Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

Розробка програмного забезпечення з застосуванням CI і CD

Текстова частина до курсової роботи за спеціальністю „Комп'ютерні науки"

Керівник курсової роботи
доктор техн. наук, декан ФІ
Глибовець А.М.

_____
(підпис)
"____" _____ 2020 р.

Виконав студент Благов В. С.
"____" _____ 2020 р.

Київ 2019

Зміст

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

доктор техн. наук, декан ФІ

_____ А. М. Глибовець

(підпис)

“____”_____ 202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Благову Владиславу

3-го курсу факультету інформатики

ТЕМА: Розробка програмного забезпечення з застосуванням CI і CD

Вихідні дані:

- Серверний застосунок із підтриманням технології CI і CD

Зміст ТЧ до курсової роботи:

Вступ
Анотація
1. Аналіз предметної області та наявних проблем і задач
2. Теоретичні засади розробки програмного забезпечення з застосуванням CI / CD
3. Реалізація практичної частини роботи
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі “___” _____ 201_ р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Розробка програмного забезпечення з застосуванням CI і CD

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад 2019 р.	
2.	Огляд літератури за темою роботи	листопад - грудень 2019 р.	
3.	Оволодіння навичками розробки вільних від блокувань структур даних	грудень - лютий 2019 р.	
4.	Створення практичної частини роботи	лютий - березень 2020 р.	
5.	Написання пояснювальної роботи	березень 2020 р.	
6.	Створення слайдів для доповіді та написання доповіді	березень 2020 р.	
7.	Надання роботи керівнику для перевірки	березень - квітень 2020 р.	
8.	Корегування роботи за результатами перевірки керівником	квітень 2020 р.	
9.	Остаточне оформлення пояснювальної роботи та слайдів	квітень-травень 2020 р.	
10.	Захист курсової роботи	15 травня 2020 р.	

Студент Благов В.С. _____

Керівник Глибовець А.М. _____

“_____” _____ _____ р.

Вступ

У розробці будь-якого програмного забезпечення швидка і якісна збірка продукту – є головною конкурентною перевагою. Мобільні додатки, веб-розробка, ігри - ті галузі, де послідовне проходження стадій проектування, побудови та тестування повинно бути максимально швидким. Замість місяців на кожен етап рахунок тут йде на години і навіть на хвилини.

Методика безперервної інтеграції, доставки і розгортання (CI / CD) - невід'ємна частина сучасного процесу розробки, покликана знизити кількість помилок під час інтеграції і розгортання і підвищити швидкість реалізації проектів. CI / CD - це одночасно філософія і набір практик, які часто зміцнюються високонадійними інструментами, які забезпечують автоматичне тестування на кожному кроці конвеєра програмного забезпечення [1].

Анотація

Роботу присвячено застосуванню технології CI / CD для автоматизації виявлення змін у системі контролю версій, збірки проектів, виконанню всіх функціональних тестів (Unit tests), створення артефактів та їх розверненням на хмарі, скрипти міграції бази даних. На основі результатів дослідження було розроблено сервіс для зберігання замовлень довільних товарів та аутентифікації користувача. Застосунок буде розміщений на хмарній платформі із підключенням БД.

1. Аналіз предметної області та наявних проблем і задач

1.1. Проблеми у комерційній розробці програмного забезпечення

Традиційно, або, скажімо, 10 років тому, команду розробників можна було приблизно розділити на розробників - людей, які вміли писати код та операції. Хто ж ці спеціалісти, котрі писали ці самі операції?

Вони є адміністраторами - системними адміністраторами, мережевими адміністраторами, адміністраторами баз даних та всіма іншими людьми, які знають інфраструктуру проекту.

