МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ

Кафедра Комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
Аліна САВЧЕНКО.
«_____» _____ 2023р.

КВАЛІФІКАЦІЙНА РОБОТА

(ДИПЛОМНИЙ ПРОЄКТ, ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ **"БАКАЛАВР"**
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
"ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: "Бібліотека для системного тестування Apache Spark"

Виконавиця: студентка групи УС-411Б Демчук Софія Олегівна

Керівник: к.т.н., доцент Єгоров Сергій Вікторович

Нормоконтролер: Олександр ШЕВЧЕНКО

Київ – 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних наук та технологій*

Кафедра *Комп'ютерних інформаційних технологій*

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології"

ЗАТВЕРДЖУЮ

Завідувач випускової кафедри

_____ Аліна САВЧЕНКО

«____»_____2023р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи студентки

Демчук Софії Олегівни

(прізвище, ім'я, по батькові)

1. Тема роботи: «Бібліотека для системного тестування Apache Spark» затверджена наказом ректора від "01" травня 2023 р. за № 623/ст.

2. Термін виконання роботи: 15.05.2023 – 25.06.2023

3. Вихідні дані до роботи: розробка бібліотеки для системного тестування фреймворку обробки даних Apache Spark

4. Зміст пояснювальної записки: вступ, загальний огляд платформи для обробки даних Apache Spark, основи тестування і огляд програмних засобів, розробка тестової бібліотеки.

5. Перелік обов'язкового графічного матеріалу: екосистема Apache Spark; архітектура застосунку Spark; схема уніфікованого стеку Spark; схема рівнів тестування; структура модулів бібліотеки; діаграма послідовностей бібліотеки, діаграма класів бібліотеки.

6. Календарний план-графік

№ п/п	Завдання	Термін виконання	Підпис керівника
1.	Аналіз предметної області дослідження	15.05.2023–16.05.2023	
2.	Збір, огляд і аналіз наукової літератури за тематикою дипломного проекту, розробка архітектури бібліотеки	17.05.2023–27.05.2023	
3.	Написання Розділу 1 дипломної роботи	27.05.2023–29.06.2023	
4.	Розробка бібліотеки для системного тестування та перевірка її роботи	29.06.2023–08.06.2023	
5.	Написання Розділу 2 дипломної роботи	08.06.2023–09.06.2023	
6.	Написання Розділу 3 дипломної роботи	09.06.2023–10.06.2023	
7.	Написання пояснювальної записки для дипломного проекту	11.06.2023–13.06.2023	
9.	Підготовка демонстраційного матеріалу та доповіді	13.06.2023–16.06.2023	

7. Дата видачі завдання: «15» травня 2023 р.

Керівник дипломної роботи _____ Сергій ЄГОРОВ
(підпис керівника) (П.І.Б.)

Завдання прийняла до виконання _____ Софія ДЕМЧУК
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Бібліотека для системного тестування Apache Spark»: 122 с., 25 рис., 2 табл., 21 літературне джерело.

Об’єкт дослідження: фреймворк обробки даних Apache Spark.

Мета роботи: розробити бібліотеку для системного тестування фреймворку Apache Spark, його клієтських бібліотек та застосунків.

Методи дослідження: порівняльний аналіз, логічний аналіз, синтез, моделювання, обробка літературних джерел.

Результати роботи рекомендується використовувати для тестування Spark-застосунків та роботи його клієнтських бібліотек.

ВЕЛИКІ ДАНІ, ФРЕЙМВОРК, ОБРОБКА ДАНИХ, APACHE SPARK, APACHE HADOOP, КЛАСТЕР, МЕНЕДЖЕР КЛАСТЕРА, ВИКОНАВЕЦЬ, ДРАЙВЕР, MAPREDUCE, СТЕК SPARK, РІВНІ ТЕСТУВАННЯ, СИСТЕМНЕ ТЕСТУВАННЯ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ЛОГУВАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

ПЗ	Програмне забезпечення
BSD	Ліцензійна угода, вперше використана для розповсюдження Unix-подібних операційних систем BSD, чию назву вона і отримала.
SQL	Structured query language. Мова для взаємодії із базами даних
K8s	Kubernetes - система управління застосунками в контейнерах
API	Application Programming Interface - набір правил та протоколів, які визначають, як програмні компоненти взаємодіють між собою
DAG	Directed Acyclic Graph - це ациклічний направлений граф. В контексті обробки даних та аналітики, DAG є моделлю виконання операцій, де кожна операція представлена вузлом графа, а залежності між операціями відображаються ребрами графа
RDD	Resilient Distributed Dataset - основний абстрактний тип даних в Apache Spark
EDA	Exploratory Data Analysis - процес дослідження та аналізу даних з метою виявлення основних характеристик, закономірностей та взаємозв'язків між змінними
Баг	Помилка, вада або дефект в комп'ютерній програмі або системі, що викликає в ній неправильний або неочікуваний результат чи неочікувану поведінку
JVM	Java Virtual Machine – віртуальна машина, що виконує Java код
ML	Machine Learning – машинне навчання
ETL	Extract, Transform, Load - процес отримання даних з різних джерел (вилучення), їх перетворення (трансформація) і

	завантаження (завантаження) до цільової системи або бази даних
RAM	Random Access Memory - тип комп'ютерної пам'яті
IDE	Integrated Development Environment - інтегроване середовище розробки
WSL	Windows Subsystem for Linux - підсистема в операційній системі Windows, яка дозволяє виконувати Linux-дистрибутиви та запускати Linux-програми на Windows-платформі
noSQL	Not Only SQL - підхід до управління базами даних, який використовує моделі даних, відмінні від традиційних реляційних баз даних
ОС	Операційна система
CLI	Command-Line Interface - інтерфейс командного рядку
JAR	Java Archive - JAR є архівним файлом, який використовується для зберігання та розповсюдження Java-класів, ресурсів та інших елементів, необхідних для виконання Java-програми
Jedis	Java-клієнт, яка надає простий інтерфейс для взаємодії з Redis - нереляційною базою даних типу ключ-значення

ВСТУП

Із постійним розвитком технологій, в сучасному світі спостерігається величезна цифрова еволюція. Персональні комп'ютери, мобільні телефони і планшети, так само як і Інтернет і соціальні мережі, все більше і більше стають частиною нашого життя. За статистикою, щодня виробляється 2,5 квінтиліонних байт даних. Певна річ, завдяки цифровій революції кількість зростає з кожним роком лише все більше і більше. Можна сказати, живемо у світі потопу даних.

Зміни відбуваються і для сучасних компаній у різних галузях – як великих, так і малих, що активно зростають, накопичуючи величезну кількість даних, які збираються відусюди. Цінність наборів даних почала зростати і стала ключовою як для бізнес-рішень, як і досліджень у аналітиці великих даних – набору даних, що характеризується гігантським об'ємом, а також великою різноманітністю.

Не викликає жодних сумнівів, що останніми роками одним із найголовніших пріоритетів для технологічних компаній-гігантів стало отримання переваги над конкурентами шляхом обробки великих даних.

Актуальність теми полягає в тому, що Apache Spark є потужним інструментом для обробки великих обсягів даних у реальному часі, що протягом останніх років набув широкого використання для різних завдань, таких як машинний аналіз, графові дослідження, інтерактивний аналіз даних, обробка великих обсягів даних і аналітика в реальному часі.

Проте, використання Spark може призвести до необхідності постійного оновлення програмного забезпечення. Це може бути складним і тривалим процесом для компаній, що використовують фреймворк. Одним з рішень є розробка власних клієнтських бібліотек, які відповідають потребам компаній.

Оскільки використання Spark і його клієнтських бібліотек є критичними для багатьох компаній, необхідно проводити системне тестування, щоб переконатися в правильності функціонування, надійності та ефективності всіх компонентів. Для

полегшення та автоматизації цього процесу було розроблено спеціалізовану бібліотеку, яка спрощує проведення системного тестування Apache Spark і його клієнтських бібліотек.

Метою даної дипломної роботи є розробка бібліотеки для системного тестування Apache Spark, яка надасть зручні та ефективні засоби для створення, виконання та аналізу тестових сценаріїв. Бібліотека дозволить розробникам виконувати системні тести, що охоплюють різні аспекти функціональності та продуктивності Spark. Бібліотека для системного тестування Apache Spark буде допомагати забезпечити правильність функціонування та ефективність фреймворку, спрощуючи та автоматизуючи процес тестування його компонентів і клієнтських бібліотек.

Відповідно до поставленої мети роботи, визначено **основні завдання** дослідження:

- Провести аналіз наукової літератури про Spark і дослідити принципи його роботи;
- Поглибити знання про основи тестування та їх рівня;
- Проаналізувати мови програмування для розробки бібліотеки і роботі за Spark та обрати найкращу;
- Проаналізувати особливості різних середовищ розробки та обрати найефективніший;
- Розробити бібліотеку для системного тестування Spark та провести перевірку її роботи на прикладі Spark-застосунку.

Отже, у ході роботи проводиться аналіз наукової літератури про Spark, вивчаються принципи його роботи, а також розглядаються основи тестування та вибір мови програмування та середовища розробки для реалізації бібліотеки. Також розробляється бібліотека для системного тестування Spark і проводиться перевірка її роботи на прикладі Spark-застосунку.

Об’єктом дослідження є фреймворк для обробки даних Apache Spark.

Предметом дослідження є розробка бібліотеки для системного рівня тестування Apache Spark.

Методи дослідження, що використовувалися у процесі виконання роботи: порівняльний аналіз, логічний аналіз, обробка літературних джерел.

Наукова новизна полягає в дослідженні принципів роботи та основних складових фреймворку, порівняльний аналіз Apache Spark із іншим популярним фреймворком обробки даних Apache Hadoop, дослідженні основ тестування та автоматизації процесу тестування Spark шляхом розробки бібліотеки для написання автоматизованих тестів.

Практичне значення отриманих результатів. Результати роботи рекомендується використовувати розробникам, тестувальникам, дослідникам даних та іншим спеціалістам у галузі обробки даних, чия діяльність пов’язана із використанням Spark, для тестування власних Spark-застосунків та клієнтських бібліотек з метою забезпечення стабільності, продуктивності та зменшення ризиків під час розробки власних програмних продуктів та їх експлуатації.

РОЗДІЛ 1
ЗАГАЛЬНИЙ ОГЛЯД ПЛАТФОРМИ ДЛЯ ОБРОБКИ ДАНИХ APACHE SPARK

1.1. Визначення поняття великих даних

Перш ніж перейти до детального огляду Apache Spark, необхідно ознайомитися із поняттям «великі дані» (англ. «Big Data»), що являє собою величезний набір структурованих, напівструктурованих та неструктурованих даних, що були зібрані організаціями і можуть використовуватися для вирішення таких питань як машинне навчання, прогнозному моделюванні тощо.

Важливим є підкреслити, що звичайні бази даних просто не мають можливості обробляти ці великі набори даних. З метою вирішення цієї проблеми і були розроблені і поширені масштабовані кластери із розподіленими архітектурами систем, які ефективно зберігають, обробляють, а також використовують великі набори даних.

Згідно із даними, зібраними на 2023 рік, великі дані по суті є «золотом ери цифрових технологій» - щодня Google оброблює 8,5 мільярда пошукових запитів (99 тис/сек), користувачі WhatsApp обмінюються 65 мільярдами повідомлень тощо. До 2025 року світ буде виробляти трохи більше 10^9 терабайт даних, адже кількість інформації, що створюється, копіюється, передається, а також оброблюється у цифровому просторі є майже неосяжною.

В свою чергу, існує величезна кількість джерел, з яких безпосередньо генеруються великі дані, серед яких слід визначити друковані ЗМІ, дані соціальних мереж, таких як Facebook, Twitter, Snapchat тощо, фінансові дані, демографічні дані тощо.

Існують так звані «5 V's» великих даних, що являють собою визначальні властивості або розміри великих даних:

- Обсяг (англ. "volume") – визначає кількість даних;
- Швидкість (англ. "velocity") – визначає безпосередньо швидкість, з якою накопичуються дані з таких джерел, як мобільні телефони, комп'ютери тощо;
- Різноманітність (англ. "variety") – визначає природу даних. Може бути структурованою, напівструктурованою або неструктурованою із різноманітних джерел;
- Правдивість (англ. "veracity") – визначає якість та точність даних, адже у даних, що збираються, є ймовірність, що фрагменти будуть відсутні, або ж самі дані можуть бути неточними. Іншими словами, правдивість відноситься до рівня довіри до зібраних даних;
- Значення (англ. "value") – визначає здатність отримувати цінність із великих даних.

