НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ
імені адмірала Макарова

Навчально-науковий інститут автоматики та електротехніки
(повне найменування інституту, факультету)

Кафедра комп'ютеризованих систем управління
(повна назва кафедри)

«Допущений до захисту»
Завідувач кафедри
Черно О.О.
«__» _____ 2022 р.

Пояснювальна записка

до кваліфікаційної роботи

на здобуття першого (бакалаврського) рівня вищої освіти

на тему: **Автоматизація процесів розробки та тестування програмного забезпечення на основі CI/CD практик**

Виконав студент **4** курсу, **4341** групи спеціальності
«151 - Автоматизація та комп'ютерно-інтегровані технології»
(шифр і назва напряму підготовки)
Черній М.О.
(прізвище та ініціали)
Керівник **Герасін О.С.**
(прізвище та ініціали)
Рецензент **Гаврилов С.О.**
(прізвище та ініціали)

м. Миколаїв – 2022 року

Національний університет кораблебудування імені адмірала Макарова

Інститут	автоматики і електротехніки
Кафедра	комп'ютеризованих систем управління
Освітньо-кваліфікаційний рівень	**«бакалавр»**
Спеціальність	**151 «Автоматизація та комп'ютерно-інтегровані технології»**
Освітня програма	**«Комп'ютеризовані системи управління та автоматика»**

ЗАТВЕРДЖУЮ
Завідувач кафедри КСУ

_____ Черно О.О.

«____» _____ 2022 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Чернiй Максим Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи: *Автоматизація процесів розробки та тестування програмного забезпечення на основі CI/CD практик*

керівник роботи Герасін Олександр Сергійович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "04" травня 2022 р. №254

2. Строк подання студентом роботи *20.06.2022*

3. Вихідні дані до роботи *Створити універсальний фреймворк для програмного автоматизації процесів розробки та тестування забезпечення (ПЗ), Прискорити тестування ПЗ мінімум у 2 рази*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз проблематики автоматизованого тестування та розробки програмного забезпечення (ПЗ)
Програмні засоби автоматизації процесів розробки та тестування ПЗ
Розробка та моделювання системи для автоматизації розробки та тестування ПЗ
Охорона праці

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

Використання гнучких технологій в розробці та тествуванні

Життєвий цикл розробки і тестування ПЗ

DevOps методологія. Практики непреривної інтеграції та розгортування

Програмні засоби автоматизації процесів розробки та тестування ПЗ

Моделювання системи для автоматизації розробки та тестування ПЗ

Структура фреймворку модельованої системи

Алгоритм роботи скрипта environment.py

Алгоритм роботи скрипта setup_environment.py

Алгоритми do_register_new_client do_add_data_center

Структура Gradle скриптів

Алгоритм роботи preparation.sh

Алгоритм роботи pull_services.sh

Структура Jenkins Pipelines

Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Охорона праці	*Герасін О.С., доцент каф. КСУ*	*17.05.2022*	*31.05.2022*

7. Дата видачі завдання: «17» січня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів підготовки кваліфікаційної роботи	Строк виконання	Примітка
	Аналіз проблематики автоматизованого тестування та розробки програмного забезпечення (ПЗ)	24.03.22	виконано
	Програмні засоби автоматизації процесів розробки та тестування ПЗ	08.04.22	виконано
	Розробка та моделювання системи для автоматизації розробки та тестування ПЗ	11.05.22	виконано
	Охорона праці	31.05.22	виконано

Студент _____ Черній М.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Герасін О.С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Метою даної дипломної роботи є розробка програмних засобів автоматизації процесів розробки та тестування програмного забезпечення на основі CI/CD практик.

В ході роботи проведено аналіз існуючих систем автоматізації, а також засобів та методів найбільш поширених для вирішення проблем бізнесу. Наведено структуру та програмні засоби реалізації мультифункціональної системи автоматизації для IT проекту.

В результаті проведено моделювання системи на базі практик постійної інтеграції та розгортання програмного забезпечення, а також здійснено порівняльний аналіз ефективності автоматизації на проекті. Також, в роботі розглянуті питання охорони праці, а саме: аналіз небезпечних і шкідливих факторів для здоров'я інженера при розробці системи автоматиції в розрізі сучнах методик ведення розробки та використання гнучких технологій ведення проекту; можливі заходи для забезпечення підвищення безпечності робіт.

Обсяг кваліфікаційної роботи становить 82 сторінки, робота містить 59 рисунків та 9 використаних джерел.

Ключові слова: Agile, автоматизація процесів, DevOps, CI/CD.

ABSTRACT

The purpose of this thesis is to develop software to automate the development and testing of software based on CI / CD practices.

In the course of the work the analysis of the existing automation systems, and also means and methods most widespread for the decision of problems of business is carried out. The structure and software for the implementation of a multifunctional automation system for an IT project are presented.

As a result, the system was modeled based on the practices of constant integration and software deployment, as well as a comparative analysis of the effectiveness of automation on the project. Also, the issues of labor protection are considered in the work, namely: analysis of dangerous and harmful factors for the health of the engineer in the development of automation system in terms of current methods of development and use of flexible technologies for project management; possible measures to ensure the safety of work.

The volume of qualifying work is 82 pages, the work contains 59 figures and 9 sources used.

Keywords: Agile, process automation, DevOps, CI / CD.

ЗМІСТ

ВСТУП

Завдання, що вимагають автоматизувати процеси в рамках IT проекту, виключно різноманітні і зустрічаються в різних областях відповідальності.

Автоматизація розробки та тестування - необхідна в випадках, коли розробка має обмеження за часом, за підтримки великих чи заснованих на гнучких методиках проектів.

Актуальність в автоматизації процесів зростає в міру збільшення команди, функціональностей, над якими працюють як інженери з розробки ПЗ, так і з тестування та складності підготовки продукту до випуску. До характерних завдань належать: створення та підтримка оточення для інтеграції та розсортування продукту; автоматизація процесу потрапляння версії продукту до потрібного оточення; автоматизація процесу тестування перед випуском версії продукту та під час її розробки.

На сьогоднішній день актуальною є задача створення оточення та програмних засобів для автоматизації процесів під час розробки програмного забезпечення. Орієнтація такої системи посилює швидкість та надійність в розробці та тестуванні програмного забезпечення.

					151.4341.КР.ПЗ.Р1		
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Черній М.О.*			АНАЛІЗ ПРОБЛЕМАТИКИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ТА РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ (ПЗ)	*Літ.*	*Арк* 8 / *Аркушів* 16
						НУК ім. адм.. Макарова	
Норм. контр		*Вінниченко І.Л.*					
Керівник		*Герасін О.С.*					
Зав. Каф.		*Черно О.О.*					

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ТА РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ (ПЗ)

Коли мова заходить про програмне забезпечення та автоматизацію процесів, при розробці та тестуванні – насамперед – це інтереси та проблеми бізнесу, які потрібно вирішувати швидко та ефективно. Потрібен певний підхід, який гарантуватиме, що проект автоматизації буде інтегрований у більш широкий бізнес, та буде відповідати нормативним вимогам та іншим обмеженням. Найкраще на цю роль підходить використання Agile методології.

1.1 Актуальність гнучких технологій з розробки та тестування

Agile - це ітеративний підхід до управління проектами та розробки програмного забезпечення, який допомагає командам швидше і з меншими труднощами приносити користь своїм клієнтам. Замість того, щоб робити ставку на великий вибух, agile-команда виконує роботу невеликими, але видатковими частинами. Вимоги, плани та результати постійно оцінюються, тому команди мають природний механізм швидкого реагування на зміни. [1]

У той час, як при традиційному «водоспадному» підході одна дисципліна робить свій внесок у проект, а потім «перекидає його через стіну» наступному учаснику, agile вимагає спільних крос-функціональних команд. Відкрите спілкування, співпраця, адаптація та довіра між членами команди лежать в основі Agile. Хоча керівник проекту, або власник продукту, зазвичай, розставляє пріоритети у роботі, яка має бути виконана, команда бере на себе ініціативу у вирішенні того, як виконуватиметься робота, самоорганізовуючись навколо окремих завдань та призначень.

Agile не визначається набором церемоній чи конкретними методами розробки. Швидше, Agile - це група методологій, що демонструють відданість жорстким циклам зворотного зв'язку та постійному вдосконаленню.

Початковий Agile-маніфест не наказував двотижневі ітерації або ідеальний розмір команди. Він просто виклав набір основних цінностей, які ставлять людей

на перше місце. Те, як команда житимете з цими цінностями сьогодні - чи буде виконуватися Scrum за правилами або змішувати елементи Kanban і XP - повністю залежить від інженера.

Команди вибирають Agile, щоб вони могли швидко реагувати на зміни на ринку чи відгуки клієнтів, не порушуючи річний план. Планування «достатньо» та постачання невеликими частинами дозволяють команді збирати відгуки про кожну зміну та інтегрувати їх у майбутні плани з мінімальними витратами.

Як описано в Маніфесті Agile, справжня людська взаємодія важливіша за жорсткі процеси. Співпраця з клієнтами та товаришами за командою важливіша за заздалегідь певні домовленості. І надання працюючого вирішення проблеми клієнта важливіше за наддетальну документацію. [1]

Agile-команда об'єднується під загальним баченням, та втілює їх у життя, оскільки вважає за потрібне. Кожна команда встановлює власні стандарти якості, зручності використання та повноти. Потім їх «визначення готовності» інформує про те, як швидко вони виконуватимуть роботу. Хоча спочатку це може бути незвичним, керівники компаній виявляють, що коли вони довіряють agile-команді, ця команда відчуває більше почуття причетності і піднімається, щоб відповідати (або перевершувати) очікуванням керівництва.

Публікація Agile Manifesto в 2001 знаменує собою народження Agile як методології. З того часу з'явилося безліч гнучких фреймворків, таких як Scrum, Kanban, Lean та Extreme Programming (XP). Кожен по-своєму втілює основні принципи частого повторення, безперервного навчання та високої якості. Scrum і XP віддають перевагу командам розробників програмного забезпечення, у той час як Kanban — улюбленець команд, орієнтованих на обслуговування, таких як IT або відділ кадрів. [2]

Сьогодні багато agile-команд комбінують практики з декількох різних фреймворків, заправляючи їх унікальними для команди практиками. Деякі команди приймають деякі agile-ритуали (такі як регулярні стендапи, ретроспективи, беклоги тощо), тоді як інші створили нову agile-практику (agile-маркетингові команди, які дотримуються Agile Marketing Manifesto).

Agile-команди майбутнього цінуватимуть свою ефективність вищеусталеним стереотипам про розробку та тестування. Відкритість, тісна інтеграція між усіма членами команди і їх деяка автономність від безпосереднього начальства - будуть візитною карткою компанії, які хочуть залучати найкращих людей та отримувати від них максимальну віддачу. Такі компанії вже доводять, що методи можуть різнитися у різних командах, якщо вони керуються правильними принципами.

1.2 Життєвий цикл розробки і тестування ПЗ

Методологія Software Development Lifecycle Phases (SDLC) заснована на спільному прийнятті рішень між командами та циклічному, ітеративному процесі створення працюючого програмного забезпечення. Робота виконується спринтами, що регулярно повторюються, які зазвичай тривають кілька тижнів.

Це помітно контрастує із тимчасовими рамками традиційного управління проектами, які іноді розтягуються на невизначений термін.

Модель SDLC наголошує на адаптованість процесів і задоволеність клієнтів за рахунок швидкої доставки працюючого програмного продукту.

Методи Agile розбивають продукт на невеликі додаткові збирання, що надаються в ітераціях. Наприкінці ітерації, розробники показують замовнику та іншим зацікавленим сторонам робочий продукт, що містить весь функціонал замовника.

Кожна ітерація, зазвичай, триває від одного до трьох тижнів і включає:

- Планування;
- Аналіз;
- Архітектурний дизайн;
- Кодування;
- Модульне тестування;
- Приймальне тестування;
- Доставка;
- Зворотній зв'язок.

Кроки повторюються, поки всі елементи в цілому продукту не будуть завершені. Процес швидше циклічний, ніж лінійний.

Згідно з моделлю Agile, кожен проект обробляється по-різному, а існуючі методи адаптуються, щоб найкращим чином відповідати вимогам проекту. Відбувається зосередження на більш можливих цілях, які визначають негайні дії вашої команди.[1]

Основні принципи гнучкої розробки

Коли 17 розробників програмного забезпечення вигадали новий спосіб розробки програмного забезпечення майже 20 років тому, вони не знали, як швидко їхні ідеї поширюється за межі їхньої галузі. Ці принципи поступового розвитку зробили модель Agile такою, якою вона є сьогодні.

Головним пріоритетом є задоволення потреб клієнтів за рахунок своєчасного та безперервного постачання цінного програмного забезпечення.
Зміни у гнучких процесах використовуються для створення конкурентної переваги для клієнтів.

Зміна вимог можлива навіть на пізнішому етапі розробки.

Працююче програмне забезпечення доставляється від кількох тижнів до декількох місяців, переважно більш короткі терміни. А це є головним показником прогресу.

Роботодавці та розробники мають бути на зв'язку щодня, протягом усього проекту. Мотивовані люди створюють проекти у правильному середовищі та за належної підтримки. Самоорганізовані команди створюють найкращі вимоги, архітектури та проекти. Найбільш ефективним методом передачі інформації команді розробників і в ній є спілкування віч-на-віч.

Команда повинна регулярно переглядати свої показники продуктивності, відповідним чином налаштовувати та коригувати свою продуктивність. Гнучкі процеси сприяють сталому розвитку. Інвестори, розробники та користувачі повинні рухатися у постійному темпі. Невпинна увага до технічної досконалості та бездоганного дизайну підвищують маневреність. Простота - мистецтво максимізувати обсяг незавершеної роботи – має важливе значення.

Ключові етапи життєвого циклу гнучкої розробки програмного забезпечення.

Життєвий цикл Agile [1] - це структурована серія етапів, якими проходить продукт. Він складається із шести фаз:

Вимоги. Зацікавлені сторони проводять загальну оцінку проекту, щоб визначити час та ресурси, необхідні для процесу розробки. На цьому ж етапі власник оцінює ризики, розставляє пріоритети для різних функцій залежно від їхньої цінності для бізнесу.

Дизайн. Власник програмного забезпечення зустрічається з командою розробників та знайомить їх із вимогами, викладеними на першому етапі. Потім група обговорює послідовність введення функцій та визначає основні інструменти – мову програмування, синтаксичні бібліотеки та основні фреймворки. На цьому ж етапі команди розробників програмного забезпечення можуть створити прототип очікуваного інтерфейсу користувача.

Розробка. Після узгодження плану із замовником, команда розробляє сам продукт. Продукт поставляється поетапно, окремими спринтами, кожен з яких призначений для поліпшення поточної версії продукту. Початковий випуск, ймовірно, зазнає безліч змін, щоб забезпечити покращену функціональність та нові функції. Кожен цикл включає тестування, і кінцевий продукт також повинен пройти фінальне тестування. На цьому етапі можна використовувати методологію Scrum і Kanban, процес розробки на основі окремих завдань.

Інтеграція та тестування. У цей момент, продукт стає доступним для споживачів, тому команда повинна провести серію тестів, щоб переконатися, що програмне забезпечення є повністю функціональним. Якщо буде виявлено потенційні помилки чи недоліки, розробники негайно їх виправлять. На цьому етапі вони також збирають відгуки споживачів.

Впровадження та розгортання. Тепер, програмне забезпечення повністю розгорнуте та доступне клієнтам. Ця дія переводить їх у фазу обслуговування. На

цьому етапі, команда розробників проекту забезпечує постійну підтримку, щоб надати безперебійну роботу системи та виправити будь-які нові помилки. Згодом, можливі подальші ітерації для оновлення наявного продукту або додавання інших функцій.

Огляд. Це останній етап циклу розробки Agile. Після завершення всіх попередніх етапів розробки, команда розробників представляє власнику результат, досягнутий під час виконання вимог. Після цього етапи Agile-розробки програмного забезпечення починаються заново - або з новою ітерацією, або з переходом на наступний етап та масштабування Agile.

Software Testing Life Cycle (STLC)

Життєвий цикл тестування програмного забезпечення (STLC) є послідовністю певних дій, що виконуються в процесі тестування для забезпечення досягнення цілей якості програмного забезпечення. STLC включає як верифікацію, так і валідацію. Попри поширену думку, тестування програмного забезпечення – це не просто окрема діяльність. Воно складається з низки методологічних процесів, які допомагають сертифікувати програмний продукт.

STLC - це високоякісна стратегія, безпосередньо пов'язана з життєвим циклом розробки програмного забезпечення (SDLC) і є його частиною, який, у свою чергу, є структурою, заснованою на 6 основних принципах:

- Аналіз вимог;
- Планування випробувань;
- Розробка тестового прикладу;
- Налаштування тестового середовища;
- Виконання тесту;
- Закриття циклу тестування.

Аналіз вимог. Один із найважливіших етапів, тому що саме на ньому можна практично безкоштовно виправити недоліки проекту. Етап аналізу вимог також виявляє потенційну потребу в автоматизованому тестуванні та дозволяє проводити економічні розрахунки трудовитрат на основі оцінки проекту. Це також час, коли критерії входу та критерії виходу обговорюються та

документуються.

Планування тестування. На цьому етапі формується план тестування. Це конкретизація всіх етапів самого тестування, термінів, учасників та обов'язків. В результаті цього ми отримуємо данні про:

- Учасники та їх ролі у тестуванні;
- Необхідні інструменти тестування;
- Необхідне тестове середовище.

Розробка тестів. Передбачає використання ручного та автоматизованого тестування для досягнення повного охоплення функціональності програмного забезпечення, при цьому процес ґрунтується на заздалегідь встановлених вимогах. Найчастіше тест-кейси для автоматизованого тестування пишуться окремо, оскільки кейси для ручного тестування описуються як замітки.

Налаштування тестового середовища. У плані тестування ясно вказується, яке тестове середовище слід використовувати. На цьому етапі STLC налаштовуються операційні системи та віртуальні машини, розгортаються інструменти тестування, такі як Selenium, Katalon Studio, а також тестове середовище та бази даних проекту. Також, потрібні запити до DevOps та адміністраторів, якщо потрібна підтримка.

Виконання тесту. Тести виконуються на основі готової тестової документації та правильно налаштованого тестового середовища. Усі результати тестування заносяться до системи управління тестуванням. Негативно пройдені тести, де фактичний результат відрізняється від очікуваного, фіксуються як помилки та передаються команді розробників на доопрацювання з подальшою перевіркою після виправлення.

Закриття циклу тестування. Останнім етапом STLC є створення звітів про тестування для клієнта. Вони повинні включати: витрачений час, відсоток знайдених помилок до позитивних результатів тесту, загальну кількість знайдених та виправлених помилок. Що стосується відділу тестування, то це момент для аналізу його роботи, підбиття підсумків, аналізу його продуктивності та можливості внести пропозиції щодо покращення якості тестування.

Знаючи, що STLC є комплексом заходів, можна припустити, що він включає різні етапи, такі як планування, контроль, впровадження, стандартизація і так далі.

Все це призводить до того, що STLC потрібен не тільки для випробувань продукту, що розробляється, але і для наступного:

- Усунення його недоліків на ранній і найбільш вигідний стадії розвитку;
- Підвищення якості та прозорості процесу розробки;
- Максимальний контроль якості продукту, що розробляється на всіх стадіях SDLC;
- Вплив на застосування Agile, Scrum, SAFe тощо;
- Надання якісного продукту не лише Клієнту, а й Користувачам.

Роль STLC у SDLC

Як згадувалося раніше, життєвий цикл тестування програмного забезпечення та життєвий цикл розробки програмного забезпечення тісно пов'язані один з одним, але одночасно переслідують різні завдання однієї і тієї ж мети, а саме: збір вимог у бажаному вигляді та розробка заявленого функціоналу (як для SDLC); аналіз вимог, допомога клієнту та команді розробників, підтвердження якості реалізованого функціоналу (як для SDLC).

Загальна мета – задоволення клієнта та отримання максимально можливого балу на етапах перевірки. Роль STLC у SDLC можна представити на дволінійному графіку, показаному на рис. 1.1.

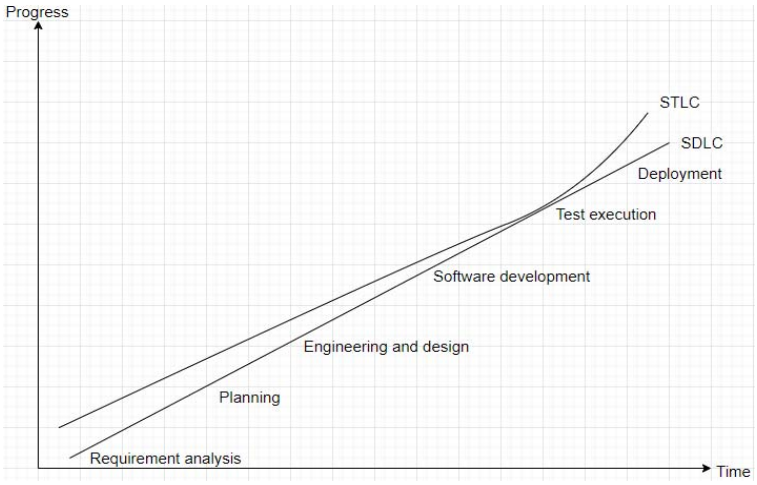


Рис. 1.1. Перетин на етапах STLC та SDLC

Де лінії SDLC та STLC рухаються паралельно протягом більшої частини проекту, але починають швидко сходитися на етапі розробки програмного забезпечення із глибокою синхронізацією на етапі тестування SDLC. Цей графік актуальний для безлічі різних типів проектів, а не тільки великих або незалежних, і залишиться однаковим для реалізації завдання в рамках проекту та зворотним для проекту з величезною кількістю ітерацій (але в цьому випадку лінії частіше розходитимуться і сходитимуться).

