Міністерство освіти і науки України

Національний університет «Києво-Могилянська Академія»

Факультет інформатики

Кафедра мережних технологій

Кваліфікаційна робота

освітній ступінь - бакалавр

на тему: «**РОЗРОБКА БАГАТОРІВНЕВОГО ВЕБ-ЗАСТОСУВАННЯ НА ХМАРНІЙ ПЛАТФОРМІ «AWS»»**

Виконав: студент 4-го року навчання,

Спеціальності

121 Інженерія програмного забезпечення

Безрука Анастасія Володимирівна

Науковий керівник:

Черкасов Дмитро Іванович

кандидат технічних наук, старший викладач

Рецензент: _____

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

“___” _____ 202_ р.

Київ - 202_

ЗМІСТ

Міністерство освіти і науки України

Національний університет «Києво-Могилянська Академія»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри мережних технологій

Малашонок Г. І.

_____

(підпис)

“____” _____202_ р.

ЗАВДАННЯ ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

студенту Безрукій Анастасії Володимирівні факультету інформатики 4-го курсу

ТЕМА: Розробка багаторівневого веб-застосування на хмарній платформі «AWS»

Вихідні дані:

- Програмний код рівня бізнес-логіки
- Конфігурація розгортання компонентів

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Анотація

Вступ

1 Огляд існуючих рішень

2 Структурна розробка власного рішення

3 Детальна розробка та розгортання компонентів

Висновки

Список літератури

Додатки

Дата видачі “___” _____ 202_ р.

Науковий керівник Черкасов Д. І. _____

(підпис)

Завдання отримав _____

(підпис)

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Примітки
1.	Узгодження теми та плану кваліфікаційної роботи	жовтень 2023		
2.	Вивчення джерел та літератури за темою роботи	листопад-грудень 2023		
3.	Огляд існуючих рішень	грудень 2023		
4.	Аналіз архітектурних підходів, базових технологій розгортання та платформ для розміщення інфраструктури	січень 2023		
5.	Розробка структури застосування, опис компонентів та функціонування системи	лютий-березень 2023		
6.	Детальна розробка та розгортання компонентів	березень-квітень 2023		
7.	Написання висновків, обговорення роботи із науковим керівником	квітень 2023		
8.	Оформлення роботи згідно з вимогами	початок травня 2023		
9.	Створення презентації та написання доповіді для захисту кваліфікаційної роботи	початок травня 2023		
10.	Коригування роботи за результатами попереднього захисту	травень 2023		
11.	Здача кваліфікаційної роботи на відгук, рецензію та перевірку на відповідність вимогам академічної доброчесності	20 травня 2023		
12.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	31 травня, 1-2 червня 2023		

Графік узгоджено “____”_____ 202_ року

Науковий керівник Черкасов Дмитро Іванович

Виконавець кваліфікаційної роботи Безрука Анастасія Володимирівна

Анотація

Метою даної роботи є розробити багаторівневе веб-застосування на хмарній платформі «AWS» для ресторанного бізнесу. Додаток містить електронне меню, керування замовленням, систему відгуків та збір статистичних даних про заклад.

У роботі зроблено огляд існуючих рішень, аналіз архітектурних підходів, базових технологій розгортання та платформ для розміщення застосунку. На основі цього було спроектовано структуру рішення, з описом компонентів та взаємодію між ними. У заключному розділі роботи продемонстровано детальну розробку модуля API та розгортання необхідної інфраструктури у хмарному провайдері «AWS», за допомогою конвеєра неперервної інтеграції та доставки і технології «інфраструктура як код».

ВСТУП

Протягом останніх років все більше застосунків будуються з використанням хмарних обчислень. Причиною такої тенденції є певні переваги такого підходу, такі як: надійність, гнучкість, безпека, ефективне масштабування та конфігурування. Також використання хмарних технологій дає можливість розробникам абстрагуватися від апаратної платформи, що спрощує процес розробки та підтримки застосувань. За певних обставин (наприклад, у разі помірної інтенсивності використання) хмарні технології дозволяють знизити вартість підтримки застосунку, оскільки оплата здійснюється лише за ті ресурси, які реально використовуються.

На сучасному ринку є чимало хмарних платформ. Найбільшої популярності здобули «Amazon Web Services» («AWS»), «Microsoft Azure», «Google Cloud Platform» [1]. В даній роботі використовується «Amazon Web Services» («AWS»). Ця платформа містить чимало додаткових сервісів, що допомагають спросити розробку, моніторинг та управління застосунком, а також вона інтегрується з багатьма мовами програмування, зокрема з Java, яку буде використано у розробці.

У ролі веб-застосування було обрано застосунок для автоматизації ресторанного бізнесу, оскільки такий вид підприємництва є досить поширеним. Додаток міститиме коротку інформацію про заклад, електронне меню, можливість оформлення замовлення, без допомоги персоналу, його оплата різними способами, резервування столиків та систему відгуків про обслуговування.

Цей застосунок потребує масштабованості, адже ресторан може розширитись до мережі, високої доступності, оскільки завжди є відвідувачі, гнучкості, тому що кількість клієнтів у певні дні та години буде вищою, у інші ж навпаки досить малою. Він узгоджується з основними принципами хмарних обчислень.

Дана робота має на меті розробити багаторівневий веб-застосунку для ресторану, що допоможе автоматизувати деякі процеси цього бізнесу. На додачу, виконати огляд існуючих рішень, проаналізувати архітектурні підходи, технології розгортання та платформи для розміщення інфраструктури, розробити структуру застосунку. І ще одним важливим завданням є детальна розробка модуля API та його автоматичне розгортання у хмарному провайдері «AWS».

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд застосунків, що використовуються у ресторанному бізнесі

Ресторанний бізнес існує вже достатньо довго, а автоматизація у цій сфері не є чимось новим. Більшість закладів громадського харчування використовує застосунки хоча б для деяких процесів. На ринку є багато програмного забезпечення, яке вирішує задачі, що з’являються під час ведення ресторанного бізнесу. Оскільки існує багато застосунків для різних цілей, спочатку виділимо функції, що будуть реалізовані у власному рішенні, і проведемо аналіз вже існуючих з подібним функціоналом.

1.1.1 Короткий опис функціоналу

Очевидно, що повністю автоматизувати заклад громадського харчування на даному етапі розвитку технологій не є доцільним. До прикладу, приготування страв роботом, цілком реальне, але ризиковане та дороге рішення. Проте є й процеси, автоматизація яких можлива, за допомогою, застосунку, без використання великої кількості машин. До них належать наступні: облік ресурсів, прийом та передача замовлень на відповідний виробничий процес та їхня оплата, документообіг, резервування столиків, обрахунок робочого часу та інші. [2]

І хоча автоматизація внутрішніх процесів є важливою, я вважаю, що на успіх закладу громадського харчування найбільше впливає задоволеність клієнтів. Саме тому, у власному рішенні буде реалізовано ті функції, які використовуються безпосередньо гостями ресторану. Наведемо їхній перелік:

а) електронне меню;

б) керування замовленням гостями, його оплата (як у закладі, так і на самовиніс чи доставку);

в) резервування столиків;

г) система відгуків про обслуговування.

Для кращого розуміння функціоналу на рисунку 1.1 зображено діаграму зміни станів застосунку.

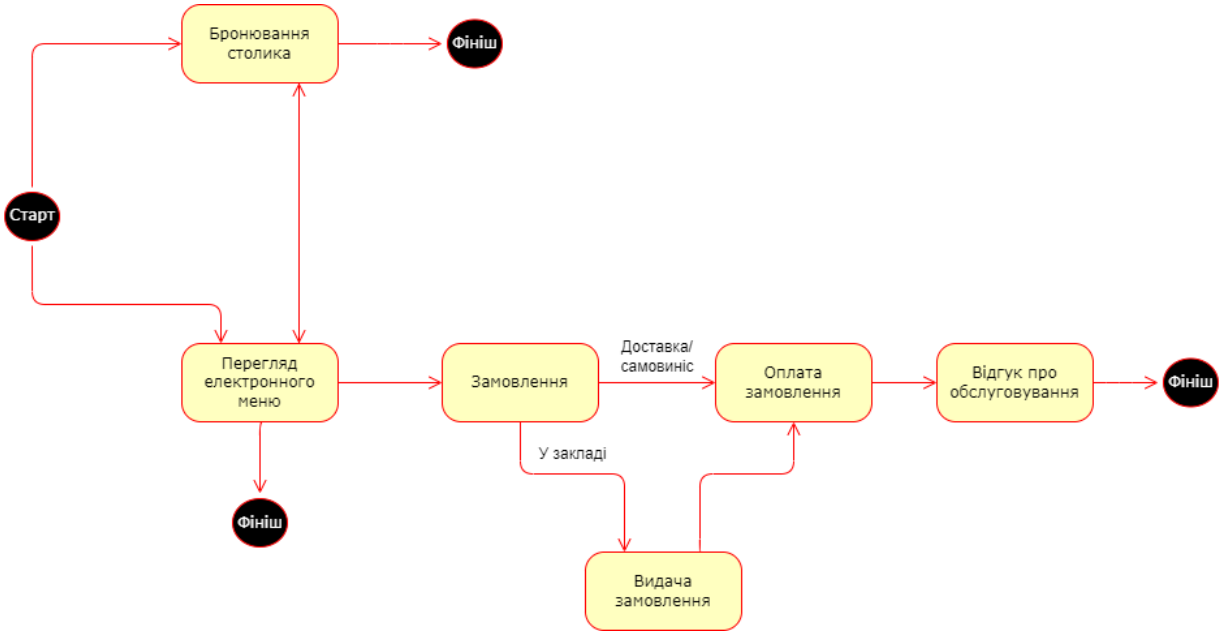


Рисунок 1.1 - Діаграма зміни станів застосунку

1.1.2 Порівняння застосунків для ресторанного бізнесу

Розглянемо застосунки, що хоча б частково реалізовують наведений вище функціонал. Варто зазначити, що детальний аналіз рішень не є метою даної роботи. На жаль, зрозуміти, яка їхня архітектура та де його розміщено, що було б найбільш релевантним до даної роботи, не вдалося. Саме тому порівняння базується лише на досвіді їхнього використання та наявному у них функціоналі.

Список застосунків, які оглянемо:

а) «Choice»;

б) «OddMenu»;

в) «expirenza».

Для їхнього огляду було знайдено заклади, що користуються цими рішеннями. У додатку А наведено знімки екрану інтерфейсу застосунків, як з персонального комп’ютера так і телефону та посилання на них, за винятком «expirenza», для якого було знайдено лише мобільну версію.

1.1.2.1 Огляд застосунку «Choice»

Для огляду додатку «Choice» було обрано ресторан «California» у місті Івано-Франківськ, що використовує дане рішення. На кожному зі столів у закладі розміщено спеціальний QR-код, який переправляє нас на сайт. Якщо у пошуку в браузері ввести назву закладу, ми також легко побачимо даний веб-застосунок у списку результатів. На сайті присутнє меню з фото, контакти закладу та соцмережі. У ньому є можливість оформити замовлення, не шукаючи офіціанта, інтегрована доставка та присутня можливість залишити відгук. Оплату картою також можна здійснити на сайті чи обрати готівку та розрахуватись пізніше. Інтерфейс зручний та інтуїтивно зрозумілий. Є можливість обрати мову, проте переклад автоматичний, що і зазначено в застосунку. Серед недоліків варто відзначити відсутність інформації про сам заклад, його концепцію, історію тощо, що було б своєрідною візиткою.

1.1.2.2 Огляд застосунку «OddMenu»

Схожим до «Choice» є застосунок «OddMenu». Його було оглянуто на прикладі бару «Copper head», що розташований у місті Івано-Франківськ. Принцип роботи такий же - потрібно просканувати QR-код. Варто відзначити, що на відміну від попереднього рішення, просто ввівши назву закладу в пошуковий рядок, знайти швидко сайт не вдасться. У застосунку відображається меню, адреса та контактні дані. На жаль, зробити замовлення чи

оформити доставку в цьому рішенні не вийде. Інтерфейс простий, проте зручний. На відміну від «Choice» є можливість переглянути меню лише трьома мовами: українською, англійською та російською. При цьому локалізація погано працює: в англійській версії опис багатьох страв та напоїв все одно залишається українською. Серед недоліків також відсутність можливості залишити відгук та інформації про ресторан.

1.1.2.3 Огляд застосунку «expirenza»

Останнім популярним застосунком є «expirenza», який було оглянуто на прикладі кафе «Manufactura». Посилання на застосунок не було знайдени у браузері: відкрити його можливо лише після сканування QR-коду. У ньому можна переглянути меню, проте зробити замовлення можливо лише на доставку (не у всіх версіях рішення), оплатити рахунок за допомогою «Google Pay» та залишити відгук. У «expirenza» також відсутня інформація про заклад. Користувацький інтерфейс теж зручний, та інтуїтивно зрозумілий. Що стосується локалізації: застосунок доступний українською та англійською, а переклад відображається правильно.

1.1.4 Висновок

Отже, хоча оглянуті застосунки і містять свої переваги, вони не реалізовують увесь функціонал, який було описано. Всі вони мають доволі зручний та інтуїтивний інтерфейс та імплементують електронне меню. «Choice» та «expirenza» містять у собі дещо більше функціоналу, зокрема система відгуків та оплата замовлень, на відміну від «OddMenu». У жодному із застосунків немає можливості забронювати столик. Також, недоліком двох застосунків із трьох, окрім «Choice», є те, що вони не відображаються при пошуку в браузері закладу. Що стосується архітектури та розміщення

інфраструктури, то, на жаль, не вдалося з’ясувати знаходиться вона у публічній хмарі чи у приватному дата-центрі.

Тому у даній роботі ми спробуємо спроектувати застосунок, що реалізовував б усі бажані функціональні вимоги, та реалізуємо певні компоненти. При цьому використаємо переваги публічних хмарних платформ та продемонструємо їх.

1.2 Огляд архітектурних підходів

Архітектура застосунку достатньо широке поняття. Найкоротшим визначенням цього терміну, з яким погоджується більшість, є «найбільші компоненти системи і їхня взаємодія між собою».[3, с.2] Найпоширенішим підходом є розділення застосунку на рівні. Вони взаємодіють один з одним - зазвичай вищі використовують певні сервіси нижніх. Відповідно до їхньої кількості виділяють монолітну, 2-рівневу, 3-рівневу та багаторівневу системи. Також окремо розглянемо мікросервісну архітектуру, яка є наслідком подальшого розвитку горизонтального масштабування застосунків.

1.2.1 Монолітна архітектура

При монолітній архітектурі усі компоненти системи є одним цілим. Серед переваг такого підходу простота розробки та розгортання й економія витрат на ресурси. Найбільшим недоліком такого розміщення є висока зв’язність компонентів, що ускладнює додавання нового функціоналу. [4] Також це призводить до того, що помилка у роботі якогось компоненту впливає на працездатність усієї системи, і навіть на ті частини застосунку, що не мають відношення до проблеми.

Отже, беручи до уваги переваги та недоліки такого підходу, можна зробити висновок, що він підходить для невеликих і простих застосунків, які не будуть розширювати свій функціонал й оновлюватися достатньою часто.

1.2.2 2-рівнева архітектура

У 90х роках XX ст. з розвитком клієнт-серверних застосувань почали набувати популярності 2-рівневі застосунки. Архітектура таких систем поділена на клієнтську та серверну частину. На початку свого розвитку застосунок був поділений таким чином, що один рівень містив інтерфейс і бізнес-логіку, а інший - реляційну базу даних. [3, с.18] При такому підході у нас буде «товстий клієнт». Зараз ж домінує інший поділ - переважна більшість операцій виконується на серверній частині. Такий застосунок є моделлю із «тонким» клієнтом. Переваги другого підходу є очевидними: він не вимагає від користувачів значної обчислювальної потужності.

На мою думку, 2-рівневу архітектуру варто використовувати лише для нескладних застосунків. Хоч такий підхід і надає нам більшу гнучкість, ніж монолітна система, проте компоненти все ще є сильно зв’язними.

1.2.3 3-рівнева архітектура

Поступово бізнес-логіка ставала все складнішою, а оновлювати функціонал застосунку доводилось частіше. Саме через ці чинники поступово виникла 3-рівнева архітектура. ЇЇ підхід полягав у розділенні системи на три частини: користувацький інтерфейс, бізнес-логіка та база даних. [3, с.18-19] На рисунку 1.2 зображено схему 3-рівневої архітектури.

Рисунок 1.2 - Схема 3-рівневої архітектури

Цей підхід надає більше можливостей для масштабування. Таку систему значно легше підтримувати та оновлювати: можна змінювати лише один рівень. Відповідно помилка у одному з компонентів, не впливає на працездатність усього застосунку. [5]

1.2.4 Багаторівнева архітектура

Багаторівнева архітектура, як і 3-рівнева містить користувацький інтерфейс і базу даних. Різниця у тому, що між цими шарами, сервер бізнес-логіки може бути розділений на декілька, або додаються проміжні рівні, такі як: балансувальник навантаження, сервіс кешування, фільтрація запитів, тощо. На рисунку 1.3 розміщено приклад реалізації такої архітектури з деякими із можливих додаткових компонентів.

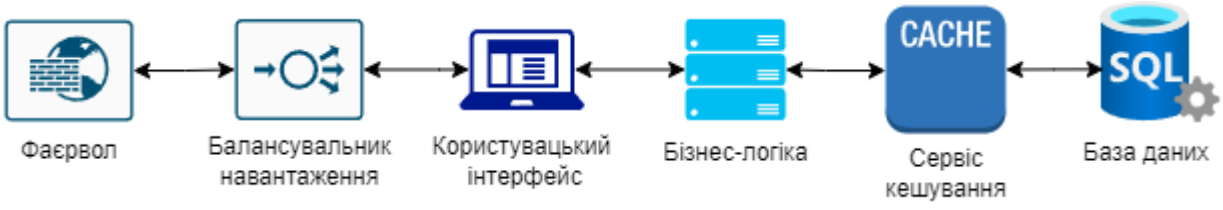


Рисунок 1.3 - Приклад реалізації багаторівневої архітектури

Переваги у такої системи, такі ж як і в 3-рівневої архітектури. При цьому збільшується продуктивність, гнучкість та масштабованість, безпека, у випадку додавання рівня фаєрволу, швидкодія, якщо використовувати сервіси кешування тощо. [5]

1.2.5 Мікросервісна архітектура

Мікросервісна архітектура набула широкої популярності в останні декілька років. При цьому підході застосування поділяється на декілька слабо зв’язаних сервісів, які комунікують між собою за допомогою різних протоколів, не обов’язково HTTP. При цьому для них може бути спільна база даних чи у кожного своя, в залежності від вимог до системи. [6] Схематично архітектуру розміщено на рисунку 1.4.

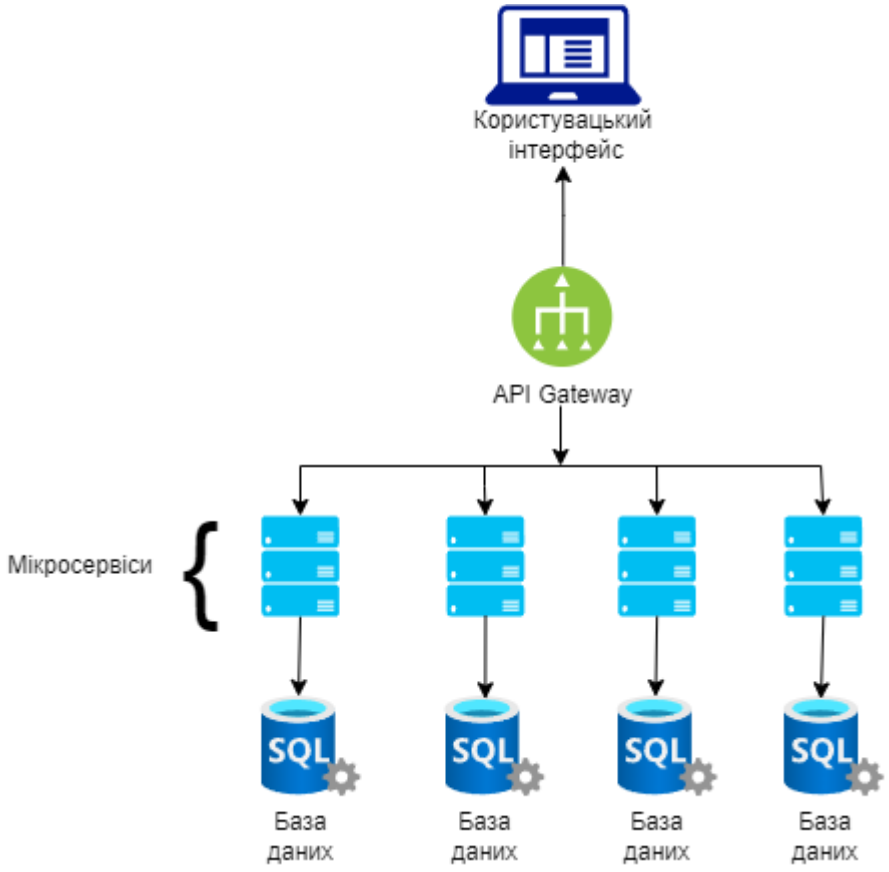


Рисунок 1.4 - Мікросервісна архітектура

Оскільки компоненти слабо пов’язані між собою, це надає нам гнучкість у розробці, оновленні та масштабуванні, зокрема надає можливість використовувати різні технологічні стеки для різних сервісів. Також помилки конкретного компоненту ізольовані. Недоліком є те, що зі збільшенням кількості сервісів зростає складність керування ними. [7] Також, у випадку

використання кількох розділених баз даних важче стає підтримувати узгодженість між ними.

1.2.6 Висновок

Отже, на основі перелічених переваг та недоліків кожної із архітектур, було вирішено розробляти багаторівневу систему з такими додатковими компонентами як: балансувальник навантаження, служба доставки контенту (CDN), фаєрвол для веб-застосунків. Можливо, при зростанні складності застосунку його варто буде розділити на мікросервіси за потреби.

1.3 Огляд базової технології

У цьому розділі оглянемо базові технології розгортання застосунку такі як: віртуальні машини, контейнери та безсерверні обчислення.