Саме через ці властивості і виникає проблема неможливості обробки даних традиційними обчислювальними системами, унаслідок чого з'являється необхідність розробки системи, що надає можливість обробки великих даних і отримувати максимальну користь із них. В результаті всього зазначеного вище, почали виникати розподілені файлові системи та системи обробки, такі як, наприклад, Hadoop MapReduce і HDFS. В результаті розширення властивостей даних систем і еволюції Hadoop, згодом була і розроблена безпосередньо Apache Spark, що детально розглядається в даній роботі.

Одним із найбільш широко використовуваних фреймворків, тобто програмного середовища, що орієнтоване на спрощення і автоматизацію процесу розробки програмного забезпечення, є Apache Hadoop, що застосовується у різних галузях для аналізу наборів даних і базується на простій моделі програмування MapReduce. Apache Hadoop, так само як і Apache Spark, є фреймворком із відкритим вихідним кодом, що полегшує обробку великих наборів даних. Детальне порівняння обох фреймворків, розроблених Apache Software Foundation буде розглянуто пізніше.

Apache Spark був розроблений із метою збільшення швидкості обчислювального процесу Hadoop. Взагалі, Apache Spark являє собою високопродуктивну систему розподілених обчислень і потокової обробки неструктурованих або слабоструктурованих даних. Іншими словами, це Big Data фреймворк. Apache Spark має відкритий вихідний код і активно використовується у сучасному світі розробниками і дослідниками даних (англ. “data scientists”). Apache Spark виділяється за рахунок своєї простоти у використанні, швидкості, і також гнучкості – поєднання цих трьох властивостей робить фреймворк одним із самих широко застосовуваних у сфері технологій, розробки програмного забезпечення і тестування. Тисячі сучасних малих і великих компаній у різних галузях, такі як Facebook, Microsoft, Netflix, LinkedIn, Apple, IBM тощо, активно використовують дану масштабовану систему обробки даних.

Все вище сказане надає можливість зазначити, що з 2009 року і на даний момент, Apache Spark став одним із найключовіших фреймворків обробки великих даних в світі із відкритим кодом, із більш ніж 1000 активних учасників. Враховуючи активний розвиток технологій, слід зазначити, що популярність Apache Spark буде зростати і далі, як і кількість випадків його використання. Spark все частіше буде використовуватися у якості ядра для обробки великих даних і буде надалі користуватися великим попитом у обробці фреймворків.

1.2. Історія розвитку Apache Spark

Як було зазначено раніше, історія Apache Spark розпочинається у 2009 році. Тоді фреймворк являв собою дослідницький проект в Berkley AMPLab – лабораторії у Каліфорнійському університеті, що досліджувала аналітику великих даних. Нині AMPLab відома як RISELab. Метою, що поставили перед собою дослідники проекту, було вирішення питання неефективності фреймворку Hadoop MapReduce при управлінні інтерактивними, а також ітеративними випадками обробки даних. Саме тому автори проекту придумали способи подолання цих проблем шляхом використання таких ідей, як обчислення у пам’яті і ефективний спосіб вирішення

проблеми усунення несправностей. Багато ідей, що в результаті були взяти у якості основи системи, були представлені лабораторією у різноманітних наукових роботах протягом багатьох років. Головною метою було створити фреймворк, що буде простішим і швидшим за MapReduce.

Як тільки практичне рішення, що перевершило Hadoop MapReduce, було розроблено у результаті досліджень, в 2010 році Apache Spark стає відкритим ресурсом під ліцензією типу BSD. Проект став успішним і вже в 2013 році став проектом вищого рівня Apache.

В результаті цього, група дослідників, що вела роботу над проектом, тобто рання команда AMPlab вирішила створити компанію, що нині називається «Databricks» і є основним комерційним управителем Spark. Це було зроблено із метою посилення проекту і його поширення.

Згодом використання проекту починає розширюватися і застосуватися понад 30 організаціями вже за межами Каліфорнійського університету. Отже, в 2014 році Spark починає використовуватися Databricks для сортування великомасштабних наборів даних, у результаті чого задає новий світовий рекорд.

На даний момент спільнота Apache Spark продовжує робити регулярні, постійні релізи і вносити у нього все більше нових функцій. Так, у 2014 році було випущено Spark 1.0, внаслідок чого було додано Spark SQL для роботи із структурованими даними, що являють собою таблиці із фіксованим форматом даних. Це надало можливість здійснювати нові потужні оптимізації між бібліотеками та API.

В свою чергу, вже у 2016 році було випущено Spark 2.0. Основна суть оновлення полягала у збільшенні зручності використання API, покращення продуктивності, операційні поліпшення і також оновлення структурованої потокової передачі тощо. Згідно даним офіційного сайту Apache Spark, реліз включав понад 2500 патчів від більш ніж 300 учасників.

На момент написання даної роботи, останньою версією Apache Spark є версія 3.4.0, що була випущена 13 квітня 2023 року, в якій було покращено Python API і зроблено його більш інтуїтивно зрозумілим, додано ряд нових функцій і покращена підтримка K8s.

Таким чином, на сьогоднішній день Spark став одним із найактивніших проектів в екосистемі Hadoop, що використовується для обробки даних. Починаючи з 2009 року, він отримав внесок понад 1000 розробниками із більш ніж 200 організацій.

1.3. Визначення Apache Spark

Отже, розглянемо трохи детальніше, що являє собою Apache Spark. Apache Spark - це багатомовний Big Data фреймворк для здійснення інженерії даних, та машинного навчання. Іншими словами, це уніфікований обчислювальний рушій і набір бібліотек для паралельної обробки даних на комп'ютерному кластері. Від своїх аналогів Spark відрізняється унікальним дизайном і інтерфейсом, а також високорівневим, простим і інтуїтивно зрозумілим API. Однією із головних його переваг є швидкість, що в багато разів перевищує швидкість обробки Hadoop MapReduce. Spark забезпечує простий спосіб вивчення API, а також являє собою потужний інструмент для інтерактивного аналізу даних.

Spark включає у собі різноманітні бібліотеки із складними API: для машинного навчання, для роботи із SQL для інтерактивних запитів, для обробки потокових даних, для обробки графіків тощо. Детальніше бібліотеки будуть розглядатися пізніше.

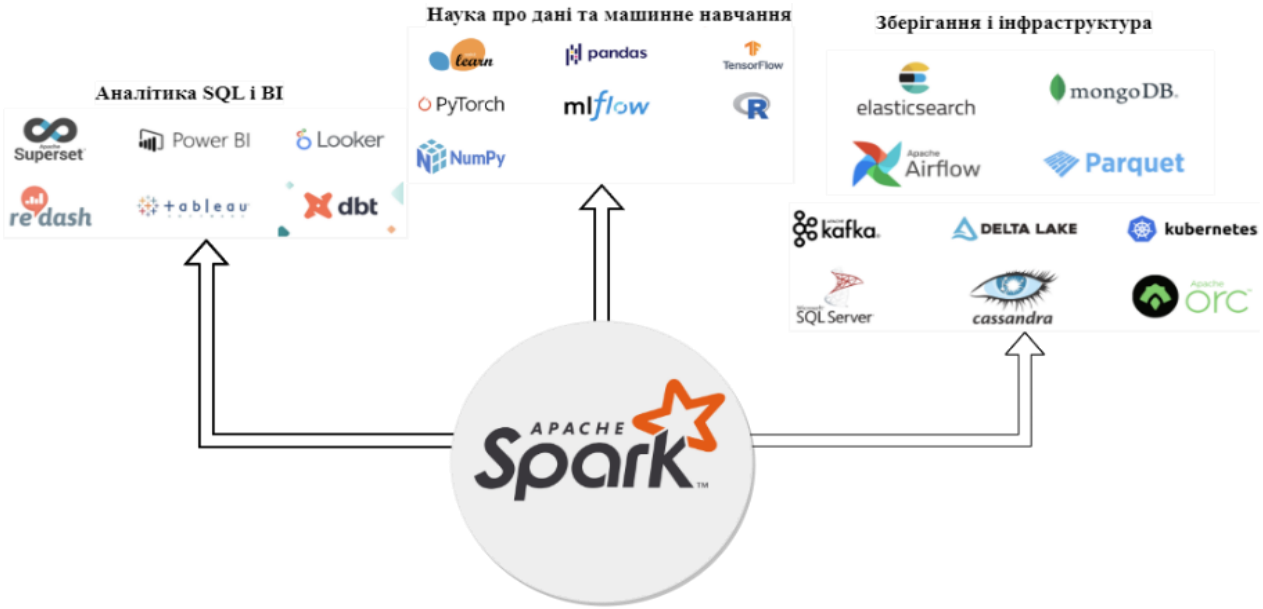

Рис. 1.1. Екосистема Apache Spark

Екосистема Apache Spark інтегрується з широким переліком різних фреймворків, завдяки чому можна набагато полегшити процес роботи із великими даними (рис. 1.1.).

Ключовими характеристиками Spark, на основі яких застосунок створювався, є:

- *Швидкість.* Однією із головних цілей, які ставили перед собою розробники Spark, було збільшення швидкості обробки даних. Досягнення високої швидкості було здійснено завдяки декільком способам. Перш за все, внутрішня реалізація Spark надає можливість використовувати усі переваги сучасних товарних серверів, такі як дешева вартість, сотні гігабайт пам'яті, використання паралельної обробки. По-друге, важливим є зазначити, що Apache Spark будує свої обчислення запитів як DAG, завдяки чому створюється ефективний обчислювальний граф, поділений на завдання. Завдання безпосередньо виконуються паралельно так званими «працівниками кластера». По-третє, завдяки можливостям фізичного рушія виконання під кодовою назвою Tungsten, що фокусується на підвищенні ефективності пам'яті та процесора для застосунків Spark, продуктивність набагато збільшується шляхом використання генерації цільного коду для створення компактного коду для виконання.
- *Простота використання*. Spark є простим у використанні, що було досягнуто завдяки фундаментальній абстракції простої логічної структури даних RDD, на основі чого будуються інші структуровані абстракції вищого рівня, такі як DataFrames і Datasets. Таким чином, Spark пропонує просту модель програмування, яку можна використовувати для побудови великих застосунків на різних мовах програмування.
- *Модульність*. Spark надає можливість застосовувати операції різних типів завдяки підтримці різних мов програмування, таких як Scala, Python, SQL і R. Spark складається із уніфікованих бібліотек, наприклад, Spark SQL, Spark MLlib, GraphX, що включають у собі різні компоненти і працюють під один рушій. Детальніше кожна із бібліотек буде розглядатися пізніше. І що є найважливішим, що Spark не вимагає використання різних рушіїв для різних робочих навантажень і є універсальним, тобто

необхідність вивчати окремі API відсутня. Іншими словами, усі зазначені вище бібліотеки вищого рівня, можна легко використовувати у одній програмі.

- *Розширюваність*. Тут мається на увазі можливість підтримки списку сторонніх пакетів і використання Spark для читання і обробки даних, що зберігаються у різноманітних джерелах, наприклад, Apache Cassandra, Apache HBase, Apache Hadoop, Apache Hive, Mongo DB, RDBMS тощо. Дані можна читати і з інших джерел, таких як Apache Kafka, Kinesis, Amazon S3, Azure Storage, т.д.

Перелік джерел даних Spark є досить широким, а також є незліченні джерела, що були розроблені безпосередньо спільнотою Spark. Серед основних джерел даних Spark виділяють:

- CSV - бібліотека Java, яка читає та записує файли у форматі Comma Separated Value;
- JSON – являє собою формат файлів і обміну даними, що використовується для представлення структурованих даних на основі синтаксису об’єктів Java Script;
- Parquet – формат файлів із відкритим кодом, що призначений для ефективного зберігання і пошуку даних, та доступний на декількох мовах;
- ORC – формат файлів, розроблений для подолання обмежень інших форматів файлів Hive;
- Підключення JDBC/ODBC – застосунків на основі SQL, що використовується для доступу до баз даних через SQL;
- Звичайні текстові файли.

Також, як зазначалося раніше, Spark має безліч джерел даних, які були створені безпосередньо спільнотою. У якості прикладу слід навести: Cassandra, HBase, MongoDB, AWS Redshift, XML, і так далі.

Серед ключових особливостей Apache Spark можна виділити:

- *Пакетні/потокові дані.* Spark надає можливість уніфіковувати обробку даних партіями, використовуючи бажані мови;
- *Аналітика SQL.* Відноситься до можливості виконувати швидкі розподілені запити ANSI SQL для інформаційних панелей і спеціальних звітів;

- *Наука про дані у масштабі.* Відноситься до можливості здійснювати EDA на петабайтному масштабі;
- *Машинне вивчення.* Відноситься до можливості тренувати алгоритми машинного навчання на ноутбуці і використовувати той самий код для масштабування відмовостійких кластерів тисяч машин.

