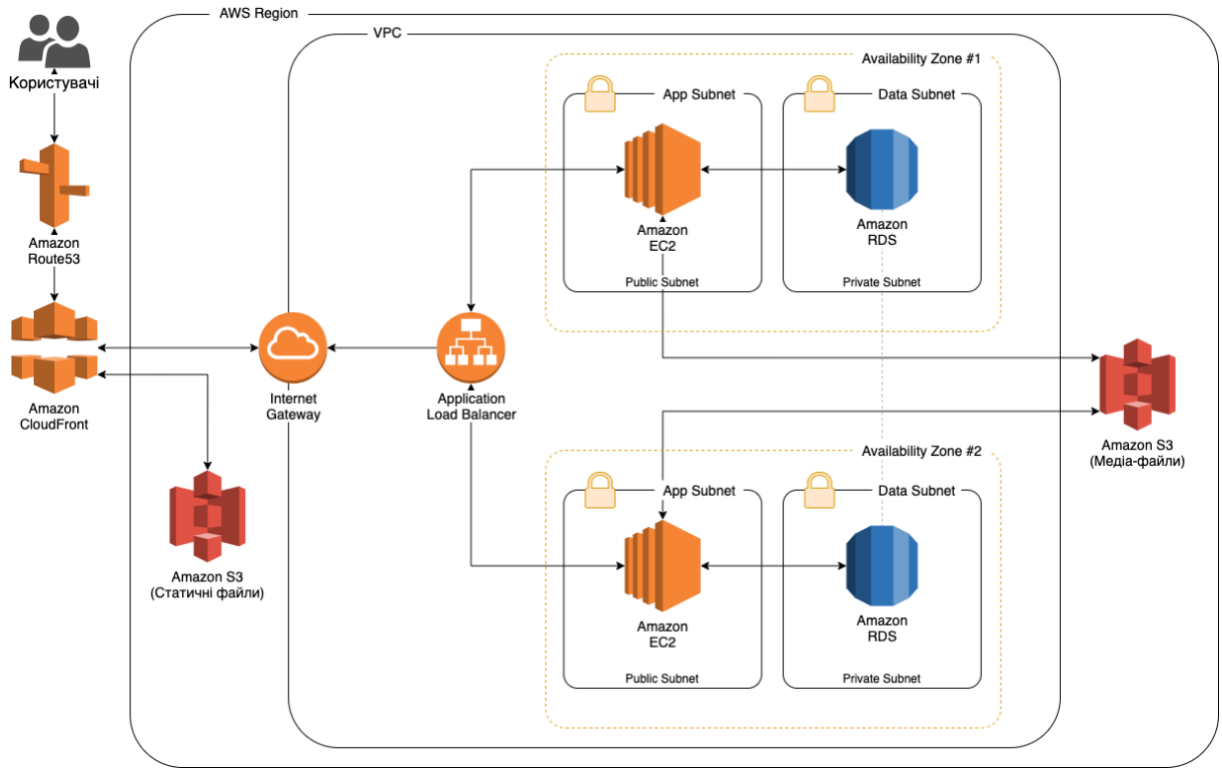

Рисунок 2.1 - Структурна схема веб-застосування

2.2 Опис компонентів веб-застосування

2.2.1 Користувацький інтерфейс

Для розміщення та доставки компонента користувацького інтерфейсу використовується зв'язка сервісів AWS S3 та AWS CloudFront. S3 дозволяє зберігати статичні файли (HTML, CSS, JS) веб-застосунку у хмарі, звертатися до них, налаштовувати права доступу до цих файлів, тощо. На додачу, S3 надає змогу хостингу статичного веб-сайту. Для того, щоб налаштувати подібний статичний веб-сайту потрібно виконати декілька кроків. По-перше, потрібно створити так званий bucket, який би містив статичні файли сайту. За замовчуванням, усі нові файли, які були додані в bucket приватні, проте при створенні bucket'у є можливість змінити ці

налаштування. Отже, для коректної роботи сайту, усі статичні файли повинні бути відкритими для читання. Останнім кроком є вибір точки входу або кореневого HTML файлу. Таким чином, було створено статичний сайт на базі S3, проте є змога покращити даний підхід із використанням AWS CloudFront. Сервіс CloudFront є мережею доставки контенту (CDN). Використання CDN дозволяє пришвидшити завантаження статичних файлів завдяки географічно розподіленій групі серверів [11]. Налаштування AWS CloudFront є досить простим та зводиться до вибору джерела статичних файлів та вибору стратегії кешування. У результаті, користувачу надається домен до якого можна звернутися, або використати згодом із сервісом роутингу AWS Route 53.