1.3.1 Віртуальні машини

Віртуалізація - це технологія створення віртуальних представлень чогось, до прикладу сервера, сховища, мережі тощо. Віртуальні машини фактично є цифровою копією фізичної. Вони дозволяють організаціям встановлювати кілька операційних систем і створювати різні середовища на одному сервері. Ця технологія дозволяє ефективно використовувати обчислювальні потужності фізичної машини, при цьому віртуальна не може безпосередньою взаємодіяти з нею. Для такого підходу потрібен гіпервізор - програмне забезпечення, що виступає посередником між апаратним забезпеченням та операційною системою хоста і віртуальною машиною. [8]

Серед переваг цієї технології, окрім ефективного використання ресурсів, також портативність і гнучкість. На додачу, віртуальні машини покращують безпеку кількома способами. По-перше, це фактично файл, який можна перевіряти на наявність вразливостей. По-друге, за допомогою знімків віртуальної машини, заражену програму можна відновити до стану, що був у певний момент часу в минулому. [9]

1.3.2 Контейнери

Контейнеризація - це упакування програмного коду, разом із залежностями, необхідними для роботи застосунку на будь-якій інфраструктурі. [7] Тобто ця технологія, як і віртуальні машини, забезпечує портативність застосунку - він може працювати на будь-якому пристрої чи операційній системі. Фактично, у обох підходів, використовується файл - зображення («image»), який можна запустити будь-де. Як і віртуальні машини, контейнери можна масштабувати створенням декількох екземплярів.

Однак дана технологія дещо відрізняється від попередньої. По-перше, для віртуальних машин потрібне встановлення програмного забезпечення для віртуалізації - гіпервізора. Для контейнерів ж потрібен застосунок, що буде посередником між ними та операційною системою. Найпопулярнішим на даний момент таким програмним забезпеченням є «Docker», хоча є й інші, до прикладу «LXC» («Linux Container»). По-друге, контейнери є більш легкими, їхній розмір значно менший. Пов’язано це з тим, що віртуальні машини містять власну операційну систему. Також контейнери надають менше контролю над середовищем поза контейнером. Віртуальні машини ж надають можливість керувати усім оточенням. [10] На рисунку 1.5 зображено схеми віртуальної машини та контейнера, для кращого розуміння вище зазначених відмінностей.

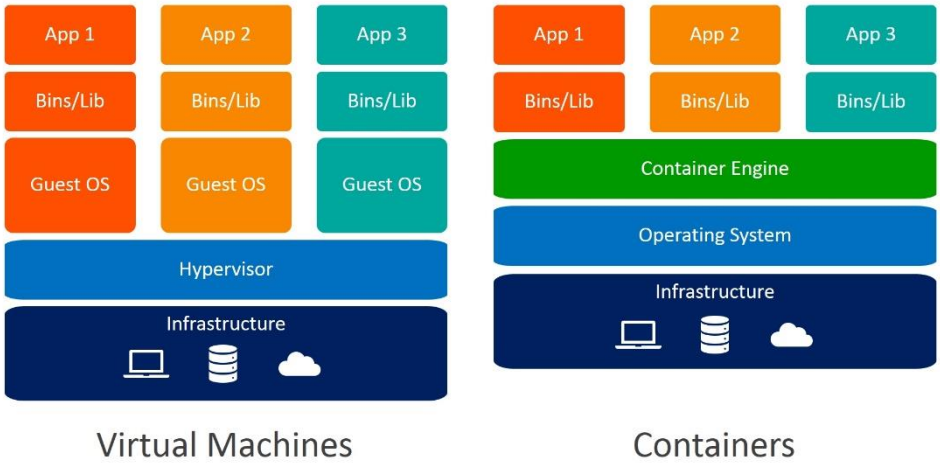


Рисунок 1.5 - Різниця між віртуальними машинами та контейнерами [11]

1.3.3 Безсерверні обчислення

Безсерверні обчислення тісно пов’язані з хмарними технологіями. При такому підході розробники створюють та запускають програми без необхідності керувати інфраструктурою. Нею керує провайдер хмарних обчислень. Використовуючи безсерверні обчислення, розробники можуть зосередитись на продукті, а не на інфраструктурі та середовищі виконання. [12] Ще однією перевагою, є автоматичне масштабування при використанні даної технології. Серед основних недоліків підходу: втрата контролю, складність інтеграційного тестування та погіршення безпеки, адже постачальник може запускати код різних користувачів на одному сервері. [13]

Усі хмарні провайдери надають можливість створювати програми, за допомогою безсерверних обчислень: у «AWS» він представлений сервісом «AWS Lambda», у «Azure» - «Azure Functions», у «Google Cloud Platform» - «Google Cloud Functions».

1.3.4 Висновок

Отже, розглянувши базові технології, що можуть бути використані при розгортанні, було прийнято рішення використовувати контейнери. Ця технологія надає нам чимало переваг: портативність, масштабування, швидкість. Також на відміну від віртуальних машин, їхній розмір в рази менший, що полегшує розгортання.

Безсерверні обчислення ж надають розробника можливість зосередитись на застосунку, а не інфраструктурі. Цю технологію також використаємо для двох додаткових компонентів. По-перше, за допомогою безсерверних обчислень розробимо систему збору статистики ресторану - які страви були популярними протягом певного періоду, визначення пікових годин тощо. По-друге, цей підхід буде корисним для автоматичної очистки бази даних від застарілих даних, до прикладу резервувань столиків, що були додані в систему давно. Звісно можна додати цю функціональність і в рівень бізнес-логіки, проте створення окремих компонентів зменшить навантаження на основний застосунок, а використання безсерверних обчислень полегшить їхню розробку та розгортання.

1.4 Огляд варіантів розміщення інфраструктури

Коли архітектуру та технологію обрано, потрібно вирішити, де розмістити інфраструктуру. Розглянемо два найбільш популярні варіанти: власний дата-центр та хмарні обчислення. Зазначимо, що існує також варіант розміщення на орендованих ресурсах, проте він не є широко поширеним. Також дуже часто виникає гібридне рішення - частина системи розташована локально, а частина у хмарі.

1.4.1 Розміщення інфраструктури на власних ресурсах (On-premise)

До появи хмарних обчислень майже усі застосунки розміщувались локально. Для цього бізнесу доводилось будувати власний дата-центр, що складався з самих обчислювальних ресурсів, мережі, сховищ даних та програмних продуктів. Протягом усього життєвого циклу застосунку потрібно обслуговувати інфраструктуру, відповідати за безпеку, налаштовувати і конфігурувати та оновлювати програмне забезпечення самостійно. Це є як перевагою, адже бізнесу належить повний контроль, так і недоліком, оскільки зростають витрати коштів і часу та потрібно наймати більше працівників. Також такий підхід довше впроваджувати, проте він зменшує залежність від зовнішніх факторів, таких як швидкісне підключення до мережі Інтернет.

1.4.2 Розміщення на хмарних платформах

Хмарні обчислення - це доступ до IT-ресурсів, таких як сервери чи програмне забезпечення, через Інтернет. Замість того, щоб будувати та підтримувати дата-центр, можна отримати доступ до технологічних послуг за потреби від хмарного постачальника. При цьому оплата здійснюється лише за те, що використовується. [14] Таким чином, компанія може зекономити кошти та час, делегуючи обслуговування дата-центру, провайдеру хмарних послуг.

Виділяють три основні типи хмарних обчислень, які ми коротко опишемо:

а) інфраструктура як послуга (IaaS) - доступ до мережевих функцій, комп’ютерів (віртуальних чи ні) і сховищ даних;

б) платформа як послуга (PaaS) - дозволяє розробляти та керувати власними програмами без необхідності керувати інфраструктурою (апаратним забезпеченням і операційними системами);

в) програмне забезпечення як послуга (SaaS) - надає готовий застосунок, розроблений провайдером, який можна використати. [14]

1.4.3 Порівняння розміщення на власних ресурсах та у хмарі

Для того, щоб краще зрозуміти переваги та недоліки розміщенні інфраструктури у хмарі та на власних ресурсах порівняємо їх за такими критеріями: розгортання, витрати, контроль, безпека, масштабованість, доступність. У таблиці 1.1 наведено основні відмінності цих рішень.

Таблиця 1.1 - Розміщення на власних ресурсах чи у хмарі

Критерії оцінювання	**Розміщення на власних ресурсах**	**Розміщення на хмарній платформі**
Розгортання	Власник самостійно забезпечує себе інфраструктурою: якщо наявних серверів не достатньо, щоб розгорнути чи розширити застосунок, потрібно докупити і налаштувати обладнання.	Провайдер надає всі ресурси за потребую, що надає можливість швидко розгортати інфраструктуру.
Витрати	Такий підхід потребує початкових інвестицій в апаратне та програмне забезпечення, а також чимало операційних витрати на обслуговування дата-центру (персонал, електроенергія, оновлення ліцензій, тощо).	Оплата здійснюється лише за ті ресурси, що використовуються, що у більшості випадків призведе до менших витрат.
Контроль	Усю систему контролює власник застосунку.	Організація віддає контроль над обладнанням та програмним забезпеченням на ньому постачальнику, а відповідає лише за власний застосунок.
Безпека	Оскільки організація сама контролює свій дата-центр, у неї є можливість забезпечити максимальну безпеку, дотримуючись різних практик.	Безпека хмари достатньо висока, проте провайдер не відповідає за застосунок, це відповідальність розробників.

Продовження таблиці 1.1

Критерії оцінювання	**Розміщення на власних ресурсах**	**Розміщення на хмарній платформі**
Масштабованість	Масштабованість контролюється власником, проте вона обмежена реальними розмірами дата-центру. Також часто виникає проблема, що не усі ресурси дата-центру використовуються, а їхнє обслуговування та пов'язані з цим витрати все одно лягають на компанію.	Оскільки розгортання інфраструктури відбувається на вимогу, масштабування швидке та гнучке.
Доступність	Завдяки повному контролю, доступність застосунку забезпечується організацією і зазвичай є достатньо високою.	Саме хмарний провайдер відповідає за доступність інфраструктури. Проте він також часто надає можливість розмістити свої ресурси у різних зонах чи регіонах, завдяки чому компанія може отримати максимальну доступність.

1.4.4 Висновок

Розміщення як на власних ресурсах, так і у хмарі має свої переваги та недоліки, які варто використати в залежності від потреб. Локальний дата-центр надає повний контроль, що призводить до високої безпеки та доступності. Хмарні обчислення ж спрощують розробку продукту, даючи змогу зосередитись на застосунку. Вони також надають нам швидке розгортання та масштабованість, і зменшують витрати. Оскільки додаток для ресторанного бізнесу не потребує підвищеної безпеки чи особливого апаратного забезпечення доцільним буде розмістити його у хмарі.

2. СТРУКТНА РОЗРОБКА ВЛАСНОГО РІШЕННЯ

2.1 Розробка архітектури рішення

На рисунку 2.1 зображено архітектуру власного рішення з такими рівнями: база-даних, бізнес-логіка, користувацький інтерфейс, два балансувальники навантаження (розташовані перед різними рівнями), кожен з яких захищений фаєрволом, а також служба доставки контенту. Застосунок розміщуватиметься у приватній мережі, у регіоні «eu-central-1», у двох зонах доступу («eu-central-1a» та «eu-central-1b»). При збільшенні кількості користувачів, наприклад у випадку, коли ресторан стає мережею, можна розміщувати застосунок навіть у декількох регіонах.

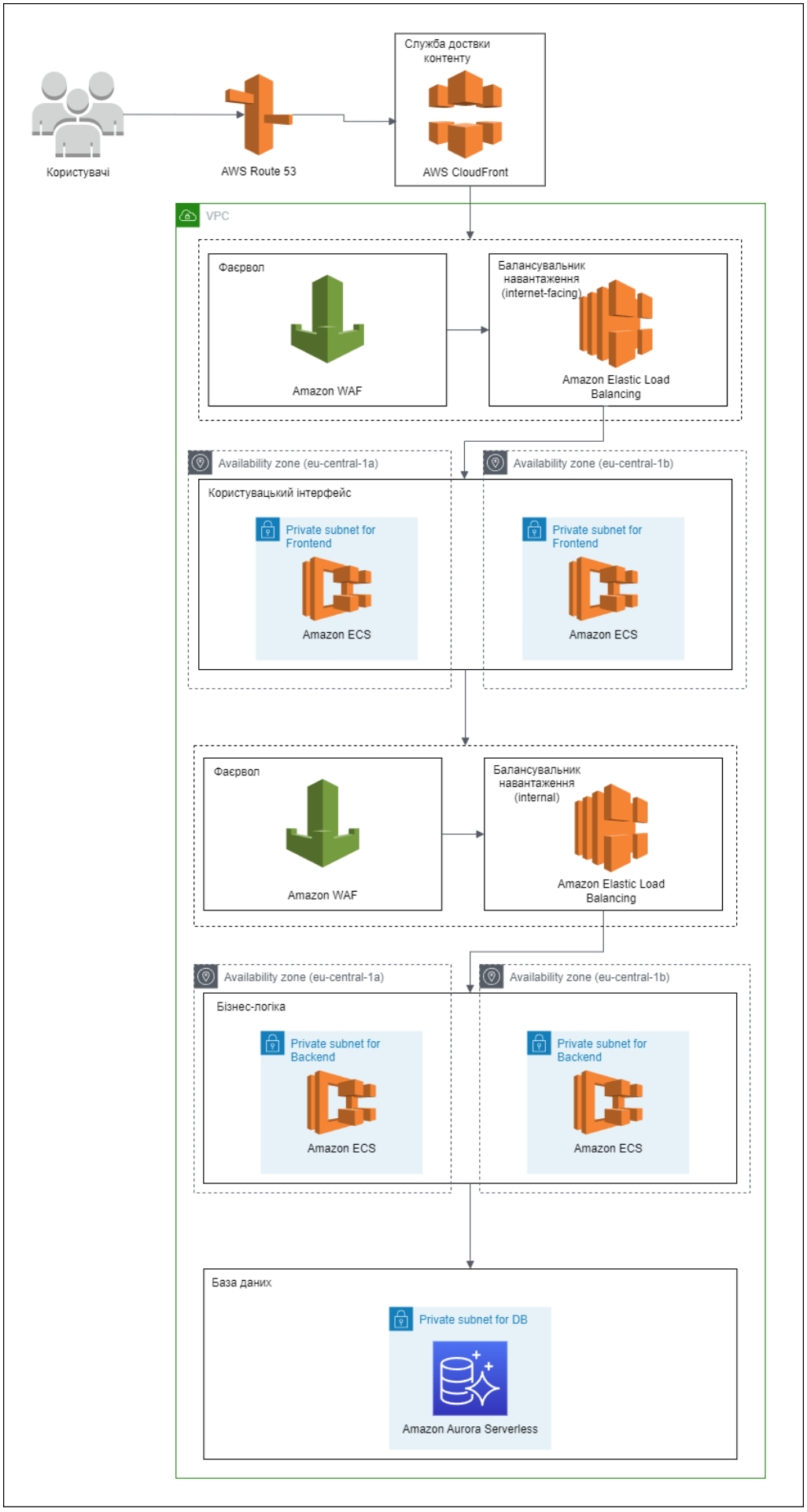


Рисунок 2.1 - Схема архітектури власного рішення

На рисунку 2.2 зображено схему «AWS Lambda» для збору статистики про ресторан адміністратором, а на рисунку 2.3 - безсерверну архітектуру для автоматичної очистки бази даних. Детальніше розглянемо їх у підрозділі 2.4.

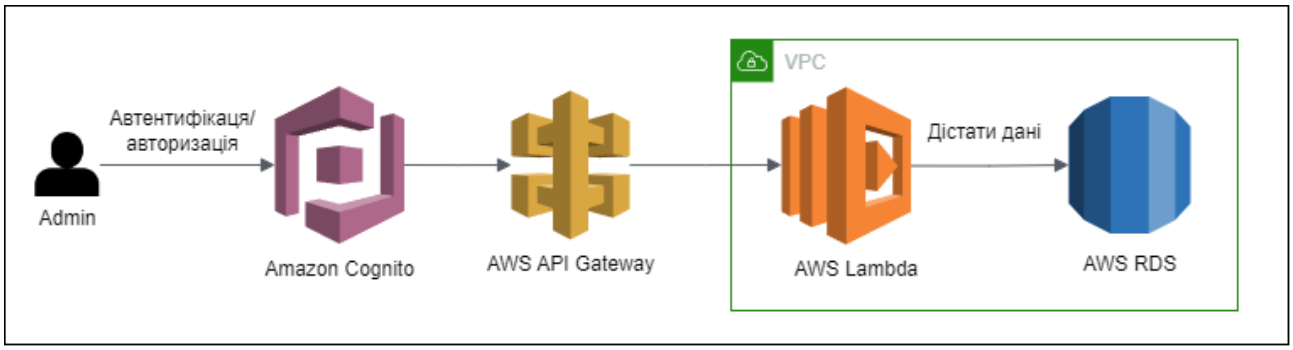


Рисунок 2.2 - Архітектура компоненту для збору статистики ресторану

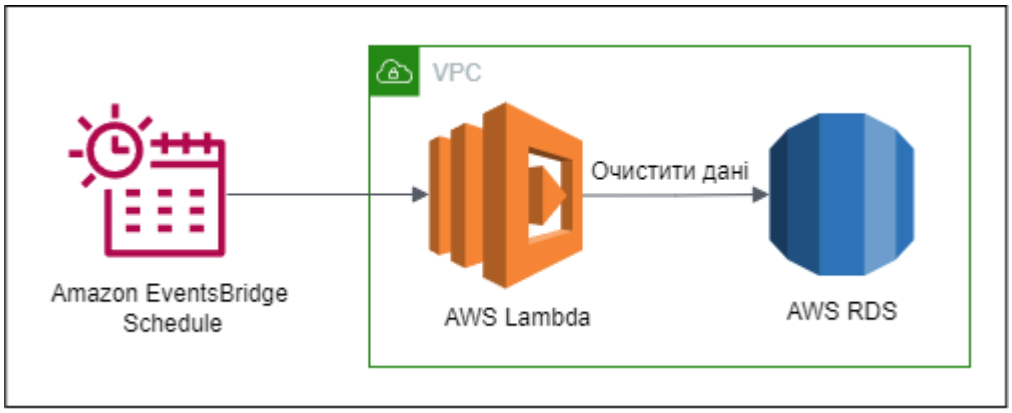


Рисунок 2.3 - Архітектура компоненту для очистки бази даних

Варто зазначити, що на рисунку зображено лише ті сервіси, що є компонентами застосунку, проте додатково будуть використані й інші: «AWS S3» - сховище статичних файлів, до прикладу зображень страв з меню, та конфігураційних файлів; «AWS Cloudwatch» - для моніторингу та як сховище логування; «AWS Secrets Manager» - для збереження секретів таких, як пароль до бази даних; «AWS IAM» - для керування дозволами компонентів системи та користувачів; «AWS ECR» - як репозиторій зображень контейнерів. При зростанні список додаткових сервісів, що використовуються може зростати.

2.2 Опис компонентів системи

2.2.1 База даних

Для рівня бази даних було обрано розширення сервісу «AWS RDS» («Relational Database Service») - «Aurora». Він надає чимало переваг розробникам, адже фактично провайдер відповідає не тільки за обладнання, а й за операційну систему та програмне забезпечення. Цей сервіс містить можливість резервного копіювання: як автоматичного, так і вручну. «RDS», як і «Aurora» надає способи підвищити відмовостійкість системи за допомогою налаштування екземплярів у різних зонах доступу. [15] Проте при використанні другого сервісу, розробка сильно спрощується, адже він автоматично слідкує за станом серверів та масштабує базу даних. Ця перевага «Amazon Aurora» є основною причиною, чому було зроблено такий вибір. [16] Розміщено базу даних буде у віртуальній приватній хмарі («AWS VPC) в окремій підмережі.

Що стосується двигуна бази даних, то було обрано «PostgreSQL». Сервіс «Amazon Aurora» наразі доступний лише для нього та «MySQL». [16]

Варто також згадати про зберігання паролю до бази даних. Для цього використаємо сервіс «AWS Secrets Manager». Цей сервіс дозволяє зберігати не тільки паролі, а і складніші структури, до прикладу повну інформацію про обліковий запис. На додачу «Secrets Manager» зберігає дані у зашифрованому вигляді, за допомогою іншого сервісу - «AWS KMS». Очевидно, що пароль потрібно час від часу змінювати. У цьому «Secrets Manager» також може допомогти, адже надає можливість автоматичної ротації. Для цього використовується «AWS Lambda». Для «PostgreSQL» уже є готовий шаблон, що виконує це, тому розробляти власний код для ротації не обов’язково. [17]

2.2.2 Бізнес-логіка

Як зазначалося у попередньому розділі, як базову технологію було обрано контейнери. Звісно їх можна розмістити і за допомогою «AWS EC2», який фактично надає віртуальні сервери, проте краще використати спеціалізовані сервіси. У даного хмарного провайдера доступні два варіанти: «AWS ECS» («Elastic Container Service») та «AWS EKS» («Elastic Kubernetes Service»). Другий варіант чудово підійде для великих мікросервісних застосунків. «AWS EKS» надає переваги, і «Kubernetes», і хмарних обчислень. Проте, для нашого застосунку на початковому етапі достатньо буде можливостей «AWS ECS».

Цей сервіс є оркестратором контейнерів, він допомагає керувати та розгортати їх. Також надає можливість масштабування в залежності від вказаних параметрів. Є два варіанти розгортання контейнерів:

а) «Fargate» - розгортання контейнерів без керування серверами;

б) «EC2» - розгортання контейнерів на віртуальних серверах («EC2»), що керуються власником програмного забезпечення. [18]

Для даного застосунку було вирішено використовувати «Fargate». Він чудово підходить для нескладних застосунків, як у нашому випадку, та пришвидшує розробку, адже нам не доведеться керувати інфраструктурою. При використанні розгортання за допомогою «Fargate», контейнери автоматично розміщуватимуться у тих зонах доступу, які визначені у підмережі кластеру.

У сервісі буде налаштовано автоматичне масштабування в залежності від навантаження. На додачу «AWS» надає нам можливість створювати так звані масштабування за розкладом. Їх також варто використати, як для збільшення кількості розгорнутих копій застосунку в пікові години, коли у ресторані велика кількість відвідувачів, так і зменшення вночі, коли застосунком навряд чи користуватимуться активно. [19]

Рівень бізнес-логіки буде написаний на «Java» та використовуватиметься фреймворком «Spring». Для доступу до сервісів хмарний провайдер надає «AWS SDK for Java» - набір бібліотек, що спрощують взаємодію з ним. [20] Для створення образів контейнера застосовуватимемо «Docker».