Адміністратори були зацікавлені у збереженні стабільності, щоб мінімізувати ймовірність конфліктів із програмним забезпеченням. Розробники в основному ж дбали про нові функції, нові версії а також, виправлення функціональних помилок. Головною проблемою усього вищесказаного була відсутність добре налаштованої співпраці та спілкування. Як результат, програмне забезпечення не могло бути доставлене в потрібному швидкому темпі,
оскільки тестувальники витрачали багато часу на знаходження багів, а деплой продукту займав чимало часу (10 – 8 годин), через це розробники працювали та виправляли свій код на локальній машині. Які ж проблеми могли виникати? [2]

По-перше, не можна вгадати, як ваша програма поведе себе при великій кількості користувачів. Якщо ваш код локально виконує поставлену задачу за 0.3 секунди, то при кількості людей вже 10-12 чоловік він може працювати вже 3 секунди.

По-друге, недбалий розробник може нехтувати фінальним стартом unit тестів, тим самим запровадивши помилку у програмі, яку невідомо коли вже можна буде відслідкувати. Треба пам’ятати, чим пізніше вдається знайти неправильний функціонал у програмі, тим дорожче буде його виправити.

Розглядаючи галузь веб-сервісної розробки, раніше та й зараз весь програмний код знаходився на одному сервері. Якщо потрібно збільшити його

потужність, то можна було запросити у хмарної платформи, такій як Amazon або Azure, просто потужніше залізо і все знову начебто добре. Але чим більше ми просимо, тим більше ми платимо і ріст цін зростає у геометричній прогресії. Можемо уявити, що гроші для нас не проблема, але з'являється ще одна проблема. Чим більше стає наша система, тим складніше її підтримувати. Наприклад, новому працівнику може знадобитися навіть місяць, щоб розібратися у цій системі, і не факт, що він зможе повністю її зрозуміти. А коли помилка виникне в деякій частині цієї системи, вона повністю зламається, що стане причиною збитків замовника.

Мікросервісна архітектура проекту дозволяє гнучко та надійно будувати нашу систему. Звісно, це складніше, чим писати за монолітною архітектурою, але як не дивно, ускладнюючи собі життя, ми тим самим і спрощуємо його. Яким чином?

По-перше, виникнення збою на одному з серверів, ніяк не вплине на роботу інших серверів. Наприклад, користувач, продивляючись каталог товарів, може, навіть не помітити, що функціонал стосовно його замовлення поки не доступний.

По-друге, команду кожного з серверів можна назвати "Two pizza". Чому саме так? Саме 2 піц повинно бути достатньо для обіду команди. Чим менша команда, тим скоріше може пристосуватися до проекту новий співробітник.

1.2 Dev-Ops як метод автоматизації процесів розробки

Dev-Ops – процес, який призначений для пришвидшення взаємодії розробників із системними адміністраторами.

Завдання DevOps полягає в узгодженні розробки й постачання програмного забезпечення із його використанням. Це завдання часто вирішується за допомогою автоматичних засобів.

Потреба в DevOps зросла через прагнення організацій готувати випуски частіше й швидше. [3]

DevOps — це командна робота (між співробітниками, що займаються розробкою, виробництвом і перевіркою якості), немає єдиного інструменту «DevOps»: це скоріше добірка (або «інструментальний ланцюжок DevOps») декількох інструментів. Як правило, інструменти DevOps вписуються в одну або декілька з цих категорій, що відображує ключові аспекти розробки і доставки програмного забезпечення:

Code — розробка та аналіз коду, інструменти контролю версій, злиття коду;

Build — процес збирання проекту;

Test — інструменти безперервного тестування, що повідомляють про функціональні недоліки, баги у програмі;

Спакування — репозиторій артефактів, підготовка програми для випуску;

Release — офіційне затвердження випуску, встановлення версій, автоматизація виробництва;

Моніторинг — відстеження продуктивності застосунків;

Попри наявність багатьох інструментів, деякі їх різновиди мають вкрай важливе значення для налаштування інструментальних засобів DevOps

Одним із цих інструментів і є CI / CD [4].

1.3 CI / CD як практика розробки програмного забезпечення

1.3.1 Неперервна інтеграція (англ. Continuous Integration)

Практика розробки програмного забезпечення, яка полягає у виконанні частих автоматизованих складань проекту для якнайшвидшого виявлення та вирішення інтеграційних проблем. У звичайному проекті, де над різними частинами системи розробники працюють незалежно, стадія інтеграції є завершальною. Вона може непередбачувано затримати закінчення робіт. Перехід до неперервної (постійної) інтеграції дозволяє знизити трудомісткість інтеграції і зробити її передбачуванішою за рахунок найбільш раннього виявлення та усунення помилок і суперечностей.