2.2.2 Бізнес-логіка

Екземпляр сервісу EC2 відповідає за бізнес-логіку веб-застосування. Даний сервіс опрацьовує запити від користувачів, які здійснюються за допомогою користувацького інтерфейсу. Завдання коректної обробки цих запитів повністю лежить на розробнику, адже при створенні екземпляру EC2 надається порожній віртуальний сервер. Під час створення надається можливість обрати AMI (Amazon Machine Instance), кількість обчислювальних ресурсів, мережні налаштування, налаштування безпеки, тощо. До EC2 можна під’єднатися за допомогою SSH, або використовуючи веб-інтерфейс. Також надається можливість налаштувати автоматичне масштабування, балансувальник навантаження та змінити кількість обчислювальних ресурсів. Враховуючи, що за замовчуванням користувачам надається сервер без допоміжного програмного забезпечення, розробники повинні самостійно налаштувати веб-сервер. Найпопулярнішими з яких наразі є Apache та nginx. Для взаємодії Python-застосування з веб-сервером використовується стандарт Web Server

Gateway Interface (WSGI). Варто зазначити, що стандарт WSGI підтримує тільки синхронні Python-застосунки, проте для підтримки асинхронності можна використовувати Asynchronous Server Gateway Interface (ASGI). Реалізації цих стандартів (наприклад gunicorn та uwsgi) фактично надає API для взаємодії Python-застосунку та веб-сервера, вирішує низькорівневі задачі, наприклад адміністрування сокетів та виділення так званих workers. Компонент бізнес-логіки є REST-подібним сервісом, який має змогу обробляти запити у форматі JSON. Даний компонент містить декілька сутностей пов'язаних із предметною областю: User (Користувач), Conversation (Бесіда), Message (Повідомлення). Сутність User містить дані про користувача; Conversation містить ідентифікатори користувачів які ведуть обмін повідомленнями; Message може містити текст повідомлення або посилання на медіа-файл. REST API описує такі ресурси:

- GET "api/conversations/" - отримання усіх бесід
- POST "api/conversations/" - створення нової бесіди
- GET "api/conversation/pk/" - отримання однієї бесіди та повідомлень
- POST "api/conversation/pk/" - створення нового повідомлення
- DELETE "api/conversation/pk/" - видалення бесіди
- POST "api/authenticate" - отримання JWT-токена
- POST "api/refresh" - оновлення JWT-токена
- POST "api/register" - реєстрація нового користувача

2.2.3 База даних

Сервіс Amazon RDS адмініструє розподілену реляційну базу даних. При створення екземпляру RDS надається можливість обрати двигун бази даних (MySQL, PostgreSQL, тощо), стратегію оптимізації бази даних, обчислювальні ресурси, базові налаштування доступу до бази даних (ім'я,

пароль) та якій віртуальній мережі даний екземпляр належатиме. Варто зазначити, що з міркувань безпеки, за замовчуванням доступ до бази даних надається тільки сервісам які знаходяться у тій самій віртуальній мережі. Встановлення з’єднання між компонентом бізнес-логіки та бази даних здійснюється за допомогою URL-адреси.

2.3 Опис функціонування веб-застосування

При навігації на адресу веб-застосування користувачу виводиться форма авторизації. Також, користувач має змогу зареєструватися використовуючи свої контактні дані. Опісля реєстрації, користувач авторизується та отримує JWT-токен. Запити для обміну повідомленням вимагають передачі JWT-токена для отримання об’єкту користувача. При введені коректного псевдоніма та пароля, користувач перенаправляється на основну сторінку веб-застосування. На ній виводиться список активних бесід та надається можливість створити нову бесіду використовуючи псевдонім іншого користувача. При переході до певної бесіди, користувачу виводяться попередні повідомлення, та форма введення нового повідомлення і можливість прикріпити медіа-файл. Важливим аспектом роботи системи є автоматичне відображення нових повідомлень. Таке автоматичне оновлення досягається завдяки використанню протоколу WebSocket.