2.2.3 Користувацький інтерфейс

Для користувацького інтерфейсу, як видно з рисунку 2.1 використаємо «AWS ECS» із типом розгортання «Fargate». Обрано його було через ті ж причини, що і для бізнес-логіки. Обидва рівні на даному етапі розміщуватимуться на одному кластері, проте в майбутньому їхнє розділення також можливе за потреби.

2.2.4 Балансувальники навантаження

Балансувальник навантаження - девайс, що відповідає за ефективний розподіл вхідного мережевого трафіку поміж групою (фермою) серверів. Він також слідкує за їхнім станом - якщо один із пристроїв вийшов з ладу, запити перенаправлятимуться на інші. Балансувальник навантаження допомагає підвищити надійсність, доступність та масштабованість застосунку. [21]

Хмарний провайдер «AWS» надає нам можливість створювати та керувати цими девайсами. Для застосунків, що розгортаються за допомогою «ECS» доступно два види балансувальників навантаження:

а) балансувальник навантаження програми («Application Load Balancer») - приймає рішення щодо маршрутизації запитів на програмному рівні (HTTP/HTTPS/gRPC);

б) балансувальник навантаження мережі («Network Load Balancer») - примає рішення щодо розподілу запитів на транспортному рівні (TCP/UDP/TLS). [22]

Оскільки застосунок для ресторану не вимагає надзвичайно високої продуктивності, проте потребує гнучкості в керуванні, буде використано балансувальник навантаження програми. Він також буває внутрішнім («internal») та публічним («internet-facing»). У застосунку, що розробляється, для підвищення безпеки балансувальник навантаження рівня бізнес-логіки буде приватним - доступним лише з нашої віртуальної мережі, тоді як для рівня користувацького інтерфейсу від буде публічним.

2.2.5 Фаєрвол

Фаєрвол - це пристрій для безпеки мережі, що відстежує вхідний і вихідний трафік і вирішує, дозволяти чи блокувати його на основі визначеного переліку правил. [23] Хмарний провайдер «AWS» дозволяє налаштувати нам його для застосунків. Для веб-додатків можна використати сервіс «WAF». Потрібно створити список керування доступом («ACL») та пов’язати його з одним чи кількома ресурсами, що хочемо захистити. У нашому застосунку це обидва балансувальники навантаження, кожен зі своїм списком керування доступом. [24] Можна захистити також службу доставки контенту, адже «AWS» надає нам і таку можливість.

Список керування доступом містить набір правил, що визначають як перевіряти запити, та як діяти, якщо ті відповідають критеріям. Існують вже готові групи правил, що керуються хмарним «AWS» та «AWS Marketplace». Вони пропонують широкий захист від відомих загроз та вразливостей. Більшість із них можна використовувати без додаткової плати. [25] Також можна створити власні правила.

2.2.6 Служба доставки контенту

Коли користувач переходить за посиланням застосунку, запит проходить через велику сукупність взаємопов’язаних мереж. Для того, щоб пришвидшити взаємодію з додатком можна використати службу доставки контенту.

«CloudFront» сервіс «AWS», що виконує цю функцію. Він направляє кожен запит через магістральну мережу хмарного провайдера найближче до користувачів, найчастіше до так званого граничного сервера. Таким чином «CloudFront» підвищує швидкодію застосунку. Він також забезпечує кращу надійність і доступність, оскільки зберігає копії файлів у кількох місцях по всьому світу. [26] На рисунку 2.4 схематично зображено роботу «CloudFront»

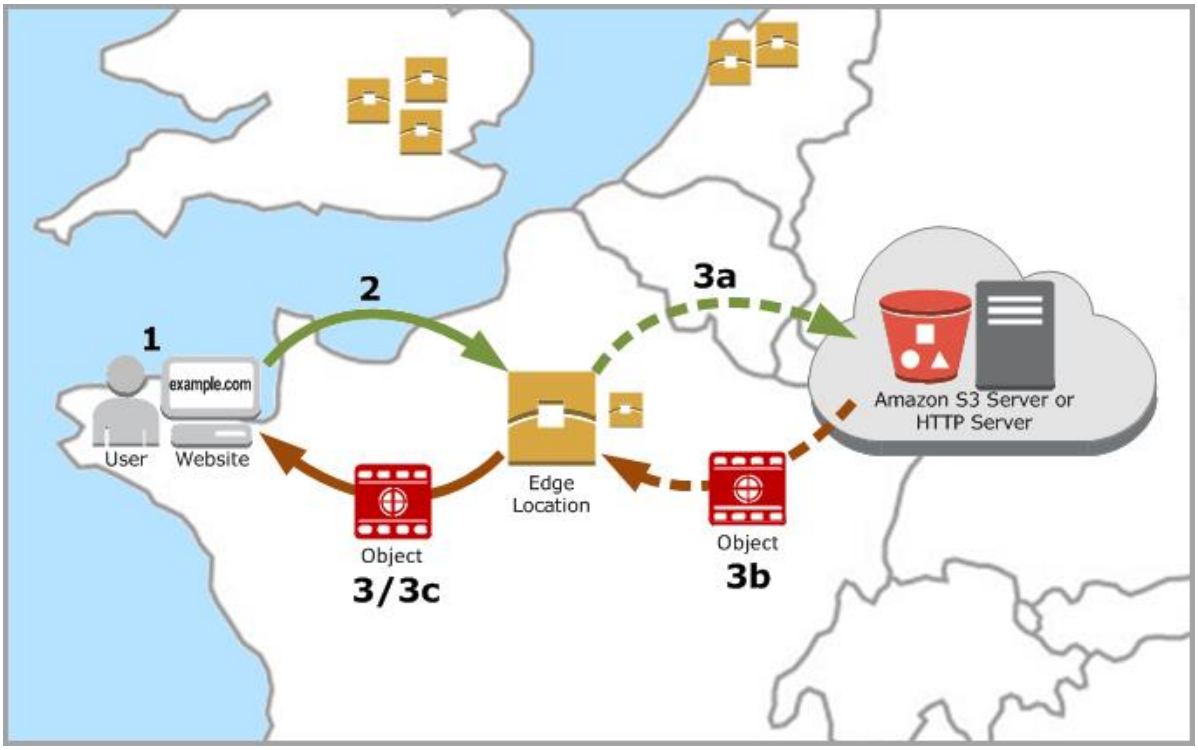


Рисунок 2.4 - Схема роботи сервісу «CloudFront» [27]

2.3 Опис функціонування системи

Важливо розуміти, як усі компоненти описані вище будуть взаємодіяти між собою та як користувачі використовуватимуть систему.

2.3.1 Взаємодія користувачів із застосунком

Для кожного столика у ресторані буде згенеровано унікальний QR-код, що переправлятиме на застосунок та автоматично обиратиме відповідне місце у закладі для подальшої роботи із застосунком. Також додаток можна знайти у пошуковій системі браузера. На головній сторінці користувачу відображатиметься інформація про ресторан. Сторінка меню міститиме страви та напої і буде поділена на категорії. Користувач зможе додавати страви до свого замовлення, записувати примітки і потім відправляти його на приготування. Якщо у гостя є активне замовлення, він може оплатити його просто із застосунку за допомогою сервісів «Google Pay»/«Apple Pay». Також додаток реалізовуватиме резервування столика у закладі. При навігації на відповідну сторінку користувач обиратиме дату та час, після чого йому відображатиметься схема залу з вільними місцями. Він зможе обрати столик та створити бронювання вказавши ім’я, номер телефону, кількість осіб.

2.3.2 Взаємодія компонентів системи

Доменне ім’я буде зареєстроване через DNS-сервіс «Amazon Route 53», який також перенаправлятиме користувачів на балансувальник навантаження користувацького інтерфейсу. Статичні сторінки, як от інформація про ресторан, будуть одразу повертатися користувачу, а зображення страв зберігатимуться в сервісі «S3». Для динамічного контенту, рівень користувацького інтерфейсу звертатиметься до балансувальника навантаження бізнес-логіки за допомогою REST API. Той в свою чергу звертатиметься до бази даних та діставатиме потрібну інформацію.

2.4 Опис функціонування безсерверних компонентів

2.4.1 Збір статистичних даних про ресторан

Ресторанний бізнес, як і будь-який інший, повинен постійно розвиватись, щоб залишатись конкурентоспроможним. Потрібно аналізувати потреби гостей закладу і приймати на основі цього певні рішення. Саме ці чинники зумовили структурну розробку додаткового компоненту, що надавав би власнику ресторану статистику: страви, що замовляють найчастіше, години та дні пікової активності, середня сума замовлення тощо.

Цей компонент системи потребує автентифікації - підтвердження, що користувач є в системі та авторизації - перевірка, що користувачу дозволяється виконувати ті чи інші дії. Для цього можна використати сервіс «Amazon Cognito», у якому є така функціональність. Додатково він надає можливість автентифікувати користувача через сторонні сервіси, до прикладу «Google». [28] Обробляти запити буде «AWS Lambda», а «Amazon API Gateway» буде її запускати. Ці два сервіси легко інтегруються і хмарний провайдер надає детальну документацію щодо цього. [29] Код буде написаний мовою програмування «Java». «AWS Lambda» підтримує як це оточення, так й інші: «Node.js», «Python», «.NET» тощо. Лямбда-функція буде діставати потрібні дані з бази даних та виконувати їхню обробку.

2.4.2 Автоматична очистка бази даних

У нашій системі зберігатиметься багато даних - щодня ресторан обслуговує сотні чи навіть тисячі гостей, створюються нові бронювання столиків тощо. Важливо, щоб вони були доступними впродовж певного терміну для отримання статистики, проте з часом вони втрачають свою актуальність і лише сповільнюють роботу рівня бази даних. Саме тому було спроектовано компонент, що буде автоматично очищувати їх.

Основною частиною цього компоненту буде «AWS Lambda», написана на мові програмування «Java». За допомогою змінних середовища функції можна буде налаштувати термін, протягом якого дані зберігатимуться. І хоча можна було б запускати лямда-функцію, як і в попередньому компоненті, за допомогою сервісу «Amazon API Gateway», проте це вимагає від адміністратора ресторану додаткових дій - надсилання запиту. Ми ж хочемо повністю автоматизувати цей процес. Для цього використаємо сервіс «Amazon EventBridge Scheduler». Раніше для подібних цілей застосовували «Amazon CloudWatch Events», проте хмарний провайдер пропонує використовувати новий сервіс. [30] За допомогою нього можна створити розклад виконання певної дії, у нашому випадку запуску лямбда-функції. Після цього вона видалятиме застарілу інформацію з бази даних.

2.5 Додаткові інструменти для неперервного розгортання та доставки коду (CI/CD)

З поширенням хмарних технологій все більшої популярності набула методологія «DevOps» - філософія, набір практик, що інтегрують процеси між розробниками («Dev») та командою, що відповідає за інфраструктуру і розгортання («Ops»). Вона та її інструменти покращують швидкість розробки й доставки застосунку, його якість і надійність. «DevOps» методологія наголошує на автоматизації процесів. [31]

Однією з її практик є «CI/CD» - «Continuous Integration/Continuous Delivery (Deployment)». Вона автоматизує етапи збірки, тестування та розгортання нового коду. [31] Для імплементації неперервної інтеграції та доставки коду використовуватимемо інструмент «Gitlab CI».

Іще однією практикою «DevOps», яку ми застосуємо буде «IaC» («Infrastructure as Code») - керування інфраструктурою за допомогою певних файлів, що її описують. Без цієї методики керування серверами та іншими

ресурсами є повільним та хаотичним процесом: системні адміністратори вручну створюють та оновлюють все, що потрібно. У команди немає цілісного уявлення про інфраструктуру, збій системи виникає значно частіше. «IaC» ж використовує файли, що легко розуміються розробниками. Вони зберігаються у репозиторії, тому вся команда має доступ до них. [32]

Одним із інструментів «IaC» є «Terraform». Він використовує декларативний підхід - розробники описують бажаний стан системи, а «Terraform» порівнює її з існуючою та робить відповідні зміни. [33] Щоб зберігати файл стану, цьому інструменту потрібен додатковий компонент - «backend», який можна налаштовувати по-різному. Для нашої системи ми також використаємо хмарний провайдер «AWS». Тип «backend» «s3» зберігає файли стану в сервісі «AWS S3». Додатково розгортається нереляційна база даних цього провайдера - «DynamoDB». ЇЇ мета - блокувати одночасне редагування інфраструктури. Таким чином, тільки один розробник зможе вносити зміни в систему. [34]

На рисунку 2.5 зображено інструменти, що використовуються для неперервного розгортання та доставки коду, створення інфраструктури та їхню взаємодію.

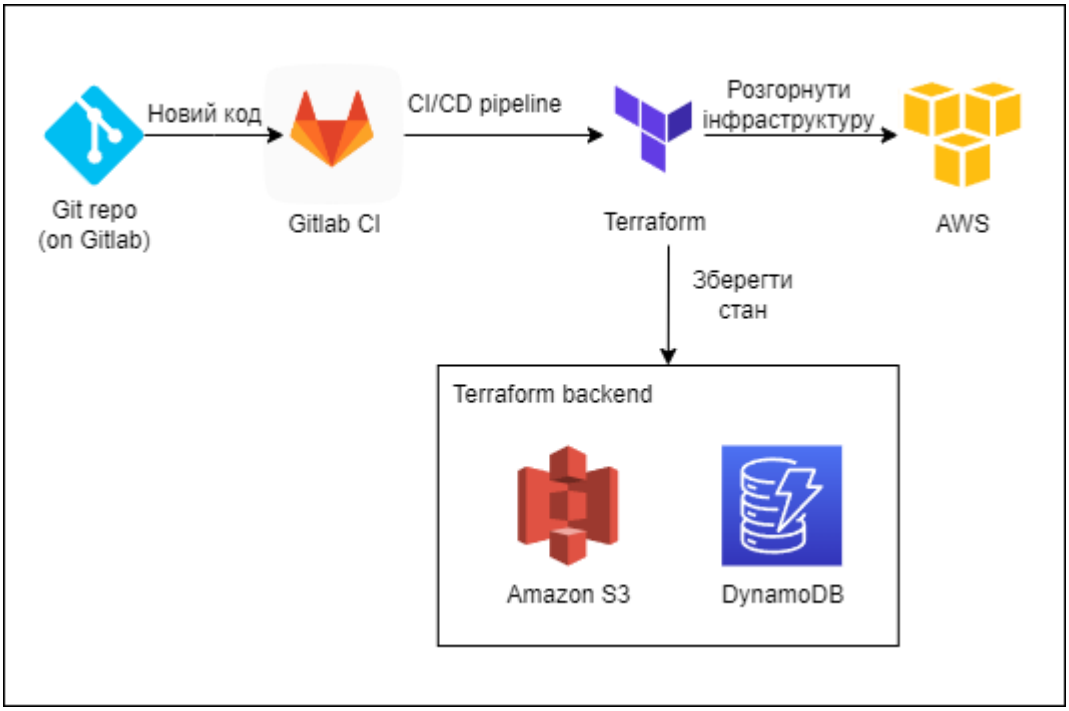


Рисунок 2.5 Схема неперервного розгортання та доставки коду

2.6 Висновок

У даному розділі було розроблено структуру майбутнього застосування. На основі аналізу різних можливостей, що надає хмарний провайдер «AWS» спроектовано багаторівневу систему. Для неї характерна висока доступність та відмовостійкість - завдяки розміщенню у кількох зонах, гнучка масштабованість - її надає розгортання через «AWS ECS», безпека - оскільки у системі присутній фаєрвол та використано приватну мережу, ефективність - через застосування балансувальників навантаження, служби доставки контенту. Детально описано рівні та їхню взаємодію і функціонування застосунку. Додатково було розроблено два компоненти, що використовують безсерверні обчислення - це допоможе зменшити навантаження на основний додаток. Також описано процес й інструменти, що використовуватимуться для неперервної інтеграції та доставки застосунку, інфраструктури як коду - «Gitlab CI» та «Terraform» відповідно.

3. ДЕТАЛЬНА РОЗРОБКА ТА РОЗГОРТАННЯ КОМПОНЕНТІВ

У даному розділі розробимо та розгорнемо компоненти нашого застосування: базу даних та бізнес-логіку з балансувальником навантаження. Таким чином у результаті отримаємо готове до використання REST API. При цьому продемонструємо на практиці розгортання застосунку у хмарному провайдері «AWS» з технологією контейнерів, використання практик «CI/CD» з таким інструментом як «Gitlab CI» та технології «IaC» з «Terraform».

3.1 Технічне завдання

І хоча у попередніх розділах досить детально описувався функціонал, у цьому підрозділі перерахуємо основні завдання, що має виконувати наше API. Проаналізувавши вимоги до застосунку виділимо наступні можливості:

а) перегляд категорій меню;

б) перегляд страв та напоїв за відповідною категорією, включаючи усі підкатегорії;

в) створення замовлення;

г) перегляд активного замовлення за відповідним столиком;

д) додавання нових позицій у замовлення, якщо воно активне;

е) позначення замовлення як завершеного (оплаченого);

ж) додавання відгуку про замовлення, якщо воно завершене;

з) перегляд столиків, що ще не зарезервовані для конкретної дати та часу;

и) додавання бронювання столику.

Що стосується оплати замовлення, цей функціонал реалізовуватиметься на рівні користувацького інтерфейсу, за допомогою використання готового API

«Google Pay» та «Apple Pay». У випадку успішної транзакції, цей компонент надсилатиме запит на балансувальник навантаження бізнес-логіки про зміну статусу замовлення на завершене.

3.2 Розробка компонентів

3.2.1 Проектування реляційної схеми бази даних

Проаналізувавши подане вище технічне завдання та доменну область було спроектовано реляційну модель зображену на рисунку 3.1.

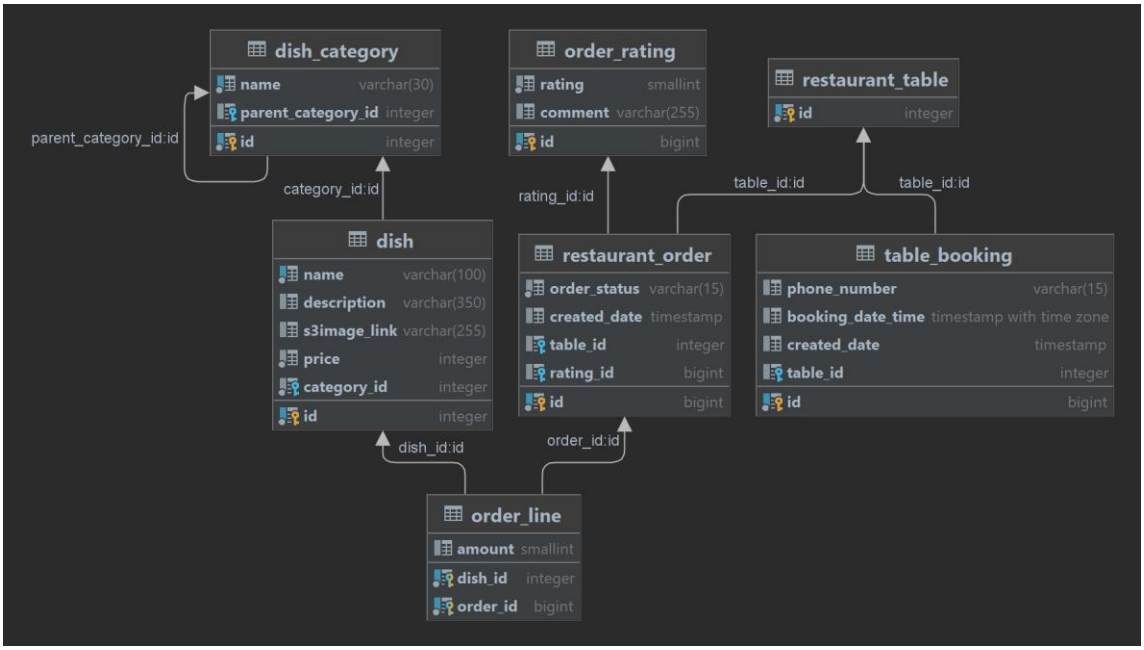


Рисунок 3.1 - Реляційна модель бази даних

У системі будуть наступні сутності:

а) Категорія страви - до прикладу закуски, напої тощо. Ця сутність має рекурсивний зв’язок сама із собою, таким чином утворюючи ієрархію.

б) Страва - міститиме опис, назву, ціну, категорію, а також для деяких страв посилання на зображення, що буде збережене у сервісі «S3» в окремому бакеті. Вони будуть доступні для публічного перегляду, що дозволить легко відображати їх на користувацькому інтерфейсі.

в) Столик - сутність, у якої на даному етапі є лише одна властивість - ключ.

г) Бронювання столику - містить номер телефону, для легкої ідентифікації того, хто резервував, дату та столик. До неї також додано атрибут - дата створення.

д) Замовлення - сутність має зв’язок зі столиком, дату створення та статус, що наразі може набувати двох значень - активне та завершене. Також вона пов’язано зв’язком «багато до багатьох» із стравами за допомогою допоміжної таблиці.

е) Відгук - кожне замовлення може бути зв’язане із цією сутністю, якщо гість виявив бажання залишити оцінку. У ньому зберігається сама оцінка (від 1 до 5) та коментар до неї.

3.2.2 Розробка SQL-скриптів міграцій

Для налаштування схеми бази даних використаємо інструмент для міграцій «Flyway». Такий підхід допомагає перевести структуру реляцій із поточного стану в новий. При цьому всі зміни зберігаються в репозиторії, разом із іншим кодом, що допоможе легко слідкувати за оновленнями схеми. Серед переваг також автоматизація процесу створення та видалення таблиць, даних, зв’язків тощо. [35] Завдяки інтеграції фреймворку «Spring» із «Flyway», усі міграції застосовуватимуться при запуску застосунку. Протягом розробки додатку було додано чотири міграції, зображені на рисунку 3.2. У додатку Б розміщено їхній вміст. Перша із них «V1.0__Initial.sql» створює всі таблиці та зв’язки між ними, тоді як три інші заповнюють їх початковими даними: категоріями меню, стравами та столиками закладу.