Головна користь для розробників полягає в тому, що завдяки Spark вони мають можливість позбавитися частини програмного тягаря певних завдань за допомогою Spark API, який безпосередньо абстрагує велику частину важкої роботи розподілених обчислень і обробки великих даних.

1.4. Розбір ключових складових Spark

По своїй суті Spark є розподіленою системою, яка призначена для обробки великого обсягу даних швидко і ефективно. Ця розподілена система зазвичай розгортається у якості колекції машин, відомі також як кластер Spark, розмір якого буде як великим, так і малим. Найбільший публічно оголошений кластер Spark у світі налічує понад 8000 машин.

Розуміючи важливі компоненти архітектури Spark, розробники можуть розкрити потужність цього інструменту для створення масштабованих високопродуктивних програм.

На рис. 1.2, що наведено нижче, схематично проілюстровано архітектуру застосунку Apache Spark. Взагалі, застосунок Spark складається із двох частин. Перший являє собою логіку обробки даних додатків, що виражається безпосередньо за допомогою Spark API. Інша частина – це драйвер Spark, що є центральним координатором програми і взаємодіє із так званим «менеджером кластера» із метою з’ясування, на якій машині запускати логіку обробки даних. Точніше, для кожної машини, драйвер надсилає запит, щоб менеджер кластера запустив процес, що називається виконавцем Spark (англ. “Spark executor”) – компонент, що безпосередньо виконує призначені завдання.

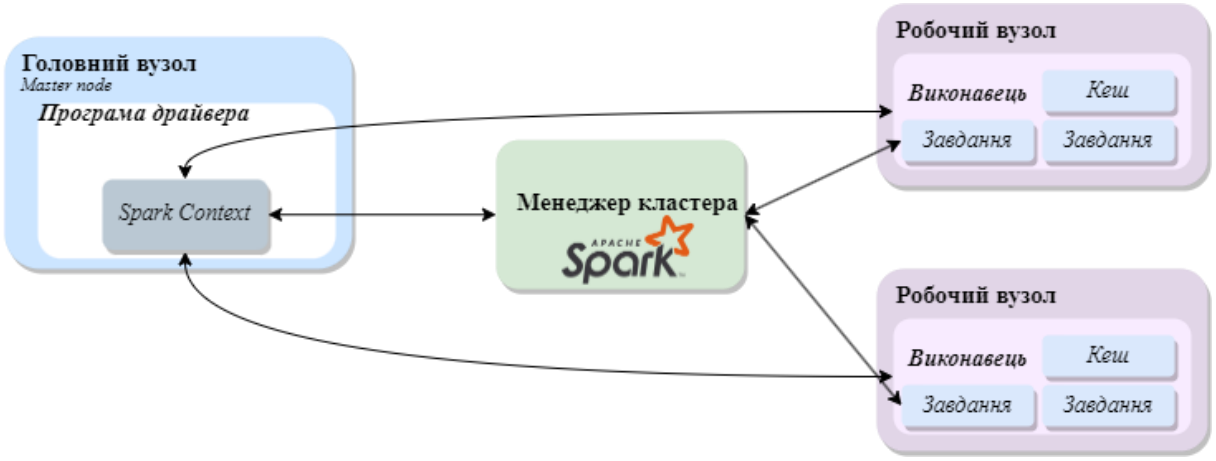

Рис. 1.2. Архітектура застосунку Apache Spark

Доповнюючи сказане вище, необхідним є визначити, що являє собою менеджер кластера. Менеджер кластера або кластер-менеджер – це компонент, що використовується для управління системою і відповідає за збереження інформації про те, де знаходяться виконавці, їх об'єм пам'яті, кількість процесорних ядер. Іншими словами, його основним обов'язком є організація роботи і призначення завдань кожному виконавцеві. Відповідно до цього, кожен виконавець має певні ресурси (наприклад, пам'ять), що він пропонує менеджеру кластера і після цього виконує призначене йому завдання. На сьогоднішній день, Spark усього підтримує три види кластерних менеджерів: Standalone Cluster Manager, Apache Mesos і Hadoop YARN. Standalone Cluster Manager включений у застосунок по замовчанню.

Отже, іншими словами слід сказати, що на фундаментальному рівні Apache Spark складається із двох компонентів:

- Драйвер, перед яким стоїть задача перетворення коду користувача в кілька завдань, які можна розподілити між робочими вузлами і є по суті серцем застосунку Spark.
- Виконавець, що виділяється в конкретну програму Spark і є відповідальним за збереження інформації про застосунок Spark, відповідь на програму або введення користувача, і зокрема, як було зазначено раніше, аналіз, розподіл і планування роботи між виконавцями.

Розглядаючи роль драйвера більш детальніше при роботі із Apache Spark, слід зазначити, що застосунок складається лише з одного драйвера, в той час як виконавець може бути один або декілька. Користувач має можливість вказувати кількість виконавців, що потрапляють на кожний робочий вузол через конфігурації.

Виконавці, у свою чергу, є відповідальними за фактичне виконання робіт, який їм призначає драйвер. Інакше виражаючись, виконавець по суті відповідає за виконання коду, що йому присвоюється і повідомляє про стан обчислення.

Важливим також є зазначити, що Apache Spark дотримується архітектури «Майстер-раб» (англ. «Master-Slave»), суть якого полягає в тому, що у якості центрального координатора використовується драйвер, що спілкується із усіма виконавцями, і кілька розподілених робочих вузлів. Кожен із робочих вузлів складається або з одного, або з декількох виконавців, що виконують певну задачу і реєструються у драйвері. Таким чином, драйвер постійно має інформацію про виконавців.

Отже, враховуючи зазначену вище інформацію, визначимо основні задачі, що стоять перед драйвером:

- Запуск програми за допомогою методу main(), в результаті чого створюється SparkSession або SparkContext.
- Перетворення коду користувача у завдання. Це відбувається наступним чином: драйвер аналізує код користувача і визначає можливі завдання за допомогою Lineage – компонент, що відноситься до RDD – також важливої складової Spark, що буде розглядатися пізніше.
- Допомагає створювати логічний і фізичний план. Логічний план – це абстракція усіх кроків перетворення, що треба виконати. Іншими словами, це внутрішнє представлення логіки обробки даних користувача у вигляді дерева операторів і виразу.

В свою чергу, фізичний план – це план, що генерується після логічного і визначає, як буде виконуватися логічний план на кластері. Кількість фізичних планів може дорівнювати одиниці, але також їх може бути і декілька. По суті, фізичний план

відповідає за виконання Spark. Під час фізичного плану також виконуються певні керовані оптимізації.

- Як тільки фізичний план генерується, драйвер визначає план виконання завдань і погоджує це із менеджером кластера.
- Координує роботу із усіма виконавцями і переглядає поточний набір виконавців, плануючи завдання.
- Відстежує дані у пам’яті виконавця.

У свою чергу, виконавець, що знаходиться у робочому вузлі і запускається у результаті координації із менеджером кластера, є відповідальним за наступне:

- Власне виконання окремого завдання.
- Повернення результатів драйверу.
- Зберігання даних у робочому вузлі.

1.5. Принцип роботи Apache Spark

Розглянувши основні складові застосунку Spark, перейдемо до опису принципів його роботи.

Взагалі, Spark часто розглядають як альтернативу Apache MapReduce – програмною парадигмою, або, іншими словами, моделлю програмування для обробки великих наборів даних із паралельним, розподіленим алгоритмом, який використовується в рамках Hadoop. Як і Spark, MapReduce також може використовуватися для розподіленої обробки даних разом із Hadoop. Проте, важливим є сказати, що Spark суттєво відрізняється від принципів MapReduce, як і його внутрішня система від багатьох інших традиційних систем. Щоб повністю зрозуміти потенціал фреймворку, важливим є детально розібрати принципи його роботи.

Як було сказано раніше, головною метою при розробці Apache Spark було вирішення обмежень MapReduce шляхом обробки у пам’яті, а також зменшенні кількості кроків у завданні та повторного використання даних через кілька паралельних операцій. Повторне використання здійснюється завдяки так званим

«DataFrames», абстрації над RDD, тобто стійким або ж незмінним розподіленим набором даних, що по своїй суті є колекцією об’єктів, закешованих у пам’яті. Завдяки цьому швидкість Spark в кілька разів більша, ніж MapReduce, особливо у процесі виконання машинного навчання та інтерактивної аналітики.

Важливим є зазначити, що самостійно Spark не можна назвати рішенням для зберігання даних, його завданням є оброблення даних та обчислення на JVM Spark – віртуальній машині Java. Зазвичай Spark використовують у тандемі із розподіленою системою зберігання, такою як, наприклад, HDFS, Cassandra, S3. Важливу роль у його діяльності також відіграє кластерний менеджер, детальніше про якого розповідалося раніше.

Отже, суть роботи Spark полягає у наступному: користувач пише програму для драйвера (або ж головного вузла) на кластерній обчислювальний системі, яка, в свою чергу, здатна виконувати операції над даними паралельно. Великі набори даних в Spark представляються як RDD – незмінні розподілені набори даних або колекції об’єктів, про які вже згадувалося раніше.

Необхідним є зупинитися на RDD, що є основною абстракцією Spark, і розглянути це поняття детальніше. RDD містить у собі так звані «розділи», що мають можливість обчислюватися на різних вузлах розподіленої системи.

Незмінні розподілені набори даних містяться і зберігаються у підлеглих вузлах, тобто виконавцях. Формально, незмінні розподілені набори даних є доступними лише для читання. Вони забезпечують функціональність для виконання обчислень у пам’яті відмовостійким способом.

Виділимо важливі характеристики, що пов’язані із поняттям стійких розподілених наборів даних, і є невід’ємною частиною моделі API програмування RDD, що лежить в основі усієї функціональності вищого рівня, а також безпосередньо використовуються Spark при роботі із RDD:

- Залежності (англ. dependencies) – певний список, що інструктує Spark для відтворення результатів. Завдяки цій характеристиці RDD можна назвати стійкою.

- Розділи, що вже згадувалися раніше. Мають у собі певну інформацію про місце розташування. Завдяки розділам Spark може розділити роботу для того, щоб здійснити розпаралелювання обчислень на розділах між виконавцями.
- Функція обчислення. Відноситься до обчислювальної функції, що виробляє ітератор для даних, які будуть зберігатися в RDD.

RDD можуть містити будь-який тип об'єктів Python, Java або Scala, включаючи визначені користувачем класи. RDD можуть експлуатуватися паралельно і є відмовостійким набором елементів. Створити стійкі розподілені набори даних можна наступними способами:

- Розпаралелювання вже існуючої колекції записів. Іншими словами, перетворення вже існуючого RDD.
- Шляхом посилання на набір даних у зовнішній системі зберігання даних.
- Перетворення DataFrame або DataSet.

Після запуску менеджер кластерів обробляє його і здійснює розподіл виконавців Spark по розподіленій системі згідно параметрам конфігурації, що були встановлені програмою Spark.

Далі сам рушій виконання Spark здійснює розподіл даних між виконавцями для обчислення.

Серед ключових характеристик RDD можна визначити:

1) *Лінива оцінка.*

Важливу роль у роботі Spark відіграє так звана «лінива оцінка». Суть цього поняття полягає в тому, що застосунок Spark замість того, щоб оцінювати кожне перетворення, оцінює RDD ліниво, тобто обчислює перетворення лише в тому випадку, коли остаточні дані RDD треба обчислити. В результаті перетворення RDD ми отримуємо новий RDD, адже стійкі розподілені набори даних є незмінними.

Отже, зупинимося на терміні «лінива оцінка» і що конкретно мається тут на увазі. На відміну від інших систем, що засновані на «дрібнозернистих» оновленнях змінюваних об’єктів, оцінка RDD є лінивою. Тобто, замість виклику певної комірки в таблиці шляхом зберігання проміжних результатів, Spark не починає обчислення розділів, поки не буде викликана дія – операція, що запускає оцінку розділів. Її

запускає планувальник (DAG), який визначає порядок обчислення RDD. Це дозволяє ефективно виконувати складні конвеєри, включаючи кілька етапів перетасування та агрегації. Також планувальник DAG будує спрямований ациклічний граф на основі залежностей між перетвореннями стійких розподілених наборів даних. У якості прикладу дії можна навести повернення даних до драйвера, чи запис даних до зовнішньої системи зберігання даних, тощо. Інакше виражаючись, по суті Spark оцінює дію «працюючи назад» для того, щоб визначити перелік кроків, які він повинен зробити, аби створити кожен об'єкт у кінцевому RDD. Сформулювавши цей перелік або ряд кроків – план виконання, - планувальник обчислює відсутні розділи для кожного етапу, поки не обчислить результат.

