МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образование «Полоцкий государственный университет имени Евфросинии Полоцкой»

Факультет информационных технологий

Кафедра технологий программирования

**Лабораторная работа №4**

по дисциплине: «Разработка и анализ требований»

на тему: «Автоматизация доставки проекта»

|  |  |
| --- | --- |
| Выполнил | Студент гр. 23-ИТ-2  Скрипко В.В. |
|  |  |
| Проверил | Старший преподаватель кафедры ТП  Струк Т.С. |

Полоцк 2025

**Цель работы:** освоить основы работы с облачной платформой непрерывной интеграции и доставки CircleCI: создать аккаунт и подключить репозиторий; написать конфигурационный файл .circleci/config.yml; автоматизировать сборку и тестирование простого приложения; проанализировать результаты выполнения пайплайна.

**Задание:**

Задание 1.

1. Создать простое Python-приложение с тестами.

2. Разместить его в публичном репозитории на GitHub.

3. Подключить репозиторий к CircleCI.

4. Написать конфигурационный файл , который:.circleci/config.yml

* устанавливает зависимости;
* запускает тесты с помощью pytest;
* завершается успешно только при прохождении всех тестов.

5. Выполнить пайплайн и проанализировать его результаты.

Задание 2 (повышенной сложности - по желанию).

* Добавьте второй job, который проверяет форматирование кода с помощью .black--check
* Настройте workflow так, чтобы тесты и проверка стиля запускались
* параллельно.
* Добавьте кэширование зависимостей () для ускорения сборки.pip

**Ход работы**

В ходе выполнения лабораторной работы было создано несколько файлов: app.py, test\_app.py и .circleci/config.yml. Содержимое данных файлов представлено в листингах 1.1-1.3 соответственно.

Листинг 1.1 – Код app.py.

1. def add(a, b):
2. return a + b

Листинг 1.2 – Код test\_app.py.

1. import pytest
2. from app import add
3. def test\_add():
4. assert add(2, 3) == 5
5. assert add(-1, 1) == 0

Листинг 1.3 – Содержимое config.yml.

1. version: 2.1
2. jobs:

Продолжение листинга 1.3

1. test:
2. docker:
3. - image: cimg/python:3.11
4. steps:
5. - checkout
6. - run:
7. name: Install dependencies
8. command: pip install -r requirements.txt
9. - run:
10. name: Run tests
11. command: pytest
12. workflows:
13. version: 2
14. build-and-test:
15. jobs:
16. - test

Также был создан файл requirements.txt, в котором содержаться библиотеки, используемые в файлах .py.

Далее проект был добавлен в репозиторий circleci-lab на GitHub. Результат представлен на рисунке 1.1.

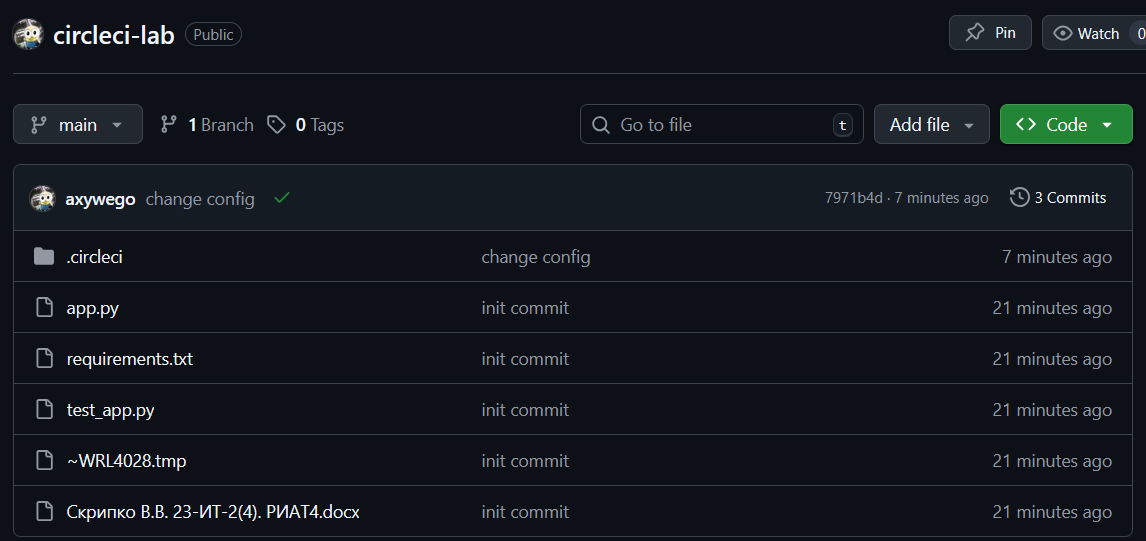


Рисунок 1.1 – Репозиторий circleci-lab.