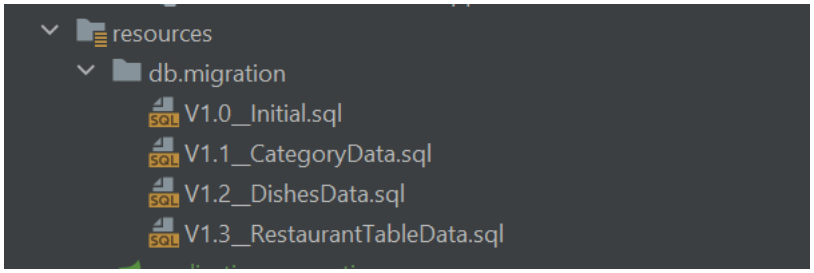


Рисунок 3.2 - Скрипти міграції схеми бази даних

3.2.3 Розробка коду рівня бізнес-логіки

Застосунок був розроблений на мові програмування «Java» із використанням фреймворку «Spring» та його модулів. Структурно застосунок поділено на рівні:

а) Моделі - містить сутності бази даних, класи DTO (Data Transfer Object) патерну та користувацькі виключення (exception).

б) Репозиторії - класи для доступу до бази даних.

в) Сервіси - класи, що містять власне бізнес-логіку.

г) Контролери - класи, що обробляють запити REST API.

На рисунку 3.3 зображено ієрархію пакетів Java відповідно до цього.

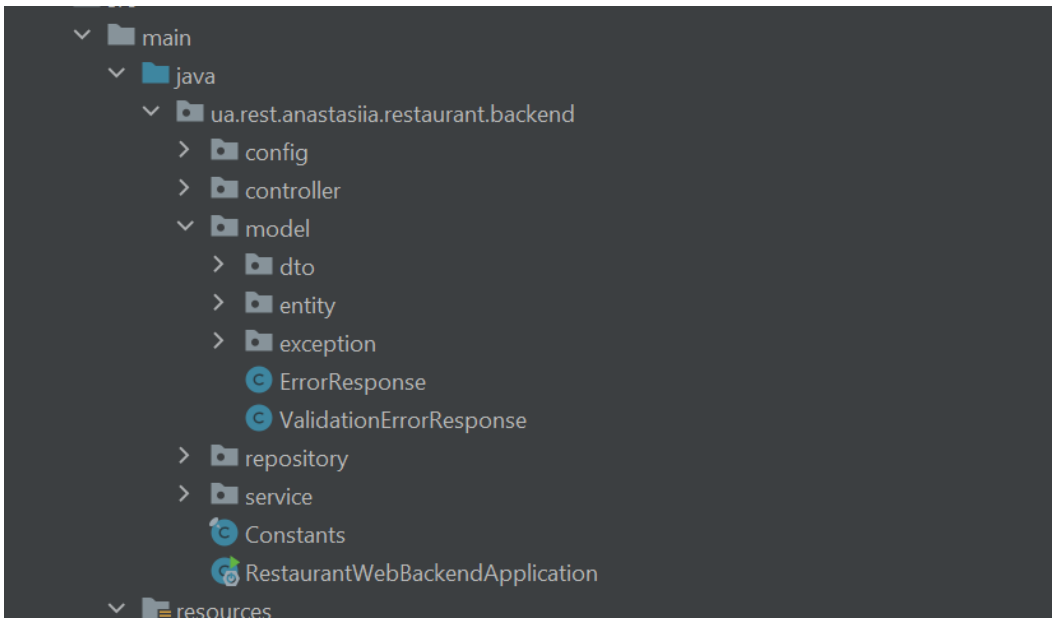


Рисунок 3.3 - Структура застосунку бізнес-логіки

Що стосується залежностей, то для керування ними використовувався «Maven». Для репозиторіїв та класів сутностей було застосовано фреймворк «Spring Data JPA», що реалізовує стандарт «Jakarta Persistence» для «ORM» («Object-Relational Mapping») - підхід, що пов’язує рівень бази даних з об’єктами в застосунку. [36] Бібліотека «Lombok» - для генерації шаблонного коду в моделях. Для серіалізації та десеріалізації тіла запиту застосовували залежність «Jackson». Додатково було додано перевірку на коректність класів DTO, за допомогою «Jakarta Bean Validation». Готове REST API автоматично документується за допомогою «OpenAPI» та візуалізується через «SwaggerUI», що зображено на рисунку 3.4.

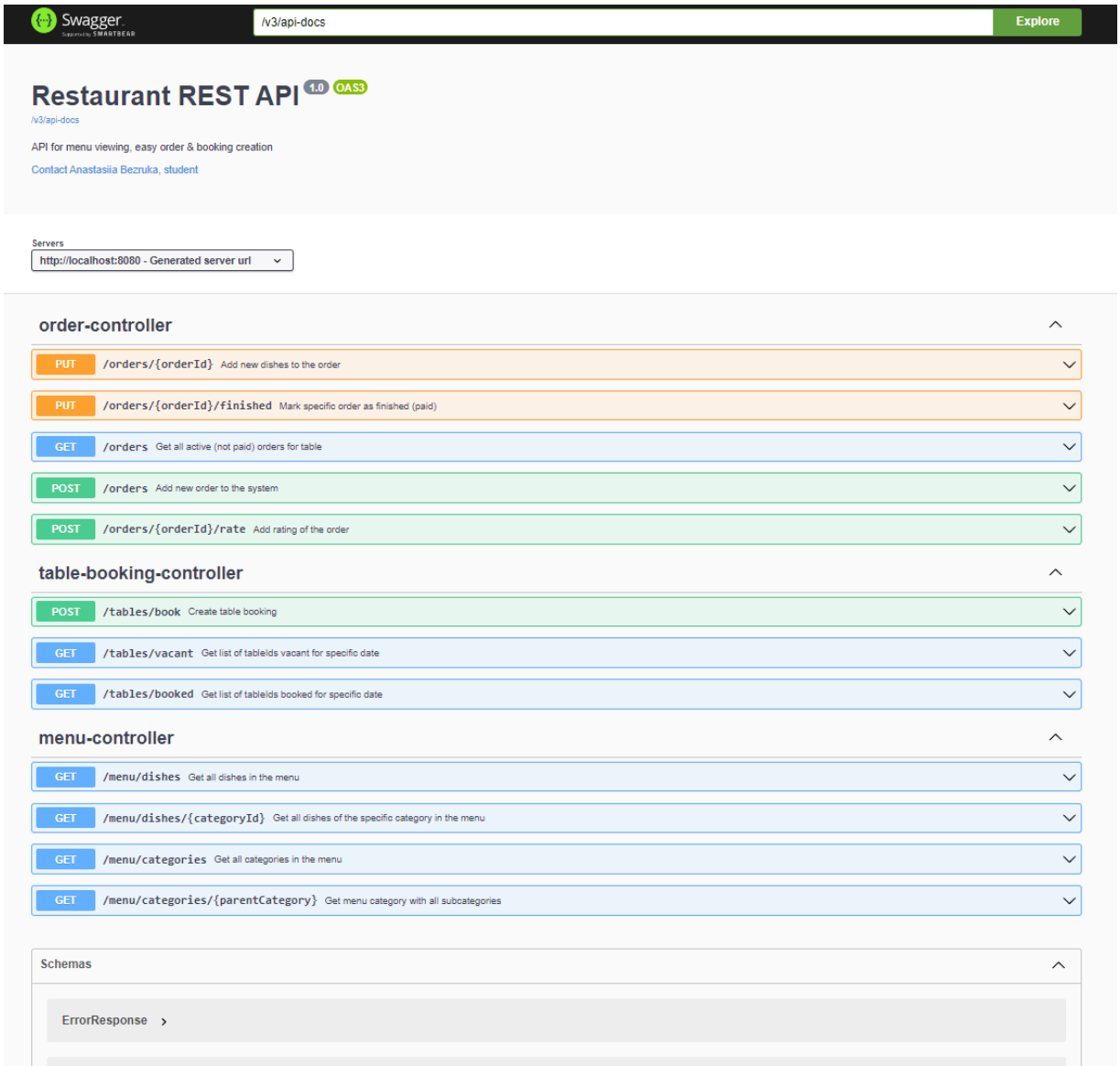


Рисунок 3.4 - SwaggerUI застосунку

На додачу, завдяки залежності «Spring Boot Actuator» у застосунок автоматично додано запит для перевірки працездатності - «health», який є важливим для розгортання в хмарному середовищі. З його допомогою, контейнери, що з певної причини працюють не коректно, будуть автоматично зупинені, а натомість додаватимуться нові.

Також у застосунку визначено два профайли: один для розгортання в локальному середовищі, інший - для хмари. Основна відмінність між ними у конфігурації з'єднання бази даних, оскільки та розташована у приватній підмережі і не доступна з Інтернету. На додачу, у хмарному середовищі використовуватиметься бібліотека «AWS Secrets Manager JDBC», що автоматично діставатиме пароль, збережений як секрет. [37] У додатку В розміщено конфігураційні файли як спільні, так і для кожного з профайлів.

Модульними тестами було покрито рівень сервісів, оскільки у ньому фактично розміщена основна логіка застосунку. Інший функціонал перевірявся вручну, як у локальному середовищі, так і у хмарі.

3.3 Автоматизоване розгортання системи

3.3.1 Розгортання «terraform backend»

Як зазначалося у попередньому розділі, для збереження файлів стану інструменту «Terraform» також використаємо сервіси «AWS». На відміну від інших ресурсів, бекенд створимо вручну. Це пов'язано з тим, що ці ресурси не є частиною застосунку і зазвичай є незмінними. Отже, за допомогою веб-консолі «AWS» ми створили бекенд: «S3» бакет - «restaurant-tfstate-euc1» та таблицю «DynamoDB» - «restaurant-tf-locks-euc1». Для усіх «Terraform» скриптів бекенд вказується так, як зображено на рисунку 3.5.

```
terraform {
  backend "s3" {
    bucket         = "restaurant-tfstate-euc1"
    key            = "euc1/shared-resources.tfstate"
    encrypt        = true
    region         = "eu-central-1"
    dynamodb_table = "restaurant-tf-locks-euc1"
  }
}
```

Рисунок 3.5 - Приклад визначення бекенду

3.3.2 Розгортання спільних ресурсів

3.3.2.1 Конвеєр неперервної інтеграції та доставки

У процесі розробки застосунку було створено окремий репозиторій для спільних ресурсів. Саме у ньому визначені файли конфігурації для «VPC», «S3» бакета з зображеннями страв, кластера «ECS», бази даних. У цьому репозиторії також міститься користувацький модуль для балансувальника навантаження, проте розгортання відбувається разом із рівнем бізнес логіки. На рисунку 3.6 зображено його структуру.

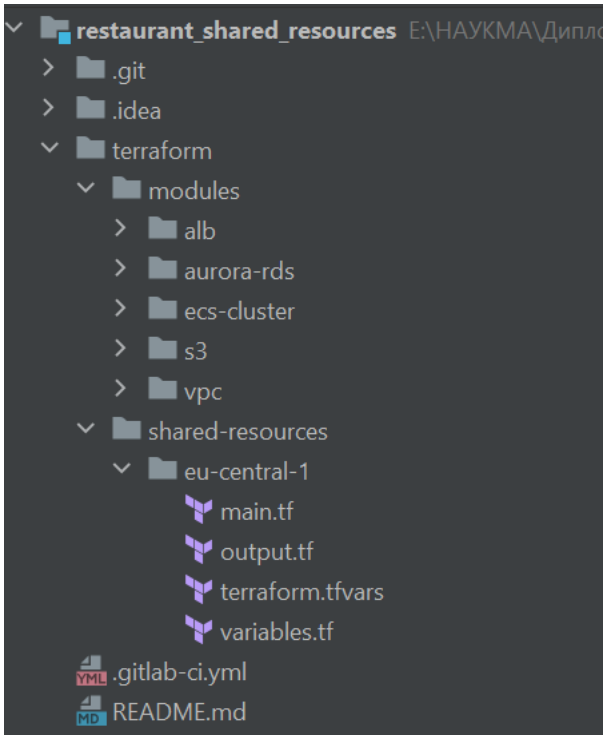


Рисунок 3.6 - Структура репозиторію із спільними ресурсами

Що стосується «CI/CD», то конвеєр складається із двох етапів та трьох завдань. Вони виконують відповідні команди «Terraform».

а) етап «shared-resources-build»:

- завдання «terraform_validate» - перевіряє файли на відсутність помилок, посилаючись при цьому лише на конфігурацію та не звертаючись до віддалених служб. [38]
- завдання «terraform_plan» - створення плану виконання змін інфраструктури та збереження їх у файл. [39]

б) етап «shared-resources-deploy»:

- завдання «terraform_apply» - створює інфраструктуру, використовуючи збережений план виконання із попереднього завдання. [40] Цей крок, на відміну від інших, не виконуватиметься поки розробник не запустить його вручну.

На рисунку 3.7 візуалізовано конвеєр, а файли його конфігурації подано у додатку Г. Визначення ресурсів, за допомогою «Terraform» подані разом із програмним кодом роботи.

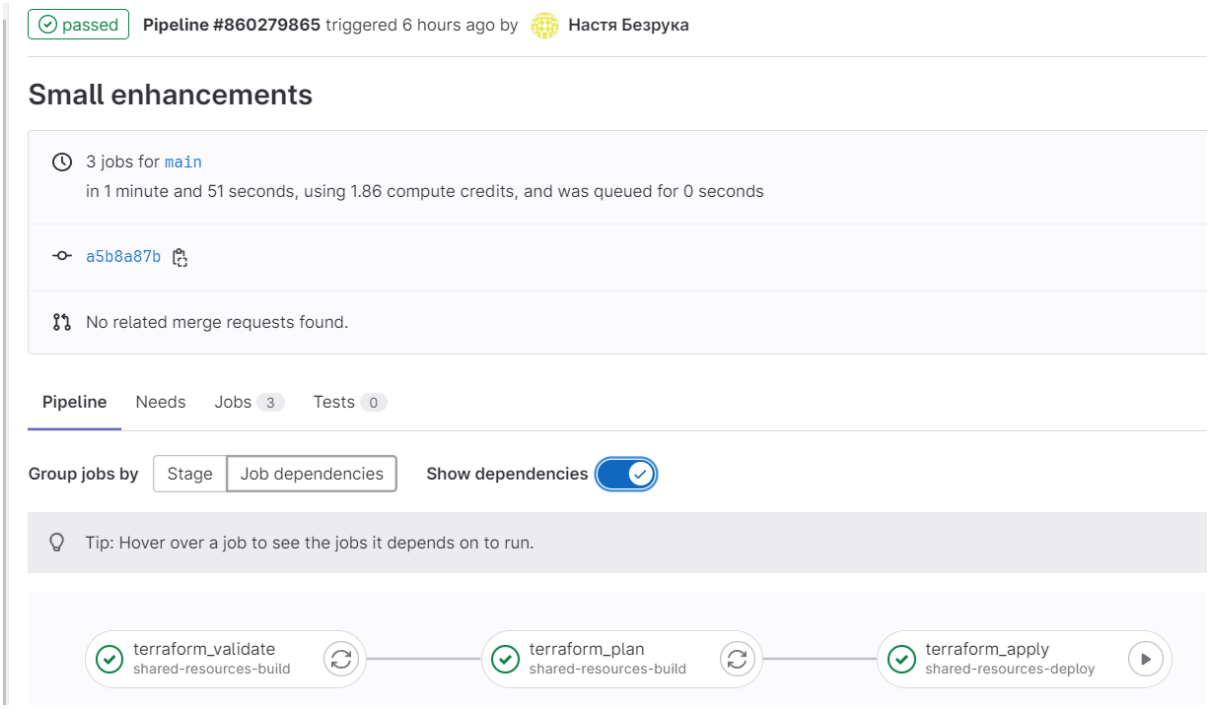


Рисунок 3.7 - Конвеєр неперервної інтеграції та доставки для «shared-resources»

3.3.2.2 Розгортання «VPC»

У результаті роботи конвеєра неперервної інтеграції та доставки було розгорнуто «VPC», що містить по дві приватні підмережі для рівня бази даних, та бізнес-логіки і дві публічні. Щоб останні могли взаємодіяти з ресурсами поза межами приватної хмари, було створено Інтернет-шлюз та загальнодоступні таблиці маршрутів для них. [41] Для підмереж рівня бізнес логіки було створено шлюз NAT (Network Address Translation). Він дозволяє цьому компоненту підключатися до Інтернету, але при цьому зовнішні служби не можуть ініціювати з’єднання з ним. [42] Це дозволить рівню бізнес-логіки взаємодіяти з сторонніми програмними інтерфейсами за потреби. Для рівня бази даних обов’язково потрібно створити підмережеву групу. На рисунку 3.8 зображено схему вже розгорнутої «VPC», що відображається у веб-консолі «AWS».

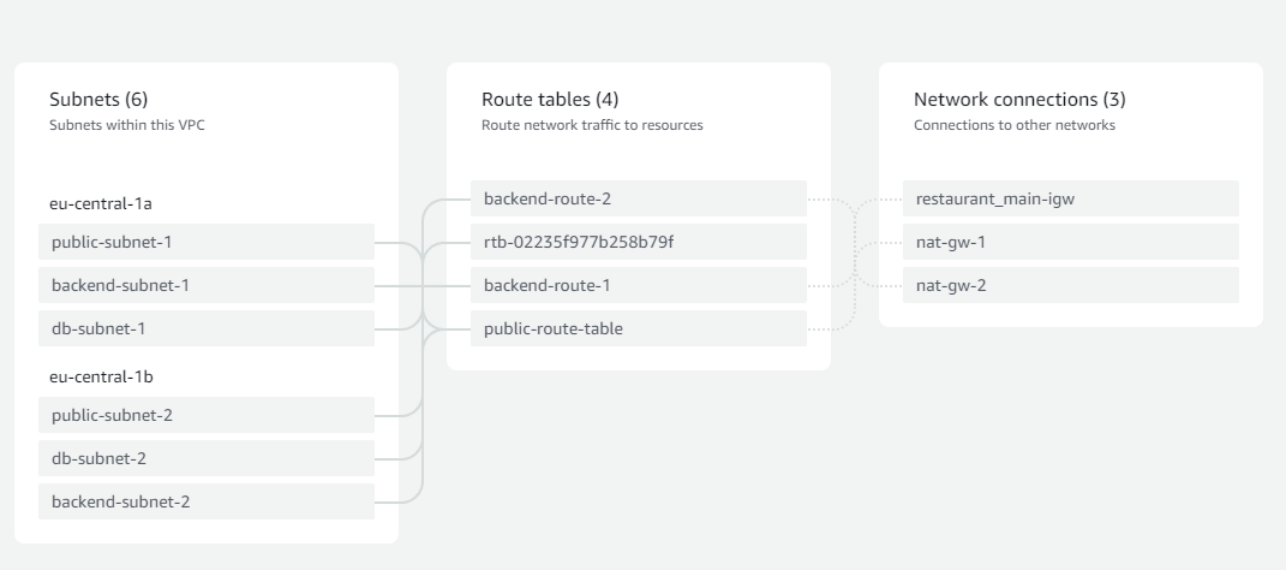


Рисунок 3.8 - Структура ресурсів «VPC»

3.3.2.3 Розгортання «S3» для зображень страв

У репозиторії «shared-resources» також було розгорнуто «S3» бакет, де міститимуться фото страв чи напоїв. До нього було додано публічний список

доступу для читання, щоб легко відображати зображення на користувацькому інтерфейсі. На додачу, дані на стороні сервера шифруватимуться керованими ключами «Amazon S3» («SSE-S3»). На рисунку 3.9 зображено вже розгорнутий ресурс, з доданими вручну зображеннями деяких страв.

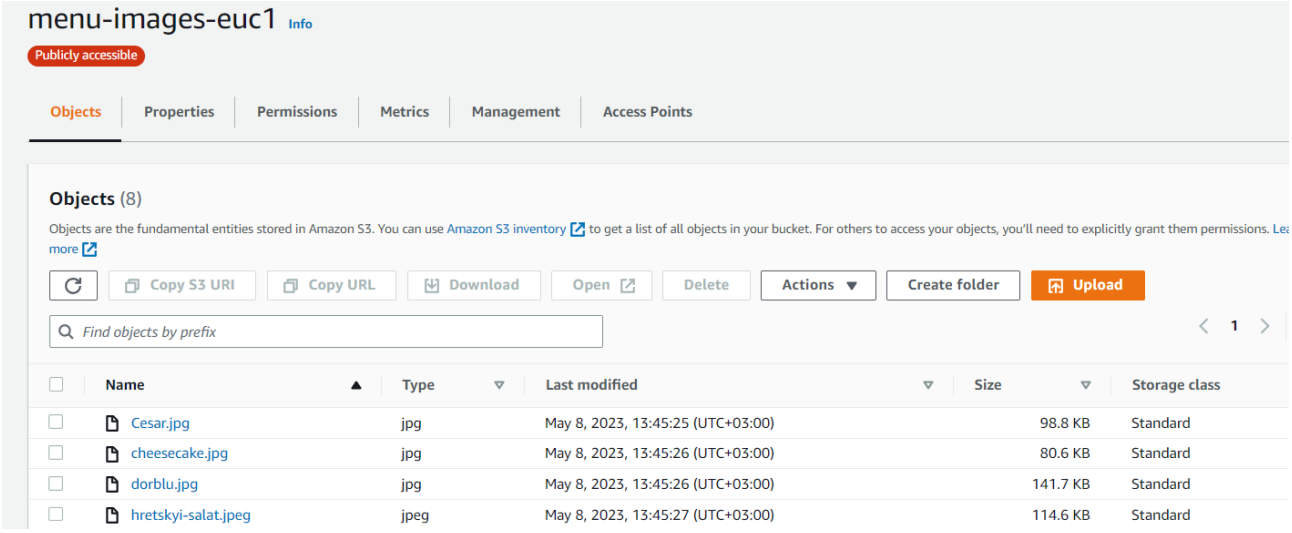


Рисунок 3.9 - «S3» бакет із зображеннями страв

3.3.2.4 Розгортання кластеру «ECS»

Оскільки ми використовуємо тип розгортання «Fargate», єдиний ресурс, який потрібно визначити, - сам кластер. Було також увімкнено опцію «containerInsights», для того, щоб «AWS» моніторив стан системи. На рисунку 3.10 зображено метрики нашого кластера, зібрані автоматично хмарним провайдером. А на рисунку 3.11 - визначення цього ресурсу, за допомогою інструменту «Terraform».