2) *Обчислення в пам'яті.*

Не менш важливу роль у діяльності Spark відіграє обчислення в пам'яті, завдяки чому вирішується одна із головних проблем MapReduce. Через здійснення обчислення безпосередньо в пам'яті, швидкість загального часу обробки Spark набагато збільшується. Відбувається це завдяки тому, що дані, обчисленні в пам'яті, зберігаються в оперативній пам'яті (Random-Access Memory), в результаті чого вартість пам'яті зменшується і з'являється можливість більш ефективного аналізу великих даних. Основними методами, що використовуються для цього є методи cashe () та persist ().

3) *Відмовостійкість.*

Відмовостійкість відноситься до того, що ймовірність втрати даних або ж повернення неточних результатів при роботі зі Spark є дуже і дуже низькою. Це пов'язано із можливістю відстеження інформації про лінії даних і таким чином, автоматичного відновлення втрачених даних при відмові. Іншими словами, кожен розділ даних містить інформацію про залежності, що необхідні для перерахунку розділу. В результаті чого, у випадку втрачання розділу, RDD просто має достатньо інформації, щоб обчислити його повторно. Завдяки можливості також розпаралелення обчислення, відновлення може бути швидким.

4) *Незмінність.*

Звісно, незмінність також є характеристикою стійких розподілених наборів даних. Завдяки незмінності можна запобігти ряду потенційних проблем, що виникають у результаті здійснення одночасного оновлення на різних потоках.

5) *Розбиття.*

Відноситься до автоматичного розподілу розділів, що не можуть вписуватися в один вузол. За рахунок цього збільшується кількість виконавців на кластері і паралелізм.

6) *Можливість налаштування місцезнаходження.*

Суть даної характеристики полягає у тому, що швидкість обчислення даних збільшується в результаті чіткого розташування переваг для обчислення розділів, тобто визначальної інформації про місце його розташування.

Таким чином, щойно було розглянуто поняття RDD та принцип роботи Apache Spark.

1.6. Визначення уніфікованого стеку Spark

На відміну від попередників, Spark забезпечує уніфікований рушій обробки даних – Spark Unified Stack, що побудований на основі Spark Core – ядра Spark. Основною задачею, що стоїть перед Spark Core, є забезпечення всіх необхідних функціональних можливостей для управління і запуску розподілених застосунків, а також потужної і загальної абстрації програмування для обробки даних – стійкими розподіленими наборами даних (RDD), про які вже згадувалося раніше. Отже, іншими словами, Spark Core – це базовий механізм для масштабованої паралельної і розподіленої обробки даних, що відповідає за управління пам’яттю, усунення несправностей, взаємодію із системами зберігання, планування, моніторинг і розподіл завдань у кластері тощо.

Ядро Spark складається із двох частин:

- Інфраструктура розподілених обчислень, що є відповідальною за розподіл, координацію, а також планування обчислювальних завдань, а також обробку збоїв і переміщення даних (перетасування даних).
- Абстракція програмування RDD, що вже розглядалася раніше. Тут можна додати, що API RDD підтримує декілька мов програмування і дозволяє користувачам передавати локальні функції для запуску на кластері.

Решта компонентів у стеці Spark призначені для запуску поверх ядра Spark. Таким чином, будь-які поліпшення або оптимізації, зроблені в Spark Core між версіями Spark, будуть автоматично доступні іншим компонентам.

1.7. Ключові бібліотеки Spark

На рис. 1.3 проілюстровано компоненти, що є частиною Spark Unified Stack і власне побудовані поверх ядра Spark, а саме:

- Spark SQL;
- Spark Steaming and Structured Streaming;
- Spark MLlib;
- GraphX.

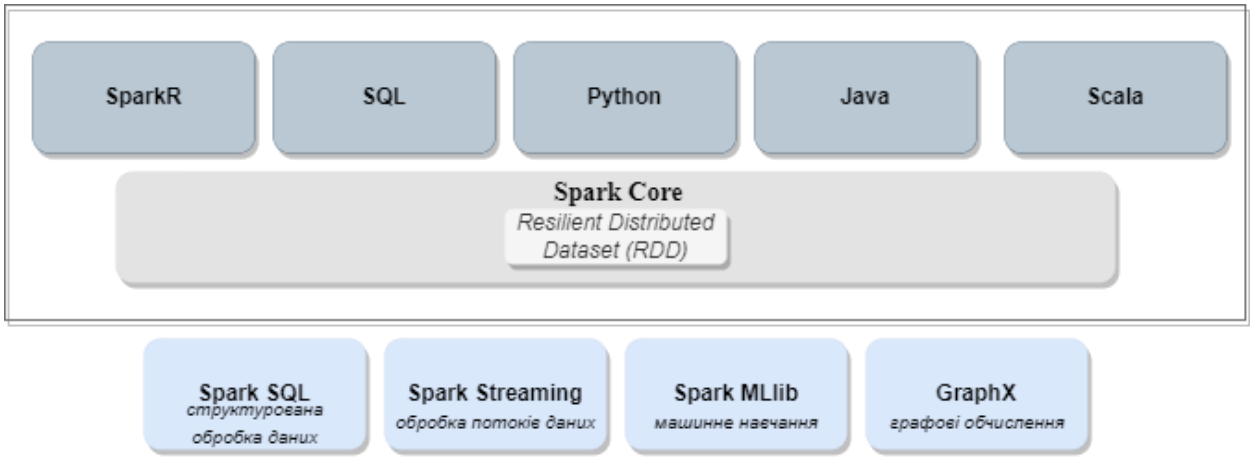

Рис. 1.3. Основні складові уніфікованого стеку Spark

Завдяки наявності різноманітних бібліотек, Spark забезпечує більш специфічну функціональність обробки даних. Розгляньмо кожну із вбудованих бібліотек детальніше.

1.7.1. Spark SQL

Переважна кількість операційних даних, що зберігаються сьогодні, зберігаються у табличному форматі в реляційних системах баз даних. Для роботи з ними і була створена бібліотека Spark SQL.

Девізом Spark SQL є «Писати менше коду, читати менше даних і дозволяти оптимізатору виконувати важку роботу».

Як можна зрозуміти по назві, Spark SQL (Structured Query Language) – це модуль, що надає можливість структурованої обробки даних в масштабі, що є важливим компонентом фреймворку обробки даних Spark і одним із найпопулярніших, що пов'язано із збільшенням рівня гнучкості, простоти використання та продуктивності. Головною задачею, що стоїть перед Spark SQL є обробка SQL-запитів із великими об'ємами даних.

Spark SQL включає у собі оптимізатор витрат, стовпчате сховище, а також генерацію і може масштабуватися до тисяч вузлів і багатогодинних запитів за допомогою Spark. Завдяки цьому модулю користувачі можуть використовувати абстракцію високо рівня, що виставлена через API DataFrames – розподілена колекція даних, що організована в іменовані стовпці. Виражаючись інакше, можна сказати, що концептуально DataFrame еквівалентний таблиці в реляційній базі даних. Важливим є додати, що API DataFrames здатні виконувати реляційні операції на кожному джерелі даних, як зовнішні джерела.

Деякі помилково припускають, що Spark SQL – база даних, проте це твердження не є коректним. Spark SQL не є базою даних, а модулем, що використовується для абстрагування DataFrames. Взагалі, Spark SQL може використовуватися для переліку різних завдань. Наприклад, при пакетній обробці, ітеративних робочих навантаженнях, потоковому передаванні тощо.

Процес запуску Spark SQL локально в системі є дуже простим, адже Spark працює на усіх операційних системах: і на Windows, і на Linux, і на macOS.

Spark SQL характеризується наступними особливостями:

1) *Інтеграція зі Spark.* Spark SQL є по суті компонентом застосунку Spark, тобто запити Spark SQL інтегровані із програмами Spark. Тобто, таким чином, ми можемо розбити запити структурованих даних всередині Spark, і робиться це за допомогою SQL чи API DataFrame;

2) *Єдиний доступ до даних.* Тут мається на увазі підтримка загального способу доступу до різних джерел даних, наприклад, Hive, Avro, Parquet, JSON, JDBC;

3) *Hive-сумісність.* Відноситься до можливості Spark SQL запускати немодифіковані запити Hive на поточні дані;

4) *Стандартне підключення,* що здійснюється через JDBC/ODBC;

5) *Продуктивніть і масштабованість.* Складові Spark SQL надають можливість читати інформацію з декількох джерел, а також використання додаткової інформації для виконання додаткової оптимізації, завдяки чому всьому досягається швидке виконання запитів.

Таким чином, ми розглянули Spark SQL – модуль Spark, що аналізує структуровані дані. Завдяки ньому можна забезпечити масштабованість, а також високу сумісність із системою.

1.7.2. Spark Streaming and Structured Streaming

Під «обробкою потоків» на увазі мається безперервна обробка нескінченних даних. Модулі Spark Streaming and Structured Streaming додають можливість обробки потокових даних із різних джерел в режимі реального часу, що надалі можуть відправлятися в бази даних, файлові системи тощо. Можливість обробляти дані є значною перевагою над конкурентами для багатьох компаній у різних галузях. Дані зазвичай можуть бути отримані з таких джерел, як наприклад, Kafka, Flume, Kinesis, Twitter, HDFS тощо.

DStream – це основна абстракція в першому поколінні Spark Streaming, суть якого полягає у реалізації інкрементної моделі обробки потоку шляхом розбиття вхідних даних на невеликі партії, що можуть регулярно поєднувати поточний стан обробки для отримання нових результатів. Інакше виражаючись, поєднання пакетних і інтерактивних запитів із потоковою обробкою можна легко зробити в Spark через уніфікований стек. DStream представив ідею мікропакетної обробки потоку, де потокове обчислення моделюється як безперервна серія невеликих завдань з пакетної обробки карт/скорочень (отже, «мікро-партій») на невеликих шматках даних потоку.

Отже, суть роботи Spark Streaming полягає у наступному: на початку дані діляться з вхідного потоку на мікро-партії, що обробляються в кластері Spark розподіленим чином із невеличкими детермінованими завданнями. Завдяки гнучкому плануванню завдань, Spark є відмовостійким, а детермінований характер завдань, в свою чергу, призводить до того, що згенеровані результати виведення будуть такими, що кожен запис введення був оброблений рівно один раз.

Після цього, в Spark версії 2.1. була представлена нова версія Structured Streaming, що була вже побудована на вершині рушія Spark SQL. Завдяки цьому діяльність розробників ще більше була спрощена, адже тепер потокові обчислення оброблялися так само, як і пакетні обчислення на статичних даних.

При роботі зі Spark Structured Streaming, зникає необхідність розробки і підтримки двох різних технологічних стеків для пакетної та потокової передачі. А також важливим є зазначити, що уніфіковані API полегшують перенесення існуючих завдань Spark на потокові завдання. Внутрішньо, за замовчуванням, Structured Streaming запити обробляються за допомогою мікропакетного процесора, який обробляє потоки даних як серію невеликих пакетних завдань, тим самим досягаючи кінцевих затримок до 100 мілісекунд.

Серед головних властивостей, що характеризують Spark Streaming, слід виділити:

- Швидке відновлення після збоїв і затримок;
- Краще балансування навантаження та використання ресурсів;

- Поєднання потокових даних зі статичними наборами даних та інтерактивними запитами;
- Вбудована інтеграція з розширеними бібліотеками обробки (SQL, машинне навчання, обробка графіків).

Таким чином, Structured Streaming забезпечує швидку, масштабовану, відмовостійку, наскрізну точну обробку потоку без потреби користувача міркувати про потокове передавання.

Проте, незважаючи на всі переваги, очевидним є і наявність певних доліків, такі як, наприклад, відсутність єдиного API для пакетної і потокової обробки, необхідність ручної оптимізації коду для того, щоб отримати максимальну продуктивність, відсутність нативної підтримки вікон часу-події (англ. «eventtime window») тощо.

Таким чином, щойно було розглянуто Spark Streaming і Structured Streaming, що є потужними і гнучкими модулями для роботи з Apache Spark.

1.7.3. Spark MLlib

Взагалі, що являє собою термін «машинне навчання»? Машинне навчання – це підгалузь штучного інтелекту, ідея якого полягає у дослідженні і побудові методів, які дозволять машинам навчатися. Іншими словами, по суті машинне навчання – це просто процес вилучення шаблонів із даних за допомогою статистики, лінейної алгебри та чисельної оптимізації. Нині застосування машинного навчання все більш і більш поширюється, починаючи від рушіїв рекомендацій до фільтрів спаму до виявлення шахрайства та багато іншого.

Spark - це єдиний механізм аналітики, який забезпечує екосистему для вживання даних, проектування функцій, навчання моделей та розгортання. Без Spark розробникам знадобиться багато розрізнених інструментів для виконання цього набору завдань, і вони все ще можуть боротися з масштабованістю.