Вимоги до проекту:

- Початковий код і все, що необхідно для побудови та тестування проекту, зберігається в репозиторії системи керування версіями
- Операції копіювання з репозиторію, складання та тестування всього проекту автоматизовані і легко викликаються із зовнішньої програми.

Організація:

На виділеному сервері організовується служба, до завдань якої входять:

- Отримання початкового коду з репозиторію;
- Складання проекту;
- Виконання тестів;
- Розгортання готового проекту;
- Відправлення звітів.

Переваги:

- Проблеми інтеграції виявляються і виправляються швидше, що виявляється дешевше

- Негайний прогін модульних тестів для свіжих змін
- Постійна наявність поточної стабільної версії разом з продуктами складань — для тестування, демонстрації, тощо

Недоліки:

- Витрати на підтримку роботи безперервної інтеграції

Потенційна необхідність у виділеному сервері під потреби безперервної інтеграції [5].

1.3.2 Безперервне розгортання (англ. Continuous deployment)

Безперервне розгортання може розглядатися як продовження безперервної інтеграції, спрямоване на мінімізацію часу що минув між розробкою, наприклад, написання однієї нової рядки коду та цим новим кодом, який використовується вже користувачами, у виробництві.

Щоб досягти безперервного розгортання, команда спирається на інфраструктуру, яка автоматизує різні етапи розробки, що призводить до швидкого розгортання, так що після кожної інтеграції продукту, що успішно пройшла тестування та відповідає усім критеріям, програма, що знаходиться на стороні клієнтів оновлюється новим кодом.

Переваги:

1. Швидке додавання нового фукціоналу, після його розробки, що не потребує значних вкладень капіталу.
2. Попередні відгуки користувачів про кожну нову функцію після її випуску у виробництво, що забезпечує паралельне тестування, щоб визначити, яка з двох можливих реалізацій краща користувачам.

Потенційні витрати:

1. Безперервне розгортання розраховує на широкий інструментарій для того, щоб функція, щойно надана користувачам, не призводила до неприємних інцидентів, як порушення функціоналу програми
2. Безперервне розгортання покладається на інфраструктуру, яка дозволяє легко створювати резервні копії нових артефактів, коли дефект не був виявлений автоматизованими тестами [6].

2. Теоретичні засади розробки програмного забезпечення з застосуванням CI і CD

Існує 8 загальних фаз для успішних конвеєрів CI / CD

2.1 Plan

Потрібно мати стратегію для частого розгортання нових функцій, оновлень або виправлень у виробничому середовищі. Оскільки середовище виконання запускається на контейнерах Docker, можна скористатися перевагою контейнерізації та за рахунок неї впровадити постійну інтеграцію та безперервне розгортання (CI / CD) конвеєру. Конвеєр CI / CD допомагає вам автоматизувати процеси в життєвому циклі розвитку, починаючи з того, коли розробник перевіряє код, до розгортання коду у виробничому середовищі. Правильне настроювання конвеєру може допомогти частіше випускати оновлення, або з поступовими змінами, або з повними версіями.

Частина CI конвеєру полегшує упакування спеціального коду та створення спеціальних зображень Docker. Частина CD-конвеєра спрощує розгортання користувальницьких зображень у нових або існуючих середовищах.

Перед тим, як вирішити, які інструменти ви хочете застосувати, щоб полегшити роботу з CI / CD, переконайтесь, що ви досліджуєте та плануєте відповідно. Подумайте, яке програмне забезпечення найкраще працює для вашого бізнесу. Наприклад, ви можете захотіти реалізувати, використовуючи Jenkins, IBM Urban Code Deploy або комбінацію обох [7].