1.3 Сучасні методології та практики розробки та тестування

Серед сучасних методологій та практик розробки та тестування ПЗ особливе місце посідають конвеєри CI/CD. Конвеєр доставки програмного забезпечення є центром в автоматизації, оскільки він забезпечує швидку та безпечну доставку продукту.

CI/CD DevOps практики і автоматизація - є основними компонентами розробки програм та значно покращують цей процес, надаючи переваги як членам команди розробників програмного забезпечення, так і замовнику.

Конвеєри CI/CD дозволяють інтегрувати невеликі пакети коду, що спрощує вирішення можливих проблем пізніше. Більш того, ці дрібніші частини можна точно протестувати відразу, після їх злиття з репозиторієм коду. Це дозволяє команді виявляти та вирішувати проблему одразу, а не після того, як буде виконано занадто великий обсяг роботи. І, нарешті, використання конвеєрів CI/CD забезпечує швидку ідентифікацію локалізації збоїв. Це відмінне рішення для великих команд, особливо для тих, у які входять розробники, що працюють віддалено, коли спілкування між членами команди може бути ускладнене. [6]

Також, цей процес забезпечує зростання темпу випуску за умови, що розробники постійно створюють код. Іншими словами, конвеєри CI/CD забезпечують узгоджену інтеграцію та розгортання коду, готуючи код до випуску.

Завдяки своєму потенціалу прискорення процесу розробки програмного забезпечення, автоматизовані конвеєри доставки допомагають підприємствам краще реагувати на потреби ринку.

Автоматизовані конвеєри доставки забезпечують меншу кількість помилок,

а у разі виникнення - швидке виявлення помилок. Це означає, що членам команди не доведеться вносити численні зміни до коду, щоб була можливість зосередитись на розробці якісного продукту. А, якщо поєднаєти швидкість і продуктивність, то можно створити продукт, який допоможе перемогти конкурентів.

Безперервна інтеграція (CI). Безперервна інтеграція - це метод, який дозволяє автоматично інтегрувати будь-які зміни до коду від різних членів команди. По суті, безперервна інтеграція забезпечує якість коду і включає аналіз коду, компіляцію, упаковку коду і модульне тестування. [6]

Розробники, що використовують CI, послідовно інтегрують свої зміни до основної галузі. Ці зміни затверджуються за допомогою поточного тестування створеного складанням.

Ось найголовніше: команді більше не доведеться переживати день релізу.

Автоматизоване тестування - ключова перевага безперервної інтеграції, яка заощаджує час та ресурси. По-перше, автоматизоване тестування забезпечує виявлення помилок на ранніх стадіях і зводить до мінімуму потрапляння помилок до кінцевого користувача. По-друге, CI - спрощує складання релізу, оскільки допомагає усувати проблеми відразу ж, як вони виникають. Скорочуючи витрати на тестування: безперервна інтеграція дозволяє запускати кілька тестів за кілька секунд.

Нарешті, CI дозволяє членам команди краще зосередитися на своїх поточних завданнях, оскільки інженери відразу отримують повідомлення про помилку і можуть миттєво виправити її, перш ніж переключаться на наступне завдання.

Безперервне постачання та Безперервне розгортання (CD)

Безперервне постачання - це підхід у розробці програмного забезпечення, який забезпечує часті та швидкі випуски продукції. [6]

Команди, що використовують безперервне постачання, виробляють розробки в короткі цикли, щоб їх можна було випустити будь-коли. Деталі невеликого розміру легко виправити у разі виникнення проблем. При безперервній доставці - потрібно буде встановити частоту випуску, наприклад

щодня або щотижня. Щоб максимізувати результати процесу розробки, команда повинна виконувати розгортання у робочому середовищі якнайчастіше.

Безперервне постачання дозволяє швидко та легко випускати продукт. Оскільки автоматизація є основним принципом безперервної доставки, та допомагає ефективно автоматизувати випуски. Команда заощадить час, зосередившись на творчих завданнях замість того, щоб виконувати контрольні списки вручну.

Крім того, CD забезпечує контроль версій, що є дуже важливим для будь-якого проекту. Це дозволяє команді ефективно співпрацювати у загальній базі даних. Завдяки контролю версій, розробники можуть легко повернутися до попередньої версії продукту. Крім того, виправлення можна відстежувати протягом усієї історії.

Безперервне розгортання

Безперервне розгортання відповідає за часту доставку функціональних можливостей програмного забезпечення. Однак воно, також, дозволяє автоматично доставляти користувачеві будь-які зміни у виробництві. Це хитрий спосіб прискорення обміну відгуками з клієнтами, використовуючи безперервне розгортання, позбавить кошмару, коли прийде день релізу. Команда може побачити, як їх розробки набирають чинності одразу після виконання завдань.

Безперервне розгортання - пропонує ще більше додаткових переваг: по-перше, це забезпечує швидший темп процесу розробки і команді не потрібно зупиняти його для випусків, оскільки кожна зміна передається автоматично; по-друге, релізи простіше з погляду виправлення будь-яких проблем, оскільки зміни вносяться у невеликих підрозділах. [6]

На додачу до всього, безперервне розгортання підвищує якість обслуговування клієнтів, оскільки вони можуть стежити за послідовними поліпшеннями.

1.4 Автоматизація розробки і тестування ПЗ

Якість є одним із принципів, якої повинні дотримуватися на проекті. Команди бажають створювати найкраще програмне забезпечення відповідно до вимог клієнтів - ефективне, безпомилкове та економічне.

Автоматизація розробки програмного забезпечення

Процес автоматизованої розробки програмного забезпечення на проекті характеризується такими характеристиками:

Створено єдиний загальний репозиторій коду. Усі розробники розміщують написаний ними код у репозиторії. В даний час Git є найпопулярнішою системою контролю версій. Код у репозиторії є єдиним джерелом програмного забезпечення у проекті.

Жодних додаткових скриптів, програм або іншого коду, що розсилаються електронною поштою або розповсюджуються в компанії будь-яким іншим чином, не передбачено.

Існує так званий «процес складання».

Процес складання – це стандартизований метод створення та побудови подальших копій програмного забезпечення. Кожен розробник, тестувальник, тестуючий скрипт і механізм використовують той самий процес для отримання поточної версії програмного забезпечення.

Процес збирання автоматизований

Отримання актуальної версії програмного забезпечення не вимагає від когось виконання великої кількості ручних дій. В ідеальній ситуації, процес складання є іншим скриптом або частиною програмного забезпечення, версія якого, також, знаходиться в репозиторії коду. Розробник завантажує останній код з репозиторію, запускає процес складання (наприклад, запускаючи скрипт) та отримує поточний стан програми.

Один і той же скрипт повинен використовуватись усіма інструментами тестування та середовищами тестування, а також для збирання демонстраційних версій.

Процес збирання швидкий. Створення пакета програмного забезпечення не триває надто довго. Це дозволяє тестувати результати та впроваджувати виправлення кілька разів.

Команда вносить зміни часто, щодня чи у разі кілька разів на день. Робочий код постійно поміщається в основну гілку у системі контролю версій.

Середовище тестування має бути максимально схожим на робоче середовище. В ідеальній ситуації це була б пряма копія виробничого середовища.

Процес запуску програмного забезпечення у виробництво автоматизований. У кращому разі внесення нових змін у робоче середовище має виконуватися натисканням однієї кнопки або запуском одного сценарію.

Автоматизоване тестування

Сучасне програмне забезпечення дуже складне, і часто складається з сотень тисяч рядків коду, розкиданих по безлічі файлів. У таких складних проектах використовують численні бібліотеки та інші залежності. Зміни в коді та бібліотеках, як правило, впливають на кілька функцій системи.

При внесенні змін розробники програмного забезпечення зазвичай перевіряють, чи код працює так, як задумано. Однак найчастіше вони не мають ні знань про систему, ні можливості, ні часу, щоб перевірити, чи впливають їх зміни на інші функції системи, над якою вони працюють.

Те саме стосується і тестувальників - вони просто не взмозі перевірити всю систему на наявність помилок, які могли б зафарбуватися при тестуванні нових функцій. Якщо в проекті вони справді виконують усі сценарії тестування вручну при кожному розгортанні, розгортання стає стомлюючим та дорогим. Кожна виявлена та згодом виправлена помилка вимагає внесення змін до кодової бази, що означає тестування всієї системи з нуля чи ризик помилок. Часто використання таких практик призводить до довгої спіралі тестів, помилок, нових тестів, нових помилок... Або, що ще гірше, недбале тестування, що призводить до того, що помилки вирушають у виробництво.

Цю проблему можна вирішити, впровадивши автоматизоване тестування.

Тести мають бути автоматизовані, і їх має бути легко запускати. Кожен розробник повинен мати можливість запускати всі тести у середовищі розробки.

Тести повинні бути швидкими, тому що вони можуть бути пропущені через те, що вони займають багато часу. Тести повинні проводитися щоразу під час зміни програмного забезпечення.

В ідеальній ситуації кожен commit має бути протестований, але розоробник обов'язково має протестувати хоча б кожен фрагмент коду, який буде відправлений в основну гілку.

Тести повинні охоплювати максимально можливий обсяг програмного забезпечення. Кількість тестів та спосіб їх планування повинні дозволяти тестувати всі важливі та значущі функції програмного забезпечення, а команда має бути впевнена, що той факт, що всі тести завершено успішно, означає, що програмне забезпечення працює правильно. Існує безліч методів оцінки покриття тестами, які можна використовувати для контролю кількості тестів.

Тести мають бути частиною збирання. В ідеальній ситуації кожне складання має автоматично запускати всі тести та відображати їх результати.

Кожна зміна має бути перевірена шляхом тестування всієї системи. Знову ж таки, в ідеальній ситуації протестували б кожен коміт, але часто досить протестувати код перед його злиттям з основною гілкою. Коли розробник оновлює гілку master кодом, що ламає програмне забезпечення, вони заважають роботі інших людей, які хочуть почати роботу над новим завданням за допомогою останнього складання - вона просто не працюватиме через помилки в коді.

Команди, які не використовують автоматизоване тестування, часто стикаються з проблемою, коли якийсь новий коміт ламає частину програмного забезпечення, і всі, хто хотів працювати над ним, зупиняються, поки помилка не буде виправлена. Автоматизоване тестування запобігає таким ситуаціям.

Досягши цілей автоматизації розробки та тестування - отримуєте безліч переваг, що роблять процес розробки програмного забезпечення більш упорядкованим.

Інженер можете швидко перебудувати середовище розробки. Одна з найбільших проблем із проектами з поганою автоматизацією - це постійні розбіжності між тим, як програмне забезпечення працює та працюватиме у виробничому середовищі, порівняно із середами розробки та тестування. Через те, що зміни між середовищами потрібно реплікувати вручну, це робиться рідко, або не робиться взагалі. Кожне середовище має деякі особливості, і якщо їх багато, ця проблема лише посилюється. Таким чином, ризик того, що код або конфігурація, створені в одному середовищі, працюватимуть інакше в іншому, стає високим.

Автоматизована швидка перебудова середовища повністю усуває проблему.

Коли інженер автоматизуватиме розробку та тестування - отримаєте механізм, у якому завжди знатимете стан свого програмного забезпечення. Що важливо, перш ніж надавати будь-які нові зміни, можна перевірити, що вони призводять до нових помилок. Завдяки цьому, завжди можна прийняти рішення про розгортання поточної версії програмного забезпечення в робочому середовищі.

Більше того, можна зробити це швидко та безпечно, тому що це буде виконуватись автоматизованим процесом, попередньо протестованим у тестовому середовищі, ідентичному виробничому.

Досягши такого рівня автоматизації, розробник можете безпечно розробляти нові функції, знаючи, що вони точно працюють і нічого не ламають, а потім розгортати програмне забезпечення. По суті виключається ризик щось зламати, а якщо знайдеться якась помилка, є можливість швидко її виправити.

Висновки за розділом 1

У цьому розділі розглянуті парадигми автоматизації тестування та розробки з використанням Agile методології. Проведено аналіз використання гнучких технологій спільно з автоматизацією через призму CI/CD DevOps практик.

Виходячи з аналізу проблематики, автоматизація розробки та тестування на проекті дає можливість отримати безліч переваг, що роблять процес випуску програмного забезпечення до кінцевого користувача більш упорядкованим, швидшим, дешевшим та безпечнішим для бізнесу.

					151.4341.КР.ПЗ.Р2		
Змн	Арк	№ докум.	Підпис	Дата			
Розроб.		Черній М.О.			ПРОГРАМНІ ЗАСОБИ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ТА ТЕСТУВАННЯ ПЗ	Літ.	Арк 24 / Аркушів 17
							НУК ім. адм.. Макарова
Норм. контр		Вінниченко І.Л.					
Керівник		Герасін О.С.					
Зав. Каф.		Черно О.О.					

РОЗДІЛ 2
ПРОГРАМНІ ЗАСОБИ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ТА ТЕСТУВАННЯ ПЗ

Знання мови програмування допомагає інженеру з автоматизації по-різному.

Крім надання доступу до інструментів автоматизації, здатність розуміти код сприяє тестуванню. Так чи інакше, це підвищує компетенцію людини та робить компанію, що займається автоматизацією процесів розробки та тестування програмного забезпечення, більш підготовленою до викликів ринку. Питання, яку мову програмування використовувати в автоматизації.

2.1 Мова програмування Python в автоматизації

Python - це мова програмування з відкритим вихідним кодом. Більше 70% розробників вважають його найпопулярнішою та затребуваною мовою. У відкритому доступі багато бібліотек, тому менше рядків вихідного коду для самостійного написання. Синтаксис Python є простим, що спрощує вивчення мови. Крім того, навколо Python створено сильну спільноту, і будь-коли можна звернутися за допомогою в Інтернеті і знайти відповідь на будь-яке питання. [3]

Ніщо не говорить про Python краще, ніж його зростаюча популярність. Але популярність — не єдина причина, чому інженери з автоматизації продовжують використовувати Python. Такі технології, як Java, C#, C++ та Ruby, часто використовуються у службах автоматизації тестування. Тим не менш, у Python є низка переваг, які роблять його оптимальним рішенням. Python простий в засвоєнні. Інженер з автоматизації повинен зосередитися на сервісах автоматизації програмного забезпечення, і вивчення нових речей не повинно бути перешкодою. Простий синтаксис робить Python найкращою мовою програмування для вивчення з нуля. Крім того, у мережі можна знайти купу корисних матеріалів.

Код Python є простим для розуміння. Він зручний для написання сценаріїв та підтримується численними інструментами. Python - (майже) універсальна мова. Python - мова загального призначення, здатна вирішувати широкий спектр

завдань. Він використовується у веб та настільних додатках, аналітиці даних, сценаріях тощо.

Він підвищує продуктивність команди. Там, де Python потребує одного рядка коду, Java використовує десять рядків. Python лаконічний, тому дозволяє вирішувати більше завдань із меншою кількістю рядків коду, залишаючи дорогоцінний час для вирішення складних завдань.

Величезні бібліотеки коду допоможуть заощадити час.

Не потрібно винаходити велосипед, а можна використати готовий код для імпорту. Автоматизація сценаріїв Python спрощує життя, оскільки може автоматизувати весь світ - від розгортання середовища до безперервної інтеграції/розгортання.

Pytest – одна з найкращих доступних платформ для комп'ютерів. Він може зустрічатися з широким комплексним тестуванням, що вже говорити про модульне, інтеграційне або наскрізне тестування. Тестові випадки для прийняття просто як функції та прийняті вхідні дані. Плагіни розширюють можливості Pytest і дозволяють приховувати код, запускати кілька тестів одночасно і поєднуватися з іншими фреймворками, такими як Django і Flask. Багата бібліотека корисних компонентів та готових інгредієнтів для значного тестування на Python.

Python є об'єктно-орієнтованим та функціональним. Це дозволяє вибрати те, що найкраще підходить для ваших завдань - функції чи класи. В розподілені функції немає побічних ефектів, а простий синтаксис робить їх читабельними.

Командний рядок може контролювати всіх. Кожен тестовий фреймворк може запускати консоль для пошуку та запуску тестів. Багата підтримка значної кількості синтезу управління тестами.

Більш того, автоматизація за допомогою Python підтримує тестування. Python використовується, щоб довести програму до точки, коли потрібно ручне тестування. Завдяки масштабованості Python добре підходить для початківців, так і досвідчених користувачів. Масштабованість охоплюється за рахунок синтаксису, чудової структури, модульності та великої екосистеми.

Також можна інтегрувати сторонні інструменти та процеси.

2.2 Налаштування контейнеризації та процесу керування. Організація архітектури кількох контейнерів

У старі часи, коли використовувався SSH, коли потрібно було зайти на виробничий сервер, потрібно було перейти до каталогу проекту та запустити git pull, щоб розгорнути свій код. Перш ніж інженер щось розгорне, на самому початку життя сервера слід встановити всі глобальні залежності для програми, швидше за все, curl, потім git, можливо, інтерпретатор для мови, яку інженер хочете використовувати, і деякі розширення для цього також, можливо, nginx в якийсь момент. Після встановлення всіх залежностей потрібно перенести програму на сервер, запустити деякі команди для встановлення та зрештою запустити програму.

На цьому етапі, як тільки розробник витягнете свій код на сервер, інженер запусте нову версію своєї програми або перезапусте веб-сервер, щоб деякі зміни вступили в силу. Ця установка, ймовірно, працювала протягом тривалого часу. Команда розробників покладалася на залежність системи, яка має іншу версію, встановлену на робочому сервері, і тепер служба не працює. Потрібно швидко відкотити зміни розробника, але інженеру в якийсь момент доведеться оновити цю залежність. Гіршим прикладом можуть бути помилки, викликані такими відмінностями залежностей у дивному місці програми, що означає, що інженер, ймовірно, не помітить, поки не буде надто пізно.

Розглянемо інший приклад, коли необхідно запустити кілька програм на одному хості, але з міркувань безпеки їх потрібно ізолювати.

В цьому випадку потрібно або перемістити програми на окремі хости, що не рентабельно, або запустити дві різні віртуальні машини на хості, що надасть ізоляцію, але ресурси споживатимуть переважно віртуальні машини, а не додаток, що все ще не найкращий спосіб. Ці проблеми існують протягом десятиліть; тримати процеси окремо - це величезний біль, і це викликало багато проблем із безпекою, а також неефективні налаштування в минулому.

Контейнери Linux. У цьому контексті контейнер — це набір ізольованих процесів і ресурсів. Linux досягає цього за допомогою просторів імен, що дозволяє процесам отримувати доступ лише до ресурсів простору імен, що дозволяє мати дерево процесів, яке повністю не залежить від решти систем.

Docker. Docker - це один із інструментів, який використовував ідею ізольованих ресурсів для створення набору інструментів, що дозволяє пакетувати додатки з усіма встановленими залежностями та запускати де завгодно. Docker визначає контейнери таким чином:

Контейнер - це стандартна одиниця програмного забезпечення, яка упаковує код і всі його залежності, щоб програма швидко й надійно працювала від одного обчислювального середовища до іншого.

Відмінності від віртуальних машин

Контейнери Docker використовують одні й ті ж системні ресурси, у них немає окремих виділених ресурсів на рівні апаратного забезпечення, щоб вони вели себе як повністю незалежні машини.

Їм не потрібно мати повноцінну ОС всередині.

Вони дозволяють запускати кілька робочих навантажень на одній ОС, що дозволяє ефективно використовувати ресурси.

Оскільки вони здебільшого включають залежності на рівні програми, вони досить легкі та ефективні.

Машина, на якій можна запускати 2 віртуальні машини- дає можливість запускати десятки контейнерів Docker без будь-яких проблем, що означає менше ресурсів = менше витрат = менше обслуговування. Образи Docker визначаються в спеціальних текстових файлах, які називаються Dockerfile, і повинні чітко визначити всі кроки всередині Dockerfile. Приклад з використанням образу для конвеєрів приймальних тестів, який включає в себе: Python 3.8 на Alpine 3.11, Chromium, Selenium з драйвером Chrome і pytest (див. рис. 2.1.).

```
FROM python:3.8-alpine3.11

# update apk repo
RUN echo "http://dl-4.alpinelinux.org/alpine/v3.11/main" >> /etc/apk/repositories && \
    echo "http://dl-4.alpinelinux.org/alpine/v3.11/community" >> /etc/apk/repositories

# install chromedriver
RUN apk --no-cache add chromium chromium-chromedriver

# install selenium
RUN pip install selenium pytest
```

Рис. 2.1. Dockerfile Python 3.8 на Alpine 3.11

Використання базового образа Python з тегом 3.8-alpine3.11, який є окремою версією Python разом із конкретною версією Alpine.

Розміщення визначенного репозиторію та оновлення індексів репозитарію.

Використання apk, менеджер пакунків Alpine Linux, для встановлення хрому та chromium-chromedriver.

Використання pip для встановлення selenium та pytest.

У більшості випадків в реальному житті програми будуть мати зовнішні залежності, такі як бази даних, черги повідомлень або інші зовнішні служби, з якими програма спілкується, і потребують надійного та легкого у використанні способу внесення цих залежностей у середовище розробки. Хоча все можна запускати як контейнери Docker, керувати контейнерами разом із самою програмою швидко стає громіздким; і Docker має дуже гарне рішення для цього.