В свою чергу, Spark MLlib – це бібліотека машинного навчання Spark, що містить у собі багато поширених алгоритмів машинного навчання, які можна

застосовувати до великих наборів даних. Іншими словами, Spark MLlib надає абстракції для управління та спрощення багатьох завдань побудови моделі машинного навчання. Завдяки даній бібліотеці машинне навчання стає масштабованим і легким. Термін «MLlib» використовується як загальних термін для творх різних пакетів машинного навчання Spark, а саме: spark.mllib і spark.ml. Моделі машинного навчання можуть бути навчені вченими даних з R або Python на будь-якому джерелі даних Hadoop, збережені за допомогою MLlib і імпортовані в конвеєр на основі Java або Scala.

Вже починаючи із версії Spark 2.0, API MLlib засновані на DataFrames і використовують усі їх переваги. Завдяки Spark MLlib, користувач має можливість легко і просто реалізовувати алгоритми машинного навчання, які носять ітеративний характер. На високому рівні Spark MLlib надає наступні інструменти:

- Перш за все, це, звісно, алгоритми машиного навчання, такі як регресія, кластеризація, класифікація, колаборативна фільтрація тощо;
- Наступним інструментом є функціонування – вилучення ознак, перетворення, зменшення розмірності та виділення;
- Пайплайн (від англ. “pipelines”) – тут маються на увазі інструменти для побудови, оцінки і налаштування машинного навчання;
- Стійкість – алгоритми збереження і навантаження;
- Утиліти – лінійна алгебра, статистика, обробки даних тощо.

Отже, можна сказати, що Spark MLlib є досить потужною бібліотекою машинного навчання із широким спектром алгоритмів і інструментів для обробки даних, навчання моделей тощо. Дана бібліотека за рахунок простоти використання, гнучкості і також потужної функціональності активно використовується при створенні застосунків машинного навчання.

1.7.4. GraphX

Одним із найновіших компонентів Spark є GraphX – це бібліотека графових обчислень Spark, що дає змогу аналізувати масштабовані дані, структуровані на основі графів. Бібліотека включає у собі зростаючу колекцію алгоритмів, які допомагають аналізувати дані користувача.

Обробка графів оперує структурами даних, що складаються з вершин і ребер, що з'єднують їх. Структура графічних даних часто використовується для представлення реальних мереж взаємопов'язаних сутностей, включаючи професійні соціальні мережі на LinkedIn, мережі підключених веб-сторінок в Інтернеті тощо. Spark GraphX - це компонент, який дозволяє графопаралельні обчислення, забезпечуючи абстракцію спрямованого мультиграфа з властивостями, прикріпленими до кожної вершини та ребра. Компонент GraphX включає колекцію загальних алгоритмів обробки графів, включаючи ранги сторінок, пов'язані компоненти, найкоротші шляхи та інші. Крім всього цього, також Spark GraphX може переглядати та маніпулювати графіками та обчисленнями.

Spark GraphX - це розподілений фреймворк обробки графів, побудований поверх Spark. GraphX забезпечує ETL (Extract, Transform & Load - витяг, перетворення та завантаження), дослідницький аналіз та ітеративні обчислення графів, щоб користувачі могли інтерактивно будувати та перетворювати структуру даних графів у масштабі. Він поставляється з дуже гнучким API і вибором розподілених алгоритмів Graph.

Spark GraphX є гнучкою, відмовостійкою і простою у використанні бібліотекою, що також забезпечує високу продуктивність.

1.7.5. SparkR

SparkR являє собою пакет R, що застосовується при роботі із Apache Spark. Взагалі, R – це популярна статична мова програмування, що підтримує обробку даних

і завдання машинного навчання, проте не може працювати із великими наборами даних.

SparkR забезпечує легкий фронт-енд для використання Apache Spark від R. Починаючи із версії Spark 3.4.0, SparkR забезпечує реалізацію розподілених кадрів даних, яка підтримує такі операції, як вибір, фільтрація, агрегація тощо, але на великих наборах даних. SparkR також підтримує розподілене машинне навчання за допомогою MLlib.

1.8. Порівняльна характеристика Apache Spark і Apache Hadoop

Apache Hadoop, як і Apache Spark, був розроблений Apache Software Foundation, і також користується високим попитом при роботі із великими наборами даних. Обидва фреймворки мають свої переваги і недоліки, а також певні суттєві відмінності, які ми і розглянемо.

Перш ніж перейти до порівняння фреймворків, необхідним є детальніше розглянути, що являє собою Apache Hadoop. Як вже зазначалося раніше, Apache Hadoop, так само як і Apache Spark – це фреймворк із відкритим вихідним кодом, що полегшує обробку великих наборів даних.

Hadoop надає можливості як і здійснювати розподілене зберігання, так і розподілені обчислення. Основними характеристиками Hadoop, які можна визначити, є розбиття даних та паралельне обчислення великих наборів даних.

Проект зародився у 2006 році у якості проекту Yahoo. Hadoop був розроблений на Java, але також доступний і на багатьох інших мовах програмування.

Як і Apache Spark, Hadoop по своїй суті є розподіленою архітектурою типу «Майстер-раб», яка складається з:

- HDFS (Hadoop Distributed File System – розподілена файлова система Hadoop), що використовується для зберігання даних. HDFS моделюється на основі Google File System, чудово працює із великими файлами, і також характеризується масштабованістю і доступністю.

- MapReduce, що використовується для обчислень. Іншими словами, це розподілений обчислювальний фреймворк. MapReduce змодельований на основі Google MapReduce. MapReduce надає можливість паралельної обробки, а також спрощує процес роботи із розподіленими системами, що надає можливість користувачеві сфокусуватися на вирішенні бізнес-потреб.

У табл. 1.1, наведеній нижче, наведено порівняння основних характеристик Apache Hadoop і Apache Spark.

Таблиця 1.1

Порівняння Apache Spark і Apache Hadoop

Характеристика	**Apache Hadoop**	**Apache Spark**
Швидкість обробки	Швидкість обробки даних із використанням MapReduce у Hadoop є повільною	Оброблює дані в 100 разів швидше ніж MapReduce завдяки обчисленням у пам'яті
Підтримувані дані	Підтримує пакетну обробку	Підтримує і пакетну обробку, і обробку в режимі реального часу
Довжина коду і мова програмування	Код довший за Spark, і так як він написаний мовою програмування Java	Код коротший і написаний мовою програмування Scala
Основні мови програмування	Java, Python	Java, R, Scala, Python, Spark, SQL
Відмовостійкість	Є відмовостійкою завдяки відтворенню блоків даних	Є відмовостійкою завдяки відстеженню інформації про лінії даних
Зберігання даних	На диску	В пам'яті
Масштабованість	Легко масштабована завдяки використанням вузлів і диску для зберігання файлів	В порівнянні з Hadoop більш складна

Отже, як бачимо, в порівнянні Spark із Hadoop слід зробити декілька висновків. З однієї сторони, швидкість обробка даних Spark є набагато вищою, ніж у Hadoop – власне, саме з цією метою в першу чергу і розроблявся Spark. З іншої сторони, незважаючи на те, що Spark підтримує пакетну обробку, не можна сказати, що він підходить для неї. Фреймворк більш орієнтований на його використання для реальної та ітеративної обробки даних. Незважаючи на певні відмінності і недоліки Hadoop у порівнянні зі Spark, Hadoop теж користується широким попитом у різних галузях.

1.9. Переваги і недоліки Apache Spark

Популярності і широкому використанні Spark сприяв ряд певних властивостей і переваг, серед яких можна виділити:

- *Швидкість обробки.* Як вже розбиралося раніше, швидкість обробки була зменшена завдяки використанню стійких розподілених наборів даних, завдяки яким можна зекономити час, що витрачається на читання і написання операцій.
- *Обчислення в пам'яті комп'ютера.* Також є однією із ключових переваг Spark, відноситься до зберігання інформації в RAM, завдяки чому швидкість доступу і аналітики набагато збільшується.
- *Відмовостійкість.* Одним із основних завдань, що стоїть перед RDD, є запобігання помилкам і невдачам, що власне і було досягнуто.
- *Зручний для використання API.* В Spark значну частину складних процесів механізму розподіленої розробки приховується за допомогою простих викликів методів, що позбавляє розробників програмного тягаря певних завдань.
- *Гнучкість.* Завдяки наявності широкого переліку різних вбудованих бібліотек, Spark може використовуватися як для пакетної обробки, так і для обробки потоків даних, а також для машинного навчання, для роботи з графами тощо.

Очевидно, що Apache Spark є гнучким і потужним фреймворком. Проте, незважаючи на усі значні переваги, Spark має і певні недоліки, серед яких можна виділити:

- *Відсутність файлової системи.* Одним із недоліків, з яким можна зіткнутися при роботі з Apache Spark, є те, що файлова система для управління файлами відсутня. Для вирішення цієї проблеми треба використовувати Hadoop, або ж іншу хмарну платформу для управління файлами.
- *Висока вартість.* Spark споживає великий об'єм пам'яті і відповідно до цього потребує величезний обсяг оперативної пам'яті для того, щоб здійснювати обчислення в пам'яті.
- *Обмеження при роботі із малими файлами.* При роботі із Spark разом із Hadoop, можна зіткнутися із певними проблемами, працюючи із невеликими файлами.
- *Необхідність ручної оптимізації.* Автоматичний процес оптимізації коду недоступний при роботі з Apache Spark, усю оптимізацію треба виконувати самостійно вручну.

Підбиваючи підсумки, слід сказати, що Spark є фреймворком із рядом значним переваг, завдяки яким він є одним із найбільш широко використовуваних фреймворків для роботи із наборами великих даних. Проте це не означає, що зіткнутися із обмеженнями чи проблемами при роботі із ним є майже неможливим.

1.10. Приклади застосування

Як зазначалося раніше, Apache Spark користується широким попитом у різних галузях, що частково пов'язано зі зростанням цінності великих даних, і частково з його потужністю, швидкістю, гнучкістю, а також зручністю використання. В основному Apache Spark використовується для:

- Обробки паралельних великих наборів даних;
- Виконання спеціальних або інтерактивних запитів для вивчення та візуалізації наборів даних;
- Побудови, навчання та оцінки моделей машинного навчання;
- Впровадження наскрізних конвеєрів даних з безлічі потоки даних;

- Аналізу графічних наборів даних.

Наведемо конкретні приклади застосування Spark у різних галузях.

1) *Використання у фінансовій індустрії.* Apache Spark набув широкого використання у банках і використовується для того, щоб здійснювати аналіз соціальних мереж, записів розмов, електронних листів, обговорень на форумів, журналів скарг тощо. За допомогою Spark компанії здатні таким чином оцінювати ризики, а також забезпечувати клієнтів кращим сервісом. Серед компаній, що використовують фреймворк, слід виділити Goldman Sachs, що є інвестиційним банковим гігантом, а також PayPal – система онлайн-оплати.

2) *Використання у туристичній індустрії.* Spark використовується туристичними компаніями для того, щоб забезпечити клієнтів ідеальною подорожжю шляхом пришвидшення персоналізованих рекомендацій клієнтів. Його використовують для прийняття швидких рішень на основі обробки в реальному часі. Серед компаній, що використовують Spark, можна виділити TripAdvisor – веб-сайт, що надає клієнтам можливість планувати подорожі до будь-якої країни світу. Також виділимо OpenTable – службу онлайн-бронювання, що також використовує Spark для тренування власних алгоритмів і тематичного моделювання.

3) *Використання у ігровій індустрії.* Попитом Spark користується і в ігровій індустрії, де він використовується для ідентифікації шаблонів із ігрових подій в реальному часі, завдяки чому компанії мають можливість збільшувати свій прибуток за використання вибіркової реклами, автоматичного коригування ігрових рівнів, моніторингу у грі тощо. Іншими словами, Spark використовується для того, щоб максимально покращити ігровий досвід користувачів, а також щоб поліпшити продуктивність і ефективність. Різні ігрові компанії користуються Spark, такі як Riot Games – незалежний творець комп'ютерних ігор, і Tencent – компанія, яка розробляє багатокористувацькі ігри. Обидві компанії за допомогою Spark намагаються покращити продуктивність обробки даних в реальному часі і виявлення причин, що призводять до сповільнення швидкості ігор.

4) *Використання в медіа-індустрії і індустрії розваг.* Spark використовується у індустрії розваг для цільової реклами, і лежить в основі

технологічного двигуна багатьох компаній. Серед прикладів можна визначити: Yahoo, Conviva, Netflix, Pinterest. Yahoo використовує Spark для персоналізації веб-сторінок, цільової реклами, а також для використання алгоритмів машинного навчання, аби покращити рекомендації новин, що отримують користувачі. Conviva використовує даний фреймворк для того, щоб покращити якість послуг своїм клієнтам, процес перегляду відео шляхом буферизації екрану, і щоб зменшити відтік клієнтів. Netflix активно використовує Spark для надання рекомендацій користувачам і обробляє 450 мільярдів подій на день завдяки йому. 0020Pinterest же використовує Spark для виявлення тенденцій і більшого розуміння поведінки користувачів на веб-сайті.