2.2 Code

2.2.1 Система контролю версій (VCS)

Організації, які пишуть будь-який тип коду, зазвичай використовують систему контролю версій вихідного коду (VCS), таку як Git. Типовим використанням VCS є те, щоб розробники писали код і зберігали його у сховищі VCS разом з будь-якими іншими важливими файлами. Це допомагає гарантувати, що кілька розробників, які працюють над проектом, не внесуть зміни, які неможливо вирішити.

VCS підтримує старі версії коду, щоб розробники могли бачити траєкторію своєї роботи, а також переходити до старих версій, якщо це необхідно. У двох словах, VCS зберігає код, впорядкований, організований та готовий до створення.

2.2.2 VCS у CI / CD

Легко подумати, що VCS це місце для збереження коду. Це суперечить важливості сучасної VCS. Це фактично важливий зв’язок між розробкою та розгортанням та ключовим компонентом контеєру безперервної інтеграції / безперервного розгортання (CI / CD). VCS зазвичай є місцем, де починається CI в CI / CD. VCS створює вихідний код запису та організовує його для складання.

Вибираючи продукт VCS, пам’ятайте про його роль в CI / CD. Система контролю версій повинен підтримувати дві функції, щоб переконатися, що вона виконує свою роль в CI / CD [10].

По-перше, для цього потрібно мати інструменти співпраці. Існують життєво важливі процеси в розробці коду, такі як огляд коду (code review), які є робочими процесами, що підключаються до загальних робочих процесів CI / CD. Ці спільні робочі процеси утворюють межу між розробкою та CI / CD. Вони допомагають підготувати базу коду до наступного кроку - збірки.

Одним з недоліків CI / CD є те, що немає жодного продукту, який би міг керувати всім pipeline (трубою). Неминуче командам DevOps доведеться

самостійно інтегрувати продукти. З цієї причини другою найважливішою особливістю VCS є надійний API. Без чітко розроблених API, інтеграція між VCS та іншими частинами ланцюжка інструментів CI / CD, наприклад, Jenkins, буде набагато складніше і схильна до помилок. В даний час більшість організацій очікують, що RESTful API, побудовані навколо HTTPS, і саме те, що постачає більшість продуктів. Саме сфера API найбільш важлива [9].

2.3 Build

Фаза збірки запускається, коли новий код потрапляє в репозиторій у системі контролю версій. Оскільки початковий код зберігаються в невеликих гілках сховища, компілятор збирає всі функції коду та його залежності, а потім компілює їх у нову збірку. Потім починається новий етап – запуск тестів [8].

2.4 Test

Ключовим фактором якості інтеграції в наше програмне забезпечення є забезпечення швидкого відгуку про вплив нових змін. Безперервне тестування допомагає зрозуміти швидкість та гнучкість, забезпечує швидше відновлення у випадку виявлення дефектів у виробі. Автоматизація тестування дає змогу швидко пройти тестування на ранніх тестах, а також надає більшій кількості зацікавлених сторін більш точну інформацію про ризик, допомагаючи приймати кращі рішення на рівні бізнесу. Для досягнення високої якості програмного забезпечення нам потрібно постійно проводити різні типи тестів у процесі доставки.

Тестування автоматизації важливе для успішного пайплайну безперервної інтеграції / безперервної доставки (CI / CD), оскільки:

- Проста і ефективна оцінка незначних змін - враховуючи, що в процесі безперервної інтеграції більшість змін є невеликими і автоматизація тестів здатна їх покрити, команда може постійно доставляти зміни, які вже були достатньо протестовані. Окрім автоматизованих тестів, деякі нечасті ручні тестувальні дії залишаються вирішальними після великого оновлення програмного забезпечення.
- Швидші тести регресії - безперервний процес доставки вимагає швидкого зворотного зв'язку. Тестування автоматизації може бути виконано набагато швидше, ніж ручне тестування, та швидко забезпечити результати. Щоб скоротити загальний час виконання, автоматичні тести можуть працювати паралельно.
- Краща спритність - з’являються нові технології та змінюються вимоги. Ось чому хороший конвеєр повинен бути гнучким для оновлення, коригування конфігурацій та впровадження нових інструментів. Занадто багато ручного тестування робить конвеєр CI / CD менш гнучким.