Docker Compose. Compose - це інструмент для визначення та запуску багато контейнерних програм Docker. За допомогою Compose - можна використовувати файл YAML для налаштування служб програми. Потім за допомогою однієї команди можна створити та запустити всі служби з конфігурації.

Приклад використання Compose (див. рис. 2.2.):

1. Одна команда для запуску всієї програми: docker-compose up.
2. Простий дозвіл для DNS за допомогою імен контейнерів: це означає, що можна викликати контейнер service-a з контейнера service-b за допомогою http://service-a.
3. Можна монтувати томи з локальної файлової системи в контейнер за допомогою одного рядка у визначенні YAML.

4. Можна перезапускати лише контейнери, у яких є зміни, що означає швидший час завантаження у разі перезапуску.

```yaml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
      REDIS_HOST: redis
      REDIS_PORT: 6379
  redis:
    image: "redis:alpine"
```

Рис. 2.2. Складання образу для redis alpine з папки проекту

Файл починається з визначення версії Docker Compose, яка визначає, які функції можна використовувати.

Ім'я контейнера (web). Визначає перший контейнер програми.

*Крок збірки (*build*).* Вказує Compose побудувати поточний шлях за допомогою docker build , що означає, що в поточному шляху є файл Docker, який слід використовувати для зображення.

Порти (ports). Визначають відображення між хостом і портами контейнера у режимі <HOST_PORT>:<CONTAINER_PORT>. У цьому прикладі localhost:5000 буде зіставлено з портом 5000 цього контейнера.

Томи (volumes). Визначають відображення файлової системи між хостом і контейнером. У цьому прикладі поточний шлях буде змонтовано до шляху /code контейнера, що дозволить вам замінити код у зображенні вашою локальною файловою системою, включаючи ваші поточні зміни.

Середовище (environment). Дозволяє встановити змінні середовища для вашого контейнера. У цьому прикладі він встановлює три змінні середовища: FLASK_ENV, REDIS_HOST і REDIS_PORT. Значення REDIS_HOST встановлено лише на redis, що можливо, оскільки контейнер Redis, наведений нижче, використовує redis як назву служби, що робить його доступним для мережі Compose з тим самим ім’ям.

Службу redis (redis). Використання зображення redis:alpine, і це єдине визначення, необхідне для цієї служби. Визначення служби досить інтуїтивно зрозуміле і дозволяє легко керувати всіма залежностями за допомогою цього одного файлу. Завдяки Docker Compose локальна розробка за допомогою контейнерів Docker стає неймовірно легкою, і після того, як ви включите цей файл у корінь проекту, усі інженери можуть просто запустити docker-compose up, щоб запустити всю програму, не потребуючи нічого встановлювати. крім самого Docker.

Існує ще одна перевага контейнеризації додатків, яку часто ігнорують: розміщення вашої програми в контейнері в кінцевому підсумку призведе до кращого використання вашої програми.

Оскільки в середовищі розробки та в реальному часі будуть працювати більш-менш ті самі контейнери, жорстке кодування різних значень конфігурації в базі коду спричинить проблеми; отже, потрібно екстерналізувати ці значення та відокремити їх від програми, що надасть вам велику гнучкість у довгостроковій перспективі щодо середовищ розгортання, будь то розробка, підготовка чи реальна експлуатація.

Працюючи з контейнерами, інженер помітить, що час від часу він збирається перезапускати контейнери, і стан в кінцевому підсумку зникне з контейнера, або через відмінності в сховищі, або контейнер потрібно буде повторно створити, або просто інженер перезавантажив комп'ютер. У різних випадках, подібних до цього, інженер помітить, що програма має внутрішній стан, і інженер захоче перенести цей стан в інше місце, що дозволить інженеру мати додаток без стану, що також дуже допоможе інженеру під час масштабування. послуги по горизонталі.

Оскільки різні члени команди будуть часто створювати образи, вони можуть зіткнутися з випадком невідповідності із залежністю підстановки в додатку, що підштовхне інженера до створення конкретних версій для всіх залежностей, що допоможе створити відтворювані збірки та розгортати з набагато більшою впевненістю.

Оскільки інженер буде запускати програми в контейнерах, кожен раз знадобляться журнали з програми. Інженер захоче мати стандартизований спосіб роботи з журналами, який використовує stdout і stderr для всіх потреб у журналі. Це дозволить локально переглядати потік стандартного виведення з терміналу, а також збирати журнали за допомогою більш складних менеджерів журналів у прямому ефірі; програма не буде піклуватися про систему реєстрації.

Інженер зможе чітко розмежувати етапи збирання, випуску та розгортання, оскільки буде створювати образи Docker, позначати їх для випусків та розгортати окремо, що ідеально відповідає фактору «Build, release, run».

Розгортання контейнерів. Хоча розробка додатків у відтворюваний спосіб є важливою, головна потреба полягає в тому, щоб мати можливість доставити додаток до реального середовища та обслуговувати реальний трафік. Адаптація контейнерів і відокремлення програм від середовища тут дуже добре окупаються, ефективно відокремлюючи програми від середовища їхнього розгортання. Його можна розгорнути на безсерверній хмарній платформі, його можна розгорнути в кластері зі 100 вузлами або на одній машині; додатку байдуже.

Коли справа доходить до розгортання контейнерів, існує досить багато хороших рішень залежно від масштабу та потреб:

Інженер може розгортати програми з робочим файлом Docker Compose за допомогою віддаленого хоста Docker на базовій хмарній віртуальній машині, наприклад AWS EC2, DigitalOcean Droplets або будь-якій іншій віддаленій машині.

Інженер може розгорнути програму на керованій платформі розгортання додатків, наприклад AWS Elastic Beanstalk або Google App Engine, яка буде займатися балансуванням навантаження, автоматичним масштабуванням і розгортанням.

Інженер може розгорнути на безсерверному рішенні, такому як AWS ECS або Google Cloud Run, що дозволить інженеру не платити, коли програма не запущена,і матиме більш гнучке рішення для контейнерування. Очевидно, ECS не є повністю безсерверним, оскільки інженеру все ще потрібно виділити власні

вузли, і він віртуалізує їх. Іншою альтернативою для прямого безсерверного рішення є AWS Fargate.

Інженер може розгорнути додаток на рішеннях для оркестровки контейнерів, таких як Docker Swarm, Kubernetes або Nomad. Спеціально для Kubernetes також доступні керовані рішення, такі як AWS EKS, Google Kubernetes Engine або DigitalOcean Kubernetes.

2.3 Вибір програмного середовища для побудови та керування CI/CD-практиками

Стабільність і надійність програми залежать від глибини тестування, проведеного перед його випуском. За останні кілька десятиліть автоматизоване тестування в поєднанні з хмарними технологіями проклало шлях для швидшого та масштабного тестування.

Як невід'ємна частина потоку Agile, автоматизоване тестування є важливим для будь-якого сучасного програмного забезпечення. Зокрема, CI/CD (безперервна інтеграція/безперервне розгортання) - це практика, яку використовують розробники та тестувальники Agile для прискорення розробки та тестування, щоб їхні продукти могли залишатися конкурентоспроможними та керованими якістю.

Реалізація CI/CD вимагає кількох інструментів, серед яких Дженкінс є найпопулярнішим.

Стабільність і надійність програми залежать від глибини тестування, проведеного перед його випуском. За останні кілька десятиліть автоматизоване тестування в поєднанні з хмарними технологіями проклало шлях для швидшого та масштабного тестування.

Як невід'ємна частина потоку Agile, автоматизоване тестування є важливим для будь-якого сучасного програмного забезпечення. Зокрема, CI/CD — це практика, яку використовують розробники та тестувальники Agile для прискорення розробки та тестування, щоб їхні продукти могли залишатися конкурентоспроможними та керованими якістю. Реалізація CI/CD вимагає кількох інструментів, серед яких Jenkins є найпопулярнішим.

Jenkins - популярний інструмент оркестрування CI/CD. Він надає численні плагіни для інтеграції з декількома інструментами й фреймворками автоматизації тестування/розробника в конвеєр розробки/тестування. Коли справа доходить до автоматизації, Jenkins надає плагіни, які допомагають запускати набори сценаріїв, збирати результати та на інформаційній панелі, а також надавати детальну інформацію про збої.

Запускає набори автоматизованих скриптів: Jenkins надає плагіни для різних тестових платформ, таких як Selenium, Cucumber, Appium, Robot framework тощо. Їх можна інтегрувати в конвеєри CI/CD для запуску автоматизованих скриптів для кожної збірки.

Більшість плагінів також підсумовують результати сценарію та відображають їх як сторінку HTML.

Jenkins відстежує результати та відображає їх у вигляді графіка тенденцій. Це дає змогу краще побачити, як сценарії працювали в минулому.

Відображення деталей про помилки тестування/розробника: результати тесту зводяться в таблицю, а помилки реєструються разом із результатами тестування.

Завдяки численним пропонованим плагінам, Jenkins у більшості випадків задовольняє всі потреби в автоматизації. Більшість популярних засобів автоматизації можна легко інтегрувати з Jenkins.

Для простих робочих процесів, які містять спрощений процес CI/CD, Jenkins ідеально підходить для автоматизації тестування. Можна викликати тестові сценарії, а також створювати звіти.

Найкраще використовувати Jenkins, якщо плагіни надходять із надійного джерела – бажано від розробників самого інструменту автоматизації. Це дозволить уникнути непокірної поведінки робочих місць і несподіваних результатів. Це також зробить інтеграцію набагато легше.

Jenkins – чудовий вибір, якщо компанія або команда готові перейти до автоматизації. Він добре працює з методологією CI/CD і допомагає команді швидше завершити процес тестування та розробки. Jenkins об'єднує все, що потрібно компаніям для підтримки автоматизації, в одному функціональному

інструменті.

Автоматизація розширює можливості команди та дозволяє їм бути впевненішими у своїй роботі, тому що вони можуть швидко побачити результати. Великий набір функцій Jenkins дає компаніям та командам величезну перевагу.

2.4 Налаштування систем контролю версій та їх інтеграція для автоматизації процесів розробки та автоматизації

Системи контролю версій необхідні передового досвіду сучасної розробки програмного забезпечення. Керування версіями дає змогу відстежувати ваше програмне забезпечення на рівні вихідного коду. Може відслідковувати зміни, повертатися до попередніх етапів та створювати альтернативні версії файлів та каталогів.

Файли багатьох програмних проектів зберігаються у репозиторіях системи контролю версій. Однією з таких є Git, а такі платформи, як GitHub, GitLab і Bitbucket, допомагають полегшити спільне використання та спільну роботу над проектами розробки програмного забезпечення.

Контроль версій Git можна використовувати не тільки для керування версіями проекту, але і для підтримки автоматизації середовищ.

Багато принципів ефективного використання Git Branching та контролю версій для автоматизації розробки та тестування - успадковані від спільноти розробників програмного забезпечення. Ці ідеї застосовуються для підтримки створення масштабованої та зручної в обслуговуванні автоматизації на проекті.

Git розгалуження в розробці.

Зазвичай конвеєр розробки розбивається на кілька артефактів. На найнижчому рівні є виробнича версія, що відправляється кінцевим користувачам,версія-кандидат, на якій ми проводимо остаточне регресійне тестування, та складання для розробки, що містить останні функції розробки. Це можна розділити ще на кілька категорій для різних середовищ, таких як підтримка канаркових випусків, інтеграція та багато іншого. Така категоризація є стандартною для великомасштабної розробки програмного забезпечення.

Для кожної категорії необхідно створювати гілки всередині репозиторію git.

Гілки - це спосіб розгалуження та незалежного просування вперед у межах ізольованої кодової бази. Коли частина продукту готова, ми можемо поєднати гілки вперед і назад з іншими гілками, що походять з того ж екземпляра основної гілки. В рамках розробки для кожного середовища використовується окрема гілка. Розробники відокремлюються від основної галузі розробки, коли вони починають розробку нової функції. З цією метою розробники можуть працювати паралельно, працюючи з ізольованим кодом, що відноситься до їхніх історій і завдань. Як тільки функція досягає визначення завершення, робиться запит на вилучення, з якого може бути схвалено злиття для інтеграції їх коду разом із роботою будь-яких інших розробників, які також виконали роботу за час, що минув з моменту створення початкової гілки. Після того, як усі функції для випуску або спринту будуть створені, можна створити кандидата у випуск, виконуючи приймальне тестування користувачів та регресію піраміди тестування.

Із запровадженням безперервної інтеграції також стало звичним постійно тестувати кожну реєстрацію коду, запускаючи середовища пісочниці і виконуючи автоматизацію тестування коду. Якщо тести проходять при кожному злитті, то він інтегрується у гілку, що випускається. Останнім етапом є переміщення гілки реліз-кандидата у виробничу гілку. Таким чином, кандидат випускається, зазвичай, з відповідними тегами, що відповідають номерам версій випуску.

Git розгалуження для платформи автоматизації

Початкова стратегія середовища автоматизації тестування, що зберігається в Git, зазвичай обертається навколо використання однієї основної гілки. Це нормально, коли над проектом працює один інженер з автоматизації; однак для ефективного масштабування фреймворку потрібна набагато коротша стратегія розгалуження.

Краще розбити гілки на категорії, що використовуються у розробці: виробництво, реліз-кандидат та розробка. Існують гілки для кожної функції, на цей раз для наших тестових активів, створення гілок функцій, але кожен створений сценарій автоматизації.

У цьому контексті інженери використовують аналогічну стратегію

розгалуження для середовища автоматизації та середовища розробки Git. По-перше, інженер з автоматизації переходить з основної (основної) гілки, коли вони починають створювати автоматизацію (об'єкти сторінки та тести) для нової функції. Створення тегів для гілки розробки також корисно цьому етапі. Після того, як активи автоматизації створені з достатнім охопленням сценаріїв, робиться запит на вилучення, з якого може бути схвалено злиття для інтеграції їх коду разом із роботою будь-яких інших інженерів з автоматизації, які зафіксували свою роботу у проміжку. Використання розгалуження у цьому контексті означає, що ми не змішуємо автоматизацію, не готову до виробництва, з базовою автоматизацією, яка використовується в нашому конвеєрі безперервної інтеграції. Після створення автоматизації для випуску можна об'єднати з гілкою випуску автоматизації.

Використання розгалуження дозволяє проводити перевірки коду з тим самим ефектом, але з автоматизацією коду. Це гарантує, що новий код відповідає високим стандартам, з підтримуваними ідентифікаторами об'єктів та гарною архітектурою. Це основний аспект створення ефективної платформи автоматизації, що масштабується з часом.

Також дозволяє розрізняти автоматизацію, готову до виробництва, та автоматизацію у процесі виробництва. Крім того, можна вставляти посилання в кожну гілку, щоб мати повну простежуваність між вихідним кодом продукту та кодом автоматизації тестування. Ще краще, якщо тестові гілки слідують тим самим угодам, це означає, що ми можемо відстежувати вихідний код і автоматизовані скрипти.

Такий підхід дозволяє створити структуру автоматизації, яка ефективно застосовує розгалуження активів автоматизації тестування.

Масштабування автоматизації по підприємству.

Автоматизація може бути успішно реалізована розрізнено по всьому підприємству, найбільш успішна автоматизація здійснюється зверху вниз за участю організації. І тут немає сенсу кожної окремої команди створювати свою власну структуру і пов'язані з нею утиліти; необхідна чітка та послідовна стратегія, яка зробить впровадження безпроблемним та максимізує стратегічну

віддачу від інвестицій.

Для більшості організацій, швидше за все, вже існує безліч фреймворків від захоплених розробників, які пішли та створили свої власні фреймворки таінструменти. У цьому випадку повинен мати місце огляд, щоб визначити якість структури і її придатність для використання у всій організації. Часто буває, що неофіційна корпоративна структура вже набирає обертів, стаючи підходом де-факто.

Саму структуру автоматизації слід розглядати як окрему лінійку продуктів, що містить інтеграції з існуючими інструментальними засобами продукту, утиліти для виклику серверних систем чи протоколів та інші корисні артефакти, притаманні автоматизації всередині організації. Потім така структура використовується командою у своїх лінійках продуктів і вбудовується в пов'язані тестові активи (об'єкти сторінок і сценарії, а також кілька інших утиліт, що настроюються для продукту).

Це хороша практика, яку слід слідувати, навіть якщо автоматизація застосовується в окремих командах, оскільки вона відокремлює структуру та декомпозує її від фактичного логічного коду автоматизації.

Якщо масштабування автоматизації для підприємства існує центр передового досвіду (CoE), то необроблена структура автоматизації має належати функціональної групі у цьому підрозділі. Ця група відповідає за надання утиліт nessacery, функцій звітності, інтеграцій та інших нефункціональних компонентів платформи. Кожна команда, яка застосовує автоматизацію тестування у своєму проекті, може використати цю структуру.

За допомогою Git можна клонувати центральний репозиторій та створювати з нього відгалуження для адаптації фреймворку. Кожна команда вбудовує власні об'єкти сторінки та тестові сценарії для лінійки продуктів, яку вони тестують. Це забезпечує узгоджений спосіб використання автоматизації у всій організації. Ще краще, якщо центральна команда вносить зміни до структури та додає нові функції (наприклад, коли організація перемикає свою систему на керування тестовими прикладами), це можна застосувати на верхньому рівні і

відфільтрувати вниз шляхом злиття для кожного окремого користувача.

Масштабування автоматизації по підприємству

Автоматизація може бути успішно реалізована розрізнено по всьому підприємству, найбільш успішна автоматизація здійснюється зверху вниз за участю організації. І тут немає сенсу кожної окремої команди створювати свою власну структуру і пов'язані з нею утиліти; необхідна чітка та послідовна стратегія, яка зробить впровадження безпроблемним та максимізує стратегічну віддачу від інвестицій.

Для більшості організацій, швидше за все, вже існує безліч фреймворків від захоплених розробників, які пішли та створили свої власні фреймворки та інструменти. У цьому випадку повинен мати місце огляд, щоб визначити якість структури і її придатність для використання у всій організації. Часто буває, що неофіційна корпоративна структура вже набирає обертів, стаючи підходом де-факто.

Саму структуру автоматизації слід розглядати як окрему лінійку продуктів, що містить інтеграції з існуючими інструментальними засобами продукту, утиліти для виклику серверних систем чи протоколів та інші корисні артефакти, притаманні автоматизації всередині організації. Потім така структура використовується командою у своїх лінійках продуктів і вбудовується в пов'язані тестові активи (об'єкти сторінок і сценарії, а також кілька інших утиліт, що настроюються для продукту). Це хороша практика, яку слід слідувати, навіть якщо автоматизація застосовується в окремих командах, оскільки вона відокремлює структуру та декомпозує її від фактичного логічного коду автоматизації. Якщо масштабування автоматизації для підприємства існує центр передового досвіду (CoE), то необроблена структура автоматизації має належати функціональної групі у цьому підрозділі.

Ця група відповідає за надання утиліт nessacery, функцій звітності, інтеграцій та інших нефункціональних компонентів платформи. Кожна команда, яка застосовує автоматизацію тестування у своєму проекті, може використати цю структуру.

За допомогою Git можна клонувати центральний репозиторій та створювати з нього відгалуження для адаптації фреймворку. Кожна команда вбудовує власні об'єкти сторінки та тестові сценарії для лінійки продуктів, яку вони тестують. Це забезпечує узгоджений спосіб використання автоматизації у всій організації. Ще краще, якщо центральна команда вносить зміни до структури та додає нові функції (наприклад, коли організація перемикає свою систему на керування тестовими прикладами), це можна застосувати на верхньому рівні і відфільтрувати вниз шляхом злиття для кожного окремого користувача.

Висновки за розділом 2

У даному розділі розглянуто актуальність мови програмування Python для ведення автоматизації на проекті, організацію контейнеризації та поєднання декількох контейнерів між собою, актуальність інструменту Jenkins для ведення CI/CD практик на проекті та налаштування систем контролю версій за допомогою Git для автоматизації процесів розробки та тестування.

Вибір мови програмування Python для побудови автоматизації дає інженеру змогу без зміни інструменту вести розробку як ПЗ так і скриптів для автоматизації, а вибір такого інструменти як Docker дає змогу швидкого збору сервісу та розгортання його у будь-якому середовищі, Jenkins пропонує деякий API для автоматизації збірки та розгортання контейнерів з урахуванням використання такої системи контролю версій як GIT.

Враховуючи переваги перерахованих вище інструментів, інженер з автоматизації має можливість в короткі сроки розгорнути якісне середовище дляавтоматизації як процесів розробки, так і тестування.

					151.4341.КР.ПЗ.Р3		
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Черній М.О.*			РОЗРОБКА ТА МОДЕЛЮВАННЯ СИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ТА ТЕСТУВАННЯ ПЗ.	*Літ.*	*Арк* 41 / *Аркушів* 34
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм.. Макарова	
Керівник		*Герасін О.С.*					
Зав. Каф.		*Черно О.О.*					

РОЗДІЛ 3
РОЗРОБКА ТА МОДЕЛЮВАННЯ СИСТЕМИ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ТА ТЕСТУВАННЯ ПЗ

3.1 Побудова фреймворку автоматизації

Найчастіше найкращою структурою для автоматизації процесів є фреймворк. Це дозволяє організувати структуру на проекті таким чином щоб можна було перевикористовувати написану автоматизацію для інших проектів - змінивши тільки вхідні дані.

Це дуже зручно - бо економить час та зусилля - не припадати під кожен новий проект - писати новий фреймворк - а лише взяти один готовий та змінити конфігурацію.