5) *Використання у індустрії охорони здоров'я.* Так, Apache Spark поступово починає набирати популярності і в даній індустрії і береться за основу багатьох медичних додатків, для аналізу записів пацієнтів, обробки даних генома і аналізу усіх хімічних сполук тощо. Більш того, за допомогою використання машинного навчання в Spark, лікарі мають можливість прогнозувати захворювання згідно інформації про поведінкові і фізіологічні атрибути пацієнтів. Серед компаній, що використовують Spark, слід виділити MyFitnessPal, що допомагає людям вести здоровий спосіб життя за допомого дієт і фізичних вправ. MyFitnessPal використовує Spark для обробки даних користувача, сканування даних про калорійність їжі тощо.

6) *Використання у індустрії електронної комерції.* Apache Spark використовується для аналізу видів продуктів, що продають клієнти для того, щоб доречно визначати магазини для зв'язку, виділення функцій із даних зображення, надання цільових пропозицій і покращення взаємодії із клієнтами, а також підвищення продуктивності. Серед компаній, що використовуються Apache можна виділити Alibaba і eBay, що є одними із найбільших у світі платформ електронної комерції.

7) *Використання у індустрії програмного забезпечення і інформаційних послуг.* Велику роль Spark відіграє і в індустрії програмного забезпечення. В першу чергу слід виділити DataBricks – компанія, що була заснована учасниками AMPLab і нині активно використовує Spark. Додатково можна виділити Hearst – лідуючу світову

інформаційну компанію, що завдяки використанню обробки потокових даних у режимі реального часу збирає дані про популярні новини.

8) *Використання у індустрії електроенергетики.* Так, Apache Spark може використовуватися для машинного навчання задля аналізу показань лічильників газу та електроенергії, даних про електроспоживання у реальному часі для того, щоб надавати клієнтам інформацію про витрати. У якості прикладу можна навести компанію British Gas, що є лідером у Великій Британії. Компанія використовує Spark у основі свого механізму інженерії даних і машинного навчання. У якості прикладу також наведемо лідера фінансових послуг FINRA, що використовує Spark для аналізу даних мільярдів ринкових подій у режимі реального часу і їх впорядкування.

Отже, як бачимо, Apache Spark є фреймворком для розподіленої обробки даних, що активно використовується у різних галузях, і значно полегшує роботу із відповідними застосунками, системами.

1.11. Висновки до розділу 1

В даному розділі було розглянуто поняття великих даних та особливості фреймворку для обробки даних, принципи його роботи і складові, а також переваги і недоліки.

Отже, із більш частим використанням великих даних, популярності набув потужний, швидкий, продуктивний, гнучкий, масштабований і простий у використанні фреймворк Apache Spark, що має відкритий вихідний код. Spark активно застосовується у різних галузях, таких як фінансова, охорони здоров’я, туристична, електронної комерції і індустрії медіа і розваг. Він активно використовується провідними компаніями світу, такими як Apple, Microsoft, Facebook, Twitter і так далі.

Apache Spark має ряд переваг, серед яких виділяють високу швидкість, відмовостійкість, простий і інтуїтивно зрозумілий API, що значно спрощує процес обробки великих даних. Незважаючи на це, при роботі зі Spark також є можливість зіткнутися із рядом обмежень і недоліків. Наприклад, однією із головних проблем

Spark є висока вартість, що відноситься до споживання великого обсягу пам’яті, через що робота із фреймворком може ускладнюватися і бути дорогою.

Однією із ключових рис Apache Spark є обчислення і обробка в пам’яті завдяки стійким розподіленим наборам даних, що сприяє збільшенню його швидкості. В результаті цього, швидкість Spark значно переважає у порівнянні із Hadoop – іншим широко використовуваним при обробці великих даних фреймворком.

Apache Spark - це потужний та широко використовуваний фреймворк обробки великих даних з відкритим вихідним кодом, який забезпечує швидку та гнучку обробку великомасштабних наборів даних. Він був розроблений для вирішення обмежень парадигми Hadoop MapReduce, і він швидко став рішенням для реального часу та пакетної обробки великих даних. Популярності Spark також сприяє великий перелік вбудованих бібліотек, що можуть використовуватися для обробки структурованих даних, машинного навчання, обробки графів тощо.

РОЗДІЛ 2
ОСНОВИ ТЕСТУВАННЯ І ОГЛЯД ПРОГРАМНИХ ЗАСОБІВ

2.1. Необхідність тестування Spark

Як було зазначено у минулому розділі, Spark – це фреймворк із відкритим кодом, що набув широкого використання лідерами у різних сферах діяльності, включаючи сферу комп'ютерних технологій. Завдяки відкритому коду, компанії мають можливість розробляти власні клієнтські бібліотеки із додатковими функціями, орієнтованими на конкретні випадки використання і потреби компанії.

Використання цієї практики набуло широкого використання у зв'язку із тим, що Spark слідує підходу на основі випуску замість оновлення версій.

Підхід на основі випуску – це підхід, суть якого полягає у випуску нових версій замість додаткових оновлень. Іншими словами, замість того, щоб вносити зміни в межах певної версії, випускається нова версія із зміненим кодом, виправленими помилками і новим функціоналом.

В свою чергу, оновлення – це внесення змін у межах певної версії програмного забезпечення замість випуску нових версій.

Таким чином, унаслідок виправлення певних помилок та багів, що були присутні у певній версії Spark, замість внесення цих змін і прямого оновлення у рамках тієї ж версії, випускається окрема нова версія фреймворку. Це вимагає від компаній постійного оновлення власного програмного забезпечення і підключення нової версії Spark, що є зазвичай дуже невигідним і довгим процесом, потребує велику кількість ресурсів і затрат.

Серед найрозповсюдженіших бібліотек, розроблених компаніями під власні потреби на основі Spark, слід виділити:

- *Cloudera (Cloudera Rel, Cloudera Libs)* – інтегроване рішення для підтримки Spark, що використовує інструменти фреймворку із додаванням необхідних інтеграцій безпеки та управління.
- *Spring Lib M* – репозиторій Maven, що використовується для роботи зі Spark.
- *Azure Synapse Analytics* – сервіс для корпоративної аналітики, що надає можливість більш швидкого аналізу сховищ даних і систем великих даних, що поєднує у собі технології SQL і Spark для роботи із великими даними.
- *IBM Analytics Engine* - хмарний дистрибутив Spark, розроблений IBM.

Отже, Spark являє собою потужний і складний застосунок, відкритий код якого надає можливість розробки клієнтської бібліотеки для специфічних потреб різних компаній, що розширюють можливості фреймворку. З його активним розвитком та поширенням випадків його використання з'явилася необхідність постійного його тестування у зв'язку із постійним введенням нового функціоналу та інструментів.

Тестування допомагає забезпечити якість та надійність застосунків Spark, виявляти та запобігати проблемам, які можуть призвести до пошкодження даних, неправильних результатів або системних збоїв. Перевіряючи функціональність і правильність застосунків Spark, тестування допомагає підвищити впевненість у їх продуктивності.

В даній роботі ми будемо розробляти бібліотеку системного тестування клієнтських бібліотек і використовувати її на прикладі самого фреймворку Spark, а саме – Spark-застосунку, що обчислює число π методом Монте-Карло. Таким чином, її можна буде використовувати як і для тестування власних бібліотек компаніями, так і для тестування безпосередньо Spark та його складових.

2.1.1. Визначення поняття тестування

Необхідним є докладніше розглянути поняття тестування, його особливості і рівні.

Тестування - це процес оцінки системи або її компонентів, щоб перевірити, чи відповідає вона заданим вимогам чи ні. Простіше кажучи, тестування - це виконання послідовності дій для виявлення прогалин, помилок і помилок, які суперечать вимогам, що допомагає оцінювати функціональність програмного забезпечення .

Тестування відіграє важливу роль у процесі розробки програмного забезпечення, що визначає помилки і проблеми у процесі розробки, щоб вони були виправлені до запуску продукту, завдяки чому якість продукту значно підвищується. Важливим є зазначити, що головною роллю тестування є не демонстрація правильної роботи, а виявлення прихованих дефектів .

Отже, завдяки тестуванню ми можемо:

- Заздалегідь визначити дефекти;
- Покращити якість програмного продукту;
- Визначити вразливі місця, незахищений код програми, чим можуть скористатися зловмисники;
- Визначити рівень масштабованості.

Тестування поділяється на статичне, що включає у собі огляди та статичний аналіз, та динамічне, що передбачає власне виконання програмного забезпечення.

Без сумнівів, такий великий і багатошаровий застосунок, як Spark, що містить у собі широкий ряд вбудованих бібліотек і інструментів, активно оновлюється і широко використовується у різних сферах, вимагає постійної перевірки його функціоналу. Ефективне тестування може допомогти виявити вузькі місця, неефективність та проблеми продуктивності в застосунках Spark. Тестування допоможе виявити та вирішити проблеми відмовостійкості, забезпечуючи, що Spark може відновитися після збоїв та продовжувати обробку без втрати даних або невідповідностей.

2.1.2. Рівні тестування

Однією із найключовіших концепцій тестування є рівні тестування - групи активностей тестування, які організовуються та управляються як єдине ціле. Кожен рівень тестування складається із різних методологій і підходів, що можуть бути застосовані під час тестування програмного забезпечення. Кожен рівень тестування переслідує специфічні цілі, тестує конкретні об’єкти і може мати перелік прогнозованих дефектів.

Виділяють наступні рівні тестування (рис. 2.1) :

- *Модульне тестування,* або ж тестування компонентів – використовується для перевірки роботи конкретних елементів програми, тобто є процесом тестування певної скомпільованої програми. Отже, модульне тестування фокусується на перевірці функціональності певних частин програми, які тестуються окремо. Зазвичай, у процесі даного рівня тестування здійснюється створення так званих «юніт-тестів».

Модульне тестування є важливою частиною процесу розробки програмного продукту, адже завдяки ньому можна зменшити ймовірність виникнення дефектів, помилок та збоїв у процесі інших рівнів тестування і зекономити ресурси.

Рис. 2.1. Схематичне зображення рівнів тестування.

Модульне тестування передбачає застосуваня відповідних інструментів, що полегшують процес автоматизації і тестування. Серед найрозповсюдженіших таких інструментів виділяють:

1) Junit – для тестування мовою програмування Java.

2) Nunit – для тестування мовами групи .net.

3) JMockit – інструмент для здійснення модульного тестування із відкритим кодом.

4) EMMA – інструмент із відкритим кодом для аналізу коду написаного мовою програмування Java.

5) PHPUnit – використовується розробниками, що працюють з PHP.

- *Інтеграційне тестування* – тестування, що здійснюється для перевірки взаємодії компонентів системи. Здійснюється для перевірки правильності взаємодії модулів, чи працюють вони так як очікується і чи не виникають дефекти при інтеграції. Іншими словами, даний рівень тестування відповідає за перевірку взаємодії модулів як групи.

Отже, основною метою, що ставиться під час інтеграційного тестування – це перевірка взаємодії модулів системи. Інтеграційне тестування, як правило, здійснюється після модульного тестування. Інтеграційне тестування є важливим при розробці багатошарової системи, що містить у собі безліч складних модулів, і завдяки ньому можна перевірити правильність реалізованої розробником логіки.

- *Системне тестування* – це тестування системи у цілому, включаючи усі її компоненти. Іншими словами, основною метою системного рівня тестування є перевірка точності роботи програми та її повноти, а також правильність розроблених функцій.

Системне тестування фокусується на усіх елементах та можливостях системи або продукту, і в його процесі необхідним є визначити, чи працюють усі компоненти системи так, як очікується. Тобто, в процесі системного тестування важливим моментом є продемонструвати правильну поведінку системи.

Об’єктом системного тестування є вся система чи продукт.

- *Приймальне тестування* – це рівень тестування, що включає у собі валідацію та демонстрацію готовності продукту до розгортання. Його основними формами є:
  - Приймальне тестування користувача;
  - Експлуатаційне приймальне тестування;
  - Контрактне і регуляторне приймальне тестування;
  - Альфа-тестування;
  - Бета-тестування.

Іншими словами, приймальне тестування фокусується на визначенні відповідності програмного продукту певним критеріям, валідації і варифікації програми.

Розробка бібліотеки для тестування Spark та клієнтських бібліотек на його основі буде для системного рівня тестування, так як Spark є багатошаровою системою, що складається із багатьох модулів, які полегшують роботу при різних видах діяльності, буть то обробка великих даних чи процес машинного навчання. Завдяки бібліотеці для системного рівня тестування, ми зможемо оцінювати застосунок в умовах, що є подібними до реальних умов, у яких зазвичай розгортається Spark, а також перевірити відмовостійкість усіх компонентів екосистеми фреймворку, його продуктивність і масштабованість.