2.4.1 Unit testing

Цей тип тесту перевіряє функціонування окремих блоків коду. Зазвичай ці тести зосереджені на місцях, де можуть ховатися помилки, та на окремих функціях. Блокові тести повинні бути дуже швидкими і охоплювати близько 80% кодової бази. Це може дати достатню впевненість, що програма на даний момент працює належним чином. Якщо одиничні тести не будуть автоматизовані та незалежними один від одного, шансів досягти успіху не буде.

2.4.2 Component Integration Testing

Інтеграційні тести виконуються для виявлення дефектів у взаємодії між інтегрованими компонентами або системами.

2.4.3 User Interface (UI) Testing

Тестування користувальницьким інтерфейсом перевіряєчи відображаються інтерфейси інтерфейсу програми та працюють, як було описано в вимогах, у всіх підтримуваних браузерах та платформах. Частіше за все для цього використовуються віртуальні машини із системою розпаралелювання.

2.4.4 API (Middleware) Testing

Тести API перевіряють сумісність між програмними системами та компонентами. Ці тести імітують виклики до кінцевих точок (endpoints) API та перевіряють відповіді з сервера [11].

2.4 Release

Після того як наш проект зібрався та пройшов усі етапи тестування, можемо починати підготування нашого артефакту до майбутнього деплою на середовище. Саме це підготування і називається Release.

Спершу створюється файл під назвою version.json. Цей файл буде містити всю інформацію о версії застосунку, о даті закінчення збірки, проведення тестів та створення артефакту. Також зазвичай в ньому міститься хеш комміта, після якого був запущений pipeline.

Така проста дія може допомогти членам команди (в першу чергу тестувальника) швидко знайти версію релізу нашого застосунку, яка вже запущена в даному середовищі, та далі відслідковувати майбутні деплої.

2.5 Deploy

Deploy це процес доставки нашого артефакту до середовища, де вже працює або буде працювати наш застосунок. На цьому етапі ми маємо можливість помістити наш артефакт на один, або 2 а то і більше середовищ. Це дуже корисно, коли ми одночасно оновлюємо застосунок у середовищі клієнта, та у тестувальному середовищі. Робиться це задля того, щоб розробники не мали доступ до даних, якими маніпулює клієнт, але тим самим мали можливість відлагоджувати некоректний функціонал. Також на цьому етапі, ми маємо можливість вибрати найбільш стабільний артефакт, знайшовши його за релізом. Це корисно в тих випадках, коли ми впевнені, що випускаємо вже готовий продукт для нашого замовника. На сьогоднішній день CD має два значення – Continuous Deployment та Continuous Delivery.

Continuous Deployment це автоматична доставка нашого артефакту після його релізу, до прописаних у файлі конфігурацій середовищ.

Continuous Delivery потребує підтвердження зі сторони розробника, про дозвіл деплою артефакту, а також дає можливість продивитися звітність стосовно його релізу та збірки.

Яку з цих концепцій використовувати залежить від розробників. Якщо ти повністю впевнений у своєму коді та його стабільності, то треба вибирати Continuous Deployment. Якщо ж розробник бажає мати повний контроль над доставкою артефакту, та піклується про свій застосунок, перевіряючи всі свої дії декілька разів, то Continuous Delivery його варіант.

2.6 Operate

Фаза Operate наступає після того, як наш артефакт буде створений у реальному часі. Ця фаза складається с моніторингу на створення Docker або Cubernetes контейнерів.

2.7 Monitoring

Один з найважливіших фаз CI/CD. Після того як наш артефакт потрапляє у середовище розробники мають можливість переглянути як веде себе оновлена система. Вже на цьому етапі можна побачити як застосунок веде себе з даними, котрі вже максимально наближені до реальних, чи не сильно збільшується її навантаження при більшій кількості користувачів, переглянути як система оброблює ті чи інші помилки. Саме на цьому етапі ми маємо останню можливість знайти недоліки у функціоналі, перш ніж віддавати наш продукт до споживача. Monitoring також дає можливість відслідкувати взаємодію одного серверного застосунка з іншим.