Структура фреймворку може бути різною - залежить від переваг і самої архітектури.

Але як показав час і практика - більш зручним буде тримати поруч проекти для яких застосовуватиметься автоматизація на основі CI/CD практик - цими проектами будуть виступати - ПЗ, що розробляється, і фреймворк з автомтизації тестування ПЗ, що розробляється. На рисунку 3.1. - приведена можлива структура проекту для побудови фреймворку автоматизації.

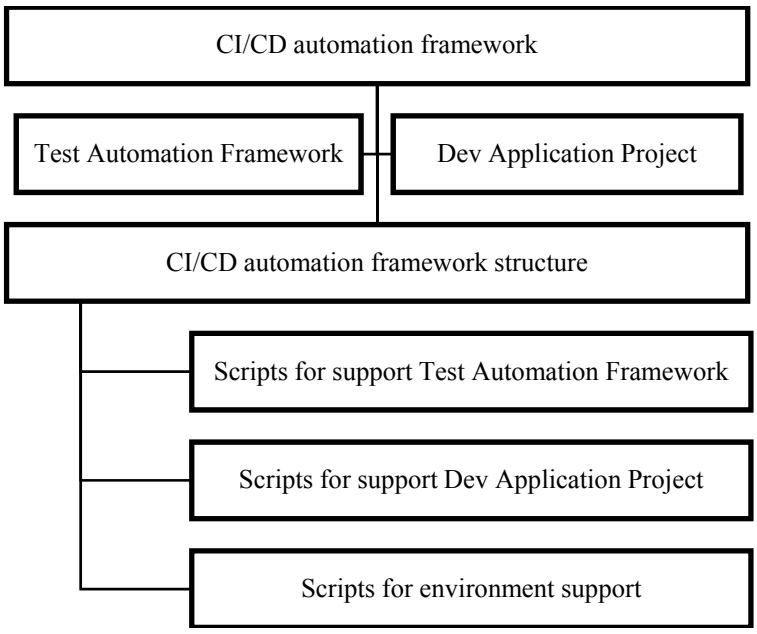


Рис. 3.1. Структура фреймворку

Скрипти підтримки тестів включають: environment.py, setup_environment.py.

Як правило для написання сриптів на підтримку тестового фреймворку вибирається мова програмування якою спроектовані і самі тести.

Робиться це з тих міркувань, що дані беруться для підтримки тестового оточення із тестового фреймворку. Та для зручності експлатація для команди тестування.

Виходячи з цих міркувань, дані скрипти були реалізовані на Python.

environment.py - отримує дані про те, яке оточення буде використане для тестування та повертає актуальне доменне ім'я оточення. Дані вираховуються з пермінних оточення - які були утсановлені під час написання тестового фреймворку. Блок схема скрипта environment.py (див. рис. 3.2). Блок схема конструктора environment.py продемонстрована – рис. 3.3., блок схема реалізації метода get_base_url (див. рис. 3.4.), код реалізацію розробленого скрипта (див. рис. 3.5.).

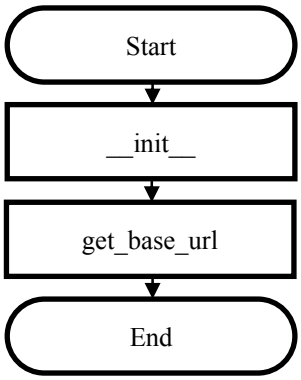


Рис. 3.2. Структура environment.py

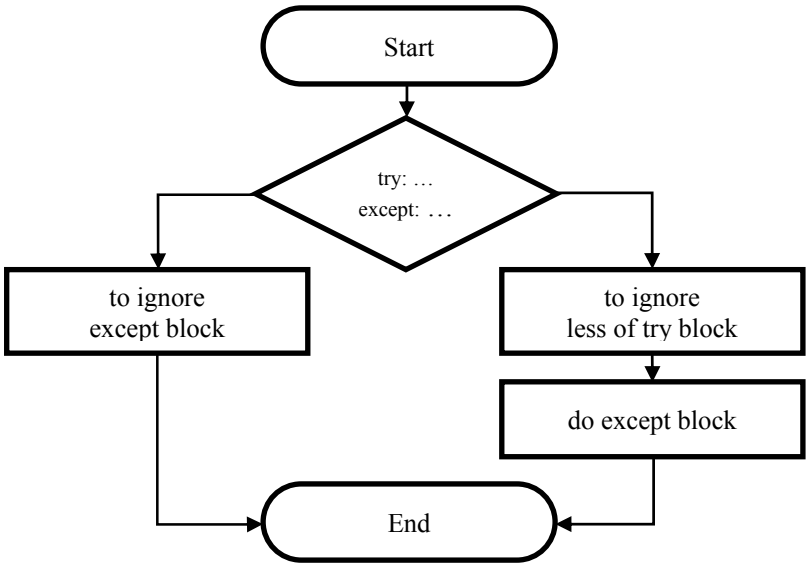


Рис. 3.3. Структура __init__

Блок схема environment.py (див. рис. 3.4.).

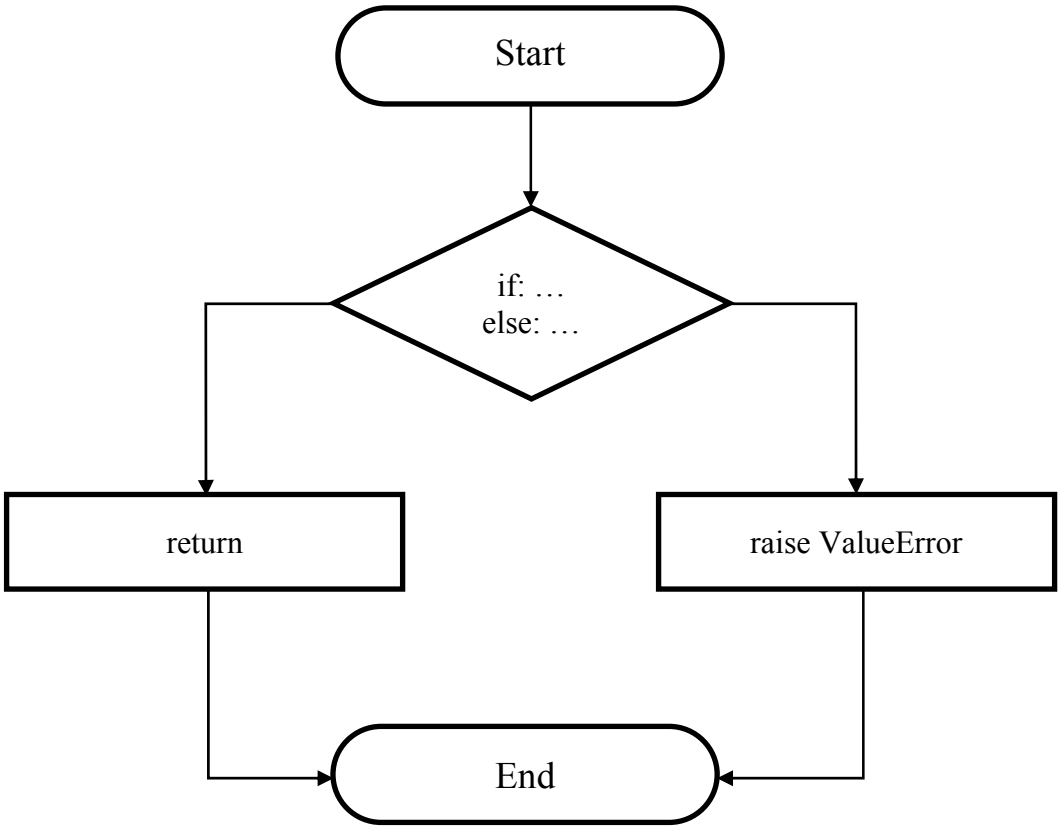


Рис. 3.4. Структура get_base_url

Код реалізація environment.py (див. рис. 3.5.).

```python
import os

You, 3 days ago | 1 author (You)
class Environment:
    PIPELINE = "pipeline"
    LOCAL = "local"
    DEV01 = "dev"

    URLS = {
        PIPELINE: f"http://app-provisioning-pipeline.gws.genesys.com",
        DEV01: f"http://docker-dev01.gws.genesys.com",
        LOCAL: f"http://localhost",
    }

    def __init__(self) -> None:
        try:
            self.env = os.environ["ENV"]
        except KeyError:
            self.env = self.PIPELINE

    def get_base_url(self) -> str:
        if self.env in self.URLS:
            return self.URLS[self.env]
        else:
            raise ValueError(f"Unknown value of ENV variable {self.env}.")


ENV_OBJECT = Environment()
```

Рис. 3.5. Реалізація environment.py

.

Для віддаленої роботи над оточенням необхідно встановлювати віддалену підключення - щоб сторонні користувачі не мали доступу до оточення.

setup_environment.py - скрипт реєструє нове підключення до оточення і створює нового клієнта для отримання токенів для авторизації та аутентифікації.

Блок схема скрипта setup_environment.py (див. рис. 3.6.), код реалізвція setup_environment.py скрипта (див. рис. 3.7.), блок схема з алгоритмом роботи функції: do_register_new_client (див. рис. 3.8.), код реалізація функції do_register_new_client (див. рис. 3.9.), структура процедури do_add_data_center (див. рис. 3.10.), код реалізація процедури do_add_data_center (див. рис. 3.11.).

Блок схема setup_environment.py (див. рис. 3.6.).

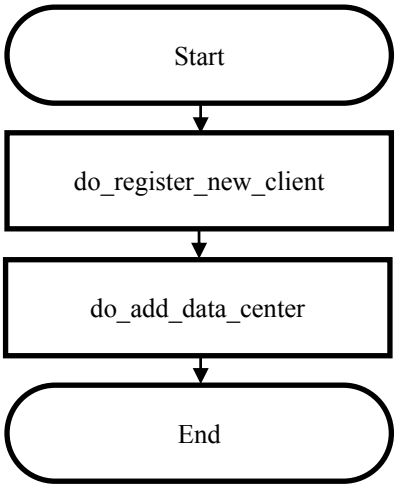


Рис. 3.6. Структура setup_environment.py

Код реалізація setup_environment.py (див. рис. 3.7.).

```python
def main() -> None:
    ingress_address = config_manager.get_provisioning_option("ingress_address")
    provisioning_host = config_manager.get_provisioning_option("provisioning_1_host")
    provisioning_port = config_manager.get_provisioning_option("provisioning_1_port")
    provisioning_1_management_port = config_manager.get_provisioning_option("provisioning_1_management_port")
    data_center_host = config_manager.get_environment_option("data_center_host")
    core_auth_host = config_manager.get_auth_option("auth_host")
    core_auth_port = config_manager.get_auth_option("auth_port")
    core_auth_management_port = config_manager.get_auth_option("auth_management_port")
    core_environment_host = config_manager.get_environment_option("env_host")
    core_environment_port = config_manager.get_environment_option("env_port")
    core_environment_management_port = config_manager.get_environment_option("env_management_port")
    external_api_client_id = config_manager.get_auth_option("external_api_client")
    basic_auth_password = config_manager.get_auth_option("ops_password")
    basic_auth_username = config_manager.get_auth_option("ops_username")
    grant_types = (
        "client_credentials",
        "authorization_code",
        "refresh_token",
        "implicit",
        "password",
    )
    redirect_uri = (
        data_center_host,
        f"http://{ingress_address}/",
        f"http://{ingress_address}:80/",
        "http://localhost/",
        f"http://{provisioning_host}",
        f"http://{provisioning_host}:{provisioning_port}",
        f"http://{provisioning_host}:{provisioning_1_management_port}",
        f"http://{core_auth_host}/",
        f"http://{core_auth_host}:{core_auth_port}/",
        f"http://{core_auth_host}:{core_auth_management_port}/",
        f"http://{core_environment_host}/",
        f"http://{core_environment_host}:{core_environment_port}/",
        f"http://{core_environment_host}:{core_environment_management_port}/",
    )

    auth_client = AuthClient(
        host=core_auth_host,
        port=core_auth_port,
        management_port=core_auth_management_port,
    )
    token = do_register_new_client(
        auth_client,
        basic_auth_username,
        basic_auth_password,
        external_api_client_id,
        grant_types,
        redirect_uri,
    )
    env_utils = RestClientEnv(
        core_environment_host,
        core_environment_port,
        core_environment_management_port,
        token,
    )
    do_add_data_center(env_utils, data_center_host)
```

Рис. 3.7. Реалізація setup_environment.py

Блок схема do_register_new_client (див. рис. 3.8.).

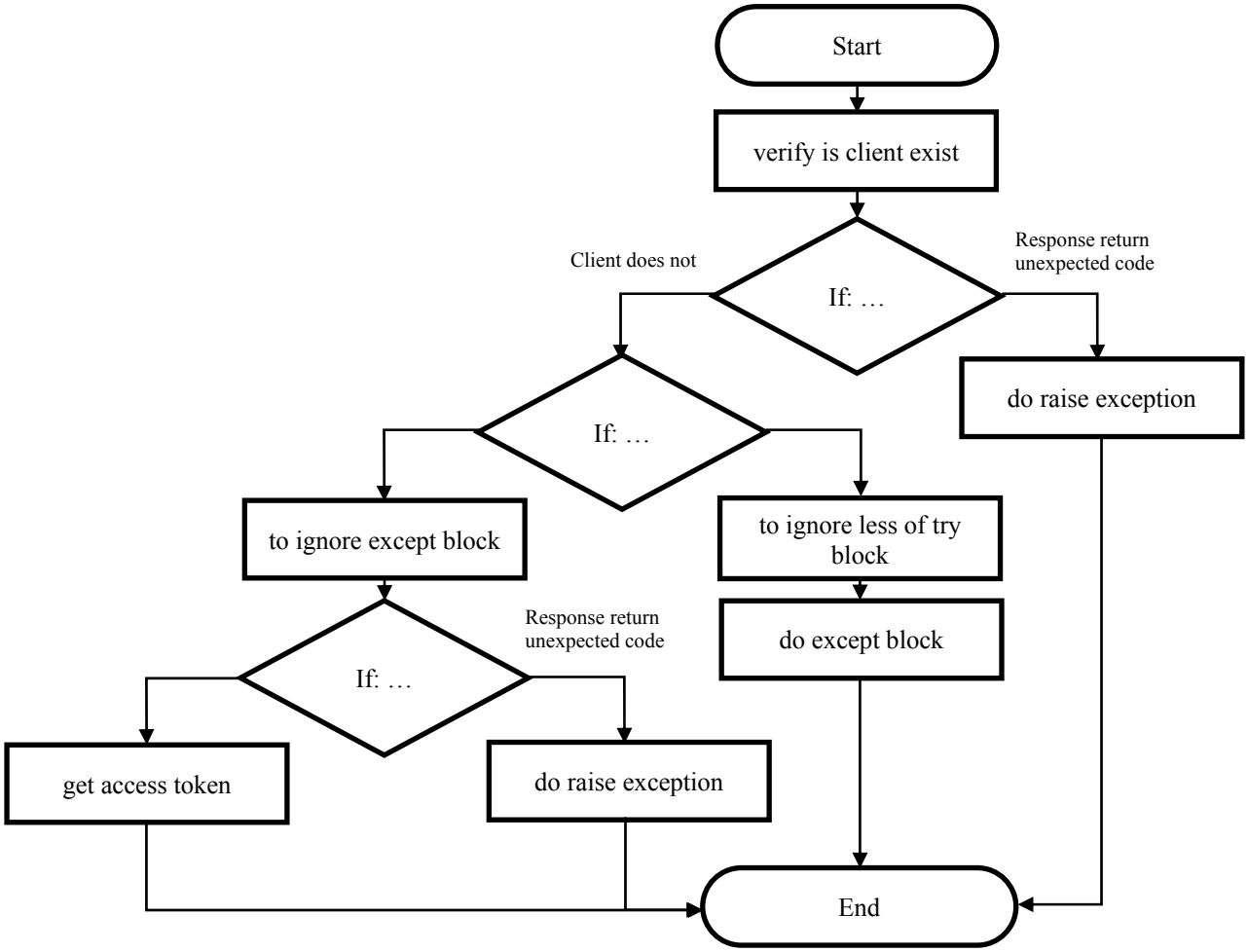


Рис. 3.8. Структура do_register_new_client

Код реалізація do_register_new_client (див. рис. 3.9.).

```
25  def do_register_new_client(
26      auth_client: AuthClient,
27      basic_auth_username: str,
28      basic_auth_password: str,
29      external_api_client_id: str,
30      grant_types: tuple,
31      redirect_uri: tuple,
32  ) -> AnyStr:
33
34      response = auth_client.get_client_by_id(
35          client_id=external_api_client_id,
36          auth=HTTPBasicAuth(basic_auth_username, basic_auth_password),
37      )
38      client = (
39          True
40          if response.status_code == 200
41          else False
42          if response.status_code == 404
43          else "Error"
44      )
45      if not client:
46          try:
47              register_response = auth_client.register_new_client(
48                  client_id=external_api_client_id,
49                  client_name=external_api_client_id,
50                  client_secret="secret",
51                  description=external_api_client_id,
52                  grant_types=grant_types,
53                  redirect_uri=redirect_uri,
54                  access_token_validity=None,
55                  refresh_token_validity=None,
56                  auth=HTTPBasicAuth(basic_auth_username, basic_auth_password),
57                  authorities=["ROLE_INTERNAL_CLIENT", "ROLE_MANAGEMENT_APPLICATION"],
58                  check_exist=True,
59              )
60          except Exception as e:
61              raise Exception("Unable to create client_id on auth service: {}".format(e))
62          if not any(
63              [
64                  "resource_already_exists" in str(register_response.content),
65                  '"code":0' in str(register_response.content),
66              ]
67          ):
68              raise Exception(
69                  "client_id neither exists nor was created. Response: {}".format(
70                      register_response.content
71                  )
72              )
73      elif client == "Error":
74          raise Exception(
75              "Get request to auth clients returned error: {}".format(response.content)
76          )
77
78      token = auth_client.get_access_token_resource_owner(
79          basic_auth_username, basic_auth_password
80      )
81
82      return token
```

Рис. 3.9. Реалізація do_register_new_client

Блок схема do_add_data_center (див. рис. 3.10.).

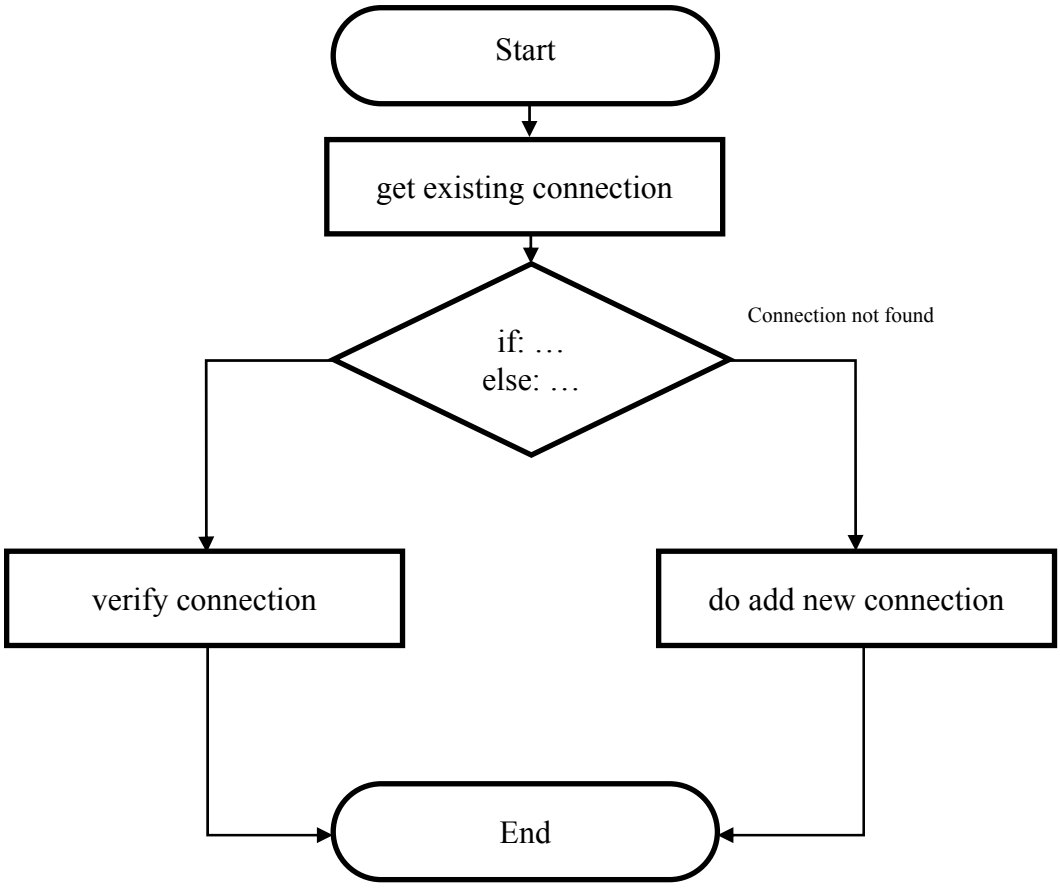


Рис. 3.10. Структура do_add_data_center

Код реалізація do_add_data_center (див. рис.3.11.) .

```
10  def do_add_data_center(env_utils: RestClientEnv, data_center_host: str) -> None:
11      dc1 = env_utils.get_data_centers(location="location/usw1")
12      if (dc1.status_code == 404) and ("not found" in str(dc1.content)):
13          env_utils.add_data_center(data_center_host, "/USW2", True)
14          env_utils.add_data_center(data_center_host, "/USW1", False)
15      else:
16          dc2 = env_utils.get_data_centers(location="location/usw2")
17          assert dc1.status_code == 200
18          assert dc2.status_code == 200
19          assert '"code":0' in str(dc1.content)
20          assert '"code":0' in str(dc2.content)
21          assert '"location":"/usw1"' in str(dc1.content)
22          assert '"location":"/usw2"' in str(dc2.content)
```

Рис. 3.11. Реалізація do_add_data_center

Скрипти для підтримки розробки включають: dev.build.gradle, test.build.gradle, Dockerfile.tests, Dockerfile.tests.python38, Dockerfile.dev.