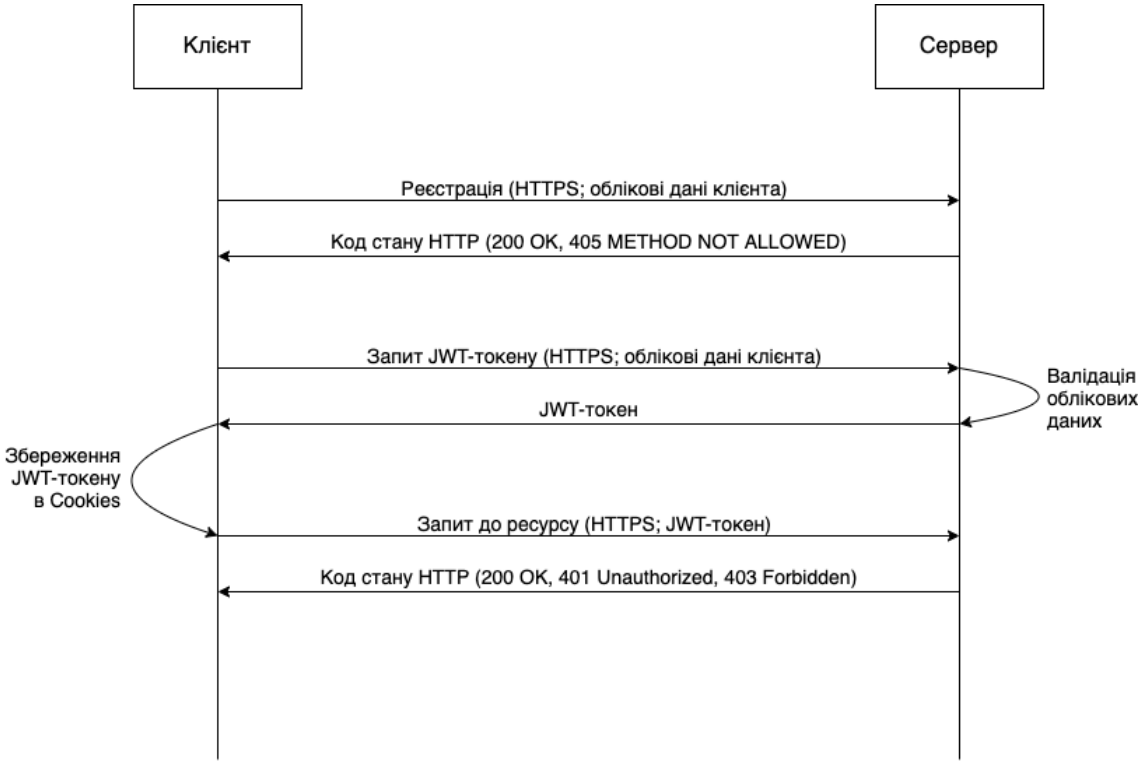

Рисунок 2.2 - Діаграма послідовності автентифікації/авторизації

РОЗДІЛ 3. Розробка компонента бізнес-логіки

Для реалізації основного компоненту-бізнес логіки було обрано мову програмування Python, веб-фреймворк FastAPI та веб-сервер uvicorn. У порівнянні із фреймворками django-rest-framework та Flask, FastAPI є набагато швидшим та підтримує асинхронність. В якості ORM-фреймворку було обрано пакет sqlaclhemy. Основний компонент бізнес-логіки REST-подібний сервіс та описує наступні ресурси:

URL	HTTP метод	Тіло запиту	Автентифікація	Результат
/api/conversations	GET	-	JWT-токен	Повертає усі бесіди
/api/conversations	POST	Тема, користувач	JWT-токен	Створює нову бесіду
/api/conversations/pk	GET	-	JWT-токен	Повертає бесіду та усі повідомлення
/api/conversations/pk	POST	Текст або файл	JWT-токен	Створює нове повідомлення
/api/conversations/pk	DELETE	-	JWT-токен	Видаляє бесіду та повідомлення
/api/users	POST	Облікові дані	-	Створює нового користувача
/api/token	POST	Облікові дані	-	Повертає JWT-токен

3.1 Налаштування середовища розробки

Загальноприйнятою практикою при розробці Python-застосунків вважається використання віртуального середовища (virtualenv). Virtualenv надає змогу розробникам встановлювати та адмініструвати додаткові Python-пакети, які використовуються у застосуванні. Таки підхід спрощує доставку та розміщення, адже усі залежності програмного продукту та їх версії містяться у файлі requirements.txt. Самі ж залежності встановлюються за допомогою вбудованої в мову Python утиліти pip. Також, обов'язковим є створення репозиторію із використанням певної системи контролю версій.