Monitoring можна проводити підчас тестів, або ж просто під час функціонування програми. Monitoring під час тестів дає можливість відстежити хіба що 80% нашого функціоналу. В інших випадках ми можемо банально помилитися. Наприклад при введенні вхідних даних. Ми не можемо наперед

знати, якими даними буде маніпулювати майбутній користувач. Також ми можемо помилитися із тестування навантаженності нашого застосунку. Розробники розраховували, що користувачі будуть активно використовувати один функціонал, але навантаження вийшло на іншу частину системи. В таких випадках всі ці дефекти можна відслідкувати в режимі Real Time за допомогою включених інструментів на хмарних платформах.

3. Реалізація практичної частини

3.1 Опис застосунку та обґрунтування обраних інструментів

Для демонстрації CI/CD можна створити будь-який застосунок, який може бути написаний на будь-якій мові, яку підтримує pipeline у вибраній нам платформі. На мою думку для демонстрації, в якості застонку підходить клієнтський або ж серверний застосунок, оскільки можна відразу після деплою артефакту побачити зміни у проекті.

Для реалізації CI / CD було розроблено сервісний застосунок з наступний функціоналом:

1. CRUD операції з товарами на замовленнями.
2. Аутентифікація користувача на основі JWT token з використанням implicit flow.

Щоб продемонструвати CI/CD у якості системи контролю версій було вибрано – GitHub, оскільки він зручний у використанні, а також особливо не відрізняється фукціоналом від інших систем керування версіями.

Щоб розгорнути застосунок нам потрібна хмарна платформа. На сьогоднішній день існує достатньо велика кількість цих платформ, наприклад: Amazon Web Services, Google Cloud Platform, Microsoft Azure і т.д. Усі ці платформи пропонують майже одні й ті самі сервіси для різних проблем і відрізняються вони хіба що мовами програмування, за допомогою яких і були

написані ці сервіси. Це стосується і pipelines, коли у різних платформах вони відрізняються хіба що назвами, так як в AWS (Amazon Web Services) це AWS CodePipelines, а в Azure це Azure Pipelines. Для розгортання застосунку було обрано саме Microsoft Azure, оскільки вона займає перше-друге місце за популярністю, а також на мою думку має один з найзручніших юзер інтерфейсів серед інших хмарних платформ.

Pipelines будуть створюватися у Azure Pipelines, Bitbucket Pipelines.

Оскільки в кінцевому результаті наш код буде запускатися за допомогою контейнерів Docker або Kubernetes, або ж за допомогою віртуальних машин, то можна сказати, що сервіси типу pipelines – кросплатформні, отже можна вибрати будь-яку мова, яку підтримує вибрана нами платформа. Для розробки веб-сервісу була вибрана мова C#, для тестів – бібліотека XUnit.

3.2 Створення Pipelines

3.2.1 Azure Pipelines

Для створення першого застосунку у платформі Azure, необхідно обов'язково пройти реєстрацію із додаванням платіжної інформації, оскільки багато сервісів платні. Після реєстрації ми маємо можливість перейти в платформу Azure Portal (https://portal.azure.com/) і створити поки порожній, свій власний App Service.

Щоб створити свій перший Pipeline та CI потрібно перейти до іншого застосунку Microsoft – Azure DevOps (https://dev.azure.com/). Створюючи pipeline ми повинні обов’язково надати доступ до нашого аккаунту, а також репозиторію у GitHub, також вказавши гілку, де після кожного нового комміту буде створюватися новий артефакт.

Після створення нашого Pipeline у нашій системі контролю версій з’явиться новий файл - azure-pipelines.yml. В цьому файлі ми прописуємо всі операції, які хочемо провести перед створення та деплоєм артефакту.

Один з етапів CI – Build.

```
- task: DotNetCoreCLI@2
    displayName: Build
    inputs:
      command: build
      projects: '**/*.csproj'
      arguments: '--configuration Release'
```

Якщо проект не вдасться зібрати, то артефакт не буде створено.