2.1.3. Переваги і недоліки автоматизованого тестування

Тестування поділяється на ручне і автоматизоване. Ручне тестування – це різновид тестування програмного забезпечення, суть якого полягає у здійсненні тестових кейсів (виконання певних дій та/або умов, що необхідні для перевірки функціональності програмного продукту) без використання автоматизованих інструментів тестування.

В свою чергу, автоматизоване тестування – це тестування, що передбачає використання скриптових послідовностей, що виконуються засобами тестування,

тобто певних інструментів тестування, які здійснюють перевірку програмного забезпечення.

Серед переваг автоматизованого тестування слід виділити:

- Спрощення процесу тестування, в першу чергу, завдяки скороченню повторювальної ручної роботи;
- Більш висока точність;
- Більш висока зручність і більш простий доступ до інформації;
- Більша ймовірність виявлення помилок та дефектів;
- Збільшене покриття – того, який відсоток коду охоплюють тесткейси.

З іншої сторони, автоматизоване тестування має свої переваги і недоліки.

Серед них слід визначити:

- Більш високі витрати;
- Висока ймовірність виникнення false positive and false negative результатів у процесі тестування, що спричиняє незручності і неточності.
- Іноді час, вартість та зусилля для введення інструменту можуть бути недооцінені;
- Складність;
- Менша гнучкість, що вимагає створення тестів для кожного нового середовища;
- Складність у розробці ремонтопридатних тестів;
- Існують певні обмеження, коли мова йде про тестування зручності, UX-тестування (перевірка зрозумілості, зручності і простоти інтерфейсу) тощо.

Тестування фреймворку Spark буде автоматизованим, спеціально для цього ми розробимо бібліотеку з використанням різних інструментів, що спростять процес тестування. У результаті, завдяки розробленій бібліотеці ми будемо мати можливість перевіряти правильність роботи компонентів Spark.

Завдяки автоматизації процесу тестування Spark, ми зможемо зробити це більш ефективно і зекономити час за рахунок повторного використання коду, тестових

випадків тощо. Бібліотека тестування також надасть можливість імітації реалістичних сценаріїв роботи зі Spark, таких як обробка великих наборів даних.

Автоматизовані тести нададуть нам можливість створювати набір регресійних тестів, які охоплюють критичну функціональність і нададуть можливість перевірки впливу оновлень коду Spark на загальну поведінку системи. Отже, враховуючи постійні оновлення, ми будемо мати можливість запускати тести і для нових версій фреймворку.

Автоматизація процесу тестування також забезпечить більше покриття, і ми зможемо охопити більшу кількість тест-кейсів, а також більш надійніше і ефективніше тестувати фреймворк, так як ручне тестування може призвести до помилок і не гарантує повного покриття, може бути трудомістким, враховуючи наскільки складним і багатошаровим є Spark.

Отже, тестування Apache Spark має важливе значення для забезпечення якості, правильності, продуктивності, сумісності, масштабованості, відмовостійкості та інтеграції застосунків Spark. Завдяки розробці бібліотеки для системного тестування Spark, ми зможемо зробити процес тестування фреймворку та бібліотек, створених на його основі, більш зручнішим і ефективнішим, а також підвищити його продуктивність.

2.2. Вибір мови програмування для бібліотеки

Як зазначалося у минулому розділі, працювати зі Spark можна з використанням наступних мов програмування: Scala, Java, Python і R. Розглянемо кожну із перелічених мов детальніше, а також проаналізуємо переваги і недоліки.

- *Scala* - назва походить від «scalable language» - масштабована мова. Саме на цій мові написаний застосунок Spark, тому аналітики даних, розробники та інші спеціалісти, що працюють з фреймворком, часто використовують її, а саму мову вважають однією із найкращих для роботи зі Spark. Основна ідея Scala полягає у поєднанні концепцій об'єктно-орієнтованого і функціонального програмування у

статично типізованій мові, завдяки чому Scala має ряд інших сильних сторін і має великий потенціал для розробки потужного програмного забезпечення.

Серед переваг Scala слід визначити продуктивність і потужність, широкий перелік бібліотек і інструментів та лаконічність. Так як Scala є нативною мовою Spark, забезпечується повний доступ до усіх функцій фреймворку.

Основним же недоліком є складність у порівнянні із Java і Python, є значно менша кількість підтримуваних інструментів та бібліотек, що частково зумовлено обмеженою спільнотою, що користується даною мовою програмування.

- *Java* – вважається однією із трьох найкращих мов для роботі зі Spark, що була випущена у 1995 році. Серед її переваг слід визначити продуктивність, простоту у вивченні мови, незалежність від платформи, завдяки чому застосунки Spark можуть працювати на будь-якій системі, сумісній із JVM.

Завдяки широкій спільноті, що виконує розробку мовою програмування Java, також у її рамках доступна велика кількість фреймворків і бібліотек, що спрощують процес обробки даних. Її відмовостійкість і масштабованість також є значними перевагами.

Нині Java є основною мовою для розробки програмного забезпечення. Основною ідеєю, що лежить у мові програмування, є використання принципів об’єктно-орієнтованих концепцій (абстракція, інкапсуляція, спадкоємність і поліморфізм) і повне ігнорування значної кількості традиційних методів програмування.

- *Python* – мова програмування, що входить у трійку найкращих мов програмування для роботи зі Spark. Це обумовлено тим, що дана мова є простою і лаконічною, є гнучкою, а її екосистема включає у собі широкий перелік бібліотек та інструментів. З іншої ж сторони, нижча продуктивність Python в порівнянні із Scala і Java є значним недоліком, а також слід виділити великі витрати пам’яті.

Мова програмування була випущена у1991 році. Python включає у собі концепції об’єктно-орієнтованого програмування, є універсальною мовою програмування із простим синтаксисом, і використовує динамічну типізацію – використання усіх типів під час виконання програми.

Усі зазначені вище переваги та ідеї, що лежать у основі Python, сприяли її популяризації серед як розробників-новачків, так і досвідчених розробників.

- *R* – мова, що використовується для статистичного аналізу, графічного представлення та звітності, що була випущена у 1993 році.

Серед її переваг слід виділити незалежність від платформи, доступність. Мова R широко використовується для машинного навчання, а також спрощує роботу із великими неструктурованими наборами даних. R також включає в собі ефективну систему обробки і зберігання даних, підтримує оператори для розрахунків на масивах (включаючи матриці), графічні засоби для аналізу даних, а також інтегрований набір проміжних інструментів для аналізу даних.

Серед недоліків слід відзначити, що по кількості доступних бібліотек та інструментів для роботи зі Spark, як і продуктивності, R значно програє вище зазначеним мовам. Мова програмування R є досить вузьконаправленою і використовується для специфічних задач. Незважаючи на це, ця мова програмування є чудовим рішенням для роботи зі Spark для аналітиків, дослідників даних тощо.

Для розробки бібліотеки для системного тестування Spark буде використовуватися Java, у зв’язку із наступними перевагами:

- Продуктивність, що обумовлена використанням JVM. Це є важливою перевагою при роботі із таким ресурсномістким фреймворком як Spark. Саме завдяки продуктивності і масштабованості Java є однією із найкращих мов для тестування.
- Широкий обсяг доступних бібліотек та інструментів.
- Java є об’єктно-орієнтованою мовою програмування, завдяки чому процес розробки бібліотеки можна спростити завдяки повторному використанню тестового коду.
- Надійність.

Таким чином, Spark пропонує широкий вибір різних мов програмування, що надають можливість вирішувати перелік різних питань. В даній роботі ми будемо використовувати Java завдяки ряду її переваг при розробці і роботі зі Spark, які були зазначені вище.

2.3. Вибір середовища розробки

Завдяки тому, що Spark підтримує різноманітні мови програмування, перелік IDE, в яких можна з ним працювати, є досить широким і різноманітним. Розглянемо найрозповсюдженіші середовища для роботи зі Spark.

1) *PyCharm* – IDE и для мови програмування Python. PyCharm надає ряд корисних функцій, таких як автоматизація тестування, аналіз продуктивності, підтримка широкого переліку інструментів для роботи із Python.

Так як Python є однією із найбільш використовуваних мов при роботі із Spark наряду із Java і Scala, дане середовище розробки набуло широкого використання при роботі із фреймворком обробки великих даних.

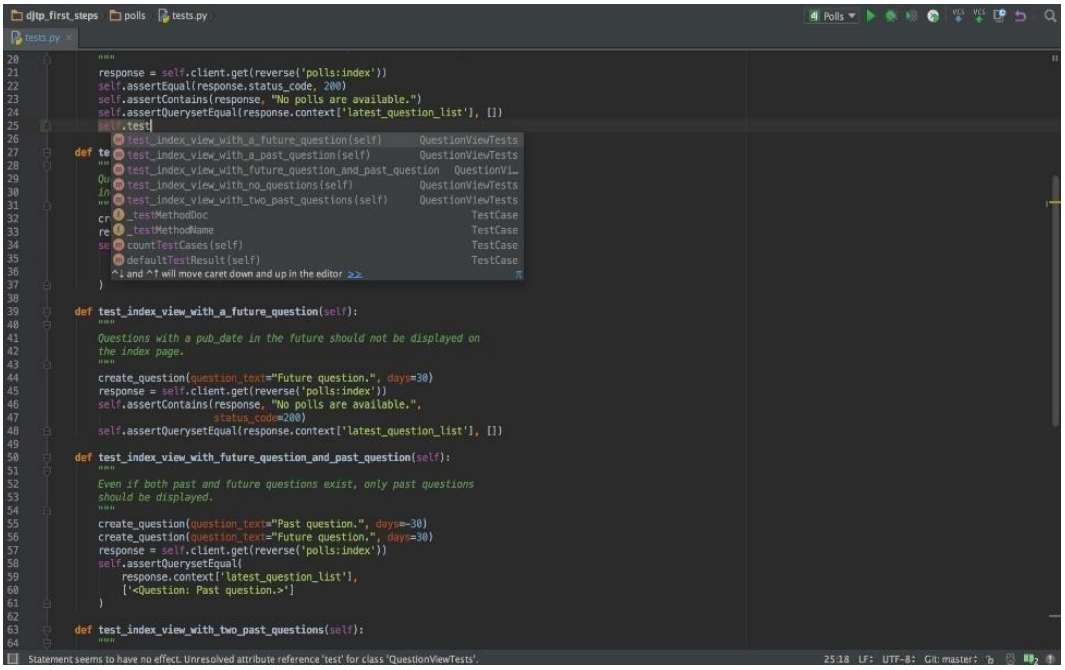

Рис. 2.2. Інтерфейс PyCharm

Отже, серед переваг PyCharm слід виділити:

- Висока продуктивність;
- Підтримка усіх необхідних інструментів для роботи із Python;
- Можливість збільшення якості коду завдяки інтелектуальному завершенню коду;

- Відмінні функції співпраці;
- Широка інтеграція екосистем.

З іншої сторони, серед недоліків даного середовища розробки необхідним є виділити:

- Значне використання системних ресурсів;
- Ліцензія, що підтримує більше функцій, є платною;
- Обмежена підтримка мови, іншими словами, PyCharm фокусується виключно на Python і не є найкращим рішенням для інших мов програмування;
- Незручність для початківця, що обумовлена широким переліком доступних функцій і інструментів.

2) *Eclipse IDE* – середовище розробки, що найчастіше використовується для розробки мовою програмування Java. Включає у собі велику кількість інструментів для зручної розробки, що є вбудованими, або можуть бути встановленими.

Серед переваг Eclipse слід визначити:

- Універсальність;
- Зручність;
- Величезна екосистема;
- Можливість інтеграції різних інструментів збірки;
- Інтуїтивно зрозумілий інтерфейс;
- Простий процес установки.

Незважаючи на усі переваги, Eclipse має і свої недоліки, що можуть призвести до проблем у процесі роботи:

- Значне використання системних ресурсів;
- Необхідність додаткової конфігурації;
- Нові функції і виправлення багів та дефектів рідко і повільніше випускаються у порівнянні із іншими IDE;
- Обмеження розробки мобільних застосунків.