Dockerfile.tests - виконується збір базового образу тестового фреймворку на

базі Ubuntu Focal Fosa - даний образ був обраний як базовий - оскільки для тестів використовується python3.8, а дана версія Ubuntu з коробки має python3.8 - а це означає, що достатнім буде підтягнути лише потрібні залежності для проекту. Код описаного образу (див. рис. 3.13.).

Також використовується образ саме дистрибутива Linux, а не самого Python - з тих міркувань, що образ буде розгорнутий на віддаленому оточенні і в будь-який момент може знадобитися попрацювати з проектом всередині самого контейнера – досить зручно коли підключаючись до контейнера - можна отримати повний функціонал дистрибутива Linux.

Не дивлячись на це - так само використовується збірка на образі самого python3.8(див. рис. 3.12.) - зроблено це для зручності запуску конкретних тестів - коли цікавить тільки результат виконання, тобто не потрібно редагувати код. Також цей підхід дозволяє з'єднати робочу директорію з докер контейнером (не передаючи локальні дані усередину контейнера) - а прокласти міст між контейнером та хост-машиною та при роботі відразу бачити результат виконання без створення та оновлення до нової версії докеру образу.

Dockerfile.dev - образ данного контейнера реалізован таким чином, що при запускі самого контейнера одразу стартує додаток на відповідних портах - доступних до даного розробленного додадка (див. рис. 3.14.).

Код реалізація Dockerfile.tests.python38 (див. рис. 3.12.).

```
1   FROM python:3.8
2
3   WORKDIR /gws_app_provisioning_functional_tests/
4
5   COPY requirements/ .
6   RUN pip install --requirement requirements.txt
7
8   ENV RESULT_PATH=/gws_app_provisioning_functional_tests/reports
9   ENV SUITE_NAME=test_users_permissions
10  ENV ENV=dev01
11
12  CMD python -m pytest -s -v --alluredir=$RESULT_PATH/allure_reports/$SUITE_NAME /gws_app_provisioning_functional_tests/tests/$SUITE_NAME
13
```

Рис. 3.12. Реалізація Dockerfile.tests.python38

Код реалізація Dockerfile.tests (див. рис. 3.13.).

```
FROM ubuntu:focal

RUN apt-get update && apt-get upgrade -y
RUN apt-get install software-properties-common -y
RUN apt-get install -y \
  curl \
  git \
  build-essential \
  python3-dev \
  python3-distutils \
  wget

RUN mkdir /gws_app_provisioning_functional_tests
COPY . /gws_app_provisioning_functional_tests

WORKDIR /gws_app_provisioning_functional_tests

RUN wget https://bootstrap.pypa.io/get-pip.py
RUN python3 get-pip.py
RUN pip3 install --upgrade pip
RUN pip3 install --requirement requirements/requirements.txt

```

Рис. 3.13. Реалізація Dockerfile.tests

Код реалізація Dockerfile.dev (див. рис. 3.14.).

```
FROM gws-docker-registry.artifactory.gws.genesys.com/stable/base/gws-base-provisioning
ARG _project_name
ARG _project_version
ARG _user=500

EXPOSE 8027 9027 18027

CMD ["--max_old_space_size=16384", "app.js", "--cacheTempDir=:memory:"]

ENV GWS_SERVICE_NAME=${_project_name} \
    GWS_SERVICE_VERSION=${_project_version}

COPY --chown=${_user}:root dummy ./dummy

COPY --chown=${_user}:root build/webapp ./

RUN chmod -R ug+rw /opt/genesys/application

```

Рис. 3.14. Реалізація Dockerfile.dev

Як уже згадувалося, для швидкості та простоти розгортання використовується Dcoker. Безсумнівно складання образів можна здійснювати

засобами самого докера – але це не буде настільки зручним упорівнянні з тими можливостями, які надає gradle: підключення сторонніх бібліотек для роботи з розливними складаннями. Підключення і легке використання репозиторіїв для отримання і розповсюдження об'єктів та зручний функціонал з додавання і виключення файлів для складання разом з користувальницькими функціями та системами архівації проектів при складанні.

test.build.gradle: збирає образ розроблюваного фреймворку для тестування (див. рис. 3.15.). Код реалізація кожного с етпів - демонструється на рисунках 3.16.-3.18.

Це дозволить використовувати цей образ для тестування продукту на будь-якому з існуючих оточень. Для чого достатньо буде лише отримати зібраний і образ і запустити на потрібному оточенні для проведення тестування.

Блок схема test.build.gradle (див. рис. 3.15.).

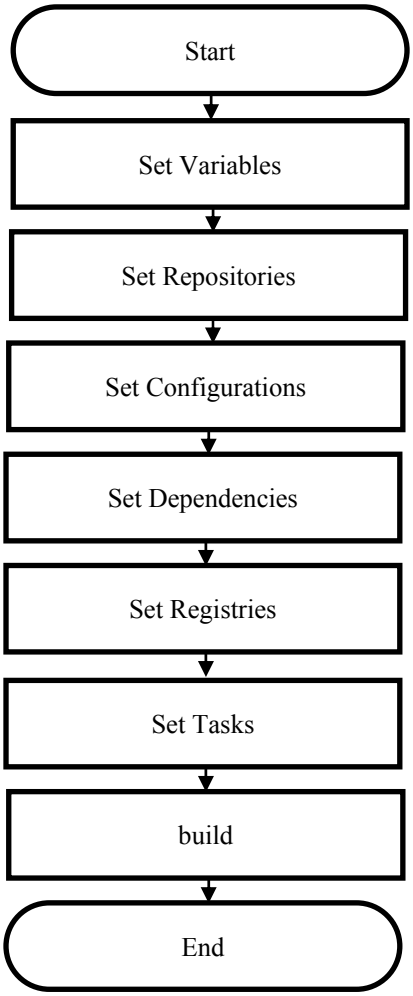


Рис. 3.15. Структура test.build.gradle

Код реалізація Set (див. рис. 3.16.).

```
File versionFile = file("VERSION.TXT")
String baseVersion = versionFile.readLines().get(0)

String buildNumber = System.getenv("BUILD_NUMBER")
if (buildNumber == null || buildNumber.isAllWhitespace()) {
    buildNumber = "dev"
}

repositories {
    ivy {
        url "http://artifactory.gws.genesys.com/artifactory"
        patternLayout {
            artifact "[organisation]/[module](-[revision]).[ext]"
        }
    }
}

configurations {
    tar
}

dependencies {
    tar "gws-generic-tools-local:gws-qaart@tar.gz"
}

version = "${baseVersion}.${buildNumber}"
group = "com.genesys.gws.v3.microservices"

def dockerRegistry = "gws-docker-registry.artifactory.gws.genesys.com"
def dockerImageName = "${dockerRegistry}/gws-ui-provisioning-functional-tests"

task downloadQaart {
    outputs.dir "${buildDir}/qaart"
    doFirst {
        configurations.tar.forEach { archive ->
            copy {
                println "[${System.currentTimeMillis()}] ARCHIVE: '${archive}'"
                from tarTree(resources.gzip(archive))
                into "${buildDir}/qaart"
            }
        }
    }
}
```

Рис. 3.16. Реалізація test.build.gradle Set … stages

Код реалізація Tasks (див. рис. 3.17.).

```
46  task copyProject(type: Copy) {
47      description = "Copies necessary test files to the Docker context directory"
48
49      into "${buildDir}/gws-ui-provisioning-functional-tests"
50      exclude "**/.*"
51      exclude "**/*.pyc"
52      exclude "**/Dockerfile*"
53      exclude "**/requirements.txt"
54      exclude { details -> details.file.name.contains('docker-compose') }
55
56      def folderList = ["tests", "utils"]
57      folderList.each() { folderName ->
58          from(folderName) {
59              into folderName
60          }
61      }
62
63  }
64
65  task copyContext(type: Copy) {
66      description = "Copies all dependencies to the Docker context directory"
67      from "Dockerfile"
68      into buildDir
69      dependsOn "downloadQaart"
70      dependsOn "copyProject"
71  }
72
73  task buildImage(type: DockerBuildImage) {
74      description = "Builds and tags the Docker image with 'version' and 'latest'"
75      dependsOn copyContext
76      inputDir = buildDir
77      images.addAll(["${dockerImageName}:${version}", "${dockerImageName}"])
78  }
79
80  task pushImage(type: DockerPushImage) {
81      description = "Pushes the Docker image to ${dockerRegistry}"
82      images = buildImage.getImages()
83  }
84
85  task removeImage(type: DockerRemoveImage) {
86      description = "Removes the Docker image from file system"
87      force = true
88      imageId = buildImage.getImageId()
89  }
```

Рис. 3.17. Реалізація test.build.gradle Set … task

Код реалізація build (див. рис. 3.18.).

```
90
91  build {
92      dependsOn buildImage
93      dependsOn pushImage
94      dependsOn removeImage
95  }
96
```

Рис. 3.18. Реалізація test.build.gradle build

dev.build.gradle: збирає образ програми, що розробляється разом з частиною модульного тестування цієї програми. Блок схема етапів скрипта (див. рис. 3.19.), код реалізація кожного з етпів – демонструється на рисунках 3.20.-3.22.

Це гарантує що отримання збірка програми вже буде протестована на рівні коду і готова для розгортання в інших використовуваних оточеннях - відмінних від локального та оточення-пісочниці.

Блок схема dev.build.gradle (див. рис. 3.19.).

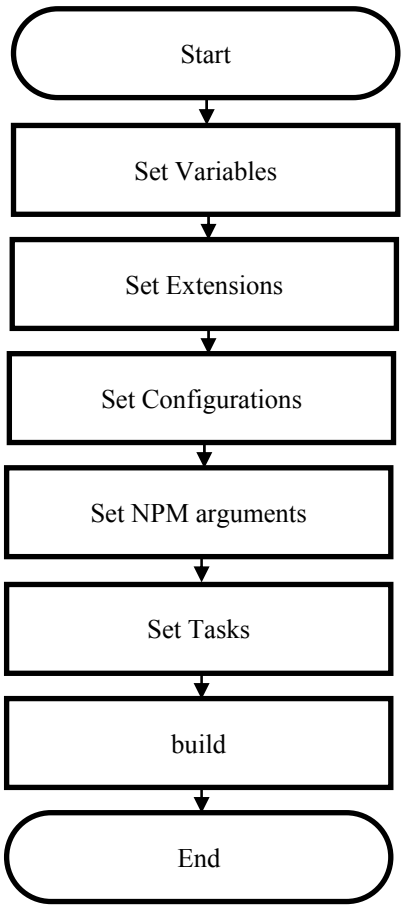


Рис. 3.19. Структура dev.build.gradle

Код реалізація Tasks (див. рис. 3.20.).

```
File versionFile = file("VERSION.TXT")
String baseVersion = versionFile.readLines().get(0)

String buildNumber = System.getenv("BUILD_NUMBER")
if (buildNumber == null || buildNumber.isAllWhitespace()) {
    buildNumber = "dev"
}

ext {
    baseVersion = readVersion.baseVersion.get()
}

group = "com.genesys.gws.v3.microservices"
version = "${project.baseVersion}.${readVersion.buildNumber.get()}"

npm_install {
    args = ["--userconfig=.npmrc"]
}
```

Рис. 3.20. Реалізація dev.build.gradle: Set stages

Код реалізація Tasks (див. рис. 3.21.).

```
task gruntBuild(type: GruntTask) {
    dependsOn = ["npm_install"]
    group "Build"
    description "Build typescript code"
    args = ["--buildVersion=${version}", "--mercurialChangeset=${System.getenv("BUILD_CHANGESET_ID")}"]
}

task copyCoverageFile(dependsOn: ["clean"]) {
    doFirst() {
        file(".dockerignore").renameTo(file(".dockerignore.old"))
        createDockerfile.copyFile("start.sh", "/opt/genesys/application/")
        createDockerfile.user("root")
        createDockerfile.copyFile("package.json", "/opt/genesys/application/")
        createDockerfile.entryPoint('./start.sh')
    }
}
createDockerfile.shouldRunAfter copyCoverageFile

grunt_test.dependsOn = ["npm_install"]

task test {
    dependsOn "grunt_test"
}

build.dependsOn = ["gruntBuild", "test"]

project.tasks["sonarqube"].dependsOn "build"
sonarqube {
    properties {
        property "sonar.projectVersion", project.baseVersion
        property "sonar.scm.provider", "git"
        property "sonar.sourceEncoding", "UTF-8"
        property "sonar.sources", "src"
        property "sonar.language", "ts"
        property "sonar.ts.tslint.configPath", "${project.projectDir}/grunt/config/tslint.json"
        property "sonar.typescript.lcov.reportPaths", "${project.buildDir}/reports/core/coverage/node/lcov.info"
    }
}
```

Рис. 3.21. Реалізація dev.build.gradle: Set Task

Код реалізація buildDockerImage (див. рис. 3.22.).

```
buildDockerImage {
    dependsOn "createVendorInfoFile"
    baseVersion = project.baseVersion
    buildChangeSetId = System.getenv("BUILD_CHANGESET_ID") ?: "unspecified"
}
```

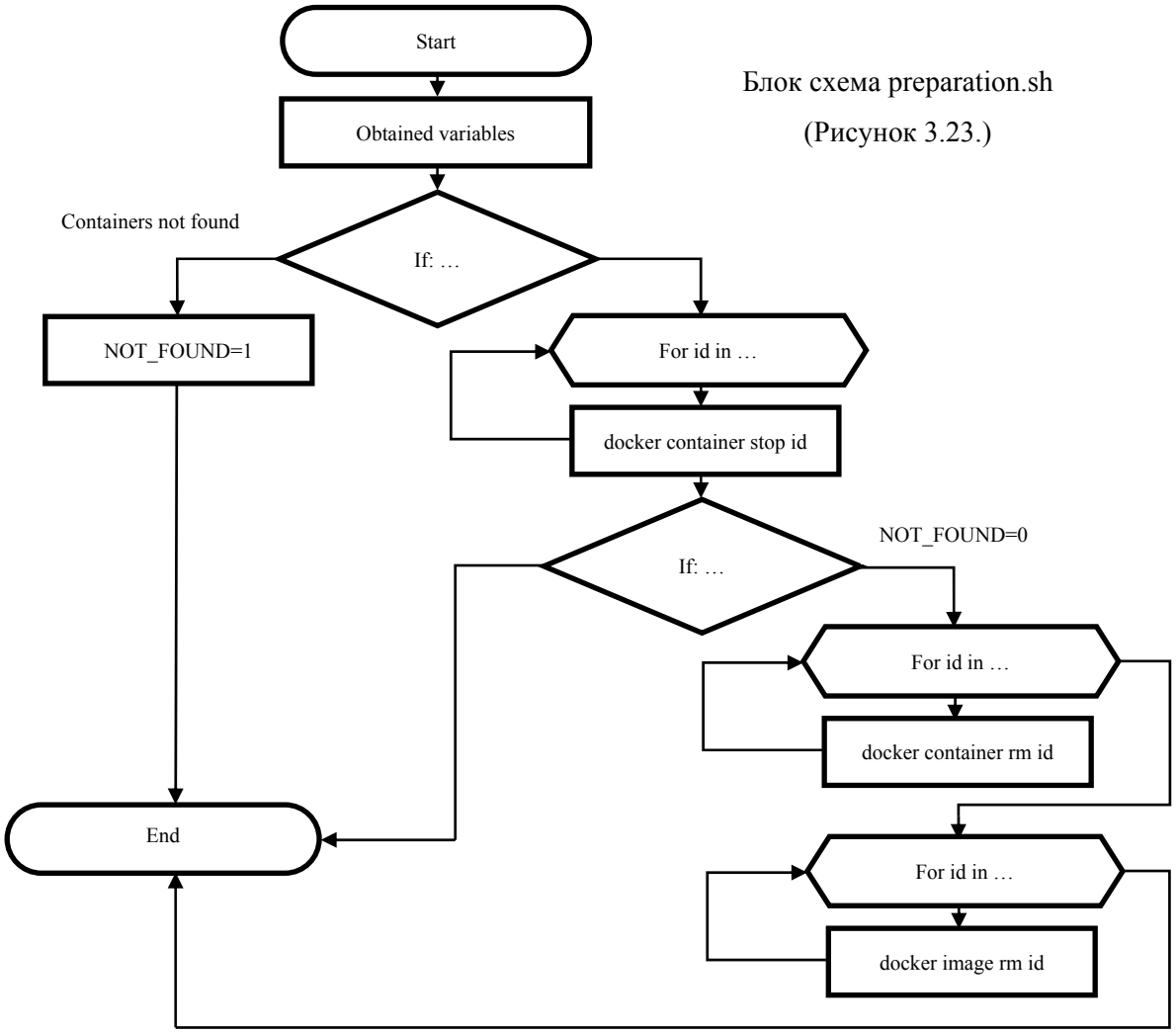
Рис. 3.22. Реалізація dev.build.gradle: build

Скрипти для підтримки оточення: preparation.sh, pull_services.sh.

В основному сервера, войс машини використовують/базуються на Linux через свою легковагість і надійність. Так само в рамках економії місця і через непотрібність тримати в образі або на сервері купу утилі - функціонал дуже мізерний: софт або образи для програми і деякі базові програми/утиліти.

І часто не відомо яке програмне забезпечення встановлено на сервері - та й в іншому і не потрібно - достатньо лише тієї інформації що розгорнут Linux і вести автоматизацію можна безпосередньо в командній оболонці, що надається з Linux: а це буде Bash - рідше Shell - різниця в функціоналі– який в основному не заважає створювати скрипти автоматизації для даної оболонки.

Так як preparation.sh, pull_services.sh скрипти будуть запущені на віддаленому сервері - швидкий і розумний спосіб писати їх на ресурсах, що надаються. Алгоритм роботи скрипта для підготовки оточення для розгорнення сервісів (див. рис. 3.23.), код реалізацію даного алгоритму (див. рис. 3.24.).



Блок схема preparation.sh

(Рисунок 3.23.)

Код реалізація preparation.sh (див. рис. 3.24.).

```
#!/bin/bash

CONTAINERS_ID=$(docker ps -a | awk '{print $1}' | tr '\n' ' ')
WITHOUT_CONTAINER_WORD=$(sed 's/CONTAINER//g' <<< "$CONTAINERS_ID")
IFS=' ' read -r -a CIDS <<< "$WITHOUT_CONTAINER_WORD"

IMAGES_ID=$(docker images | awk '{print $3}' | tr '\n' ' ')
WITHOUT_IMAGES_ID_WORDS=$(sed 's/IMAGE ID//g' <<< "$IMAGES_ID")
IFS=' ' read -r -a IIDS <<< "$WITHOUT_IMAGES_ID_WORDS"

CONTAINERS_STATUS=$(docker ps -a | awk '{print $6}')

if [[ $CONTAINERS_STATUS != STATUS ]];
then
  echo "Start stopping containers...";
  for id in "${CIDS[@]}"; do echo "Containers are still stopping..."; docker container stop "$id"; done
  echo "Done!"
else
  echo "Any containers are not found."
  NOT_FOUND=1
fi

if [[ ! $NOT_FOUND ]]; then
  echo "Start removing containers...";
  for id in "${CIDS[@]}"; do echo "Containers are still removing..."; docker container rm "$id"; done
  echo "Done!"

  echo "Start removing images...";
  for id in "${IIDS[@]}"; do echo "Images are still removing..."; docker image rm "$id"; done
  echo "Done!"
fi
```

Рис. 3.24. Реалізація preparation.sh

pull_services.sh - в свою чергу відповідає за розгортування оточення на запущеній машині та звязку контейнерів з IP-адресой машини на якій був розгорнутий контейнер. Блок схема скрипта – відображена на рисунку 3.25., код реалізація (див. рис. 3.26.).

Блок схема pull_services.sh (див. рис. 3.25.).