Для безперервної інтеграції та доставки (CI/CD) AWS надає сервіс CodePipeline [10]. CodePipeline дозволяє налаштувати процес автоматичної побудови, тестування та доставки застосування. Для побудови та тестування використовується сервіс CodeBuild, а для доставки сервіс CodeDeploy. Досить зручним підходом є використання зв'язки git-репозиторію та CodePipeline. Наразі, CodePipeline надає можливості інтеграції із GitHub. Інтеграції між цими сервісами здійснюється за допомогою так званих webhooks [12]. При певній події у репозиторії (наприклад новий коміт), надсилається HTTP запит на певний URL CodePipeline, який запускає процес інтеграції та доставки.

3.2 Доставка коду та контенту

Розміщення основного коду веб-застосування на ресурси AWS відбувається за допомогою низки допоміжних сервісів, які інтегруються із GitHub. Використано сервіс CodeDeploy для доставки коду до екземплярів EC2. CodeDeploy надає можливість створити групу розгортання, додати до цієї групи екземпляри сервісу EC2, або EC2 Auto Scaling групу та зазначити стратегію розгортання. Окрім цього, при налаштованому балансувальнику

навантаження, існує можливість додати його до групи розгортання, для досягнення коректної роботи застосування при оновленні. Фактично, CodeDeploy надає можливість оновлювати кожен екземпляр по черзі та блокує трафік до екземплярів які оновлюються. Варто також зазначити, що для налаштування процесу розгортки, репозиторій повинен містити файл appspec.yml. У цьому файлі надається можливість керування кожним розгортанням із використанням так званих подій життєвого циклу (lifecycle events). Керування здійснюється bash-скриптами, які спрацьовують на різних етапах процесу розгортання. Також, критично важливим для роботи CodeDeploy є програмний пакет CodeDeploy Agent. Даний пакет потрібно встановити на екземплярах EC2, для того, щоб використовувати ці екземпляри для розміщення.

Опісля коректного налаштування CodeDeploy, для автоматизації доставки було використано CodePipeline. Задля цього, потрібно під’єднати обліковий запис у GitHub та обрати потрібний репозиторій та гілку, сирцевий код якої буде розміщено. Також, при необхідності, можна створити екземпляр CodeBuild, який компілював би сирцевий код застосування, запускав тести, тощо. Останнім етапом є долучення раніше створеного екземпляру CodeDeploy до CodePipeline. Таким чином, було повністю автоматизовано процес доставки основного коду веб-застосування на ресурси AWS EC2.

Доставка статичного контенту на сервіс S3 також досягається із використанням AWS CodePipeline, проте налаштування процесу дещо відрізняється. В якості провайдера сирцевого коду також було обрано репозиторій GitHub та відповідну гілку. Враховуючи, що більшість сучасних підходів у побудові користувацьких інтерфейсів потребують компіляція із використанням Node.js, сервіс CodeBuild є критично важливим для доставки. Для роботи із CodeBuild, репозиторій із користувацьким інтерфейсом повинен містити файл buildspec.yml. Даний

файл містить інструкції для побудови застосування користувацького інтерфейсу. Такі інструкції вказуються у фазах: install, pre_build, build та postbuild. Зазвичай, фаза install містить команди для встановлення Node.js та пакетного менеджера (npm, або yarn); фаза pre_build містить команди для встановлення залежностей застосування (npm install або yarn install); фаза build містить команди для компіляції застосування (npm build або yarn build). Також, гарною практикою є запуск тестів на цьому етапі. У результаті цього етапу, застосування було скомпільовано у набір із декількох HTML, CSS та JS файлів. CloudPipeline підтримує доставку контенту на сервіс S3. Досягається це вибором відповідного S3 bucket та шляху. За замовчуванням, контент буде доставлений у вигляді архіву, тому варто вказати, що файли повинні бути розархівовані перед розміщенням. Таким чином, було досягнуто автоматизацію достави статичного контенту на сервіс S3.

3.3 Розробка основного коду веб-застосування

Як вже було зазначено, у застосунку використовується веб-фреймворк FastAPI для досягнення високої швидкості обробки запитів. Встановлення цього фреймворку відбувається за допомогою pip. Варто зазначити, що FastAPI поставляється із декількома важливими залежностями. По-перше, використовується залежність pydantic, яка дозволяє описувати сутності із використанням анотацій типу [14]. Це надає можливість валідувати дані на етапі ініціалізації екземпляру сутності. По-друге, використовується залежність starlette. Starlette - це фреймворк, який виступає у ролі middleware між FastAPI та асинхронним веб-сервером [15]. Він надає можливість використовувати сесії, cookies, підтримує протокол WebSocket. Також, за замовчуванням, FastAPI поставляється із пакетами, які автоматично генерують документацію для REST API.