Далее в интерфейсе CircleCI проверяем выполнение пайплайна. Результат проверки представлен на рисунке 1.2.

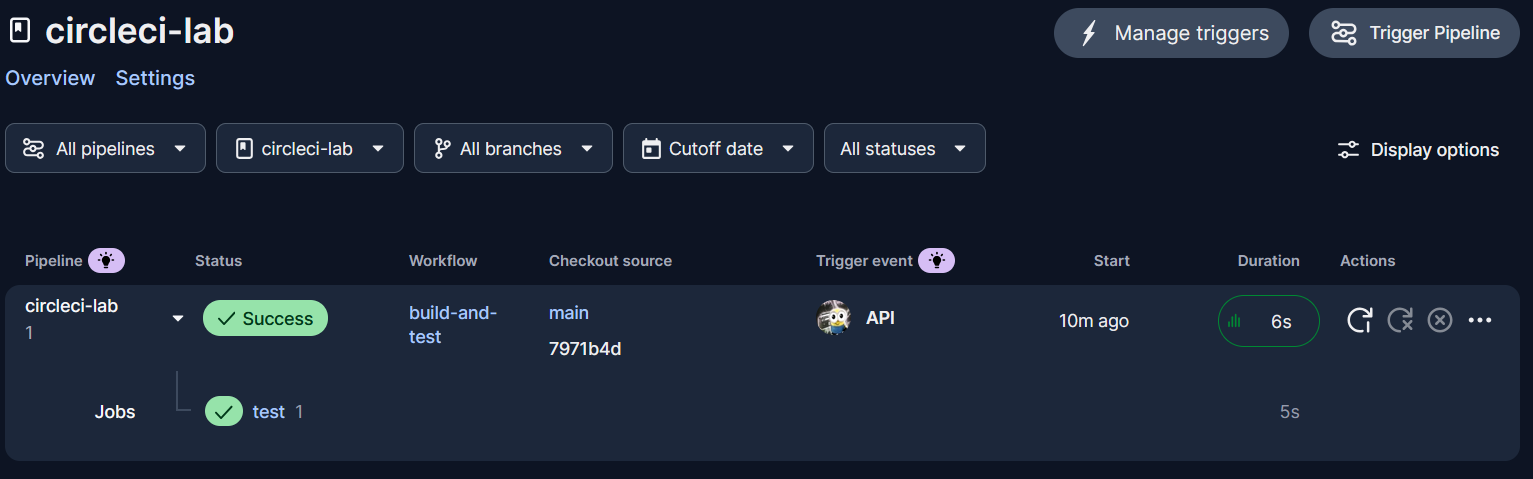


Рисунок 1.1 – Завершенный пайплайн.

Листинг 1.2 – Код 2.py.

1. import pika
2. import requests
3. import json
4. import sys
5. OPENWEATHER\_API\_KEY = "42fce9ccc5ed739d72a3b11bdae398d2"
6. CITY = "Minsk"
7. def get\_weather():
8. url = f"http://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={OPENWEATHER\_API\_KEY}&units=metric&lang=ru"
9. try:
10. response = requests.get(url, timeout=10)
11. response.raise\_for\_status()
12. data = response.json()
13. return {
14. "city": data["name"],
15. "temp": round(data["main"]["temp"]),
16. "description": data["weather"][0]["description"],
17. "humidity": data["main"]["humidity"],
18. "timestamp": data["dt"]
19. }
20. except requests.RequestException as e:
21. print(f"Ошибка получения погоды: {e}", file=sys.stderr)
22. return None
23. def send\_weather\_to\_queue(weather\_data):
24. try:
25. connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
26. channel = connection.channel()
27. channel.queue\_declare(queue='weather\_queue', durable=True)
28. message = json.dumps(weather\_data, ensure\_ascii=False)
29. channel.basic\_publish(
30. exchange='',
31. routing\_key='weather\_queue',
32. body=message,
33. properties=pika.BasicProperties(delivery\_mode=2)
34. )
35. print("Погода успешно отправлена в очередь.")
36. connection.close()
37. return True
38. except pika.exceptions.AMQPConnectionError:
39. print("Брокер недоступен.", file=sys.stderr)

Продолжение листинга 1.2

1. return False
2. if \_\_name\_\_ == "\_\_main\_\_":
3. weather = get\_weather()
4. if weather:
5. send\_weather\_to\_queue(weather)
6. else:
7. print("Не удалось получить данные о погоде.")