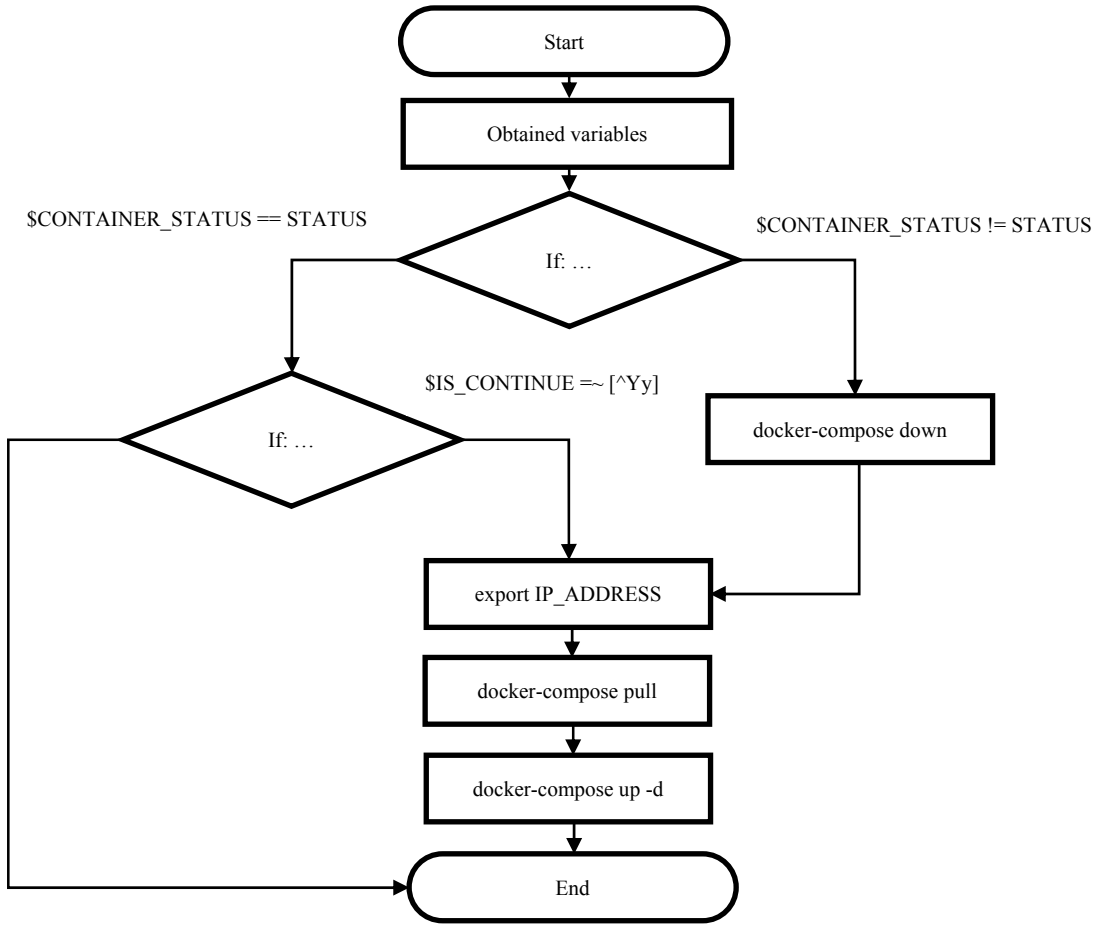


Рис. 3.25. Структкура pull_services.sh

Код реалізація pull_services.sh (див. рис. 3.26.).

```
#!/bin/bash

CONTAINERS_STATUS=$(docker ps -a | awk '{print $6}')

if [[ $CONTAINERS_STATUS != STATUS ]];
then
    docker-compose down
    unset IP_ADDRESS
else
    echo "Not of one containers are exist."
    read -r -p "Pull and Up services? [Y/N]: " IS_CONTINUE
    if [[ $IS_CONTINUE =~ [^Yy] ]]; then return; fi;
fi

echo "Getting IP Address..."
export "IP_ADDRESS=$(ifconfig en0 | grep 'inet ' | awk '{print $2}')"

echo "Pull services..."
docker-compose pull

echo "Up services on $IP_ADDRESS..."
docker-compose up -d

echo "Done."

```

Рис. 3.26. Реалізація pull_services.sh

3.2 Налаштування CI/CD Pipeline Job

Використання скриптів - дуже зручне, і дозволяє прискорити виконання процесів. Скрипти націлені на підтримку постійної інтеграції та розгортки програми та нових версій. Для автоматизації, менеджменту та збірки всіх скриптів у фреймврці - задіяний Jenkins - який дозволяє за допомогою опису спеціальних Pipeline скриптів: Jenkinsfile.dev, Jenkinsfile.tests - скомбінувати, зібрати і запустити в одному місці зі зручними інеструментами менеджера проекту з системи автоматизації.

Щоб описати CI/CD Pipeline Job - необхідно налаштувати конфігурацію необхідного сервісу, для цього:

1. Потрібно створити Pipeline Job;
2. Підключення ГІТ репозиторію для отримання файлів проекту;
3. Опис пайплайн-скрипта.

Створення Pipeline Job.

Перейти за вкладкою New Item – щоб потрапити на сторінку створення нової Pipeline Job (див. рис. 3.27.).

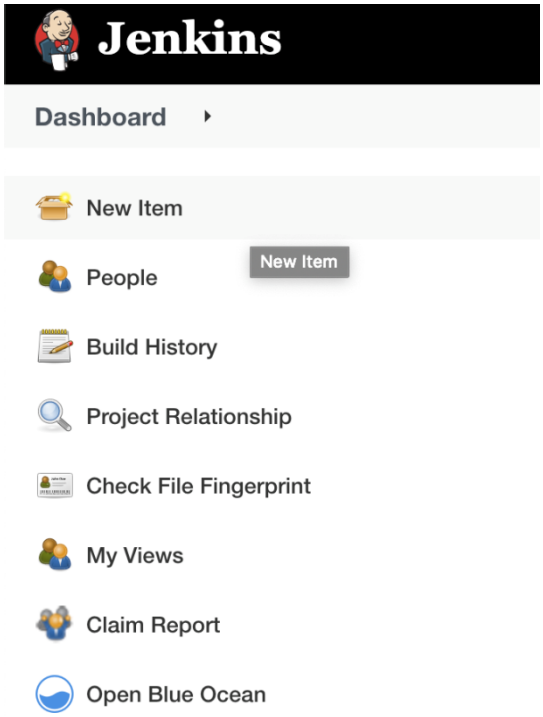


Рис. 3.27. Бокове меню головної сторінки Jenkins

Ввести ім’я нової Job та вибрати шаблон Freestyle project – для налаштування унікальної конфігурації. Вигляд меню для створення нової Job (див. рис. 3.28.).

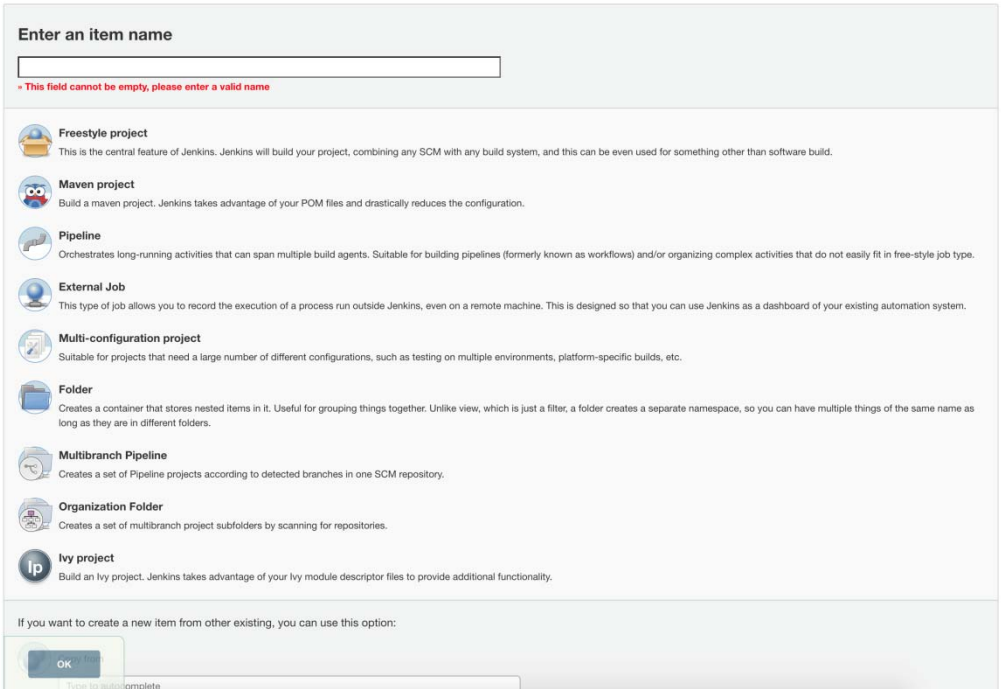


Рис. 3.28. Меню створення Jenkins Job

Ввести необхідні данні для підключення GIT репозитрію та вказати шлях до знаходження Jenkinsfile(s) – в якому описане поводження данної Job та зберегти зміни. Меню конфігурації (див. рис. 3.29.).

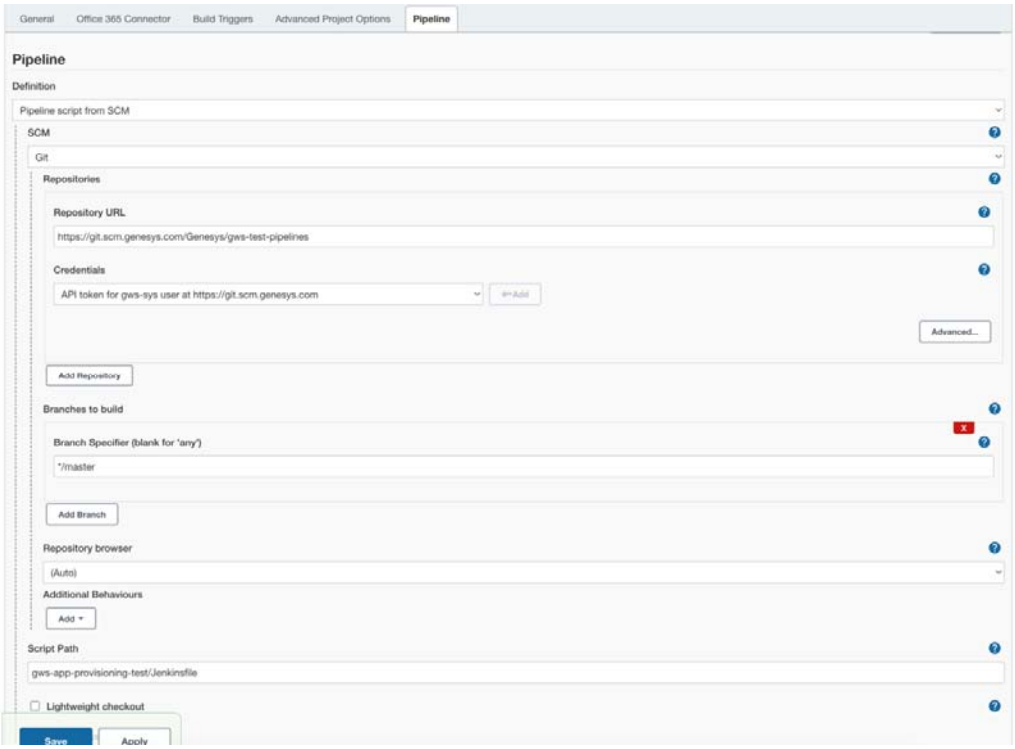


Рис. 3.29. Конфігураційне меню створеної Jenkins Job

Після проведених шагів конфігурації - отримаємо готову моніторінг - менеджемнтову автоматизовану систему CI/CD проекту, як для розробки так і для тестування. Зображення створенної системи на базі Jenkins - продемонстрованно на зображеннях 3.30.-3.31.

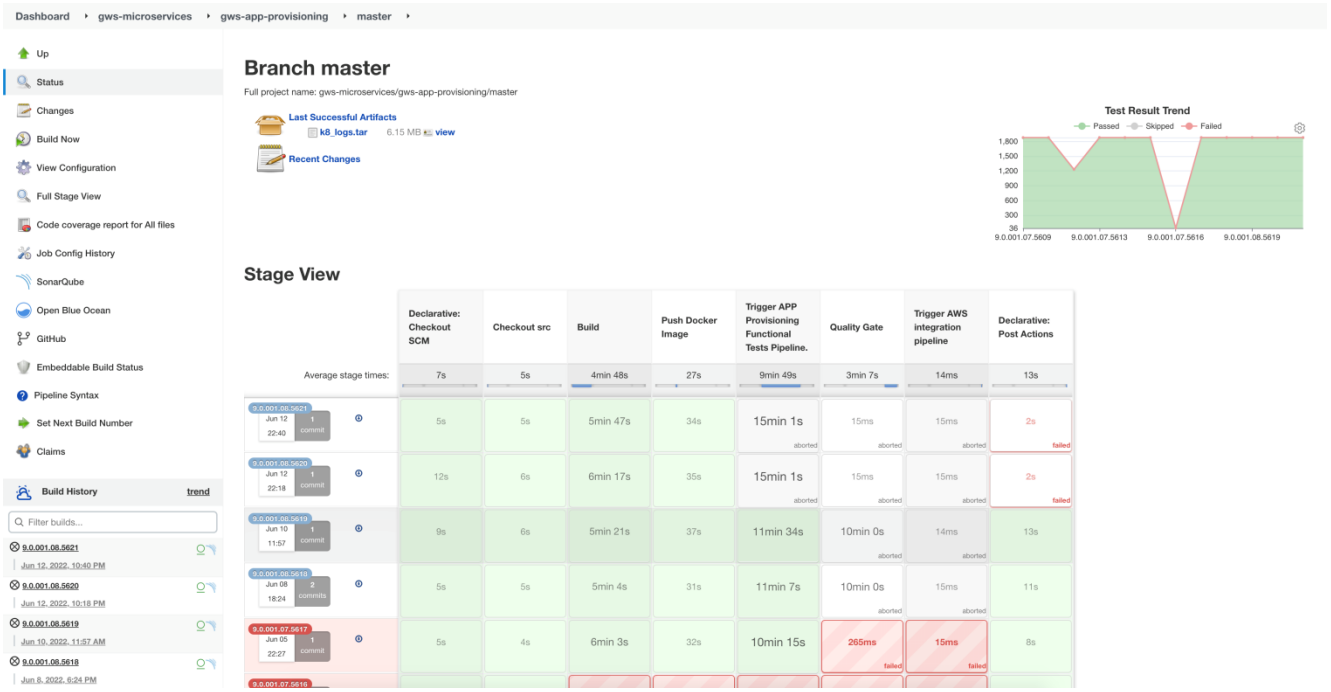


Рис. 3.30. Dev Jenkins Job з Pipeline плагіном

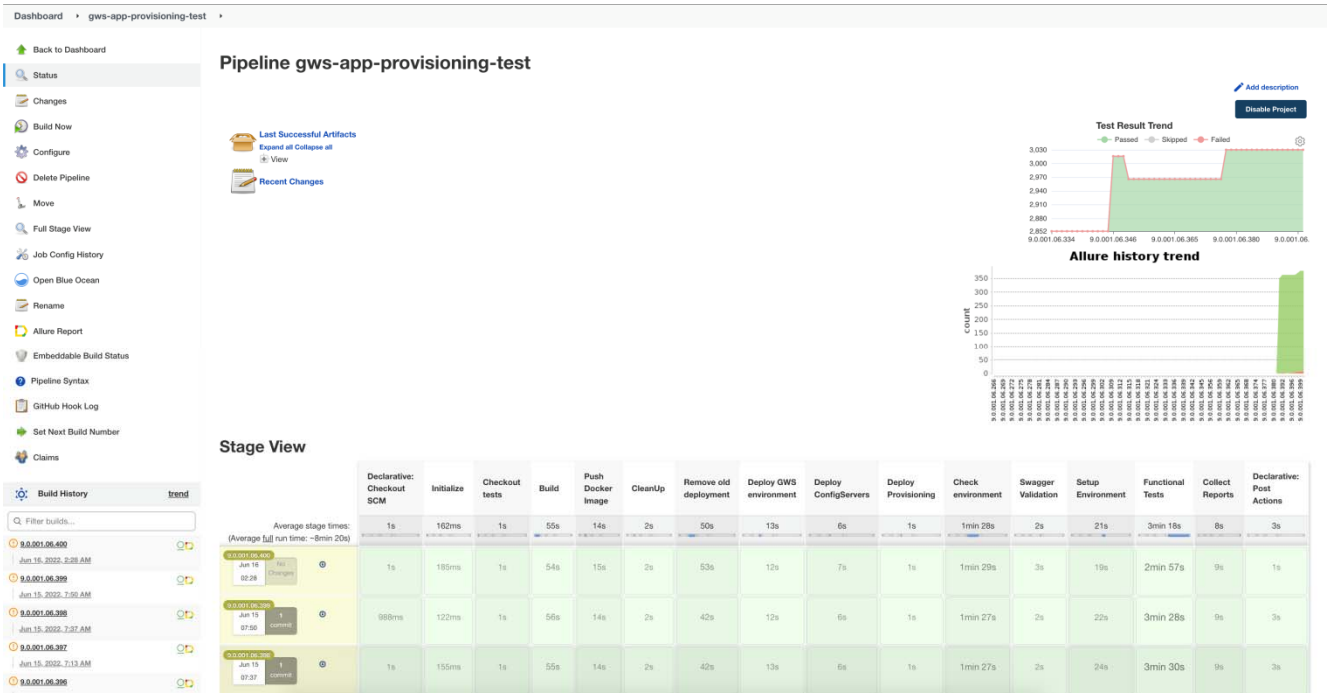


Рис. 3.31. Tests Jenkins Job з Pipeline плагіном

Використана конфігурація для CI/CD Job для автоматизації розробки Jenkinsfile.dev - pipeline скрипт написанний за підтримкої мови програмування Groovy та з використанням декларативного синтаксису. Стркутура скрипта вказана на рисунку 3.32. Реалізація коду етапів скрипту - продемонстрована на зображеннях 3.33.-3.38.

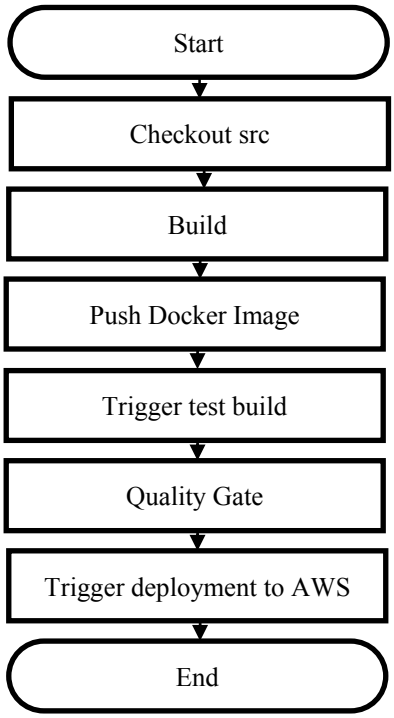


Рис. 3.32. Структура Jenkinsfile.dev

Checkout src: перевірка на актуальність і наявність вітки у репозіторії (див. рис. 3.33).

```
27      stage("Checkout src") {
28          steps {
29              checkout([
30                      $class                           : 'GitSCM',
31                      branches                         : scm.branches,
32                      doGenerateSubmoduleConfigurations: scm.doGenerateSubmoduleConfigurations,
33                      extensions                       : scm.extensions + [
34                              [$class: 'CleanBeforeCheckout'],
35                              [$class: 'RelativeTargetDirectory', relativeTargetDir: COMPONENT_NAME]
36                      ],
37                      userRemoteConfigs                : scm.userRemoteConfigs
38              ])
39
40              script {
41                  gwsAbortBuildTriggeredByFreeze()
42                  env.BUILD_VERSION = "${readFile("${COMPONENT_NAME}/VERSION.TXT").trim()}.${BUILD_NUMBER}"
43                  currentBuild.displayName = env.BUILD_VERSION
44              }
45          }
46      }
```

Рис. 3.33. Реалізація Jenkinsfile.dev: Checkout src

Build: збірка проекту розробки за допомогую gradle.dev та збір результатів (див. рис. 3.34.).

```
stage("Build") {
    steps {
        script {
            buildInfo.env.capture = true
            buildInfo.retention maxBuilds: 15, maxDays: 30, deleteBuildArtifacts: true

            def buildGradle = Artifactory.newGradleBuild()
            buildGradle.resolver server: ARTIFACTORY_SERVER, repo: 'libs-snapshot'
            buildGradle.tool = "5.4.1"

            try {
                withEnv(["BUILD_CHANGESET_ID=${GIT_COMMIT}"]) {
                    nodejs(configId: 'npm-internal-config', nodeJSInstallationName: '16.14.0') {
                        withSonarQubeEnv {
                            def gradleSwitches = [
                                    "--no-daemon",
                                    "--info",
                                    "--refresh-dependencies",
                                    "--rerun-tasks",
                                    "-Dsonar.host.url=${SONAR_HOST_URL} -Dsonar.login=${SONAR_AUTH_TOKEN}"
                            ]

                            dir(COMPONENT_NAME) {
                                buildGradle.run switches: gradleSwitches.join(" "),
                                        tasks: "clean buildDockerImage sonarqube -x artifactoryPublish",
                                        buildInfo: buildInfo
                            }

                            buildInfo.env.collect()
                        }
                    }
                }
            } catch (err) {
                stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
                throw err
            }
        }
    }

    post {
        always {
            publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: "${COMPONENT_NAME}/build/reports/core/coverage/node/lcov-report/",
                         reportFiles : "index.html", reportName: "Code coverage report for All files"])
            junit allowEmptyResults: true, keepLongStdio: true, testResults: "${COMPONENT_NAME}/build/test-results/test/*.xml"
        }
    }
}
```