Для взаємодії із базою даних та ORM використовується пакет sqlalchemy. Даний пакет складається з двох основних компонентів: core (ядро) та ORM. Core представляє собою інструментарій абстракції над SQL, що дозволяє виконувати запити за допомогою синтаксису мови Python. ORM надає можливість об'єктно-реляційного відображення сутностей.

Компонент бізнес-логіки є сервісом, який дотримується принципу REST. Він описує певну кількість REST-ресурсів, як зазначено у розділі структурної розробки веб-застосування. Дані ресурси можуть мати реалізацію певних HTTP-методів до яких може звернутися користувач за допомогою користувацького інтерфейсу. Метод GET - для отримання даних; POST - для створення об'єкту, PUT та PATCH для зміни атрибутів об'єкту; DELETE - для видалення об'єкту. Принцип REST пропагує використання серіалізованих форматів даних: XML або JSON. Через велику надлишковість формату XML, зазвичай у REST API використовується формат JSON. Окрім того, формат JSON походить від мови JavaScript, що суттєво спрощує комунікацію між користувацьким інтерфейсом та компонентом бізнес-логіки.

Першим кроком у розробці веб-застосування із використанням FastAPI є опис сутностей, які будуть використані. Дані сутності поділяються на два види: схема (schema) та модель (модель). За форматом опису сутностей ці два види є досить схожими, адже вони описують певний клас, атрибути класу та типи цих атрибутів. Проте, існує суттєва відмінність при використанні цих класів. Клас схеми наслідує клас pydantic.BaseModel. Згодом, цей клас використовується у FastAPI для валідації, серіалізації даних та автоматичного створення документації. Кожен із атрибутів класу повинен мати анотацію типу для коректної валідації та може задавати значення за замовчуванням для цих атрибутів. Клас моделі наслідує базовий клас пакету sqlalchemy. Класи моделі використовуються для об'єктно-реляційного відображення даних. Фактично, клас моделі описує

певну таблицю та її відношення. Окрім того, кожен такий клас повинен містити обов'язковий атрибут __tablename__, який дозволяє визначити назву таблиці у реляційній базі даних. Для під'єднання до бази даних використовується метод пакету sqlaclhemy create_engine. Обов'язковим параметром цього методу є URL-адреса бази даних. При успішному підключені до бази даних повертається об'єкт типу Engine, із використанням якого, згодом можна виконувати запити. Варто зазначити, що за замовчування sqlalchemy не підтримує міграції бази даних, проте існують сторонні пакети, які вирішують цю проблему: alembic, sqlalchemy-migrate.

Опісля створення потрібних сутностей схеми та моделей, гарною практикою є реалізація так званих CRUD-методів. CRUD, або create, read, update, delete - це функції, які використовуються для взаємодії із базою даних. Основною перевагою виділення цих функцій в окремі методи є можливість перевикористання коду. Такі методи зв'язують функціонал класів схеми та моделі. Наприклад, при створенні об'єкту Conversation, передається екземпляр класу схеми, який містить інформацію про об'єкт. Використовуючи цю інформацію, ініціалізується екземпляр класу моделі та згодом цей екземпляр зберігається до бази даних. Таким чином, дані валідуються двічі: при створенні екземпляру класу схеми та при записі до бази даних. У свою чергу, це дозволяє підвищити рівень цілісності даних. Безумовно, описувати потрібно не усі методи, а тільки ті, що будуть використані у основній системі.

Маючи описані методи для взаємодії із сутностями програмного продукту, варто приділити увагу створенню так званих endpoints, або точок доступу. FastAPI пропагує використання одного або декількох файлів, які відповідають за обробку запитів. Якщо веб-застосування має невелику кількість точок доступу має сенс використовувати один файл main.py. Проте, в інакшому випадку, логічним є розділенні обробників запитів за