Налаштування міграції бази даних

```
- task: DotNetCoreCLI@2
    displayName: Migrations
    inputs:
      command: dotnet ef database update
      projects: 'ProductsAndOrders.csproj'
```

Наступний етап – прогін юніт тестів

```
- task: DotNetCoreCLI@2
    displayName: Test
    inputs:
      command: test
      projects: '**/*Tests/*.csproj'
```

Деплой артефакту буде настроєно через Deployment Center у нашому застосунку в Azure.

3.2.2 Bitbucket Pipelines

Спочатку необхідно створити перший project у цій платформі, де будуть зберігатися наші репозиторії. Далі можна або запушити наш код у цей репозиторій, або ж склонувати вже існуючий. Я скористався вже існуючим репозиторієм, що знаходиться в GitHub. Після цього, створюємо наш перший pipeline за допомогою інтерфейсу. Після закінчення всіх кроків, у нашому новому репозиторії з'явиться файл з назвою bitbucket-pipelines.yml. Тут буде налаштування всього процесу.

З початку файла потрібно визначити цільову версію проекту, а також крок для виконання скриптів.

```
image: microsoft/dotnet:2.2-sdk

pipelines:
  default:
    - step:
        name: "Build and Deploy"
```

Наступний крок, виконання перших скриптів: налаштування версій, а також оновлення пакетів та істалювання zip-пакетів, для пакування артефакту.

```
script:
        - export VERSION=1.$BITBUCKET_BUILD_NUMBER
        - export ZIP_FILE=$PROJECT_NAME.$VERSION.zip
        - apt-get update
        - apt-get install zip -qq
        - export PROJECT_NAME="ProductsAndOrders"
        - export TEST_NAME="ProductsAndOrders"
```

Далі необхідно встановити усі необхідні пакети для проекту, а також виконати його збірку та прогін тестів.

```
          - dotnet restore
        - dotnet build $ProductsAndOrders
        - dotnet test $ProductsAndOrders
```

Наступний крок – скрипт міграції бази даних.

Пакування проекту, створення артефакту.

Останній крок деплой артефакту до оточення на Azure.

```
- pipe: microsoft/azure-web-apps-deploy:1.0.2
    variables:
      AZURE_APP_ID: "761280ff-d337-4b8e-bae4-1da9eaecf38d"
      AZURE_PASSWORD: "5604f44c-d7bd-4b5c-8867-2cba766fcd39"
      AZURE_TENANT_ID: "b8cbfe43-c90c-4bea-84ae-be5d6d8a5f52"
      AZURE_RESOURCE_GROUP: 'Test'
      AZURE_APP_NAME: 'test-blahov'
      ZIP_FILE: $ZIP_FILE
```

3.2.3 GitLab Pipelines

Спочатку необхідно створити репозиторій із проектом, який буде розвертати в майбутньому на хмарі. Після цього необхідно додати файл з назвою .gitlab-ci. При кожній зміні коду в репозиторії, GitLab Pipelines будуть запускатися, знаходячи цей файл. В .gitlab-ci потрібно прописати скрипти, стосовно поведінки pipelines.

Спочатку необхідно вказати який тип контейнеру ми використовуємо, для проведення всіх операцій.

```
image: microsoft/dotnet:2.2-sdk
```

Далі, які кроки ми робимо

```
stages:
  - build
  - test
  - migrations
```

Також, перед виконанням усіх скриптів, необхідно скачати усі сторонні бібліотеки, які необхідні проекту.

```
before_script:
  - 'dotnet restore'
```

Виконання збірки проекту

```
build:
 stage: build
 script:
  - dotnet build
```

Виконання тестів

```
test:
 stage: test
 script:
  - dotnet test
```

Накатуємо міграції на нашу базу даних

```
migrations:
  stage: migrations
  script:
    - "dotnet ef database update --project ProductsAndOrders/ProductsAndOrders.csproj --
startup-project ProductsAndOrders/ProductsAndOrders.csproj"
```

Після вдалого виконання усіх кроків, за допомогою Deployment Center, можна відправити наш відтестований та зібраний проект до середовища його виконання.