Рис. 3.34. Реалізація Jenkinsfile.dev: Build

Push Docker Image: відправка зібраного образу проекту до репозиторію (див. рис. 3.35.).

```

stage("Push Docker Image") {
    when {
        branch 'master'
    }

    steps {
        script {
            def rtDocker = Artifactory.docker server: ARTIFACTORY_SERVER

            env.FULL_IMAGE_NAME = "gws-docker-registry.artifactory.gws.genesys.com/${COMPONENT_NAME}"
            sh "docker tag ${FULL_IMAGE_NAME}:latest ${FULL_IMAGE_NAME}:${BUILD_VERSION}"

            rtDocker.push "${FULL_IMAGE_NAME}:latest", "gws-docker-registry"
            rtDocker.push "${FULL_IMAGE_NAME}:${BUILD_VERSION}", "gws-docker-registry", buildInfo

            setArtifactProperties artifactoryId: "artifactory.gws.genesys.com", credentialsId: "gws-artifactory-api", repository: "gws-docker-registry-local",
                    artifactName: COMPONENT_NAME, artifactVersion: BUILD_VERSION,
                    properties: [
                            "revision"     : GIT_COMMIT,
                            "build.name"   : JOB_NAME,
                            "build.number" : BUILD_NUMBER,
                            "build.version": BUILD_VERSION,
                            "qualityGate"  : "pass",
                            "status"       : "new"
                    ]
            ARTIFACTORY_SERVER.publishBuildInfo buildInfo
        }
    }
}

```

Рис. 3.35. Реалізація Jenkinsfile.dev: Push Docker Image

Trigger test build: запуск Pipeline Job для тестування продукту (див. рис. 3.36.).

```
125
126         stage("Trigger APP Provisioning Functional Tests Pipeline.") {
127             when {
128                 allOf {
129                     branch 'master'
130                     not { triggeredBy 'UpstreamCause' }
131                 }
132             }
133
134             options {
135                 timeout(time: 15, unit: "MINUTES")
136             }
137
138             steps {
139                 runFunctionalTestsInParallel testJobs: testJobs, componentName: COMPONENT_NAME,
140                         buildVersion: BUILD_VERSION, artifactoryServerId: ARTIFACTORY_SERVER_ID
141             }
142         }
143
```

Рис. 3.36. Реалізація Jenkinsfile.dev: Trigger test build

Quality Gate: збереження успішно протестованої версії продукти та очистка оточення від використаних образів (див. рис. 3.37.).

```
143
144     stage("Quality Gate") {
145         when {
146             branch 'master'
147         }
148
149         options {
150             timeout(time: 10, unit: "MINUTES")
151         }
152
153         steps {
154             dir("gws-app-provisioning"){
155                 gwsQualityGate configFilePath: "${COMPONENT_NAME}.yaml", image: "${FULL_IMAGE_NAME}:${BUILD_VERSION}"
156             }
157
158             sh "docker rmi -f ${FULL_IMAGE_NAME}:${BUILD_VERSION}"
159             sh "docker rmi -f ${FULL_IMAGE_NAME}:latest"
160         }
161     }
162
```

Рис. 3.37. Реалізація Jenkinsfile.dev: Quality Gate

Trigger deployment to AWS: розгортування та запуск стабільної версії продукту в оточені блізькому для користувальницького (див. рис. 3.38.).

```
162
163     stage("Trigger AWS integration pipeline") {
164         when {
165             allOf {
166                 branch 'master'
167                 not { triggeredBy 'UpstreamCause' }
168                 expression { currentBuild.result == "SUCCESS" }
169             }
170         }
171
172         steps {
173             echo "INFO: The build was triggered manually or by SCM change, invoke an integration pipeline."
174             build job: "gws-integration-aws-pipeline",
175                     wait: false,
176                     parameters: [
177                             string(name: "componentName", value: COMPONENT_NAME),
178                             string(name: "fullComponentVersion", value: BUILD_VERSION),
179                             string(name: "componentImageNames", value: COMPONENT_NAME)
180                     ]
181         }
182     }
183 }
```

Рис. 3.38. Реалізація Jenkinsfile.dev: Trigger deployment to AWS

Використана конфігурація для CI/CD Job для автоматизації тестування

Jenkinsfile.tests - pipeline скрипт написанний за підтримки мови програмування Groovy та з використанням декларативного синтаксису. Стркутура скрипта вказана на рисунку 3.39. Реалізація коду етапів скрипту - продемонстрована на зображеннях 3.40.-3.53.

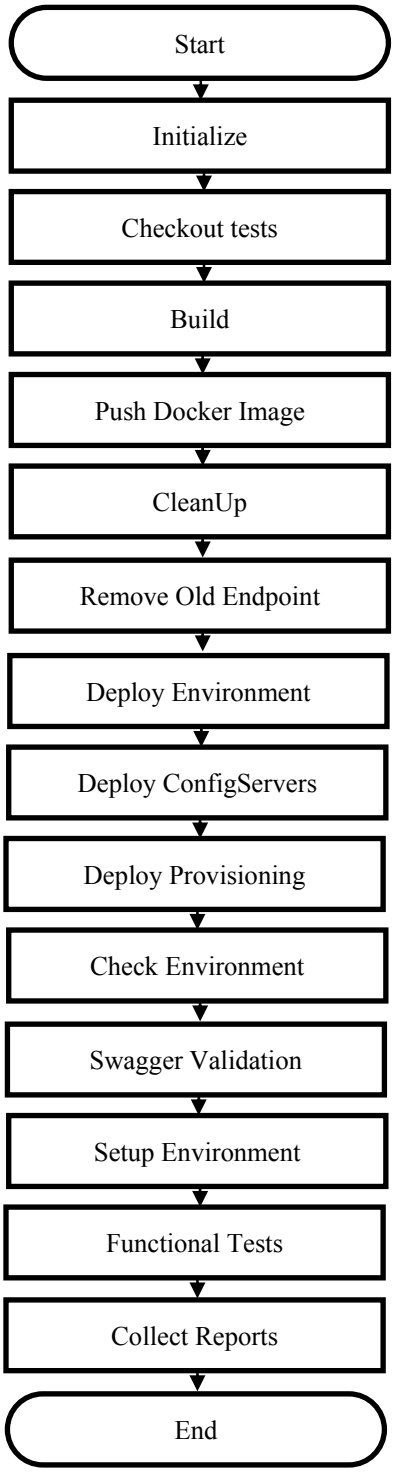


Рис. 3.39. Структура Jenkinsfile.tests

Initialize: ініціалізація глобальних змінних для позначення збірки (див. рис. 3.40.).

```
72      stage('Initialize') {
73          steps {
74              script {
75                  gwsAbortBuildTriggeredByFreeze()
76                  env.BUILD_VERSION = "9.0.001.06.${BUILD_NUMBER}"
77                  currentBuild.displayName = env.BUILD_VERSION
78              }
79          }
80      }
```

Рис. 3.40. Реалізація Jenkinsfile.tests: Initialize

Checkout tests: перевірка на актуальність і наявність тестів для проведення тестування та збірки (див. рис. 3.41.).

```
82      stage('Checkout tests') {
83          steps {
84              checkout([
85                  //[[$class: 'CleanCheckout']],
86                  $class                             : 'GitSCM',
87                  branches                           : [[name: '*/main']],
88                  doGenerateSubmoduleConfigurations  : false,
89                  extensions                         : [],
90                  submoduleCfg                       : [],
91                  userRemoteConfigs                  : [[
92                      credentialsId: 'gws-sys-github-enterprise-api',
93                      url: "https://git.scm.genesys.com/Genesys/${repositoryName}"
94                  ]]
95              ])
96          }
97      }
```

Рис. 3.41. Реалізація Jenkinsfile.tests: Checkout tests

Build: збірка проекту тестування за допомогою gradle.tests (див. рис. 3.42.).

```
98
99      stage("Build") {
100         steps {
101             script {
102                 def buildInfo = Artifactory.newBuildInfo()
103                 buildInfo.env.capture = true
104                 buildInfo.retention maxBuilds: 15, maxDays: 30, deleteBuildArtifacts: true
105
106                 def buildGradle = Artifactory.newGradleBuild()
107                 buildGradle.tool = "5.6.4"
108                 buildGradle.run switches: "--refresh-dependencies", tasks: "clean build", buildInfo: buildInfo
109
110                 buildInfo.env.collect()
111             }
112         }
113     }
```

Рис. 3.42. Реалізація Jenkinsfile.tests: Build

Push Docker Image: відправка зібраного образу проекту до репозиторію (див. рис. 3.43.).

```
115         stage('Push Docker Image') {
116             steps {
117                 script {
118                     env.FULL_IMAGE_NAME = "${gwsRegistry}/${imageName}"
119                     sh "docker tag ${FULL_IMAGE_NAME}:latest ${FULL_IMAGE_NAME}:${BUILD_VERSION}"
120 
121                     rtDocker.push "${FULL_IMAGE_NAME}:latest", 'gws-docker-registry'
122                     rtDocker.push "${FULL_IMAGE_NAME}:${BUILD_VERSION}", 'gws-docker-registry', buildInfo
123 
124                     setArtifactProperties artifactoryId: 'artifactory.gws.genesys.com',
125                                           credentialsId: 'gws-artifactory-api',
126                                           repository: 'gws-docker-registry-local',
127                                           artifactName: repositoryName,
128                                           artifactVersion: BUILD_VERSION
129 
130                     artifactoryServer.publishBuildInfo buildInfo
131                 }
132             }
133         }
```

Рис. 3.43. Реалізація Jenkinsfile.tests: Docker Image

CleanUp: очистка зібраних образів після відправки до репозиторію, щоб данні для тестів брались актуальні - а не із кешу залишеним у образах (див. рис. 3.44.).

```
135         stage('CleanUp') {
136             steps {
137                 script {
138                     sh """ docker rmi \
139                         --force ${gwsRegistry}/${imageName}:latest \
140                         --force ${gwsRegistry}/${imageName}:${BUILD_VERSION}
141                     """
142                 }
143             }
144         }
```

Рис. 3.44. Реалізація Jenkinsfile.tests: Clean Up

Remove Old Deployment: очистка тестового оточення перед початком тестування нової версії продукту (див. рис. 3.45.).

```
146         stage('Remove old deployment') {
147             steps {
148                 echo 'INFO: Start Remove Old Deployment.'
149 
150                 script {
151                     sh "nohup kubectl delete deployments -n ${appProvNamespace} --all &"
152                     sh "nohup kubectl delete service -n ${appProvNamespace} --all &"
153                     sh "nohup kubectl delete ingress -n ${appProvNamespace} --all &"
154                     sh "nohup kubectl delete pods -n ${appProvNamespace} --all &"
155 
156                     try {
157                         Boolean pods = true
158                         while (pods) {
159                             String get_namespace = sh returnStdout: true, script: "kubectl get pods -n ${appProvNamespace}"
160                             if (get_namespace) {
161                                 echo 'DEBUG: Deployments are still terminating...'
162                                 sleep 10
163                             } else {
164                                 pods = false
165                             }
166                         }
167                         echo 'DEBUG: Deployments are terminated.'
168                     } catch (e) {
169                         sh returnStatus: true, script: "kubectl delete pods --all --force --grace-period=0 -n ${appProvNamespace}" +
170                             "&& kubectl delete deployments --all --force --grace-period=0 -n ${appProvNamespace}"
171                     }
172                 }
173             }
174         }
```

Рис. 3.45. Реалізація Jenkinsfile.tests: Remove Old Deployment

Deploy GWS Environment: розгортання нових версій сервісів для підтримки оточення (див. рис. 3.46.).

```
        stage('Deploy GWS environment') {
            steps {
                echo 'INFO: Apply GWS manifest files.'

                script {
                    try {
                        dir('k8-gws-manifests') {
                            findFiles(glob: '*-service.yaml').each {
                                sh "kubectl apply -n ${appProvNamespace} -f ${it.name}"
                            }
                            findFiles(glob: '*-deployment.yaml').each {
                                sh "kubectl apply -n ${appProvNamespace} -f ${it.name}"
                            }
                        }

                        String authPorts = sh returnStdout: true, script: "kubectl get service \"gws-core-auth\" -n ${appProvNamespace} --output=json"
                        authPortsObj = readJSON(text: authPorts).spec.ports
                        String authPort = authPortsObj.find { it.name == "server-port" }.nodePort
                    } catch (err) {
                        stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
                        throw stageErrorMessage
                    }
                }
            }
        }
```

Рис. 3.46. Реалізація Jenkinsfile.tests: Deploy GWS Environment

Deploy ConfigServers: розгортання власного сервера для кожного тестового сценарія (див. рис. 3.47.).

```
    stage('Deploy ConfigServers') {
        steps {
            echo 'INFO: Apply ConfigServers manifest files.'
            script {
                try {
                    dir('k8') {
                        Map configServers = [:]
                        testSuits.each { suite ->
                            String testSuitName = suite.replace('_', '-')
                            configServers[testSuitName] = {
                                sh "sed 's+gws-configserver+gws-configserver-${testSuitName}+g' gws-configserver-service.yaml | kubectl apply -n ${appProvNamespace} -f -"
                                sh "sed 's+gws-configserver+gws-configserver-${testSuitName}+g' gws-configserver-pod.yaml | kubectl apply -n ${appProvNamespace} -f -"
                            }
                        }
                        parallel configServers
                    }
                } catch (err) {

                    stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
                    throw stageErrorMessage
                }
            }
        }
    }
```

Рис. 3.47. Реалізація Jenkinsfile.tests: Deploy ConfigServers

Deploy Provisioning: розгортання сервіса для пидтримки взаємодії між собою всіх компонентів продукту (див. рис. 3.48.).

```
    stage('Deploy Provisioning') {
        steps {
            echo 'INFO: Apply Provisioning manifest files.'
            script {
                try {

                    dir('k8-manifests') {
                        sh "kubectl apply -n ${appProvNamespace} -f gws-app-provisioning-service.yaml"
                        withCredentials([[
                            $class             : 'AmazonWebServicesCredentialsBinding',
                            accessKeyVariable  : 'AWS_ACCESS_KEY_ID',
                            credentialsId      : 'aws-credentials-staging',
                            secretKeyVariable  : 'AWS_SECRET_ACCESS_KEY'
                        ]]) {

                            String awsKeyID = env.AWS_ACCESS_KEY_ID
                            String awsSecret = env.AWS_SECRET_ACCESS_KEY

                            sh "sed 's+aws_key+${awsKeyID}+g; s+aws_secret+${awsSecret}+g; " +
                            "s+gws-docker-registry.artifactory.gws.genesys.com/gws-app-provisioning+gws-docker-registry.artifactory.gws.genesys.com/gws-app-provisioning:latest+g' "
                            "gws-app-provisioning-deployment.yaml | kubectl apply -n ${appProvNamespace} -f -"
                        }
                    }
                } catch (err) {

                    stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
                    throw stageErrorMessage
                }
            }
        }
    }
```

Рис. 3.48. Реалізація Jenkinsfile.tests: Deploy Provisioning

Check Environment: перевірка сервісів готових для використання (див. рис. 3.49.).

```
stage("Check environment") {
    steps {
        script {
            servicesPorts = getServicesPorts()
            Set manPorts = servicesPorts.findResults { it.value.management ?: null }

            echo "INFO: Check all pods in the namespace are started."
            boolean checkRunning = true
            while (checkRunning) {
                String podCurrentPhases = sh returnStdout: true,
                        script: """
                            kubectl get pods -n gws-app-provisioning-tests
                            -o custom-columns=NAME:.metadata.name,STATUS:.status.phase
                            --no-headers | grep 'gws' | awk -F ' ' '{print \$2}'
                        """
                List podCurrentPhasesList = podCurrentPhases.trim().split()
                checkRunning = podCurrentPhasesList.any { it != "Running" }
                if (checkRunning) {
                    echo "DEBUG: Some of K8 pods are not running. Will recheck."
                    sleep 10
                }
            }
            echo "INFO: All services are in Running phase."

            echo "INFO: Get readiness status of all containers in the namespace."
            boolean checkReadiness = true
            while (checkReadiness) {
                String containerCurrentStatuses = sh returnStdout: true,
                        script: """
                            kubectl get pods -n gws-app-provisioning-tests
                            -o custom-columns=NAME:.metadata.name,STATUS:.status.containerStatuses[*].ready
                            --no-headers | grep 'gws' | awk -F ' ' '{print \$2}'
                        """
                List containerCurrentStatusesList = containerCurrentStatuses.trim().split()
                checkReadiness = containerCurrentStatusesList.any { it.contains("false") }
                if (checkReadiness) {
                    echo "DEBUG: Some of K8 containers are not ready."
                    sleep 10
                }
            }
            echo "INFO: All services are Ready."
        }
    }
}
```

Рис. 3.49. Реалізація Jenkinsfile.tests: Check Environment

Swagger Validation: перевірка документації на наданий API (див. рис. 3.50.).

```
stage("Swagger Validation") {
    steps {
        dir("nightly") {
            sh "mkdir -p reports"
            script {
                try {
                    def validationStatus = sh returnStatus: true, script: "python2 test_validate_swagger_file.py --doc-url ${docURL}"
                    junit keepLongStdio: true, healthScaleFactor: 0.0, testResults: "reports/swagger-validation.xml"
                    archiveArtifacts artifacts: "reports/swagger-validation.xml"
                    if (validationStatus == 0) {
                        currentBuild.result = "SUCCESS"
                        echo "INFO: Swagger validation is SUCCESS."
                    } else {
                        currentBuild.result = "FAILED"
                        error "Swagger validation failed."
                    }
                } catch (err) {
                    stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
                    throw err
                }
            }
        }
    }
}
```

Рис. 3.50. Реалізація Jenkinsfile.tests: Swagger Validation

Setup Environment: загрузка та збір актульного образу для взаємодії з сервісом авторизації та аутентифікації - та запуск скрипта для регістрації нового хоста та клієнта для взаємодії з даним сервісом на всіх єтапах і інтеграціях при проведенні розробки чи тестування на розгорнутом оточені (див. рис. 3.51.).

```
329         stage("Setup Environment") {
330             steps {
331                 script {
332                     try {
333 
334                         currentBuild.rawBuild.getAction(PodTemplateAction.class)?.stack?.clear()
335                         podTemplate(
336                             label: "setup-environment",
337                             containers: [
338                                 containerTemplate(
339                                     name: "jnlp",
340                                     image: "gws-docker-registry.artifactory.gws.genesys.com/jenkins/jnlp-slave",
341                                     args: '${computer.jnlpmac} ${computer.name}'
342                                 ),
343                                 containerTemplate(
344                                     name: "gws-app-provisioning-test",
345                                     image: "gws-docker-registry.artifactory.gws.genesys.com/${imageName}:latest",
346                                     alwaysPullImage: true,
347                                     ttyEnabled: true
348                                 )
349                             ],
350                             namespace: "gws-app-provisioning-tests",
351                             volumes: [
352                                 emptyDirVolume(mountPath: '/tmp', memory: true)
353                         ]) {
354 
355                             node("setup-environment") {
356                                 container("gws-app-provisioning-test") {
357                                     def setupEnvironment = sh returnStatus: true, script: """
358                                         export CSHost=None &&
359                                         export CSPort=None &&
360                                         export POSTGRES_PORT=None &&
361                                         export POSTGRES_HOST=None &&
362                                         export CLEAN_ENVIRONMENT=1 &&
363                                         export TEST_DEPRECATED_FEATURES=1 &&
364                                         export DOMAIN=None &&
365                                         export READ_ONLY_DOMAIN=None &&
366                                         python3 -W ignore /gws_app_provisioning_functional_tests/setup_environment.py
367                                     """
368                                     if (setupEnvironment == 0) {
369                                         currentBuild.result = "SUCCESS"
370                                         echo "INFO: Setup Environment is SUCCESS."
371                                     } else {
372                                         currentBuild.result = "FAILED"
373                                         error "Setup Environment validation failed."
374                                     }
375                                 }
376                             }
377 
378                         }
379 
380                     } catch (err) {
381                         stageErrorMessage = "*${STAGE_NAME} Stage is failed* - \n${err.getMessage()}"
382                         throw err
383                     }
384                 }
385             }
386         }
387 
```

Рис. 3.51. Реалізація Jenkinsfile.tests: Setup Environment

Functional Tests: паралельний запуск автоматизованих тестових сценаріїв (див. рис. 3.52.).

```
388         stage('Functional Tests') {
389             steps {
390                 script {
391                     Map testBranches = [:]
392 
393                     testSuits.each { suite ->
394                         String testSuitName = suite.split(':')[0]
395                         String nodeLabel = "gws-app-provisioning-functional-tests-${UUID.randomUUID().toString()}"
396                         String voiceMachine = "gws-configserver-${testSuitName.replace('_', '-')}"
397                         String voiceMachinePort = '8888'
398                         Boolean virtual_confserver = true
399 
400                         testBranches[testSuitName] = {
401                             currentBuild.rawBuild.getAction(PodTemplateAction.class)?.stack?.clear()
402                             podTemplate(
403                                 label: nodeLabel,
404                                 containers: [
405                                     containerTemplate(
406                                         name: "jnlp",
407                                         image: "gws-docker-registry.artifactory.gws.genesys.com/jenkins/jnlp-slave",
408                                         args: '${computer.jnlpmac} ${computer.name}'
409                                     ),
410                                     containerTemplate(
411                                         name: "gws-app-provisioning-test",
412                                         image: "gws-docker-registry.artifactory.gws.genesys.com/${imageName}:latest",
413                                         alwaysPullImage: true,
414                                         ttyEnabled: true
415                                     )
416                                 ],
417                                 namespace: "gws-app-provisioning-tests",
418                                 volumes: [
419                                     emptyDirVolume(mountPath: '/tmp', memory: true)
420                             ]) {
421                                 node(nodeLabel) {
422                                     container("gws-app-provisioning-test") {
423                                         String allureResultPath = "${env.WORKSPACE}/allure-results"
424                                         String resultPath = "${env.WORKSPACE}/reports/${testSuitName}"
425                                         def status = sh returnStatus: true, script: """
426                                             export CSHost=${voiceMachine} &&
427                                             export CSPort=${voiceMachinePort} &&
428                                             export POSTGRES_PORT=${servicesPorts['gws-postgres'].server} &&
429                                             export POSTGRES_HOST='k8-200.gws.genesys.com' &&
430                                             export CLEAN_ENVIRONMENT=1 &&
431                                             export TEST_DEPRECATED_FEATURES=1 &&
432                                             export DOMAIN=${testSuitName} &&
433                                             export READ_ONLY_DOMAIN="read_only_${testSuitName}" &&
434                                             mkdir -p ${allureResultPath} &&
435                                             mkdir -p ${resultPath} &&
436                                             python3 -W ignore -m pytest -s -v --disable-warnings --alluredir=${allureResultPath} /gws_app_provisioning_functional_tests/tests/${testSuitNa
437                                         """
438 
439                                         stash name: "allureReports_${testSuitName}", includes: "allure-results/*", allowEmpty: true
440                                         junit allowEmptyResults: true, keepLongStdio: true, healthScaleFactor: 0.0, testResults: "reports/${testSuitName}/**/*.xml"
441 
442                                         echo "INFO: Archive artifacts"
443                                         archiveArtifacts "reports/**/*"
444 
445                                     }
446                                 }
447                             }
```

Рис. 3.52. Реалізація Jenkinsfile.tests: Functional Tests

Collect Reports: збір та архівація логів пройденних тестових сценаріїв (див. рис. 3.53.).

```
455     stage('Collect Reports') {
456 
457         steps {
458             script {
459 
460                 sh 'rm -rf ./reports/ ./allure-report/ ./allure-results/'
461                 sh 'mkdir -p allure-results'
462 
463                 testSuits.each { suite ->
464                     unstash "allureReports_${suite}"
465                 }
466 
467                 allure([
468                     includeProperties: false,
469                     jdk: '',
470                     properties: [],
471                     reportBuildPolicy: 'ALWAYS',
472                     results: [[path: 'target/allure-results']]
473                 ])
474 
475             }
476         }
477 
478     }
479 
480 }
```

Рис. 3.53. Реалізація Jenkinsfile.tests: Collect Reports

Аналіз розробленого фреймворку

Розроблений фреймворк включає в себе всі необхідні скрипти для підтримки автоматизації розробки та тестування. Структура данного проекту є універсальною, якщо знадобиться застосувати схожу логіку та принципи в автоматизації для іншого продукту - достатньо лише змінити проекти розробки та тестування, які знаходяться у відповідних директоріях: /dev та /tests і задати нову конфігурацію відповідну до нового проекта.