Результат выполнения программы представлен на рисунке 1.2.



Рисунок 1.2 – Результат выполнения программы из второго задания.

Для третьего задания необходимо расширить предыдущее задание, реализовав консьюмер приложение, которые подписывается на очередь и выводит полученное сообщение с данными о погоде. Код консьюмера представлен в листинге 1.3.

Листинг 1.3 – Код 3.py.

1. import pika
2. import json
3. import sys
4. def callback(ch, method, properties, body):
5. try:
6. weather\_data = json.loads(body.decode('utf-8'))
8. print("\n === Получены данные о погоде ===")
9. print(f"Город: {weather\_data['city']}")
10. print(f"Температура: {weather\_data['temp']}°C")
11. print(f"Описание: {weather\_data['description']}")
12. print(f"Влажность: {weather\_data['humidity']}%")
13. print(f"Время обновления: {weather\_data['timestamp']}")
14. print("==================================\n")
16. ch.basic\_ack(delivery\_tag=method.delivery\_tag)
18. except (json.JSONDecodeError, KeyError) as e:
19. print(f"Ошибка обработки сообщения: неверный формат данных ({e})", file=sys.stderr)
20. ch.basic\_nack(delivery\_tag=method.delivery\_tag, requeue=False)
21. except Exception as e:
22. print(f"Неизвестная ошибка: {e}", file=sys.stderr)
23. ch.basic\_nack(delivery\_tag=method.delivery\_tag, requeue=False)
24. def main():

Продолжение листинга 1.3

1. try:
2. connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
3. channel = connection.channel()
5. channel.queue\_declare(queue='weather\_queue', durable=True)
7. channel.basic\_qos(prefetch\_count=1)
9. channel.basic\_consume(
10. queue='weather\_queue',
11. on\_message\_callback=callback
12. )
14. print("Ожидание сообщений о погоде...")
15. channel.start\_consuming()
17. except pika.exceptions.AMQPConnectionError:
18. print("Ошибка: не удалось подключиться к RabbitMQ", file=sys.stderr)
19. except KeyboardInterrupt:
20. print("\n Потребитель остановлен вручную")
21. finally:
22. if 'connection' in locals() and connection.is\_open:
23. connection.close()
24. if \_\_name\_\_ == "\_\_main\_\_":
25. main()

Результат выполнения программы представлен на рисунке 1.3.

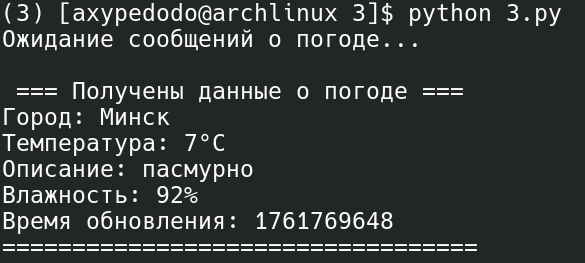


Рисунок 1.3 – Результат выполнения консьюмера из третьего задания.

Для четвертого задания необходимо разработать распределенную систему для мониторинга и обработки событий с использованием Apache Kafka. Создать продюсер, который генерирует симулированные события, отправляет их в тему Kafka. Затем создать несколько консьюмеров, которые обрабатывают и анализируют эти события. Код продюсера и консьюмера представлены в листингах 1.4 и 1.5 соответственно.

Листинг 1.4 – Код 4.py.

1. from kafka import KafkaProducer
2. import json
3. import time
4. import random
5. BASE\_LAT = 55.48
6. BASE\_LON = 28.79
7. def generate\_vehicle\_data(vehicle\_id: int):
8. return {
9. "vehicle\_id": vehicle\_id,
10. "timestamp": int(time.time()),
11. "latitude": round(BASE\_LAT + random.uniform(-0.01, 0.01), 6),
12. "longitude": round(BASE\_LON + random.uniform(-0.01, 0.01), 6),
13. "speed\_kmh": random.randint(0, 90)
14. }
15. if \_\_name\_\_ == "\_\_main\_\_":
16. producer = KafkaProducer(
17. bootstrap\_servers=['localhost:9092'],
18. value\_serializer=lambda v: json.dumps(v, ensure\_ascii=False).encode('utf-8')
19. )
20. TOPIC = "vehicle-locations"
21. print("Продюсер запущен. Отправка данных об автомобилях...")
22. try:
23. vehicle\_ids = [101, 102, 103]
24. while True:
25. for vid in vehicle\_ids:
26. data = generate\_vehicle\_data(vid)
27. producer.send(TOPIC, value=data)
28. print(f"Отправлено: {data}")
29. time.sleep(3)
30. except KeyboardInterrupt:
31. print("\nПродюсер остановлен.")
32. finally:
33. producer.flush()
34. producer.close()

Листинг 1.5 – Код consumer.py.