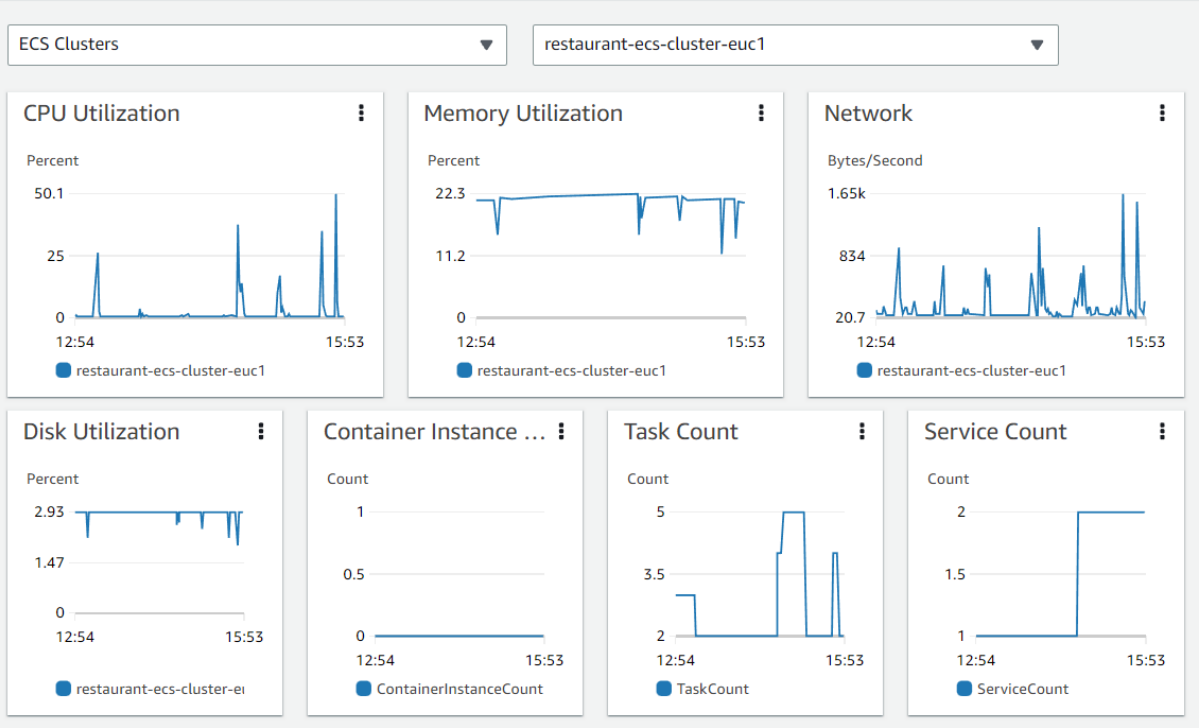


Рисунок 3.10 - Метрики кластера «ECS»

```
resource "aws_ecs_cluster" "ecs_cluster" {
  #To collect metrics
  setting {
    name  = "containerInsights"
    value = "enabled"
  }
  name = local.ecs_cluster_name
  lifecycle {
    create_before_destroy = true
  }
}
```

Рисунок 3.11 - Визначення кластеру «ECS» у «Terraform»

3.3.2.5 Розгортання рівня бази даних

Оскільки використовується сервіс «Amazon Aurora», розгортання бази даних також доволі просте. Було визначено кластер із режимом двигуна - «provisioned» і масштабуванням та один екземпляр класу «db.serverless». За такої конфігурації розгортатиметься друга версію «Amazon Aurora». (Для

першої режим двигуна повинен бути «serverless»). [43] Щоб обмежити доступ до бази даних, було додано групу безпеки, яка дозволяє вхідний трафік лише з підмережі бізнес-логіки на порт кластеру. На рисунку 3.12 зображено вже розгорнуту базу даних у веб-консолі «AWS».

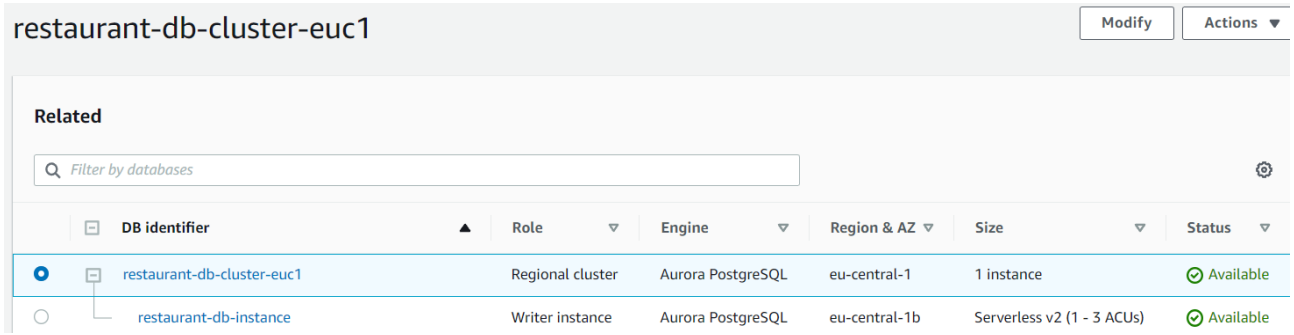


Рисунок 3.12 - Розгорнутий рівень бази даних

3.3.3 Розгортання рівня бізнес-логіки та балансувальника навантаження

3.3.3.1 Конвеєр неперервної інтеграції та доставки

Файли «Terrafrom» для цих двох компонентів розміщувались у окремому репозиторії - «restaurant_web_backend».

Було вирішено використовувати метод розгортання «Blue/Green». При використанні такої стратегії потрібно дублювати ресурси. Одне середовище («синє») - поточна версія програми, інше («зелене) - нова. Використання цього методу зменшує ризики розгортання оновлень та спрощує процес відкату. [44]

Застосування стратегії «Blue/Green» зумовило поділ ресурсів на дві директорії: «обгортка» та «застосунок». У першій міститься конфігурація балансувальника навантаження, групи логування та ролі «IAM» для сервіса. Це ресурси, які присутні в одному екземплярі і які будуть змінюватись рідко, вони розгортаються методом «Recreate». Директорія «застосунок» містить компоненти, що змінюватимуться частіше та дублюватимуться: сервіс «ECS», визначення завдань-контейнерів та їхня «IAM» роль, конфігурація

масштабування та цільова група для балансування навантаження. На рисунку 3.13 зображено структуру файлів «Terraform».

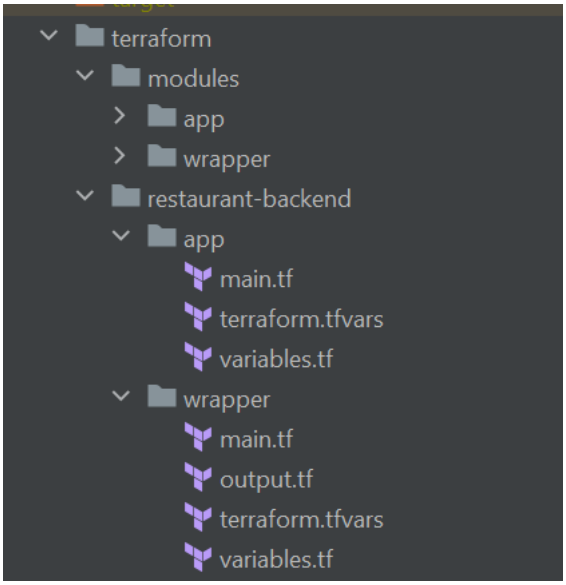


Рисунок 3.13 - Структура файлів «Terraform» репозиторію «restaurant_web_backend»

Конвеєр неперервної інтеграції та доставки містить наступні етапи і завдання:

а) етап «maven-verify-build»:

- завдання «maven_verify» - виконує відповідну команду життєвого циклу збірки застосунку інструменту «Maven»; компілює та тестує застосунок. [46]
- завдання «maven_build» - виконує команду життєвого циклу «package» з опцією пропустити тести та їхню компіляцію, оскільки це частина першого завдання; у результаті отримуємо виконуваний файл .jar. [46]

б) етап «ecr-build-deploy»:

- завдання «docker_build_image» - будує та зберігає у артефакт зображення контейнера. На рисунку 3.14 подано вміст Dockerfile.

```
FROM amazoncorretto:17-alpine

ENV APP_JAR restaurant-backend-1.0-SNAPSHOT.jar
ENV PORT 8080

COPY ./target/$APP_JAR /

EXPOSE $PORT
CMD exec java -Dserver.port=$PORT -jar $APP_JAR
```

Рисунок 3.14 - Dockerfile компоненту бізнес-логіки

- завдання «docker_push_ecr» - публікує отримане зображення у приватний регістр сервісу «ECR»; тег версії встановлюється відповідно до ідентифікатора конвеєра.

в) етап «build-wrapper» - аналогічний етапу «shared-resources-build» іншого репозиторію, завдання такі ж;

г) етап «deploy-wrapper» - аналогічний етапу «shared-resources-deploy», завдання такі ж;

д) етап «build-green»:

- завдання «terraform_app_validate»;
- завдання «terraform_app_plan» - виконує відповідну команду «terraform», перед тим створюючи чи переходячи у нову робочу область, що дозволяє дублювати ресурси. Їх є дві - «v1» та «v2». Інформація про те, яка версія активна зберігається у сервісі «AWS Systems Manager» підрозділі «Parameter Store».

е) етап «deploy-green»:

- завдання «terraform_apply_green» - створює інфраструктуру, використовуючи збережений план виконання із попереднього завдання. Далі за допомогою «AWS CLI» змінює цільову групу правила з пріоритетом 1 балансувальника навантаження. Це правило містить наступну умову - якщо у запиті є параметр з

ключем «green» і він має значення «true» - направляти запити на «зелену» версію.

ж) етап «verify-green»:

- завдання «verify_green_version» - наразі просто робить запит на API нової версії, проте при подальшому розвитку застосунку можна додати більш повне тестування у цьому кроці.

и) етап «switch-green»:

- завдання «switch_green» - перемикає версії застосунку за допомогою команд «AWS CLI». По-перше, змінює правило балансування навантаження з пріоритетом 2, умова якого виконується завжди, на нову цільову групу. Після цього змінює кількість розгорнутих контейнерів попереднього «ECS» сервісу на нуль. Останнім етапом є збереження даних у підрозділі «AWS Systems Manager» - «Parameter Store». Усього потрібно три параметри, детальна інформація про них розміщена у додатку Д.

к) етап «rollback»

- завдання «rollback_green» - надає можливість відкату до попередньої версії. Спочатку воно запускає стільки контейнерів «ECS» сервісу старої версії, скільки зараз робочих у поточної. Потім змінює основне правило балансування навантаження з пріорітетом 2, перенаправляючи його на попередню версію. Після цього змінює кількість розгорнутих контейнерів «зеленого» «ECS» сервісу на нуль. Останнім етапом є збереження даних у підрозділі «AWS Systems Manager» - «Parameter Store».

Для скриптів тих етапів, що пов’язані з розгортанням «зеленої» версії застосунку нам потрібно було зображення контейнера, що містило б у собі «AWS CLI» та «jq» - для фільтрації результатів виконання його команд, а також «Terraform». На жаль, доступні рішення не вийшло використати: у одного з них

була старіша версія інструментів, інший - не працював коректно. Ці обставини зумовили створення та публікацію в регістрі «ECR» власного зображення, що базувалося на основі знайдених. Dockerfile до нього розміщено у додатку Е.

На рисунку 3.15 візуалізовано конвеєр, а файли його конфігурації подано у додатку Ж. Визначення ресурсів, за допомогою «Terraform» подані разом із програмним кодом роботи.

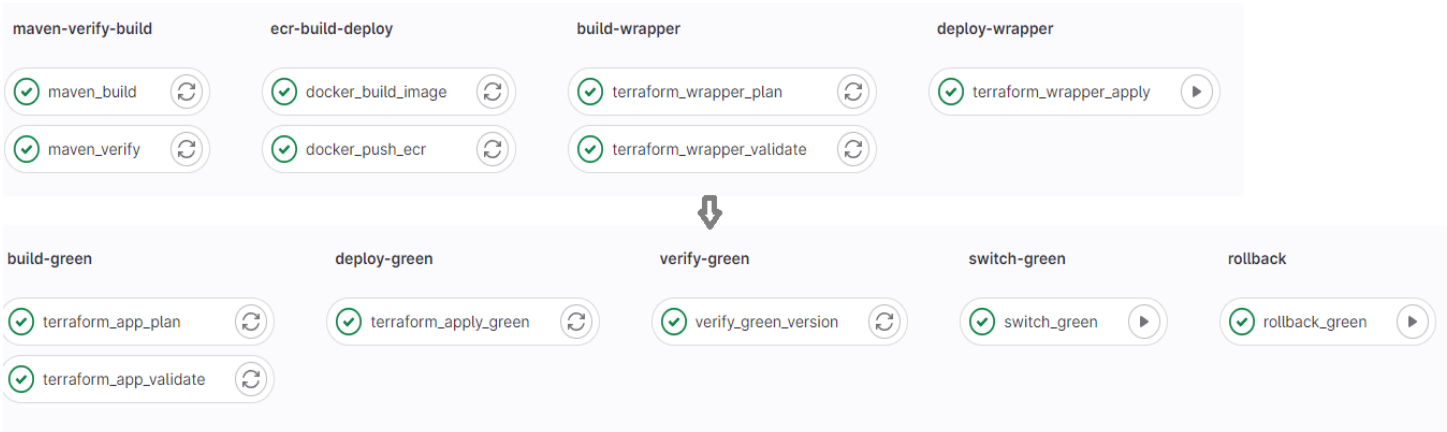


Рисунок 3.15 - Конвеєр неперервної інтеграції та доставки для «restaurant_web_backend»

3.3.3.2 Розгортання рівня бізнес-логіки

Для того, щоб розгорнути рівень бізнес-логіки було, по-перше, описано власне копію контейнеризованого застосунку - визначення завдання («task definition»). У ньому вказано порти хоста та контейнера, пам’ять та центральний процесор, «IAM» роль, налаштування логування. [46] На рисунку 3.15 зображено розгорнуте визначення завдання у веб-консолі «AWS».

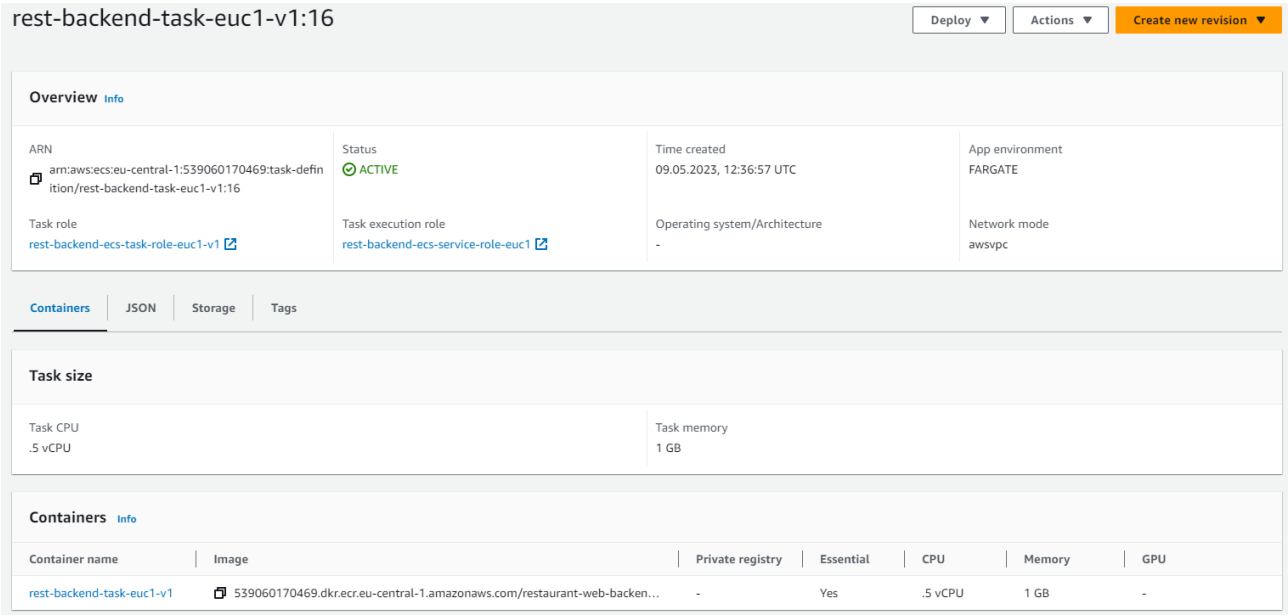


Рисунок 3.15 - Розгорнуте визначення завдання рівня бізнес-логіки

Іншим важливим компонентом є сервіс «ECS». Він керує одночасним запуском та підтримкою вказаної кількості екземплярів визначень завдань. [47] У ньому міститься конфігурація мережі - у цього рівня власна підмережа, як зазначалося раніше. Також було розгорнуто окрему групу безпеки, що дозволяє вхідний трафік лише балансувальнику навантаження. У визначенні сервісу «ECS» було вказано цільову групу. Саме вона направляє запити з балансувальника навантаження на контейнери. У ній також міститься налаштована перевірка працездатності («healthcheck»). [48] Окрім цільової групи було визначено правило балансування навантаження. На рисунку 3.16 зображено вже розгорнутий сервіс «ECS».

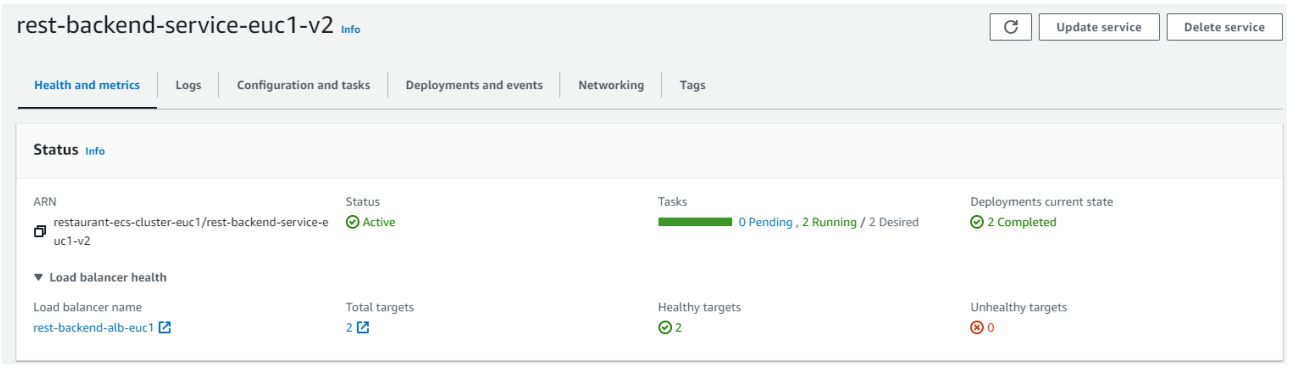


Рисунок 3.16 - Розгорнутий сервіс «ECS»

Що стосується масштабування, то було розгорнуто наступні ресурси. По перше, два алярми сервісу «CloudWatch», що базуються на середньому рівні завантаженості центрального процесора. По-друге, дві політики автоматичного масштабування, що змінюють кількість розгорнутих визначень завдань на 1 і -1. Кожна з них є дією для відповідного алярму. Налаштовано ціль масштабування - сервіс «ECS» для нашої конфігурації. На рисунку 3.17 зображено приклад подій, пов'язаних з масштабуванням.

5/9/2023, 2:46:06 PM GMT+3	service rest-backend-service-euc1-v2 has reached a steady state.
5/9/2023, 2:44:48 PM GMT+3	service rest-backend-service-euc1-v2 registered 1 targets in target-group rest-backend-euc1-tg-v2
5/9/2023, 2:44:09 PM GMT+3	service rest-backend-service-euc1-v2 has started 1 tasks: task ba4df434eb6a444d9e8a8e2f2d54308d.
5/9/2023, 2:43:57 PM GMT+3	Message: Successfully set desired count to 3. Change successfully fulfilled by ecs. Cause: monitor alarm rest-backend-ecs-cpu-high-alarm-euc1 in state ALARM triggered policy rest-backend-scale-up-euc1

Рисунок 3.17 - Масштабування сервісу «ECS»

3.3.3.3 Розгортання балансувальника навантаження

У конвеєрі неперервної інтеграції та доставки «restaurant_web_backend» було також розгорнуто балансувальник навантаження. Його розміщено у публічній підмережі та зроблено загальнодоступним через групу безпеки. При подальшому розвитку даної роботи, потрібно змінити цю конфігурацію на приватну відповідно до архітектури. Балансувальник навантаження було налаштовано публічним для спрощення тестування та демонстрації. Для його роботи також розгорнуто одного слухача на базі протоколу «HTTP». На рисунку 3.18 зображено балансувальник, а 3.19 - демонструє правила налаштовані всередині слухача.