ресурсами. Обробник запитів у FastAPI являє собою декоровану функції. Декоратори бувають різних типів та знаходяться у основному екземплярі класу FastAPI. Фактично, декоратор - це функція, яка певним чином змінює функціонал декорованої функції. Клас FastAPI містить декоратори для кожного HTTP-метода. Також, ці декоратори можуть приймають певні аргументи. Обов'язковим є аргумент path, який визначає шлях обробника. Наприклад, декоратор вигляду @app.get("/") дозволяє обробити HTTP запити із метод GET, які надсилаються на кореневу URL-адресу. Окрім того, декоратор може приймати аргумент response_model, який використовується для валідації відповіді, та автоматичного створення документації. Сама ж функція обробника може містити певні аргументи, які передаються. Наприклад, досить типовим прикладом є передача ідентифікатора об'єкту в URL-адресі. Окрім цього, FastAPI пропагує використання dependency injection. Dependency injection дозволяють оголошувати певні частини коду, які є необхідними для виконання функції. Це є досить зручним способом перевикористовувати певні частини коду в декількох функціях, ділитися з'єднаннями із базою даних, забезпечувати підтримку засобів автентифікації, тощо. Для використання такої можливості потрібно імпортувати клас Depends з модулю FastAPI, та передати в конструктор об'єкт, який можна викликати (callable).

Одним із ключових аспектів розробленої системи є автоматичний відображення нових повідомлень у бесіді без оновлення сторінки. Такий функціонал можна реалізувати декількома способами: надсилання запиту із клієнтського боку на отримання нових даних, та автоматичне надсилання даних від сервера до клієнта. Перший спосіб є досить простим у реалізації, проте великим недоліком є потреба константно надсилати запити з певною затримкою. Такий підхід створює велику кількість зайвого трафіку та суттєво навантажує веб-сервер. Протокол WebSocket дозволяє веб-серверу надсилати нові дані клієнту [16]. При навігації на певну сторінку

реєструється нове WebSocket з’єднання за допомогою JavaScript. Встановлене з’єднання існує допоки користувач знаходиться на певній сторінці веб-застосування. При реєстрації WebSocket з’єднання бекенд частина застосування отримує сесію або cookie, за допомогою яких є можливість отримати об’єкт користувача та згодом надсилати певні нові дані конкретному клієнту. Таким чином, використання WebSocket дозволяє ефективно відображати користувачам нові повідомлення за потреби.

Висновки

У результаті даної роботи було:

- проаналізовано різні архітектурні підходи у побудові веб-застосувань та різні способи розміщення;
- створено веб-застосування, яке надає користувачам можливість обмінюватися повідомленнями та файлами;
- досягнуто можливість автоматичного відображення нових повідомлень;
- використано багаторівневий принцип побудови архітектури.

Компонент бізнес-логіки було розроблено із використанням мови програмування Python та технологій FastAPI, sqlalchemy, uvicorn, starlette та pydantic. Розміщення веб-застосування відбувається за допомогою сервісів AWS: EC2, S3, RDS, CloudFront, Route53, Application Load Balancer.

Розроблене веб-застосування є прототипом, який наочно демонструє взаємодію із хмарними сервісами Amazon. Особливу увагу приділено швидкодії виконанню коду завдяки використанню веб-фреймворку з підтримкою асинхронної обробки запитів.

Список використаної літератури

1. Jadeja, Yashpalsinh, and Kirit Modi. "Cloud computing-concepts, architecture and challenges." 2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET). IEEE, 2012.
2. Dillon, Tharam, Chen Wu, and Elizabeth Chang. "Cloud computing: issues and challenges." 2010 24th IEEE international conference on advanced information networking and applications. IEEE, 2010.
3. IaaS, PaaS and SaaS [Електронний ресурс] - Режим доступу: https://www.ibm.com/cloud/learn/iaas-paas-saas
4. Amazon Elastic Compute Cloud [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/
5. AWS Lambda [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
6. Amazon Relational Database Service [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide
7. Amazon Simple Storage Service [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
8. Amazon Elastic Container Service [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonECS/latest/developerguide
9. Amazon Route53 [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide
10. Amazon CodePipeline [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/codepipeline/latest/userguide
11. What is CDN? [Електронний ресурс] - Режим доступу: https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

12. Webhooks [Електронний ресурс] - Режим доступу: https://developer.github.com/webhooks/
13. FastAPI Documentation [Електронний ресурс] - Режим доступу: https://fastapi.tiangolo.com/
14. Pydantic Documentation [Електронний ресурс] - Режим доступу: https://pydantic-docs.helpmanual.io/
15. Starlette Documentation [Електронний ресурс] - Режим доступу: https://www.starlette.io/
16. The WebSockets API [Електронний ресурс] - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Додатки

Додаток А

(Обов’язковий)

Файл appspec.yml

```
os: linux
files:
  - source: /
    destination: /
hooks:
  BeforeInstall:
    - location: scripts/install_dependencies.sh
      timeout: 300
      runas: root
  ApplicationStart:
    - location: scripts/start_server.sh
      timeout: 300
      runas: root
  ApplicationStop:
    - location: scripts/stop_server.sh
      timeout: 300
      runas: root
```