1. from kafka import KafkaConsumer
2. import json

Продолжение листинга 1.5

1. import sys
2. TOPIC = "vehicle-locations"
3. consumer = KafkaConsumer(
4. TOPIC,
5. bootstrap\_servers=['localhost:9092'],
6. auto\_offset\_reset='latest',
7. enable\_auto\_commit=True,
8. group\_id='logistics-group',
9. value\_deserializer=lambda x: json.loads(x.decode('utf-8'))
10. )
11. print("Консьюмер запущен. Ожидание данных об автомобилях...")
12. print("Анализ: отслеживание + логистика\n")
13. try:
14. for message in consumer:
15. data = message.value
16. status = "Остановлен" if data["speed\_kmh"] == 0 else "В движении"
17. print(f"[Отслеживание] Автомобиль {data['vehicle\_id']}: {status}")
18. print(f"[Логистика] ID={data['vehicle\_id']} | "
19. f"Координаты: ({data['latitude']}, {data['longitude']}) | "
20. f"Скорость: {data['speed\_kmh']} км/ч")
21. print("-" \* 60)
22. except KeyboardInterrupt:
23. print("\nКонсьюмер остановлен")
24. except Exception as e:
25. print(f"Ошибка: {e}", file=sys.stderr)
26. finally:
27. consumer.close()

Результат выполнения программы-продюсера показан на рисунке 1.4.

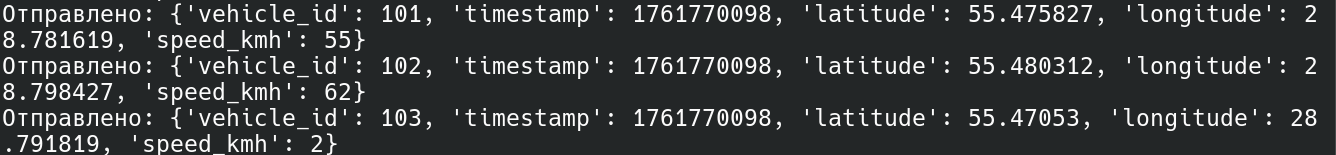


Рисунок 1.4 – результат выполнения программы-продюсера.

Результат выполнения программы-консьюмера показан на рисунке 1.5.

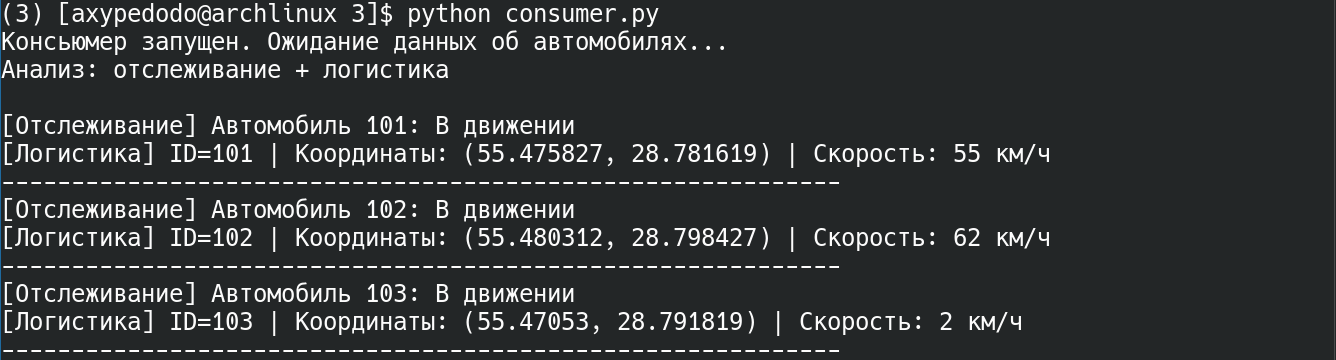


Рисунок 1.5 – результат выполнения программы-консьюмера.