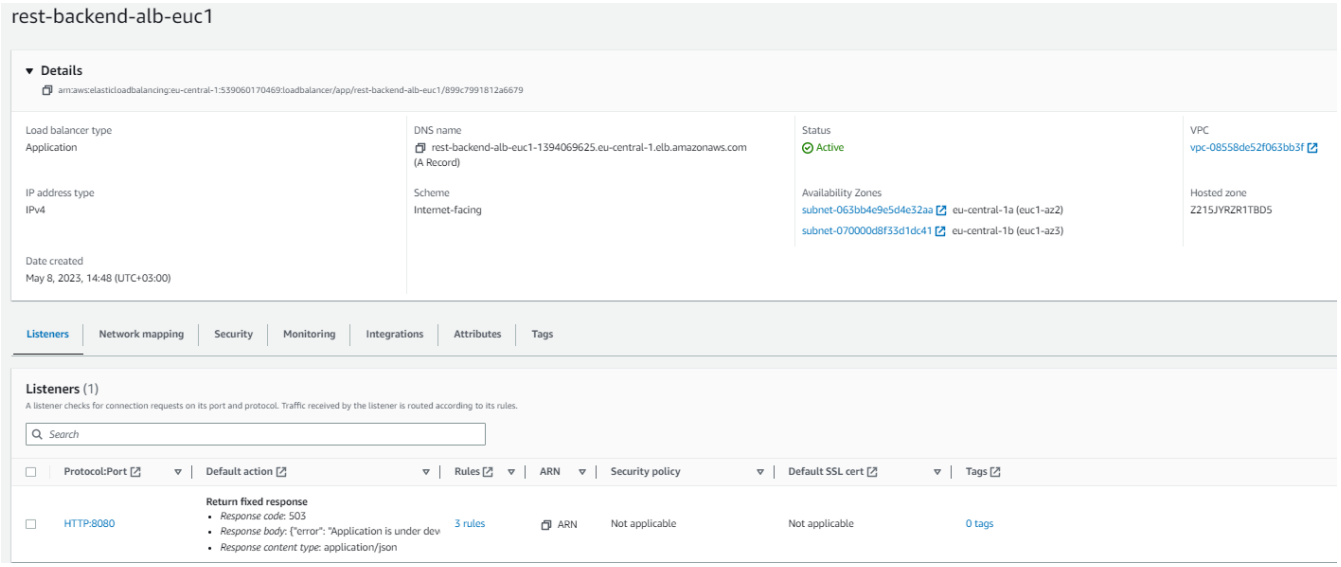


Рисунок 3.18 - Балансувальник навантаження та слухач

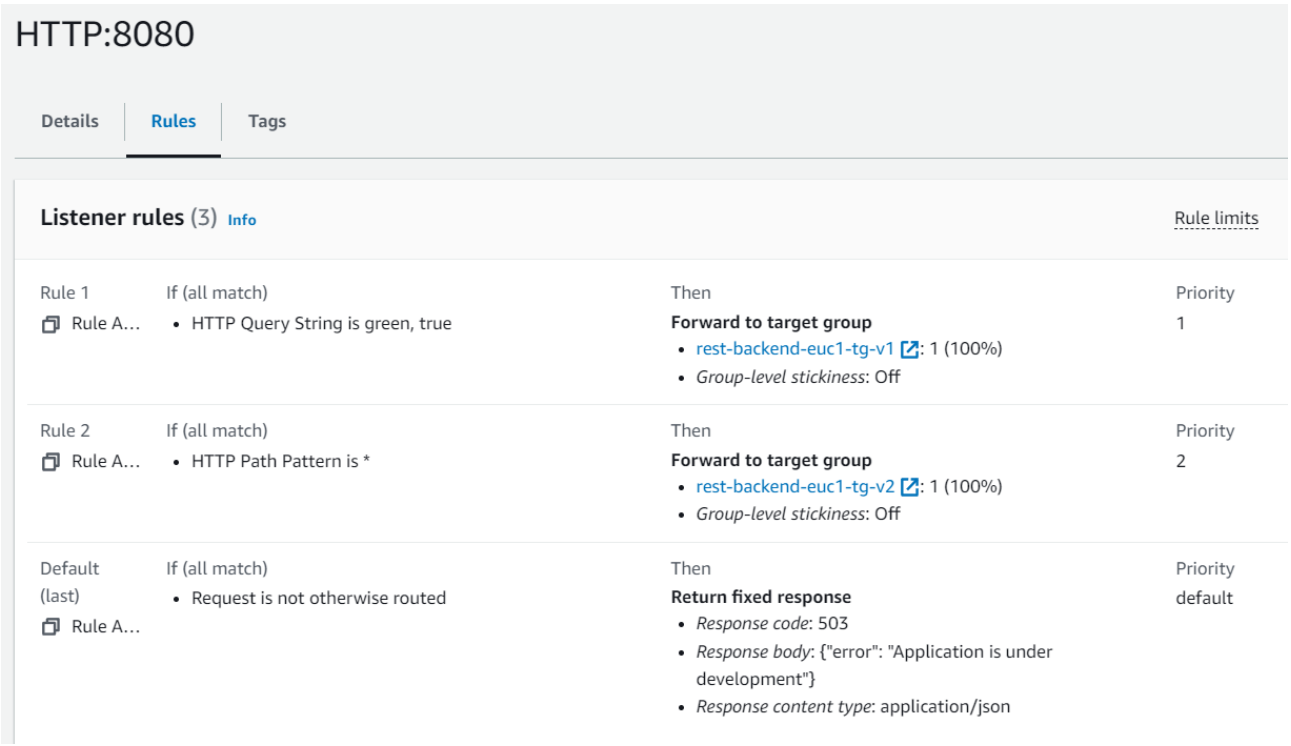


Рисунок 3.19 - Правила балансування навантаження слухача

3.4 Висновок

Отже, результатом детальної розробки та розгортання стало готове REST API. У цьому розділі описано процес написання програмного коду рівня бізнес-логіки, продемонстровано використання технології неперервного розгортання та інтеграції й інфраструктури як коду для взаємодії із хмарним провайдером

«AWS». У ньому детально показано як розгортати та налаштовувати такі компоненти: база даних з використанням керованого сервісу «Amazon Aurora», рівень бізнес-логіки розміщений на сервісі «ECS», балансувальник навантаження, «VPC», «S3» бакет та інші. Також було продемонстровано одну із можливих імплементацій стратегії розгортання «Blue/Green» у хмарному провайдері «AWS».

ВИСНОВКИ

Отже, на початку даної роботи було оглянуто існуючі рішення з подібним функціоналом. На жаль, жодне із них не реалізовує всі можливості такого застосунку для ресторанного бізнесу. Резервування місця відсутнє у всіх рішеннях, а серед трьох додатків лише «Choice» надає можливість оформлення замовлень.

У ході роботи було також проаналізовано різні архітектури та базові технології розгортання, проведено порівняння розміщення інфраструктури у хмарі й на власних ресурсах. Кожен із наведених підходів має свої переваги та недоліки і може бути корисним для вирішення певних задач. Для додатку для ресторанного бізнесу оптимальними були багаторівнева архітектура, контейнери, як базова платформа та розміщення у хмарі.

У результаті даної роботи створено структуру цього застосунку. Завдяки розміщенню у двох зонах доступу - система відмовостійка, а через створення приватної віртуальної хмари та рівня фаєрволу - вдалося підвищити безпеку. У архітектурі використано такі сервіси провайдера «AWS»: «Aurora», «Elastic Container Service» («ECS»), «Elastic Load Balancing», «WAF», «CloudFront» та інші. На додачу, вдалося спроектувати два додаткових безсерверних компоненти, продемонструвавши таким чином, ще більше особливостей цієї хмарної платформи.

Також вдалося розробити детально модуль API. Воно надає такі можливості як електронне меню, оформлення, оплата та оцінка замовлення і резервування столиків. У подальшому залишається розробити рівень користувацького інтерфейсу та безсерверні компоненти - тоді застосунок можна буде впроваджувати на підприємстві.

Було розгорнуто рівень бази даних, бізнес-логіки та балансувальник навантаження. Як наслідок - REST API доступне для використання. У роботі висвітлено автоматичне розгортання системи у хмарному провайдері «AWS», за допомогою конвеєра неперервної інтеграції та доставки «Gitlab CI» та

використання технології «інфраструктура як код» з інструментом «Terraform». Розроблені файли конфігурації пайплайну та ресурсів можна застосувати і для інших компонентів чи навіть застосунків. На додачу, було продемонстровано одну із можливих імплементацій стратегії розгортання «Blue/Green» на цій хмарній платформі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Are Gartner's 'Cautions' About Big 3 Cloud Providers? [Електронний ресурс] / David Ramel - Режим доступу: https://virtualizationreview.com/articles/2021/08/04/gartner-cloud-2021.aspx
2. Автоматизація ресторану [Електронний ресурс] - Режим доступу: https://www.vostok.dp.ua/ukr/infa1/Avtomatizatsiya/avtomatizatsiya_restorana/
3. Martin Fowler. Patterns of Enterprise Application Architecture / Martin Fowler - Pearson Education, Inc., 2003. - 517с.
4. Advantages and Disadvantages of 1 Tier Architecture [Електронний ресурс] - Режим доступу: https://www.hsslive.co.in/2023/01/advantages-disadvantages-of-1-tier-architecture.html
5. What are the advantages and disadvantages of architecture (1-tier, 2-tier, 3-tier and n-tier)? [Електронний ресурс] - Режим доступу: https://www.quora.com/What-are-the-advantages-and-disadvantages-of-architecture-1-tier-2-tier-3-tier-and-n-tier
6. Web Application Architecture: The Latest Guide 2022 [Електронний ресурс] - Режим доступу: https://www.clickittech.com/devops/web-application-architecture/#h-an-overview-of-web-application-architecture
7. Microservices 101 — A Glossary of 11 Most Popular Terms and Acronyms [Електронний ресурс] - Режим доступу: https://m.oursky.com/microservices-101-a-glossary-of-11-most-popular-terms-and-acronyms-2a1a941cd1e1
8. What Is Virtualization? [Електронний ресурс] - Режим доступу: https://aws.amazon.com/what-is/virtualization/
9. What are virtual machines (VMs)? [Електронний ресурс] - Режим доступу: https://www.ibm.com/topics/virtual-machines

10. What's The Difference Between Containers And Virtual Machines? [Електронний ресурс] - Режим доступу: https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines/
11. Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs [Електронний ресурс] - Режим доступу: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms
12. Building Applications with Serverless Architectures [Електронний ресурс] - Режим доступу: https://aws.amazon.com/lambda/serverless-architectures-learn-more/
13. Serverless Architecture Overview [Електронний ресурс] - Режим доступу: https://www.datadoghq.com/knowledge-center/serverless-architecture/
14. What is cloud computing? [Електронний ресурс] - Режим доступу: https://aws.amazon.com/what-is-cloud-computing/?nc2=h_l2_cc
15. What is Amazon Relational Database Service (Amazon RDS)? [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html
16. Amazon Aurora Serverless [Електронний ресурс] - Режим доступу: https://aws.amazon.com/rds/aurora/serverless/
17. What is AWS Secrets Manager? [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html
18. What is Amazon Elastic Container Service? [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html
19. Scheduled Scaling for Application Auto Scaling [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/autoscaling/application/userguide/application-auto-scaling-scheduled-scaling.html

20. AWS SDK for Java Documentation [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/sdk-for-java/index.html
21. What Is Load Balancing? [Електронний ресурс] - Режим доступу: https://www.nginx.com/resources/glossary/load-balancing/
22. Load balancer types [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html
23. What is Firewall? [Електронний ресурс] - Режим доступу: https://www.cisco.com/c/en/us/products/security/firewalls/what-is-a-firewall.html
24. How AWS WAF works [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/waf/latest/developerguide/how-aws-waf-works.html
25. Managed rule groups [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/waf/latest/developerguide/waf-managed-rule-groups.html
26. What is Amazon CloudFront? [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html
27. How CloudFront delivers content [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/HowCloudFrontWorks.html
28. Amazon Cognito user pools [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html
29. Using AWS Lambda with Amazon API Gateway [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html
30. Amazon EventBridge Scheduler [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/eventbridge/latest/userguide/scheduler.html

31. DevOps [Електронний ресурс] - Режим доступу: https://www.atlassian.com/devops
32. Infrastructure as code [Електронний ресурс] / Ian Buchanan - Режим доступу: https://www.atlassian.com/microservices/cloud-computing/infrastructure-as-code
33. What is Terraform? [Електронний ресурс] - Режим доступу: https://developer.hashicorp.com/terraform/intro
34. Available backends - S3 [Електронний ресурс] - Режим доступу: https://developer.hashicorp.com/terraform/language/settings/backends/s3
35. What are database migrations? [Електронний ресурс] - Режим доступу: https://www.prisma.io/dataguide/types/relational/what-are-database-migrations
36. Spring Data JPA - Reference Documentation [Електронний ресурс] - Режим доступу: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#preface
37. Connect to a SQL database with credentials in an AWS Secrets Manager secret [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/secretsmanager/latest/userguide/retrieving-secrets_jdbc.html
38. Command: validate [Електронний ресурс] - Режим доступу: https://developer.hashicorp.com/terraform/cli/commands/validate
39. Command: plan [Електронний ресурс] - Режим доступу: https://developer.hashicorp.com/terraform/cli/commands/plan
40. Command: apply [Електронний ресурс] - Режим доступу: https://developer.hashicorp.com/terraform/cli/commands/apply
41. Connect to the internet using an internet gateway [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html
42. NAT gateways [Електронний ресурс] - Режим доступу:

https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html

43. Resource: aws_rds_cluster [Електронний ресурс] - Режим доступу: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/rds_cluster
44. Blue/Green Deployments [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html
45. Introduction to the Build Lifecycle [Електронний ресурс] - Режим доступу: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
46. Amazon ECS task definitions [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html
47. Amazon ECS services [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_services.html
48. Target groups for your Application Load Balancers [Електронний ресурс] - Режим доступу: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html

ДОДАТКИ

Додаток А

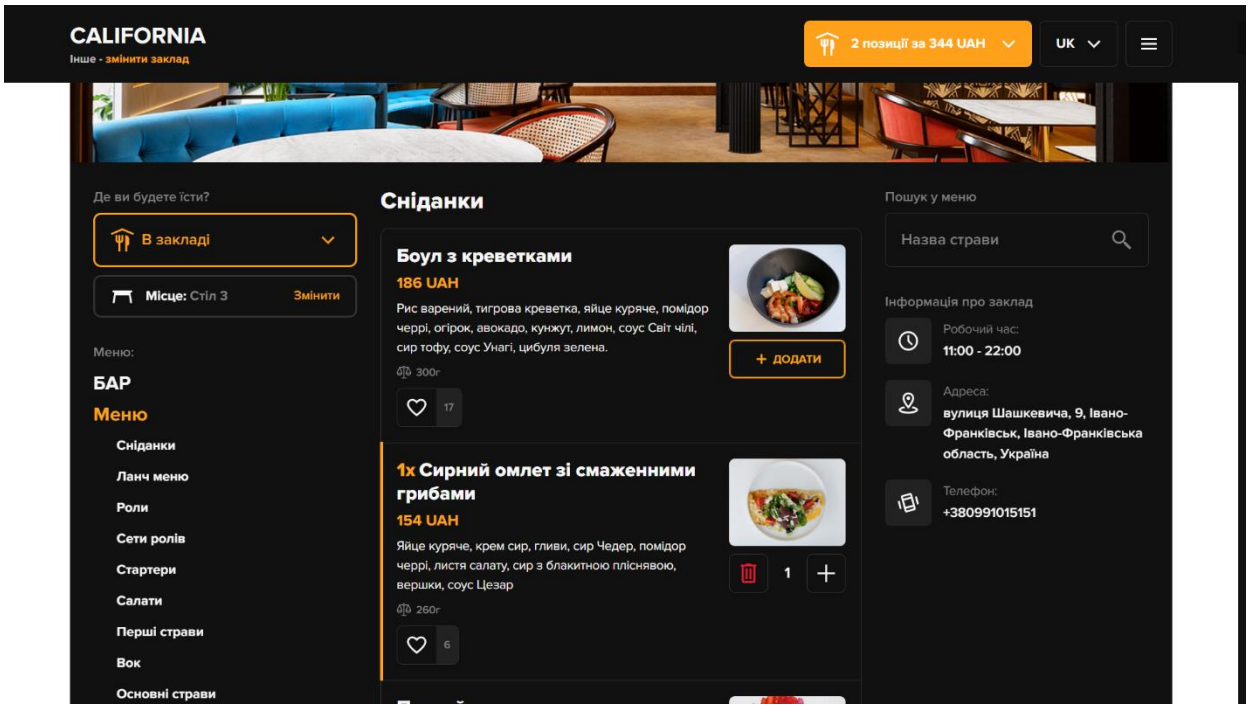
(додатковий)

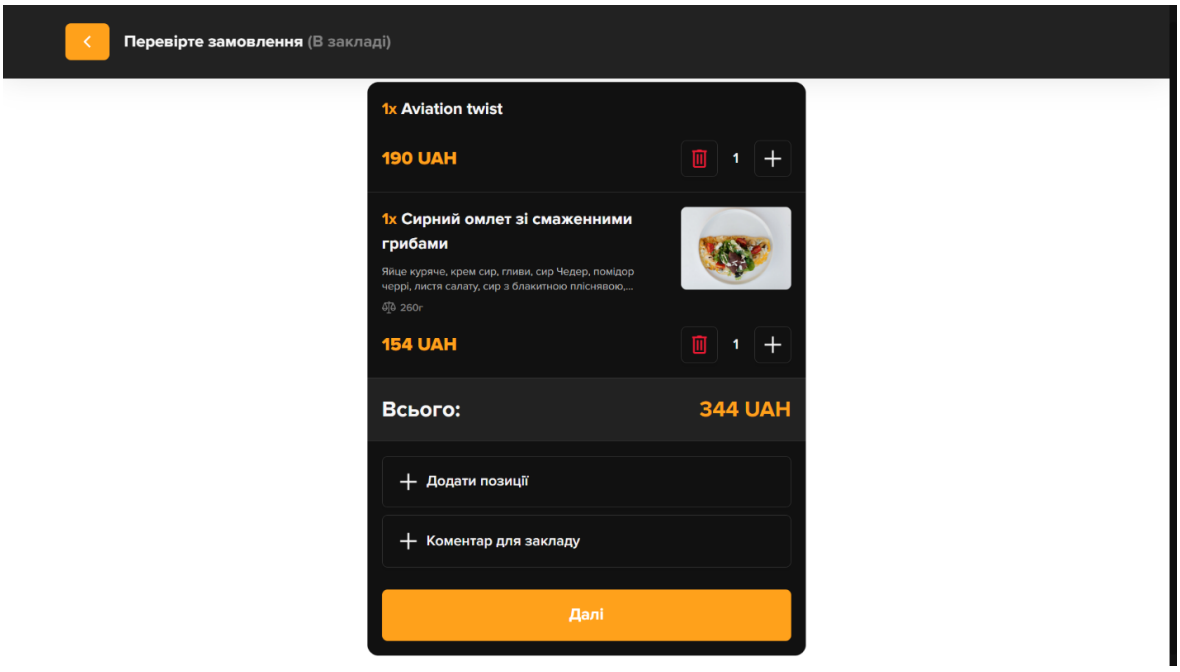
Застосунки для ресторанного бізнесу

1. Застосунок «Choice»

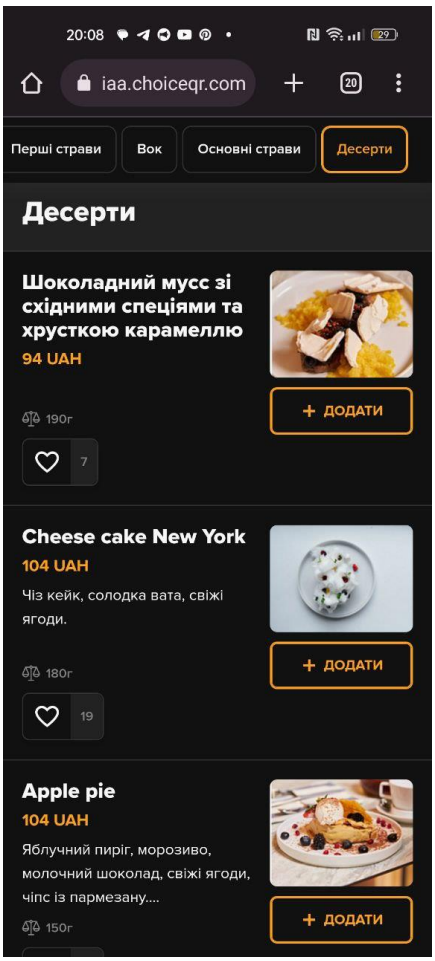
   Посилання: https://californiaa.choiceqr.com/

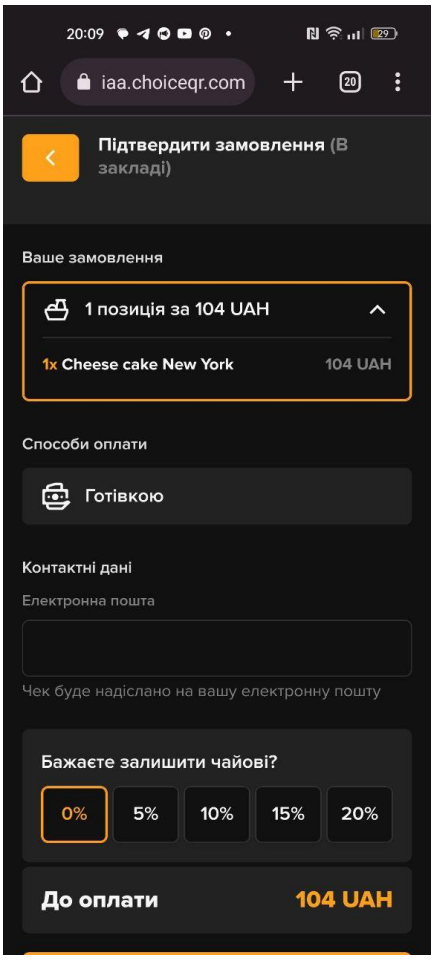
   а) декстопна версія:





б) версія для мобільних пристроїв:

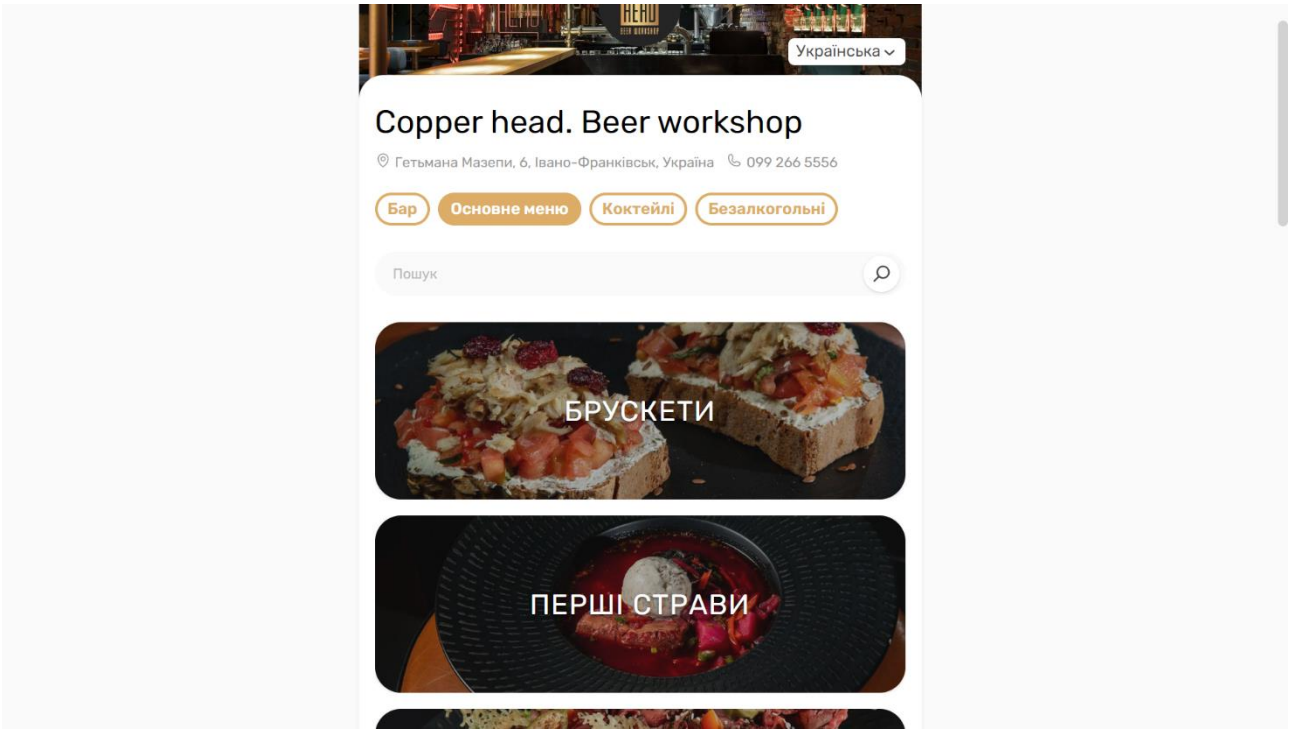


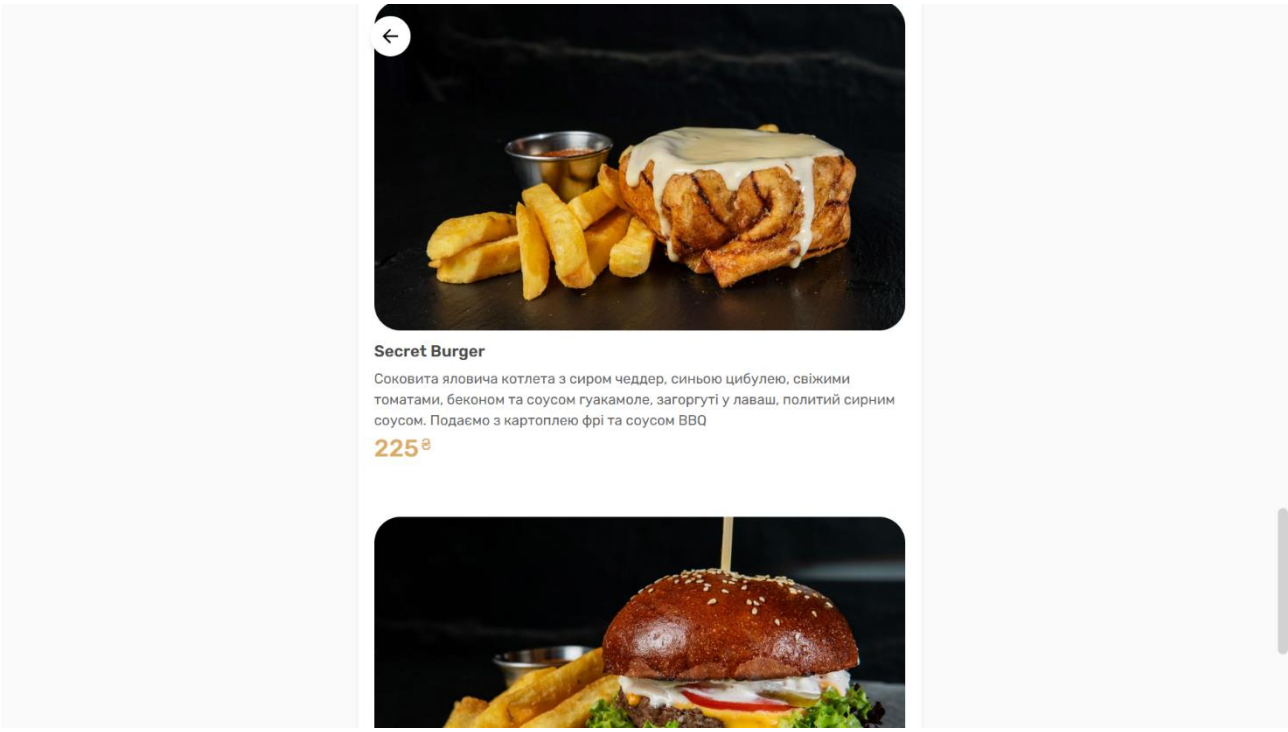


2. Застосунок «OddMenu»

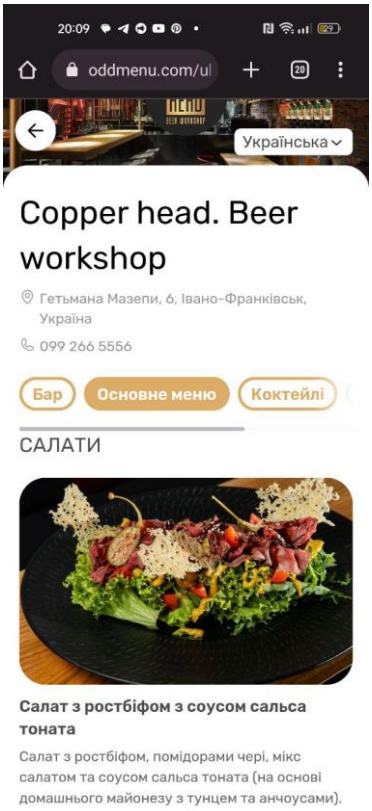
   Посилання: https://oddmenu.com/uk/p/copperhead-beerworkshop

   а) декстопна версія:

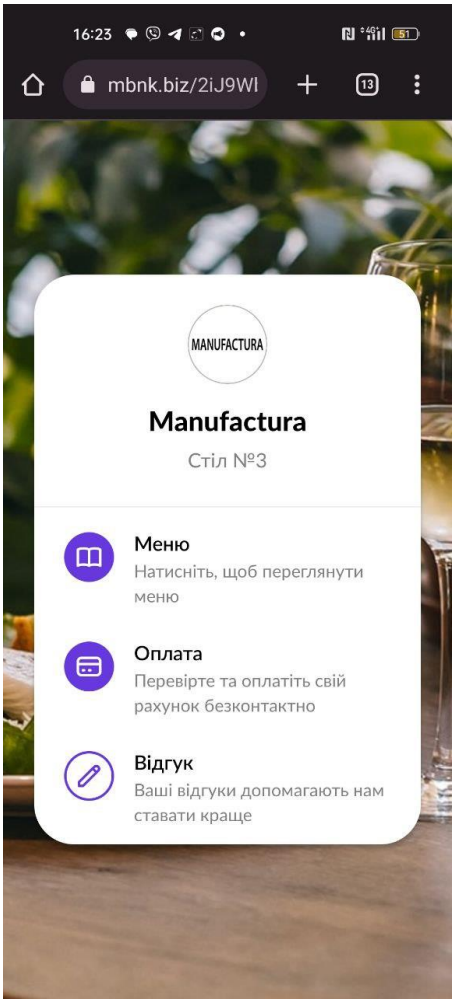


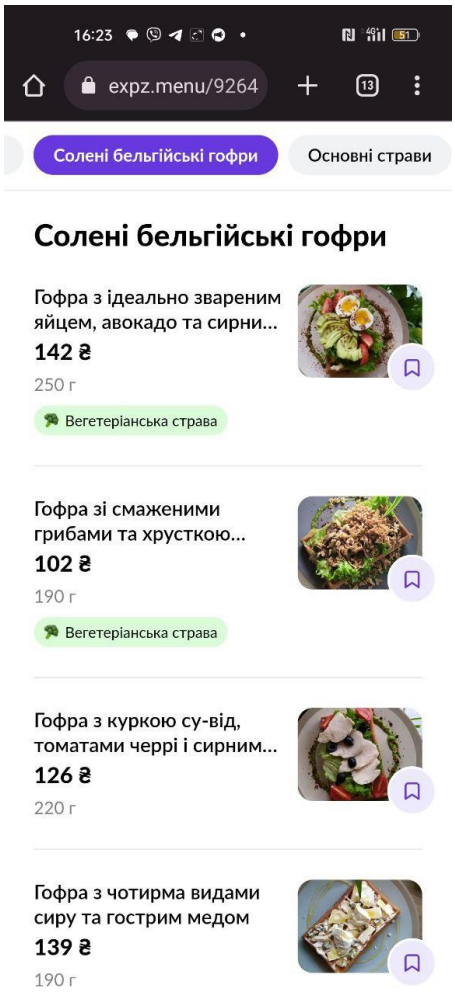


б) версія для мобільних пристроїв:



3. Застосунок «expirenza»






17:01
mbnk.biz/2iJ9WI
Сума рахунку 439 ₴
Manufactura • Стіл №3
Гість 1
Капучіно 54 ₴
Гофра з куркою та черрі 126 ₴
Разом з гостя 180 ₴
Гість 2
Капучіно 54 ₴
Гофра Чіз з куркою та беконом 205 ₴
Разом з гостя 259 ₴
До сплати 439 ₴
Це мій рахунок →

Додаток Б

(обов'язковий)

SQL-скрипти міграцій схеми бази даних

1. Файл V1.0__Initial.sql

```
DROP TABLE IF EXISTS "dish_category";

CREATE TABLE "dish_category" (
    "id" SERIAL PRIMARY KEY,
    "name" VARCHAR(30) NOT NULL UNIQUE,
    "parent_category_id" INTEGER,
    CONSTRAINT "parent_category_fk" FOREIGN KEY
("parent_category_id") REFERENCES "dish_category"("id")
);

DROP TABLE IF EXISTS "dish";

CREATE TABLE "dish" (
    "id" SERIAL PRIMARY KEY,
    "name" VARCHAR(100) NOT NULL UNIQUE,
    "description" varchar(350),
    "s3image_link" varchar(255),
    "price" INTEGER NOT NULL,
    "category_id" INTEGER NOT NULL,
    CONSTRAINT "category_fk" FOREIGN KEY ("category_id")
REFERENCES "dish_category"("id")
);

DROP TABLE IF EXISTS "restaurant_table";

CREATE TABLE "restaurant_table" (
    "id" SERIAL PRIMARY KEY
);

DROP TABLE IF EXISTS "table_booking";

CREATE TABLE "table_booking" (
    "id" BIGSERIAL PRIMARY KEY,
    "phone_number" VARCHAR(15),
    "booking_date_time" TIMESTAMP with time zone,
    "created_date" TIMESTAMP,
    "table_id" INTEGER,
    "number_of_people" INTEGER NOT NULL,
    CONSTRAINT "number_of_people_min" CHECK ( "number_of_people"
>= 1),
    CONSTRAINT "table_fk" FOREIGN KEY ("table_id") REFERENCES
"restaurant_table"("id")
);

DROP TABLE IF EXISTS "order_rating";
```

```
CREATE TABLE "order_rating"(
                                "id" BIGSERIAL PRIMARY KEY,
                                "rating" SMALLINT NOT NULL,
                                "comment" VARCHAR(255),
                                CONSTRAINT "rating_range" CHECK (
"rating" >= 1 AND "rating" <=5 )
);

DROP TABLE IF EXISTS "restaurant_order";

CREATE TABLE "restaurant_order" (
    "id" BIGSERIAL PRIMARY KEY,
    "order_status" varchar(15) NOT NULL DEFAULT 'ACTIVE',
    "created_date" TIMESTAMP,
    "table_id" INTEGER,
    "rating_id" BIGINT,
    CONSTRAINT "table_fk" FOREIGN KEY ("table_id") REFERENCES
"restaurant_table"("id"),
    CONSTRAINT "rating_fk" FOREIGN KEY ("rating_id") REFERENCES
"order_rating"("id")
);

DROP TABLE IF EXISTS "order_line";

CREATE TABLE "order_line" (
    "dish_id" INTEGER,
    "order_id" BIGINT,
    "amount" SMALLINT,
    CONSTRAINT "order_line_pk" PRIMARY KEY ("dish_id",
"order_id"),
    CONSTRAINT "dish_fk" FOREIGN KEY ("dish_id") REFERENCES
"dish"("id"),
    CONSTRAINT "order_fk" FOREIGN KEY ("order_id") REFERENCES
"restaurant_order"("id")
);
```

2. Файл V1.1__CategoryData.sql

```
INSERT INTO "dish_category"("name")
VALUES ('Напої'),
       ('Алкогольні напої'),
       ('Безалкогольні напої');

UPDATE "dish_category"
SET "parent_category_id" = "subquery"."id"
    FROM (SELECT "dish_category"."id"
          FROM "dish_category"
          WHERE "dish_category"."name" = 'Напої') AS "subquery"
WHERE "dish_category"."name" != 'Напої';

INSERT INTO "dish_category"("name")
```

```
VALUES ('Кава'),
       ('Чай'),
       ('Лимонади'),
       ('Фреші'),
       ('Інші');

UPDATE "dish_category"
SET "parent_category_id" = "subquery"."id"
    FROM (SELECT "dish_category"."id"
          FROM "dish_category"
          WHERE "dish_category"."name" = 'Безалкогольні напої')
AS "subquery"
WHERE "dish_category"."name" != 'Напої' AND
"dish_category"."parent_category_id" IS NULL ;

INSERT INTO "dish_category"("name")
VALUES ('Коктейлі'),
       ('Вино'),
       ('Пиво'),
       ('Міцний алкоголь');

UPDATE "dish_category"
SET "parent_category_id" = "subquery"."id"
    FROM (SELECT "dish_category"."id"
          FROM "dish_category"
          WHERE "dish_category"."name" = 'Алкогольні напої') AS
"subquery"
WHERE "dish_category"."name" != 'Напої' AND
"dish_category"."parent_category_id" IS NULL;

INSERT INTO "dish_category"("name")
VALUES ('Страви'),
       ('Закуски'),
       ('Салати'),
       ('Перші страви'),
       ('Основні страви'),
       ('Десерти');

UPDATE "dish_category"
SET "parent_category_id" = "subquery"."id"
    FROM (SELECT "dish_category"."id"
          FROM "dish_category"
          WHERE "dish_category"."name" = 'Страви') AS "subquery"
WHERE "dish_category"."name" != 'Напої' AND
"dish_category"."name" != 'Страви' AND

"dish_category"."parent_category_id" IS NULL;
```

3. Файл V1.2__DishesData.sql

```
CREATE OR REPLACE FUNCTION id_of_category(category_name varchar)
    RETURNS int LANGUAGE SQL AS $$
SELECT "id" FROM "dish_category" WHERE "name" = category_name;
```

```
$$;

-- страви
INSERT INTO "dish"("name", "description", "s3image_link",
"price", "category_id")
VALUES ('Цезар', 'Листя салату, черi, філе куряче, перепелине
яйце, пармезан, соус Цезар',
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/Cesar.jpg', 99, id_of_category('Салати')),
       ('Грецький', 'Фета, помідори, огірки, оливки, перець,
цибуля, олія',
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/hretskyi-salat.jpeg', 75,
id_of_category('Салати')),
       ('З лососем та авокадо', 'Рукола, лосось ,авокадо, черi',
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/losos.jpg', 120, id_of_category('Салати')),
       ('Тар-тар з тунця', NULL,
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/tartar.jpg', 120, id_of_category('Закуски')),
       ('Брускета з черi', NULL, NULL, 85,
id_of_category('Закуски')),
       ('Грибний крем-суп', 'Ніжний суп з шампіньйонів та білих
грибів. Подається з грінками',
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/kremsup.jpg', 75, id_of_category('Перші
страви')),
       ('Борщ', 'Традиційний український суп на основі свинини',
NULL, 50, id_of_category('Перші страви')),
       ('Дорадо з рисом та овочами', 'Ціла дорадо запечена у
духовці, рис, шампіньйони, перець, кукурудза', NULL, 130,
        id_of_category('Основні страви')),
       ('Курка дор-блю', 'Курка під соусом із благородного
сиру',
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/dorblu.jpg', 90, id_of_category('Основні
страви')),
       ('Медальйони з свинини з картопляним пюре', NULL,
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/medaliony.jpg', 100, id_of_category('Основні
страви')),
       ('Штрудель з яблуками', 'Традиційний віденський десерт',
NULL, 40, id_of_category('Десерти')),
       ('Чізкейк', NULL,
        'https://menu-images-euc1.s3.eu-central-
1.amazonaws.com/cheesecake.jpg', 100,
id_of_category('Десерти'));

--напої
INSERT INTO "dish"("name", "description", "s3image_link",
"price", "category_id")
VALUES ('Еспресо', NULL, NULL, 20, id_of_category('Кава')),
       ('Капучино', NULL, NULL, 30, id_of_category('Кава')),
       ('Латте', NULL, NULL, 30, id_of_category('Кава')),
```

```
       ('Чорний', NULL, NULL, 25, id_of_category('Чай')),
       ('Зелений', NULL, NULL, 25, id_of_category('Чай')),
       ('Фруктовий', NULL, NULL, 25, id_of_category('Чай')),
       ('Ківі-бузина', NULL, NULL, 40,
id_of_category('Лимонади')),
       ('Класичний', NULL, NULL, 30,
id_of_category('Лимонади')),
       ('Апельсиновий', NULL, NULL, 30,
id_of_category('Фреші')),
       ('Грейпфрутовий', NULL, NULL, 35,
id_of_category('Фреші')),
       ('Coca-Cola', NULL, NULL, 20, id_of_category('Інші')),
       ('Сік', NULL, NULL, 15, id_of_category('Інші')),
       ('Комбуча', NULL, NULL, 30, id_of_category('Інші'));
```

4. Файл V1.3__RestaurantTableData.sql

```
INSERT INTO "restaurant_table"("id")
SELECT * FROM generate_series(1,12);
```

Додаток В

(обов'язковий)

Конфігураційні файли

Файл application.properties (спільний для обох профайлів)

```
spring.profiles.active=@spring.profiles.active@
server.port=8080
#Actuator
management.endpoints.web.base-path=/
management.endpoint.health.show-details=always
management.endpoints.web.exposure.include=health
management.health.diskspace.enabled=false
```

Файл application-local.properties (для локального середовища)

```
#Db
spring.datasource.url=jdbc:postgresql://host.docker.internal:5433/
restaurant
spring.datasource.username=postgres
spring.datasource.password=postgress
spring.jpa.show-sql=true
```

Файл application-runtime.properties (для хмарного середовища)

```
#db
spring.datasource.driver-class-
name=com.amazonaws.secretsmanager.sql.AWSSecretsManagerPostgreSQLD
river
spring.jpa.database-
platform=org.hibernate.dialect.PostgreSQLDialect
spring.datasource.url=jdbc-secretsmanager:postgresql://restaurant-
db-cluster-euc1.cluster-cqt28de1uswo.eu-central-
1.rds.amazonaws.com:5432/restaurant
spring.datasource.username=restaurant-db-creds-euc1
```

Додаток Г

(обов’язковий)

Конвеєр неперервної інтеграції та доставки для «shared-resources»

Файл .gitlab-ci.yml

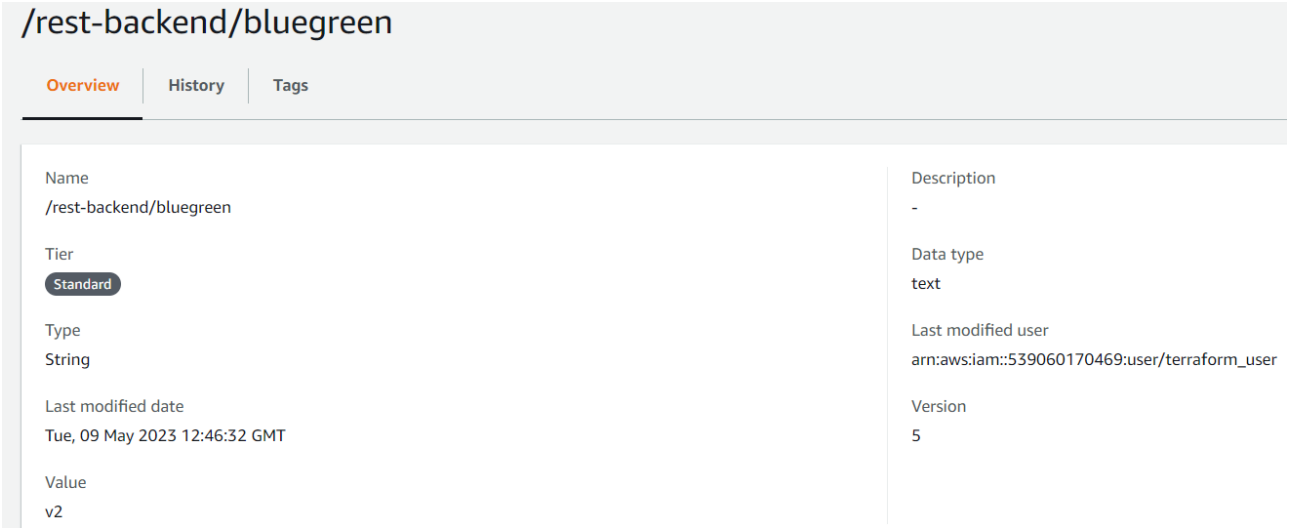
```
stages:
  - shared-resources-build
  - shared-resources-deploy

image:
  name: hashicorp/terraform:light
  entrypoint:
    - '/usr/bin/env'
    -
'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
'

before_script:
  - cd terraform/shared-resources
  - rm -rf .terraform
  - terraform --version
  - terraform init
  - terraform fmt

terraform_validate:
  stage: shared-resources-build
  script:
    - terraform validate

terraform_plan:
  stage: shared-resources-build
  needs:
    - terraform_validate
  script:
    - terraform plan -var-file=terraform.tfvars -out "tfplan"
  artifacts:
    paths:
      - terraform/shared-resources/tfplan

terraform_apply:
  stage: shared-resources-deploy
  needs:
    - terraform_plan
  when: manual
  script:
    - terraform apply -input=false "tfplan"
```