Додаток Б

(Обов’язковий)

Файл main.py

```
from datetime import timedelta
from typing import List

import jwt
import uvicorn
from fastapi import Depends, FastAPI, HTTPException, status
from jwt import PyJWTError
from sqlalchemy.orm import Session

from auth import Token, authenticate_user,
ACCESS_TOKEN_EXPIRE_MINUTES, \
    create_access_token, create_user_instance, oauth2_scheme,
SECRET_KEY, ALGORITHM, \
    get_user
from crud import crud_conversation
from database import Base, engine, SessionLocal
from schemas.conversation import Conversation, ConversationCreate
from schemas.message import Message
from schemas.user import UserLogin, UserCreate, UserInDB

app = FastAPI()
Base.metadata.create_all(engine)

def get_db():
    try:
        db = SessionLocal()
        yield db
    finally:
        db.close()

async def get_current_user(token: str = Depends(oauth2_scheme), db:
Session = Depends(get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY,
algorithms=[ALGORITHM])
        email: str = payload.get("sub")
        if email is None:
            raise credentials_exception
```

Додаток Б

(Продовження)

```
    except PyJWTError:
        raise credentials_exception
    user = get_user(db, email=email)
    if user is None:
        raise credentials_exception
    return user


@app.get("/api/conversations", response_model=List[Conversation])
def get_conversations(
        db: Session = Depends(get_db),
        current_user: UserInDB = Depends(get_current_user)
):
    return crud_conversation.get_conversations(db=db,
user=current_user)

@app.post("/api/conversations", response_model=Conversation)
def create_conversation(
        conversation: ConversationCreate,
        db: Session = Depends(get_db),
        current_user: UserInDB = Depends(get_current_user)
):
    return crud_conversation.create_conversation(db=db,
user=current_user, conversation=conversation)

@app.get("/api/conversations/{conversation_id}")
def get_conversation(
        conversation_id: int,
        db: Session = Depends(get_db),
        current_user: UserInDB = Depends(get_current_user)
):
    return crud_conversation.get_conversation(
        db=db,
        user=current_user,
        conversation_id=conversation_id
    )

@app.post("/api/conversations/{conversation_id}")
def create_message(
        message: Message,
        conversation_id: int,
        db: Session = Depends(get_db),
        current_user: UserInDB = Depends(get_current_user),
):
    return crud_conversation.create_message(
        db=db,
```

Додаток Б

(Продовження)

```
        user=current_user,
        conversation_id=conversation_id,
        message=message,
    )

@app.post("/api/users", response_model=UserInDB)
def create_user(user: UserCreate, db: Session = Depends(get_db)):
    return create_user_instance(db=db, user=user)


@app.post("/api/token", response_model=Token)
async def login_for_access_token(user: UserLogin, db: Session =
Depends(get_db)):
    user = authenticate_user(db, user.email, user.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires =
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.email}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}


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

Додаток В

(Обов’язковий)

Файл database.py

```
from sqlalchemy import create_engine

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker

import os

engine = create_engine(os.getenv("POSTGRES_URL"))

SessionLocal = sessionmaker(autocommit=False,

autoflush=False, bind=engine)

Base = declarative_base()
```

Додаток Г

(Обов’язковий)

Файл auth.py

```
from datetime import datetime, timedelta
import jwt
from fastapi import HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from passlib.context import CryptContext
from pydantic import BaseModel
import models
SECRET_KEY = "b0f59c78856f56742920afe780cf2135"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

class Token(BaseModel):
    access_token: str
    token_type: str

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token")

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)

def get_user(db, email: str):
    return db.query(models.User).filter_by(email=email).one()

def authenticate_user(db, username: str, password: str):
    user = get_user(db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user

def create_access_token(*, data: dict, expires_delta: timedelta =
None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
```

Додаток Г

(Продовження)

```
    expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY,
algorithm=ALGORITHM)
    return encoded_jwt

def create_user_instance(db, user):
    if db.query(models.User).filter_by(email=user.email).count() <
1:
        user_dict = user.dict()
        user_dict.pop('password')
        db_item = models.User(**user_dict)
        db_item.hashed_password = get_password_hash(user.password)
        db.add(db_item)
        db.commit()
        db.refresh(db_item)
        return db_item
    raise HTTPException(
        status_code=status.HTTP_405_METHOD_NOT_ALLOWED,
        detail="User already exists"
    )
```

Додаток Ґ

(Обов’язковий)

Файл crud/crud_conversation.py