3.2.4 Jenkins Pipelines

Jenkins відрізняється від всіх інших застосунків, оскільки він встановлюється і працює локально. Для завантаження потрібно перейти за лінком https://www.jenkins.io/download/ та завантажити версію згідно з операційною системою. Після цього створити адміністратора та користувача. На основній сторінці знаходиться кнопка new item. Там ми можемо створити свій pipeline. Вибравши там розділ Pipelines, потрібно в налаштуваннях вказати посилання на гіт репозиторій з паролем та назвою аккаунту для доступу. Обов’язково, в налаштуваннях Jenkins, в розділі Git повинен бути правильно заданий шлях до файлу git.exe. Далі, створюємо файл з назвою Jenkisfile, розміщуємо його в репозиторії, який вказали під час налаштування.

Структура файлу має наступний вигляд. На початку файла визначаємо тип середовища де буде виконаний pipeline

```
pipeline{
    agent any

    environment {
        dotnet ='C:\\Program Files (x86)\\dotnet\\'
    }
```

Завантажуємо потрібні для проекту пакети

```
stages {
        stage('Restore packages'){
            steps{
                bat "dotnet restore"
            }
        }
```

Виконання збірки проекту

```
stage('Build'){
            steps{
                bat "dotnet build"
            }
        }
```

Виконання тестів

```
stage('Test'){
            steps{
                bat "dotnet test"
            }
        }
```

Висновки:

Azure pipelines має простий та зрозумілий інтерфейс, багато документації, а також прикладів з інтернету, що полегшує вивчення цієї платформи та роботи з нею. Bitbucket також проявив себе, як гарна платформа. Вже з самого початку він відразу створює потрібний користувачу файл і пропонує безліч готових шаблонів для реалізації pipeline. Прикладів для реалізації на Bitbucket в інтернеті не менше ніж в документації, що також полегшує ознайомлення з ним. GitLab виявився складнішим з опануванням оскільки він пропонує обмежено документацію стосовно його використання, а також обмежена інформація в інтернеті. Також GitLab, при кожному запуску pipilines, змінює свою API адресу, що дуже сильно ускладнює міграції бази даних, коли для доступу до них треба вказувати API адреси, котрим ви довіряєте. Jenkins відрізняється від всіх платформ тим, що він запускається локально, а також потребує багато попередніх налаштувань для майбутньої роботи з ним, що ускладнює роботу з ним. Спільне у всіх цих платформ, що вони мають схожий синтаксис для написання скриптів.

Список джерел

1. Введення в кращі практики CI / CD [Електронний ресурс]. — Режим доступу: https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices-ru
2. What is DevOps [Електронний ресурс]. — Режим доступу: https://hackernoon.com/what-is-devops-and-why-i-should-have-it- e60f1ee446d2
3. Learning DevOps. Автор - Mikael Krief. Рік видання - 2019
4. DevOps – Вікіпедія[Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/DevOps
5. Continuous-Integration – Вікіпедія[Електронний ресурс]. — Режим доступу: https://uk.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BF%D0%B5%D1%80%D0%B5%D1%80%D0%B2%D0%BD%D0%B0_%D1%96%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D1%96%D1%8F
6. Continuous-Deployment [Електронний ресурс]. — Режим доступу: https://www.agilealliance.org/glossary/continuous-deployment
7. Planning a CI/CD pipeline [Електронний ресурс]. — Режим доступу: https://help.hcltechsw.com/commerce/9.0.0/install/concepts/ciginstallcicd.html
8. CI/CD Pipeline [Електронний ресурс]. — Режим доступу: https://www.katalon.com/resources-center/blog/ci-cd-pipeline/
9. Version Control Systems [Електронний ресурс]. — Режим доступу: https://www.cmswire.com/information-management/version-control-systems-the-link-between-development-and-deployment
10. Mastering Jenkins. Second Edition: Unleash the full potential of Jenkins to create advanced software delivery pipelines. Автор - Nikhil Pathania. Рік видання - 2019
11. Test automation in CI/CD [Електронний ресурс]. — Режим доступу: https://www.spritecloud.com/test-automation-with-ci-cd-pipeline