Додаток Д

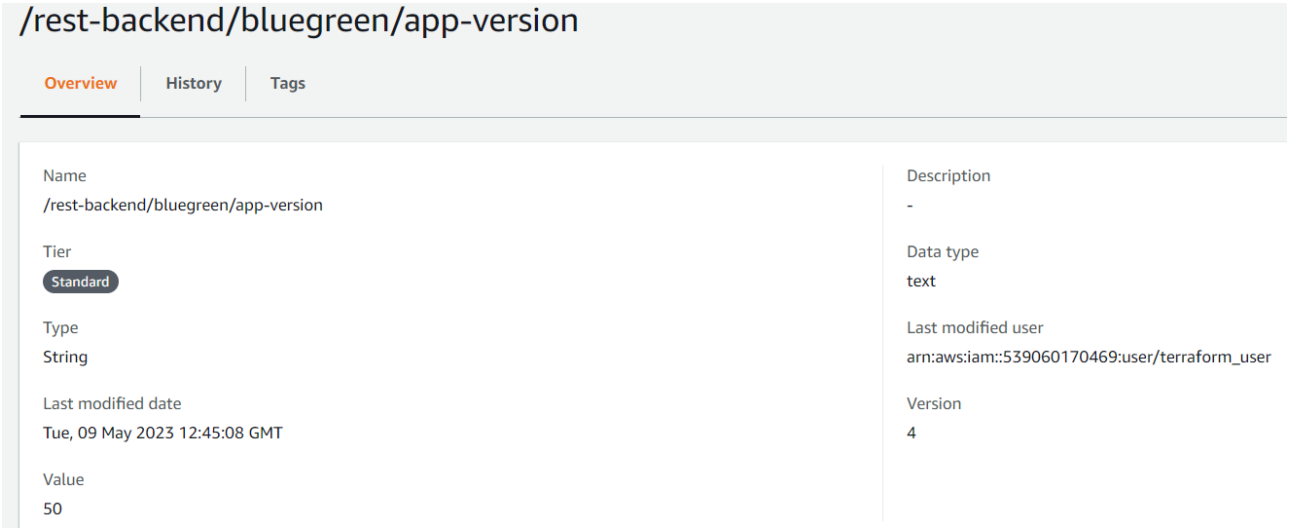
(обов'язковий)

Параметри сервісу «Parameter Store», що використовуються для стратегії «Blue/Green» розгортання

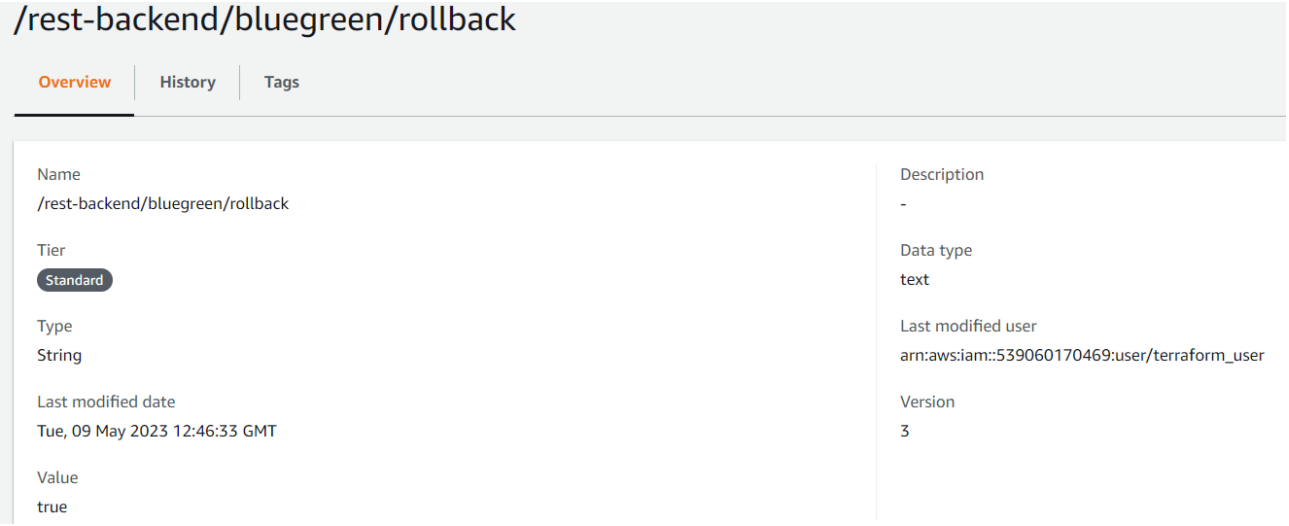
1. Параметр «/rest-backend/bluegreen» - зберігає інформацію про те, ресурси якої робочої області розгорнуті як поточна версія застосунку. Може набувати значень «v1»/«v2».



2. Параметр «/rest-backend/bluegreen/app-version» - зберігає інформацію про те, яка поточна версія збірки активна, вказуючи фактично на ідентифікатор конвеєра і версія зображення контейнера.



3. Параметр «/rest-backend/bluegreen/rollback» - зберігає інформацію про те, чи відбувався відкат для поточної версії застосунку - може набувати значення «true», якщо відбувся, та «false», якщо ні.



Додаток Е

(додатковий)

Dockerfile для зображення контейнера з «AWS CLI», «jq» та «Terraform»

```
FROM public.ecr.aws/amazonlinux/amazonlinux:2 as terraform_aws
RUN yum install -q -y unzip git &&\
    curl
"https://releases.hashicorp.com/terraform/1.4.6/terraform_1.4.6_li
nux_amd64.zip" -o terraform.zip &&\
    unzip terraform.zip &&\
    mv terraform /usr/local/bin/ &&\
    curl "https://awscli.amazonaws.com/awscli-exe-linux-
x86_64.zip" -o awscli.zip &&\
    unzip awscli.zip &&\
    cd aws &&\
    bash install --bin-dir /aws-cli-bin/

FROM public.ecr.aws/amazonlinux/amazonlinux:2
RUN yum update -y && \
    yum install -y git jq
COPY --from=terraform_aws /usr/local/aws-cli/ /usr/local/aws-cli/
COPY --from=terraform_aws /aws-cli-bin/ /usr/local/bin/
COPY --from=terraform_aws /usr/local/bin/terraform/
/usr/local/bin/terraform/
ENV PATH="${PATH}:/usr/local/bin/terraform/"
```

Додаток Ж

(обов'язковий)

Конвеєр неперервної інтеграції та доставки для «restaurant_web_backend»

Файл .gitlab-ci.yml

```
stages:
  - maven-verify-build
  - ecr-build-deploy
  - build-wrapper
  - deploy-wrapper
  - build-green
  - deploy-green
  - verify-green
  - switch-green
  - rollback

variables:
  JAR_NAME: restaurant-backend-1.0-SNAPSHOT.jar
  IMAGE_NAME: restaurant-web-backend
  ECR_REGISTRY: 539060170469.dkr.ecr.eu-central-1.amazonaws.com
  DOCKER_HOST: tcp://docker:2375
  AWS_DEFAULT_REGION: eu-central-1
  APP_NAME: rest-backend
  LB_NAME: rest-backend
  CLUSTER_NAME: restaurant

.maven_template:
  image: maven:3.9.1-amazoncorretto-17

.ecr_docker_template:
  image:
    name: amazon/aws-cli
    entrypoint: [ "" ]
  services:
    - docker:dind
  before_script:
    - amazon-linux-extras install docker

.terraform_wrapper_template:
  image:
    name: hashicorp/terraform:light
    entrypoint:
      - '/usr/bin/env'
      -
'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
'
  before_script:
    - git config --global
url."https://tf:${GIT_TF_TOKEN}@gitlab.com".insteadOf
https://gitlab.com
```

```
    - cd terraform/restaurant-backend/wrapper
    - rm -rf .terraform
    - terraform --version
    - terraform init
    - terraform fmt


.terraform_app_template:
  image: public.ecr.aws/p7d0d1e0/rest-back-terraform-aws:1.0
  before_script:
    - cd terraform/restaurant-backend/app
    - rm -rf .terraform
    - terraform init
    - terraform fmt

.terraform-plan-blue-green-plan-script: &tf-plan
  - |
    export
TF_VAR_ecr_image=${ECR_REGISTRY}/${IMAGE_NAME}:${CI_PIPELINE_IID}
    export TF_VAR_ecr_image_version=${CI_PIPELINE_IID}
    aws ssm put-parameter --name "/rest-backend/bluegreen" --value
none --type String || true
    plan_green() {
      echo "Deploying $1. Current $2"
      terraform workspace select ${1} || terraform workspace new
${1}
      terraform plan -var-file=terraform.tfvars -var current=${1}
-out "tfplan"
    }
    CURRENT_TG=$(aws ssm get-parameter --name "/rest-
backend/bluegreen" | jq -r .Parameter.Value)
    if [[ ${CURRENT_TG} == "none" ]]
    then
      plan_green "v1" "none"
    elif [[ ${CURRENT_TG} == "v1" ]]
    then
      plan_green "v2" "v1"
    elif [[ ${CURRENT_TG} == "v2" ]]
    then
      plan_green "v1" "v2"
    fi

.terraform-deploy-blue-green-script: &tf-apply
  - |
    export
TF_VAR_ecr_image=${ECR_REGISTRY}/${IMAGE_NAME}:${CI_PIPELINE_IID}
    export TF_VAR_ecr_image_version=${CI_PIPELINE_IID}
    export ACTIVE_PRIORITY=2
    export PASSIVE_PRIORITY=1
    deploy() {
      echo "Deploying $1. Current $2"
      LB_ARN=$(aws elbv2 describe-load-balancers --query
'LoadBalancers[*].[LoadBalancerArn]' --output text | grep
${LB_NAME})
```

```
      LB_LISTENER_ARN=$(aws elbv2 describe-listeners --load-
balancer-arn ${LB_ARN} --query 'Listeners[*]' | jq -r '.[] |
select(.Port==8080) | .ListenerArn')
      PASSIVE_RULE_ARN=$(aws elbv2 describe-rules --listener-arn
${LB_LISTENER_ARN} --query Rules[] | jq -r '.[] |
select(.Priority=="'${PASSIVE_PRIORITY}'") | .RuleArn')
      if [[ ${PASSIVE_RULE_ARN} != "" ]]
      then
        aws elbv2 delete-rule --rule-arn ${PASSIVE_RULE_ARN}
      fi
      terraform workspace select "${1}" || terraform workspace new
"${1}"
      terraform apply -input=false "tfplan"
    }
    CURRENT_TG=$(aws ssm get-parameter --name "/rest-
backend/bluegreen" | jq -r .Parameter.Value)
    if [[ ${CURRENT_TG} == "none" ]]
    then
      echo "Nothing was deployed before"
      terraform workspace select "v1" || terraform workspace new
"v1"
      terraform apply -input=false "tfplan"
      TG_ARN=$(aws elbv2 describe-target-groups --query
'TargetGroups[*].[TargetGroupArn]' --output text | grep
${APP_NAME} | grep v1)
      LB_ARN=$(aws elbv2 describe-load-balancers --query
'LoadBalancers[*].[LoadBalancerArn]' --output text | grep
${LB_NAME})
      LB_LISTENER_ARN=$(aws elbv2 describe-listeners --load-
balancer-arn ${LB_ARN} --query 'Listeners[*]' | jq -r '.[] |
select(.Port==8080) | .ListenerArn')
      PASSIVE_RULE_ARN=$(aws elbv2 describe-rules --listener-arn
${LB_LISTENER_ARN} --query Rules[] | jq -r '.[] |
select(.Priority=="'${PASSIVE_PRIORITY}'") | .RuleArn')

      aws elbv2 create-rule --listener-arn ${LB_LISTENER_ARN} --
priority ${ACTIVE_PRIORITY} --conditions \[\{\"Field\":\"path-
pattern\",\"Values\":\[\"*\"\]\}\] --actions
Type=forward,TargetGroupArn=${TG_ARN}
      aws elbv2 modify-rule --rule-arn ${PASSIVE_RULE_ARN} --
conditions \[\{\"Field\":\"query-
string\",\"QueryStringConfig\":\{\"Values\":\[\{\"Key\":\"green\",
\"Value\":\"true\"\}\]\}\}\] --actions
Type=forward,TargetGroupArn=${TG_ARN}

    elif [[ ${CURRENT_TG} == "v1" ]]
    then
      deploy "v2" "v1"
    elif [[ ${CURRENT_TG} == "v2" ]]
    then
      deploy "v1" "v2"
    fi

.switch-blue-green-versions: &switch-script
```

```
  - |
    export ACTIVE_PRIORITY=2
    export PASSIVE_PRIORITY=1
    switch_blue_green() {
      echo "Switching to $1. Current $2"
      LB_ARN=$(aws elbv2 describe-load-balancers --query
'LoadBalancers[*].[LoadBalancerArn]' --output text | grep
${LB_NAME})
      LB_LISTENER_ARN=$(aws elbv2 describe-listeners --load-
balancer-arn ${LB_ARN} --query 'Listeners[*]' | jq -r '.[] |
select(.Port==8080) | .ListenerArn')
      ACTIVE_RULE_ARN=$(aws elbv2 describe-rules --listener-arn
${LB_LISTENER_ARN} --query Rules[] | jq -r '.[] |
select(.Priority=="'${ACTIVE_PRIORITY}'") | .RuleArn')
      TG_ARN=$(aws elbv2 describe-target-groups --query
'TargetGroups[*].[TargetGroupArn]' --output text | grep
${APP_NAME} | grep $1)

      aws elbv2 modify-rule --rule-arn ${ACTIVE_RULE_ARN} --
conditions \[\{\"Field\":\"path-pattern\",\"Values\":\[\"*\"\]\}\]
--actions Type=forward,TargetGroupArn=${TG_ARN}
      echo "Shutting down previous version"
      ECS_CLUSTER=$(aws ecs list-clusters --query
'clusterArns[*][]' | jq -r .[] | grep ${CLUSTER_NAME})
      ECS_SERVICE=$(aws ecs list-services --cluster ${ECS_CLUSTER}
--query serviceArns[*] | jq -r .[] | grep ${APP_NAME} | grep $2)
      aws ecs update-service --cluster ${ECS_CLUSTER} --service
${ECS_SERVICE} --desired-count=0

      aws ssm put-parameter --name "/rest-backend/bluegreen" --
value $1 --overwrite --type String
      aws ssm put-parameter --name "/rest-backend/bluegreen/app-
version" --value ${CI_PIPELINE_IID} --overwrite --type String
      aws ssm put-parameter --name "/rest-
backend/bluegreen/rollback" --value false --overwrite --type
String
    }
    aws ssm put-parameter --name "/rest-backend/bluegreen/app-
version" --value none --type String || true
    CURRENT_VERSION=$(aws ssm get-parameter --name "/rest-
backend/bluegreen/app-version" | jq -r .Parameter.Value)
    if [[ ${CURRENT_VERSION} == ${CI_PIPELINE_IID} ]]
    then
      echo "Already switched to this version"
      exit 0
    fi
    CURRENT_TG=$(aws ssm get-parameter --name "/rest-
backend/bluegreen" | jq -r .Parameter.Value)
    if [[ ${CURRENT_TG} == "none" ]]
    then
      echo "First deploy, already set active version"
      aws ssm put-parameter --name "/rest-backend/bluegreen" --
value v1 --overwrite --type String
      aws ssm put-parameter --name "/rest-backend/bluegreen/app-
```

```
version" --value ${CI_PIPELINE_IID} --overwrite --type String
    elif [[ ${CURRENT_TG} == "v1" ]]
    then
      switch_blue_green "v2" "v1"
    elif [[ ${CURRENT_TG} == "v2" ]]
    then
      switch_blue_green "v1" "v2"
    fi

.rollback-blue-green: &rollback-script
  - |
    export ACTIVE_PRIORITY=2
    export PASSIVE_PRIORITY=1
    rollback() {
      echo "Switching back to $1. Current $2"
      ECS_CLUSTER=$(aws ecs list-clusters --query
'clusterArns[*][]' | jq -r .[] | grep ${CLUSTER_NAME})
      ECS_SERVICE_ACTIVE=$(aws ecs list-services --cluster
${ECS_CLUSTER} --query serviceArns[*] | jq -r .[] | grep
${APP_NAME} | grep $2)
      ECS_SERVICE_PASSIVE=$(aws ecs list-services --cluster
${ECS_CLUSTER} --query serviceArns[*] | jq -r .[] | grep
${APP_NAME} | grep $1)
      DESIRED_COUNT_ACTIVE=$(aws ecs describe-services --cluster
${ECS_CLUSTER} --service ${ECS_SERVICE_ACTIVE} --query
services[*][desiredCount][] --output text)

      echo "Restoring passive version"
      aws ecs update-service --cluster ${ECS_CLUSTER} --service
${ECS_SERVICE_PASSIVE} --desired-count=${DESIRED_COUNT_ACTIVE}

      LB_ARN=$(aws elbv2 describe-load-balancers --query
'LoadBalancers[*].[LoadBalancerArn]' --output text | grep
${LB_NAME})
      LB_LISTENER_ARN=$(aws elbv2 describe-listeners --load-
balancer-arn ${LB_ARN} --query 'Listeners[*]' | jq -r '.[] |
select(.Port==8080) | .ListenerArn')
      ACTIVE_RULE_ARN=$(aws elbv2 describe-rules --listener-arn
${LB_LISTENER_ARN} --query Rules[] | jq -r '.[] |
select(.Priority=="'${ACTIVE_PRIORITY}'") | .RuleArn')
      PASSIVE_RULE_ARN=$(aws elbv2 describe-rules --listener-arn
${LB_LISTENER_ARN} --query Rules[] | jq -r '.[] |
select(.Priority=="'${PASSIVE_PRIORITY}'") | .RuleArn')
      ACTIVE_TG_ARN=$(aws elbv2 describe-target-groups --query
'TargetGroups[*].[TargetGroupArn]' --output text | grep
${APP_NAME} | grep $2)
      PASSIVE_TG_ARN=$(aws elbv2 describe-target-groups --query
'TargetGroups[*].[TargetGroupArn]' --output text | grep
${APP_NAME} | grep $1)
      echo "Switching active rule back"
      aws elbv2 modify-rule --rule-arn ${ACTIVE_RULE_ARN} --
conditions \[\{\"Field\":\"path-pattern\",\"Values\":\[\"*\"\]\}\]
--actions Type=forward,TargetGroupArn=${PASSIVE_TG_ARN}
      aws elbv2 modify-rule --rule-arn ${PASSIVE_RULE_ARN} --
```

```
conditions \[\{\"Field\":\"query-
string\",\"QueryStringConfig\":\{\"Values\":\[\{\"Key\":\"green\",
\"Value\":\"true\"\}\]\}\}\] --actions
Type=forward,TargetGroupArn=${ACTIVE_TG_ARN}

      echo "Shutting down active version"
      aws ecs update-service --cluster ${ECS_CLUSTER} --service
${ECS_SERVICE_ACTIVE} --desired-count=0

      aws ssm put-parameter --name "/rest-backend/bluegreen" --
value $1 --overwrite --type String
      aws ssm put-parameter --name "/rest-
backend/bluegreen/rollback" --value true --overwrite --type String
    }
    ROLLBACK_HAPPENED=$(aws ssm get-parameter --name "/rest-
backend/bluegreen/rollback" | jq -r .Parameter.Value)
    if [[ ${ROLLBACK_HAPPENED} == "true" ]]
    then
      echo "You can not rollback twice, rollback already happened"
      exit 0
    fi
    CURRENT_TG=$(aws ssm get-parameter --name "/rest-
backend/bluegreen" | jq -r .Parameter.Value)
    if [[ ${CURRENT_TG} == "v1" ]]
    then
      rollback "v2" "v1"
    elif [[ ${CURRENT_TG} == "v2" ]]
    then
      rollback "v1" "v2"
    fi

maven_verify:
  extends: .maven_template
  stage: maven-verify-build
  script:
    - mvn verify -ntp

maven_build:
  extends: .maven_template
  stage: maven-verify-build
  needs:
    - maven_verify
  script:
    - mvn clean package -P runtime -Dmaven.test.skip -ntp
  artifacts:
    paths:
      - target/${JAR_NAME}

docker_build_image:
  extends: .ecr_docker_template
  stage: ecr-build-deploy
  needs:
    - maven_build
  script:
```

```
    - docker build . -t ${IMAGE_NAME}:${CI_PIPELINE_IID}
    - docker save ${IMAGE_NAME}:${CI_PIPELINE_IID} > image.tar
    - echo "Built image ${IMAGE_NAME}:${CI_PIPELINE_IID}"
  artifacts:
    paths:
      - image.tar

docker_push_ecr:
  extends: .ecr_docker_template
  stage: ecr-build-deploy
  needs:
    - docker_build_image
  script:
    - aws ecr get-login-password | docker login --username AWS --
password-stdin $ECR_REGISTRY
    - docker load -i image.tar
    - docker tag ${IMAGE_NAME}:${CI_PIPELINE_IID}
${ECR_REGISTRY}/${IMAGE_NAME}:${CI_PIPELINE_IID}
    - docker push ${ECR_REGISTRY}/${IMAGE_NAME}:${CI_PIPELINE_IID}
    - echo "Pushed image to ECR"

terraform_wrapper_validate:
  extends: .terraform_wrapper_template
  stage: build-wrapper
  script:
    - terraform validate

terraform_wrapper_plan:
  extends: .terraform_wrapper_template
  stage: build-wrapper
  needs:
    - terraform_wrapper_validate
  script:
    - terraform plan -var-file=terraform.tfvars -out "tfplan"
  artifacts:
    paths:
      - terraform/restaurant-backend/wrapper/tfplan

terraform_wrapper_apply:
  extends: .terraform_wrapper_template
  stage: deploy-wrapper
  needs:
    - terraform_wrapper_plan
  when: manual
  script:
    - terraform apply -input=false "tfplan"

terraform_app_validate:
  extends: .terraform_app_template
  stage: build-green
  script:
    - terraform validate

terraform_app_plan:
```

```
  extends: .terraform_app_template
  stage: build-green
  needs:
    - terraform_app_validate
    - terraform_wrapper_apply
    - docker_push_ecr
  script:
    *tf-plan
  artifacts:
    paths:
      - terraform/restaurant-backend/app/tfplan

terraform_apply_green:
  extends: .terraform_app_template
  stage: deploy-green
  needs:
    - terraform_app_plan
  script:
    *tf-apply

verify_green_version:
  extends: .terraform_app_template
  stage: verify-green
  before_script:
    - yum install -y perl-JSON-PP
  needs:
    - terraform_apply_green
  script:
    - LB_DNS=$(aws elbv2 describe-load-balancers --query
'LoadBalancers[*].[DNSName]' --output text | grep ${LB_NAME})
    - curl --location http://${LB_DNS}:8080/health?green=true |
json_pp
    - curl --location http://${LB_DNS}:8080/menu/dishes?green=true
| json_pp
  retry: 2

switch_green:
  extends: .terraform_app_template
  stage: switch-green
  needs:
    - verify_green_version
  when: manual
  script:
    *switch-script

rollback_green:
  extends: .terraform_app_template
  stage: rollback
  needs:
    - switch_green
  when: manual
  script:
    *rollback-script
```