```
from models import Conversation, Message

def get_conversations(db, user):
    return
db.query(Conversation).filter_by(creator_id=user.id).all()

def create_conversation(db, user, conversation):
    db_item = Conversation(**conversation.dict())
    db_item.creator_id = user.id
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item

def get_conversation(db, user, conversation_id):
    return db.query(Message).join(Conversation).filter(
        (Message.conversation_id == conversation_id) &
(Conversation.creator_id == user.id)
    ).all()

def create_message(db, user, conversation_id, message):
    db_item = Message(
        conversation_id=conversation_id,
        creator_id=user.id,
        **message.dict()
    )
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

Додаток Д

(Обов’язковий)

Файл models.py

```
from sqlalchemy import Column, Integer, ForeignKey, DateTime, String
from sqlalchemy.orm import relationship

from database import Base, engine
from models import Conversation, User

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)

class Message(Base):
    __tablename__ = "messages"
    id = Column(Integer, primary_key=True, index=True)
    conversation_id = Column(Integer, ForeignKey(Conversation.id))
    conversation = relationship(Conversation)
    creator_id = Column(Integer, ForeignKey(User.id))
    user_id = Column(Integer, ForeignKey(User.id))
    text = Column(String, nullable=True)
    attached_file = Column(String, nullable=True)
    created_at = Column(DateTime)

class Conversation(Base):
    __tablename__ = "conversations"
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String)
    creator_id = Column(Integer, ForeignKey(User.id))
    user_id = Column(Integer, ForeignKey(User.id))
    created_at = Column(DateTime)


Base.metadata.create_all(engine)
```

Додаток Е

(Обов’язковий)

Файл schemas/conversation.py

```
from typing import Optional

from pydantic import BaseModel
from datetime import datetime


class ConversationBase(BaseModel):
    title: Optional[str] = None
    created_at: datetime = datetime.now()


class ConversationBaseInDB(ConversationBase):
    id: int = None
    creator_id: int = None
    user_id: int = None

    class Config:
        orm_mode = True


class Conversation(ConversationBaseInDB):
    pass


class ConversationCreate(ConversationBaseInDB):
    title: str
    user_id: int
```

Додаток Є

(Обов’язковий)

Файл schemas/message.py

```
from typing import Optional

from pydantic import BaseModel
from datetime import datetime


class MessageBase(BaseModel):
    text: Optional[str] = None
    attached_file: Optional[str] = None
    created_at: datetime = datetime.now()


class MessageBaseInDB(MessageBase):
    id: int
    creator_id: int
    conversation_id: int

    class Config:
        orm_mode = True


class Message(MessageBase):
    pass
```

Додаток Є

(Обов’язковий)

Файл schemas/user.py

```
from pydantic import BaseModel


class User(BaseModel):
    email: str
    name: str


class UserCreate(User):
    password: str


class UserInDB(User):
    id: int

    class Config:
        orm_mode = True


class UserLogin(BaseModel):
    email: str
    password: str
```

Додаток Ж

(Обов’язковий)

Документація REST API

«___» _____ 2019 року

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

Студенту факультету інформатики спеціальності «Комп'ютерні науки та інформаційні технології» Галенку Денису Олеговичу

Вихідні дані:

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Календарний план

Анотація

Вступ

Розділ 1. Аналіз існуючих рішень

Розділ 2. Структурна розробка веб-застосування

Розділ 3. Розробка одного із компонентів веб-застосування

Список використаної літератури

Додатки

Дата видачі «_____» _____ 2019 року

Керівник Черкасов Д. І. _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання роботи:

	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	17.11.2019	
2.	Ознайомлення з предметною областю.	21.11. 2019	
3.	Формулювання вимог до розробленої системи.	25.12.2019	
4.	Пошук літератури та написання першого розділу	26.01.2020	
5.	Побудова структури веб-застосування (другий розділ)	14.02.2020	
6.	Розробка архітектури веб-застосування	26.02.2020	
7.	Початок виконання практичної частини	05.03.2020	
8.	Написання третього розділу	24.03.2020	
9.	Аналіз виконаної роботи з керівником	14.04.2020	
10.	Створення слайдів презентації та написання тексту доповіді	16.04.2020	
11.	Корегування роботи	17.04.2020	
12.	Здача роботи на перевірку на плагіат.	19.04.2020	
13.	Захист курсової роботи	24.04.2020	

Студент **Галенок Д. О.**

Керівник **Черкасов Д. І.**

“_____”_____