З реалізацією представленою в данній роботі - вдалось позбутися від що-релізної рутини по розгортуванні в ручному режимі реліз кандидата додатка на різних оточеннях, скоротити час потрапляння нової версії до релізу з декількох днів – до 20 хвилин.

Також, завдяки використанню практик неперервної інтеграції та розгортування за підтримки системи Jenkins – є можливість, в реальному часі, моніторити всі етапи потрапляння нової версії додатку до релізу та отримувати графіки і звіти за виконаною роботою скриптів автоматизації (див. рис. 3.54.).

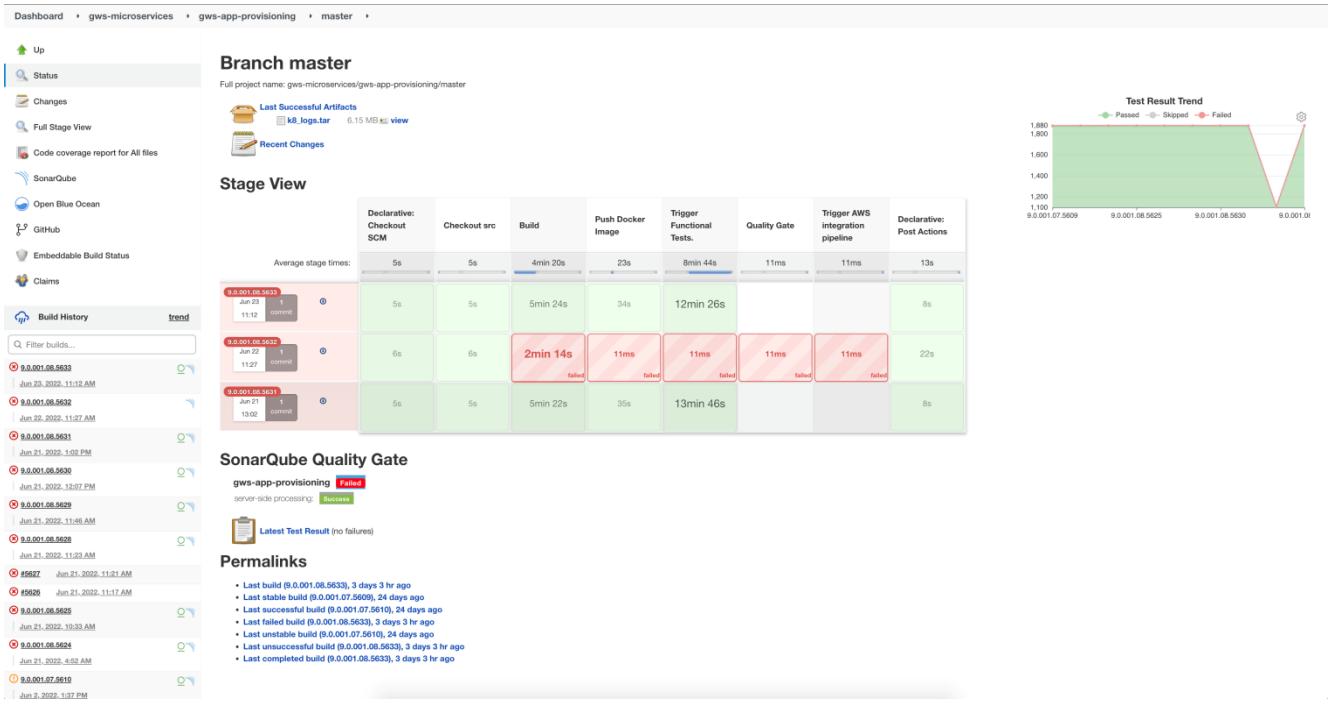


Рис. 3.54. Моніторинг сторінка з етапами збірки та графіком стабільності

Щоб отримати звіти - достаньо перейти до розділу збірки та переглянути отримані артифакти, які були сформаванні автоматично та містять всю інформацію по-минулій збірці (див.рис.3.55.).

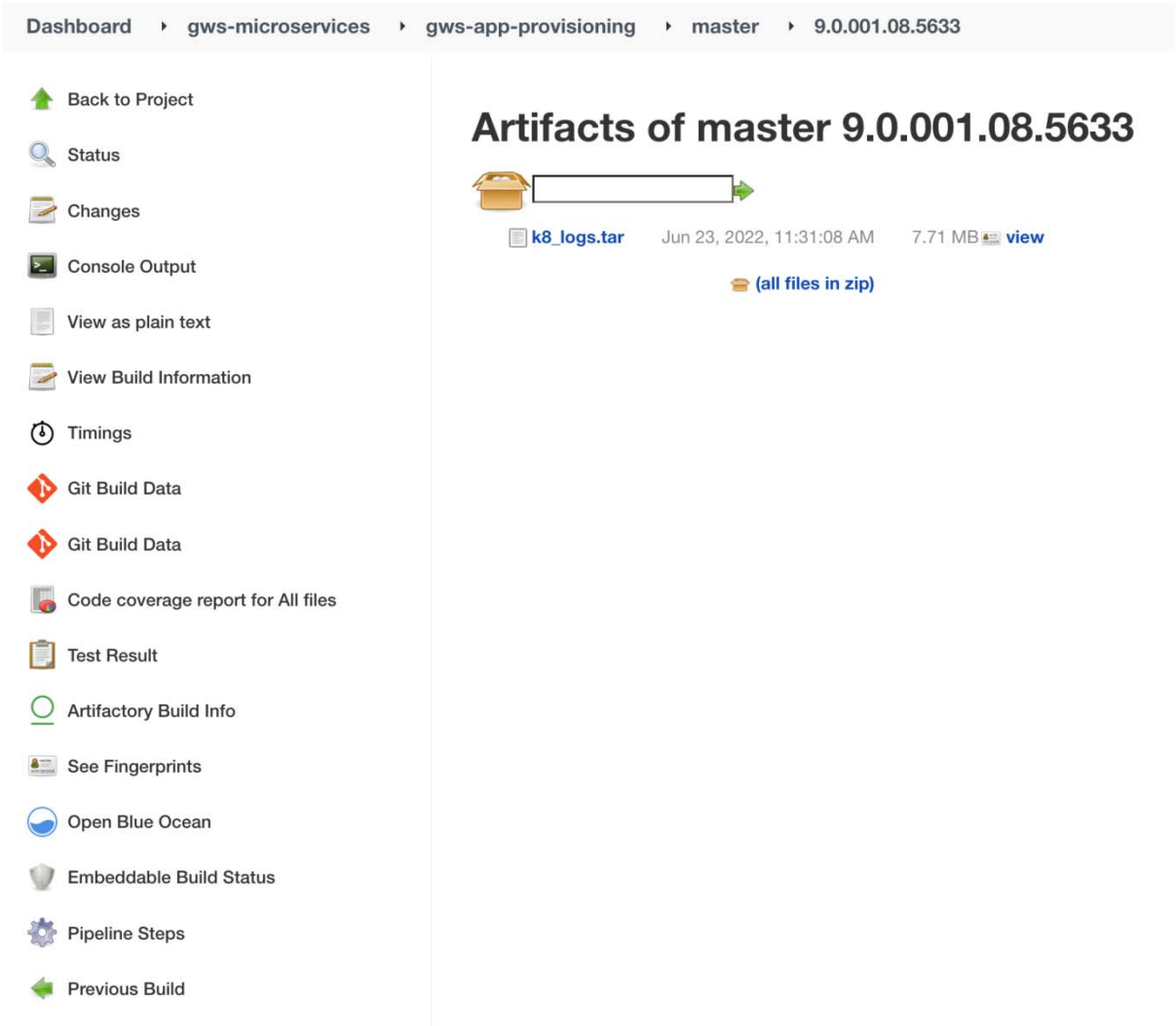


Рис. 3.55. Артефакти запуску збірки

Висновки за розділом 3

У цьому розділі синтезований універсальний фреймворк для автоматизації процесів розробки та тестування програмного забезпечення з урахуванням платформи для ведення процесів розробки. Отримане робоче середовище розроблене з використанням наступних технологій: Python, Bash/Shell, Gradle, Docker, Jenkins.

Враховуючи специфіку оточень, на яких проводиться чи проводитиметься розробка та тестування програмного продукту, даний набір інструментів для розробки фреймворку з використанням CI/CD практик є найбільш актуальним, адже робота проводиться на оточеннях, запущених навпроти Linux дистрибутивів чи використовуючих оточення Linux.

					151.4341.КР.ПЗ.Р4		
Змн	*Арк*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		*Черній М.О.*			ОХОРОНА ПРАЦІ	*Літ.*	*Арк* 75 / *Аркушів* 7
Норм. контр		*Вінниченко І.Л.*				НУК ім. адм.. Макарова	
Керівник		*Герасін О.С.*					
Зав. Каф.		*Черно О.О.*					

РОЗДІЛ 4
ОХОРОНА ПРАЦІ

Незважаючи на те, що робота инжера з автоматизації , на перший погляд, не пов'язана із загрозою життю чи здоров'ю, охорона праці цієї професійної галузі розвивається активно. Оскільки напруженість та інтенсивність роботи останнім часом зросла, виникли питання гігієни праці, оптимізації та комплексного підходу до організації робочого часу.

Умови роботи та влаштування робочого місця має відповідати вимогам гігієни, антропометрії, а також фізичним та психологічним особливостям працівника.

Найважливішими аспектами, що контролюються охороною праці в галузі, є:

1. Робочий простір. Він має бути досить просторим та дозволяти безперешкодно здійснювати необхідні для роботи та відпочинку переміщення. Крім того, стілець і стіл як головні складові простору повинні відповідати гігієнічним нормам та індивідуальним параметрам працівника. Надто низькі або надто високі меблі ускладнюють робочий процес, що призводить до серйозних проблем зі здоров'ям;
2. Устаткування. Його розміщення має відповідати гігієнічним нормам, а також не перешкоджати вільній та повноцінній експлуатації;
3. Освітлення. Один із найважливіших моментів у роботі – це достатня освітленість, недотримання норм якої може призвести до розладів фізичного та психологічного здоров'я (погіршення зору, депресії, погіршення орієнтації), а також суттєво знизити ефективність праці;
4. Шум. Рівень фонового шуму в робочому приміщенні (розмови колег, робота техніки, телефонні переговори тощо) не повинен перевищувати допустимі значення, оскільки зайва зашумленість стомлює, знижує концентрацію уваги, спричиняє головний біль та стрибки тиску;
5. Вентиляція. Приміщення - має вентилюватись постійно і на достатньому рівні. Техніка прогріває повітря і знижує кількість кисню, що може призвести

до стомлюваності і навіть непритомності;

6. Відпочинок. Регулярні перерви - обов'язкова умова будь-якої сидячої роботи. Робочий процес повинен постійно перемежуватися з фізичною активністю: це дозволить підвищити ККД роботи та запобігти проблемам зі здоров'ям.

Загальні вимоги техніки безпеки під час роботи на персональному комп'ютері.

До роботи на персональному комп'ютері допускаються особи, які пройшли навчання безпечним методам праці, вступний інструктаж, первинний інструктаж на робочому місці.

При експлуатації персонального комп'ютера на працівника можуть діяти такі небезпечні та шкідливі виробничі фактори:

1. Підвищений рівень електромагнітних випромінювань;
2. Підвищений рівень статичної електрики;
3. Знижена іонізація повітря;
4. Статичні фізичні навантаження;
5. Перенапруга зорових аналізаторів.

Працівник зобов'язаний:

1. Виконувати лише ту роботу, яка визначена його посадовою інструкцією;
2. Утримувати у чистоті робоче місце;
3. Дотримуватися режиму праці та відпочинку залежно від тривалості, виду та категорії трудової діяльності;
4. Дотримуватись заходів пожежної безпеки.

Робочі місця з комп'ютерами повинні розміщуватися таким чином, щоб відстань від екрана одного відеомонітора до тилу іншого була не меншою за 2,0 м, а відстань між бічними поверхнями відеомоніторів - не менше ніж 1,2 м. робочі місця з персональними комп'ютерами по відношенню до світлових прорізів повинні розташовуватися так, щоб природне світло падало збоку, переважно ліворуч. Віконні отвори в приміщеннях, де використовуються персональні комп'ютери, повинні бути обладнані регульованими пристроями типу: жалюзі,

завіс, зовнішніх козирків та ін. робочі меблі для користувачів комп'ютерною технікою повинні відповідати таким вимогам:

1. Висота робочої поверхні столу має регулюватися в межах 680 – 800 мм; за відсутності такої можливості висота робочої поверхні столу має становити 725 мм;
2. Робочий стіл повинен мати простір для ніг заввишки не менше 600 мм, глибиною на рівні колін не менше 450 мм та на рівні витягнутих ніг не менше 650 мм;
3. Робочий стілець повинен бути підйомно - поворотним і регульованим по висоті та кутам нахилу сидіння та спинки, а також - відстані спинки від переднього краю сидіння;
4. Робоче місце має бути обладнане підставкою для ніг, що має ширину не менше 300 мм, глибину не менше 400 мм, регулювання по висоті в межах до 150 мм і по куту нахилу опорної поверхні підставки до 20 градусів; поверхня підставки повинна бути рифленою та мати по передньому краю бортик заввишки 10 мм;
5. Робоче місце з персональним комп'ютером має бути оснащене пюпітром для документів, що легко переміщається.

Для нормалізації аероіонного фактора приміщень із комп'ютерами необхідно використовувати пристрої автоматичного регулювання іонного режиму повітряного середовища. Жінки з часу встановлення вагітності та в період годування груддю для виконання всіх видів робіт, пов'язаних з використанням комп'ютерів, не допускаються.

За невиконання цієї Інструкції винні притягуються до відповідальності згідно з правилами внутрішнього трудового розпорядку або стягненнями.

Вимоги техніки безпеки перед початком роботи:

1. Підготувати робоче місце;
2. Відрегулювати освітлення на робочому місці, переконатися у відсутності відблисків на екрані;

3. Перевірити правильність підключення обладнання до електромережі;
4. Перевірити справність проводів живлення та відсутність оголених ділянок проводів;
5. Перевірте наявність заземлення системного блоку, монітора та захисного екрану;
6. Протерти антистатичною серветкою поверхню екрана монітора та захисного екрану;
7. Перевірити правильність установки столу, стільця, підставки для ніг, пюпітра, кута нахилу екрана, положення клавіатури, положення «миші» на спеціальному килимку, при необхідності провести регулювання робочого столу та крісла, а також розташування елементів комп'ютера відповідно до вимог ергономіки та з метою виключення незручних поз та тривалих напруг тіла.

Вимоги техніки безпеки під час роботи

Працівнику під час роботи на ПК забороняється:

1. Торкатися задньої панелі системного блоку (процесора) при включеному живленні;
2. Перемикати роз'єм інтерфейсних кабелів периферійних пристроїв при включеному живленні;
3. Допускати попадання вологи на поверхню системного блоку (процесора), монітора, робочу поверхню клавіатури, дисководів, принтерів та інших пристроїв;
4. Відключати обладнання від електромережі та висмикувати електровилку, тримаючись за шнур.

Тривалість безперервної роботи з комп'ютером без регламентованої перерви має перевищувати 2-х годин.Під час регламентованих перерв з метою зниження нервово – емоційної напруги, втоми зорового аналізатора, усунення впливу гіподинамії та гіпокінезії, запобігання розвитку познотонічної втоми виконуватикомплекси вправ.

Вимоги техніки безпеки після закінчення робот:

1. Вимкнути живлення комп'ютера;
2. Упорядкувати робоче місце;
3. Виконати вправи для очей та пальців рук на розслаблення.

Питання безпеки на роботі актуальне не тільки для тих професій, які мають на увазі важку фізичну роботу або складні кліматичні, природні умови роботи, але й для робочих місць за цілком нормальних умов праці. Аварійна ситуація може статися будь-де.

Отже, про безпеку своїх підлеглих повинен думати керівник організації. Також повинен будти відділ, який відповідає за цей аспект, якщо його немає, підлеглі завжди повинні бути захищені та знати всі норми та правила поведінки для збереження своєї безпеки та безпеки оточуючих.

Висновоки за розділом 4

В даному розділі розглянуто основні правила охорони праці на IT підприємстві.

Розглянуто та проаналізовано заходи, які повинні бути реалізовані на підприємстві, та яких повинні дотримуватись інженери та роботодавці для забезпечення комфортних та безпечних умов для працівників.

ВИСНОВКИ

В кваліфікаційній роботі виконано моделювання та розробку програмних засобів автоматизації на основі CI/CD практик.

В першому розділі роботи розглянуті парадигми автоматизації тестування та розробки з використанням Agile методології. Проведено аналіз використання гнучких технологій спільно з автоматизацією через призму CI/CD DevOps практик.

У другому розділі розглянуто актуальність мови програмування Python для ведення автоматизації на проекті, організацію контейнеризації та поєднання декількох контейнерів між собою, актуальність інструменту Jenkins для ведення CI/CD практик на проекті та налаштування систем контролю версій за допомогою Git для автоматизації процесів розробки та тестування.

В третьому розділі синтезований універсальний фреймворк для автоматизації процесів розробки та тестування програмного забезпечення з урахуванням платформи для ведення процесів розробки. Отримане робоче середовище розроблене з використанням наступних технологій: Python, Bash/Shell, Gradle, Docker, Jenkins.

Основним результатом роботи є прискорення процесів інтеграції і розгортання нової версії продукту від декількох робочих днів - до 15 - 20 хвилин. Так само покращилося покриття та збір помилок при невдалих зборках та спробах розгортання. Усі звіти архівуються та сегрегуються за типами. Також, наданий підхід покращив інтеграцію тестування і розробки між собою і допоміг прискорити без того швидке автоматизоване тестування з 1 години до 7 хвилин, що досягаються налаштуванням процесів з інтеграції, доставки та розгортання.

Після будь-яких змін у коді продукту та з'єднання цих змін з головною гілкою в ПЗ, що розробляється, - одразу складається новий образ з новою версією продукту. Підготовка тестових оточень, у свою чергу, проводиться в паралельному режимі, причому виконуються всі доступні сценарії для тестування нової версії. Крім того, в роботі розглянуті основні питання охорони праці на сучасному підприємстві.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дженніфер Грін, Ендрю Стілмен: “Осягаючи Agile”.
2. Майк Кон: “Agile: Оцінка та планування проектів”.
3. Майкл Доусон: “Програмуємо на Python”.
4. Ел Свейгарт: “Автоматизація рутинних завдань за допомогою Python. Практичний посібник для початківців”.
5. Щоб зрозуміти Gradle: https://russianblogs.com/article/9460319088/.
6. Еберхард Вольф: “Continuous delivery. Практика безперервних апдейтів”.
7. Learn Shell – Free: https://www.learnshell.org/.
8. Як впровадити CI/CD на проекті з історією самотужки. Частина 1: підготовчий етап: https://jazzteam.org/ru/technical-articles/ci-cd-integration-part1/.
9. Впровадження CI/CD на проекті з історією самотужки. Частина 2: Реалізація: https://jazzteam.org/ru/technical-articles/ci-cd-integration-part2/.