**Вывод:** в ходе работы были изучены основные концепции и навыки, связанных с работой с брокерами сообщений, понимание понятия брокера сообщений, знание основных преимуществ брокеров сообщений, практические навыки работы с брокерами сообщений, готовность к разработке асинхронных приложений.

**Ответы на контрольные вопросы:**

1. Что такое брокер сообщений, и для чего он используется?

Брокер сообщений – это центральный компонент архитектуры межпрограммного взаимодействия, предназначенный для управления и маршрутизации сообщений между различными компонентами или приложениями.

Он используется для:

* организации асинхронного взаимодействия между отправителями (продюсерами) и получателями (консьюмерами);
* хранения сообщений во очередях, что позволяет устранять несоответствия в скорости отправки и обработки данных;
* обеспечения гарантированной доставки сообщений даже при сбоях;
* поддержки гибкой маршрутизации и фильтрации сообщений;
* упрощения интеграции компонентов в распределённых системах, включая микросервисы и IoT.

2. Какие популярные брокеры сообщений существуют, и в каких случаях их стоит использовать?

К популярным брокерам сообщений относятся:

* RabbitMQ – гибкий брокер на основе AMQP, подходит для сценариев с гарантированной доставкой, сложной маршрутизацией и микросервисной архитектурой.
* Apache Kafka – ориентирован на высокую пропускную способность и потоковую обработку данных в реальном времени; используется в аналитике, логировании и обработке событий.
* Apache ActiveMQ – совместим с JMS, часто применяется в Java-приложениях, поддерживает надёжную доставку и кластеризацию.
* MQTT – лёгкий протокол, идеален для IoT-устройств с ограниченными ресурсами и нестабильным соединением.
* NATS – высокопроизводительный и простой брокер, подходит для микросервисов и систем с низкой задержкой.
* Amazon SQS – управляемый облачный сервис от AWS, обеспечивает надёжность и масштабируемость без необходимости управления инфраструктурой.
* Redis Pub/Sub – простой механизм публикации/подписки, подходит для лёгких сценариев без требования сохранения сообщений.

Выбор брокера зависит от требований к надёжности, масштабируемости, задержке, объёму данных и архитектуры системы.

3. Каковы основные шаги для отправки сообщений в брокере сообщений?

Основные шаги (на примере RabbitMQ):

* Подключение к брокеру – установление соединения с использованием клиентской библиотеки (например, Pika для Python).
* Создание очереди – если очередь ещё не существует, её необходимо объявить.
* Отправка сообщения – публикация сообщения в указанную очередь (или обменник с ключом маршрутизации).
* Закрытие соединения – корректное завершение работы с брокером после отправки.

Важно: при отправке следует убедиться, что очередь/обменник существуют, а сообщение соответствует формату и не пусто.

4. Как можно подписаться на очередь и обработать полученные сообщения?

Подписка и обработка сообщений включают следующие шаги:

* Подключение к брокеру – аналогично отправке.
* Создание или проверка существования очереди – чтобы избежать ошибок при подписке.
* Определение функции-обработчика (callback) – функция, которая будет вызываться при получении каждого сообщения.
* Настройка подписки — с помощью метода вроде basic\_consume, указывающего очередь и обработчик.
* Запуск цикла ожидания сообщений – обычно бесконечный цикл (start\_consuming), который позволяет приложению постоянно слушать очередь.

При этом можно использовать автоматическое подтверждение (auto\_ack=True) или ручное подтверждение, чтобы гарантировать обработку сообщения перед его удалением из очереди.

5. Какие преимущества обеспечивает использование брокеров сообщений в распределенных системах?

Использование брокеров сообщений в распределённых системах даёт следующие преимущества:

* Асинхронность – компоненты не зависят от одновременной доступности друг друга.
* Масштабируемость – легко добавлять новые консьюмеры для распределения нагрузки.
* Гарантированная доставка – сообщения не теряются даже при сбоях благодаря подтверждениям и репликации.
* Снижение связности – компоненты взаимодействуют через абстракцию (очередь), а не напрямую.
* Буферизация нагрузки – брокер сглаживает всплески трафика, принимая сообщения быстрее, чем они обрабатываются.
* Поддержка распределённой архитектуры – компоненты могут находиться на разных серверах, в разных дата-центрах или облаках.
* Гибкая маршрутизация – сообщения могут направляться по разным правилам (темы, ключи, фильтры).
* Интеграция с разными технологиями – поддержка множества языков и протоколов упрощает взаимодействие разнородных систем.