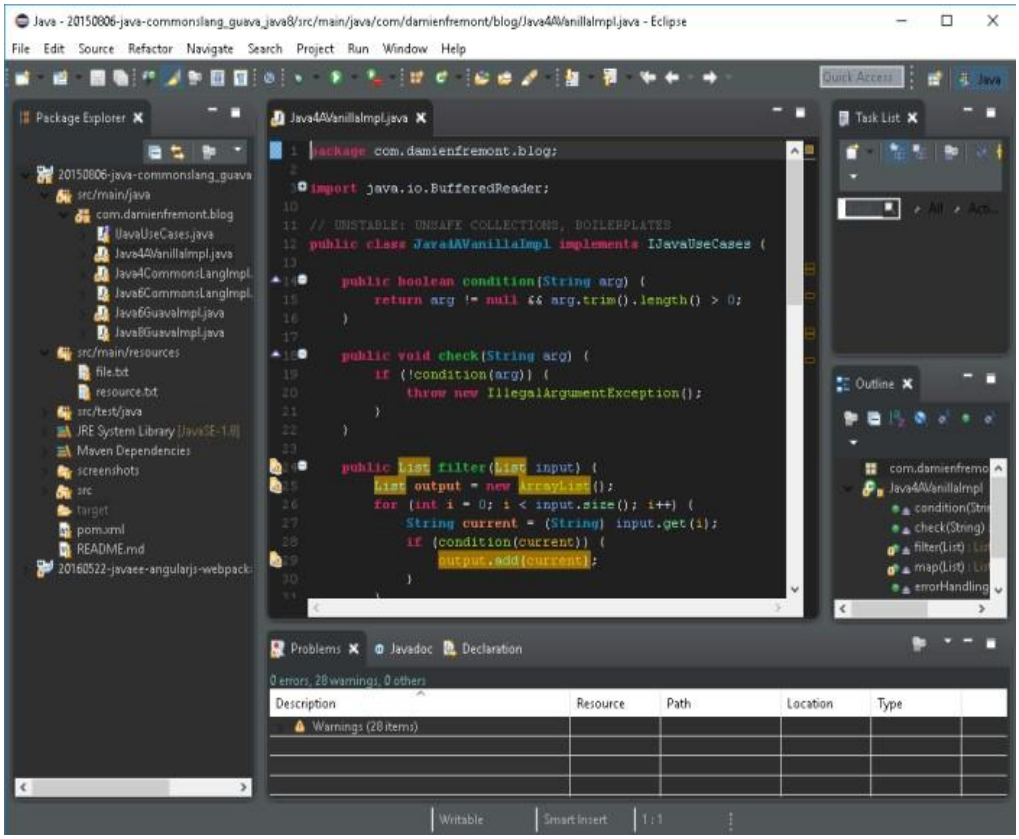

Рис. 2.3. Інтерфейс Eclipse

3) *IntelliJ IDEA* – інтегроване середовище розробки, що найчастіше використовується при роботі зі Spark. Серед його основних переваг слід виділити:

- Можливість працювати над проектами спільно із командою або віддалено;
- Широкий спектр вбудованих інструментів та бібліотек; о Глибокий аналіз коду;
- Надання програмою відповідних пропозицій згідно контексту;
- Відмінна підтримка Java;
- Широкий перелік інструментів для автоматизованого тестування;
- Розширені можливості налагодження;
- Можливість підвищення продуктивності.

Конкретно при роботі зі Spark слід зазначити, що основною причиною, що сприяла широкому використанню саме IntelliJ IDEA для роботи зі фреймворком, є підтримка Scala.

В свою чергу, важливим є зазначити наступні недоліки:

- Значне споживання системних ресурсів;
- Зосередження, в першу чергу, на Java, незважаючи на підтримку і інших мов програмування;
- Робота великих проектів може бути менш продуктивною, враховуючи значне використання ресурсів;
- Вартість.

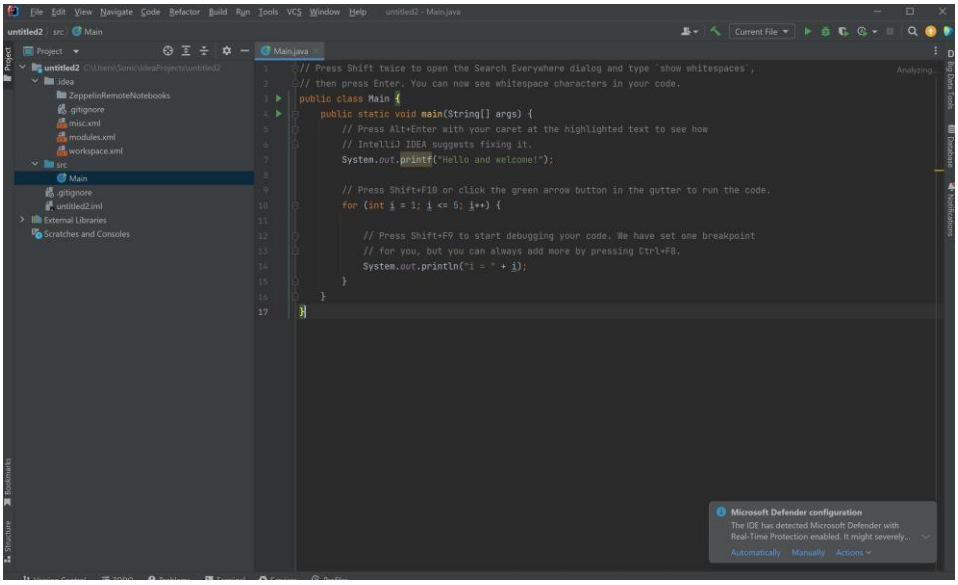

Рис. 2.4. Інтерфейс IntelliJ IDEA

4) *Visual Studio* – дане IDE часто використовується для роботи із мовою програмування C++.

Існують наступні його переваги:

- Універсальність;
- Підтримка різних мов і платформ;
- Підтримка інструментів для роботи в команді;
- Потужні можливості налагодження та тестування;
- Багата екосистема.

А серед недоліків слід виділити:

- Вартість;

- Незважаючи на підтримку кросплатформенної розробки, орієнтований в першу чергу на Windows;
- Ресурсомісткість.

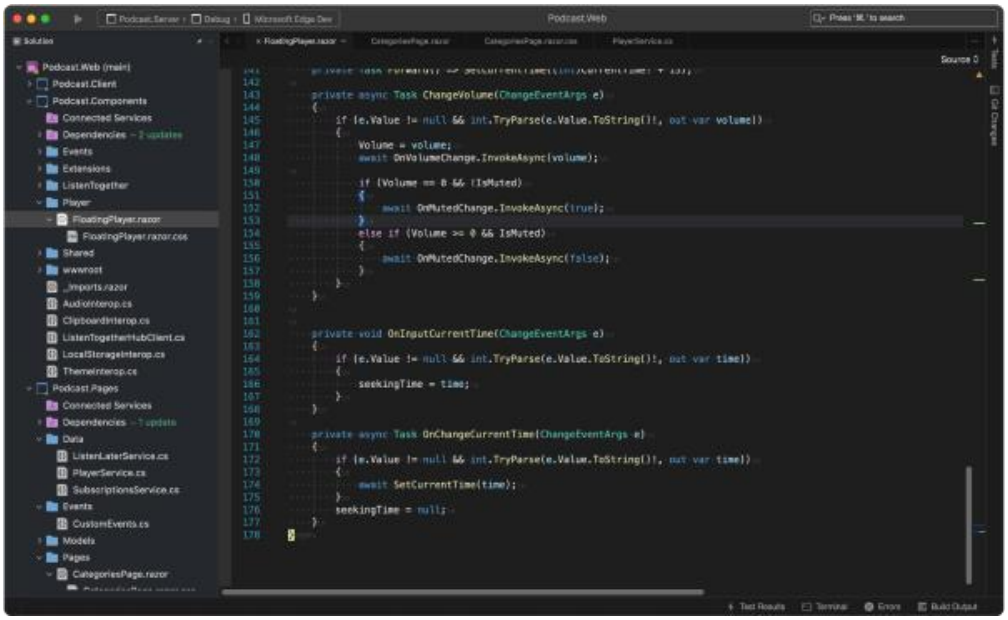

Рис. 2.5. Інтерфейс Visual Studio

Враховуючи усі потреби, для розробки бібліотеки для системного тестування, ми будемо використовувати IntelliJ IDEA. Такий вибір пов’язаний, в першу чергу, із вибором мови програмування, а саме – Java, якою буде здійснюватися розробка бібліотеки для тестування.

IntelliJ IDEA є одним із найзручніших і найпотужніших середовищ розробки при роботі зі Spark і Java, надаючи велику кількість інструментів і додаткових функцій, включаючи інструменти для роботи конкретно зі Spark. Більшість розробників, дослідників даних тощо використовують саме це IDE для роботи зі Spark.

Підтримка Scala у IntelliJ IDEA є також значною перевагою, так як застосунок Spark написаний цією мовою і ми маємо можливість у разі необхідності використовувати функції Scala.

2.3.1. Визначення підсистеми Windows для Linux

Apache Spark надає можливість здійснювати роботу зі Spark API на різних операційних системах, таких як MacOS, Windows, Linux. Однак найчастіше робота із Spark проводиться в операційній системі Linux Ubuntu.

Основною причиною для цього є ряд переваг даної системи, серед яких слід виділити стабільність, безпечність і надійність, завдяки чому є можливість мінімізації збоїв при роботі із застосунком. Окрім цього, Linux Ubuntu відмінно підтримує відкрите ПЗ і надає можливість легкого доступу до широкого переліку пакетів програмного забезпечення.

Також слід виділити у якості переваги інструмент командного рядка, завдяки якому користувач має можливість ефективно налаштовувати і керувати Spark. Екосистема Linux пропонує надійні інтерфейси командного рядка, менеджери пакетів та інструменти розробки, які добре підходять для управління та налаштування середовищ Spark.

Іншими словами, незважаючи на широкий вибір операційних систем, робота зі Spark та його тестування найчастіше ведеться в системах на базі Linux, завдяки чому мінімізуються ризики втрати даних, помилок і збоїв під час роботи.

У процесі розробки бібліотеки робота буде вестися з використанням підсистеми Windows для Linux (WSL). За допомогою цього дистрибутиву, користувач має можливість використовувати застосунки Linux, програми командної строки і службові застосунки, та усі переваги системи без необхідності встановлення середовища на базі Linux. Через нього ми зможемо запускати Spark і використовувати усі переваги системи у процесі роботи.

Крім того, WSL дозволяє легко інтегрувати з іншими інструментами та програмним забезпеченням Windows. Це особливо корисно при роботі з середовищами розробки, IDE, та графічними інтерфейсами користувача (GUI), які є специфічними для Windows. Використовуючи WSL, ми можемо використовувати IDE на базі Windows, редактори та інструменти візуалізації і таким чином, покращувати досвід розробки.

Отже, WSL – це функція операційної системи Windows, завдяки якій користувач має можливість запускати файлову систему Linux і використовувати її інструменти та застосунки, що активно застосовується розробниками, і також набула широкого використання серед спеціалістів, що працюють із Spark на Windows.

2.4. Docker

Docker – це програмна платформа із відкритим кодом, що автоматизує розгортання застосунків у контейнери, розроблена Docker Inc і випущена під ліцензією Apache. Суть ідеї, що лежить в основі Docker, є розгортання застосунків поверх віртуалізованого середовища контейнерів.

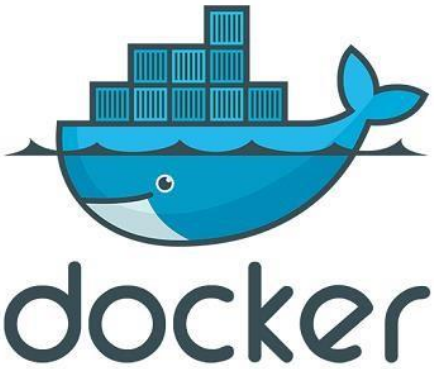

Рис. 2.6. Логотип Docker

Серед особливостей Docker слід визначити:

- Швидкість;
- Легкий і простий спосіб моделювання реальності;
- Ефективний та швидкий життєвий цикл розробки;
- Логічна сегрегація обов’язків;
- Заохочення обслуговування орієнтованої архітектури, що надає можливість масштабування, налагодження, а також самоаналізу.

Docker застосовують для створення сервісних застосунків, створення і тестування складних застосунків і архітектур, побудови багатокористувацької

архітектури, забезпечення автономних середовищ для розробки та тестування, а також навчання технологій (оболонка Unix чи мова програмування) тощо.

2.4.1. Компоненти Docker

Docker включає у собі наступні компоненти (рис. 2.7.):

- *Клієнт і сервер Docker («Docker Engine» - рушій Docker).*

Docker – це клієнт-серверний застосунок. Тобто, архітектура застосунку включає у собі набір серверів, основним завданням яких є надання інформації або певних послуг, коли до них звертаються. Іншою складовою архітектури є клієнти, які використовують сервіси, які надаються сервісами. Правила взаємодії між клієнтом і сервером називаються протоколом обміну, або ж протоколом взаємодії.

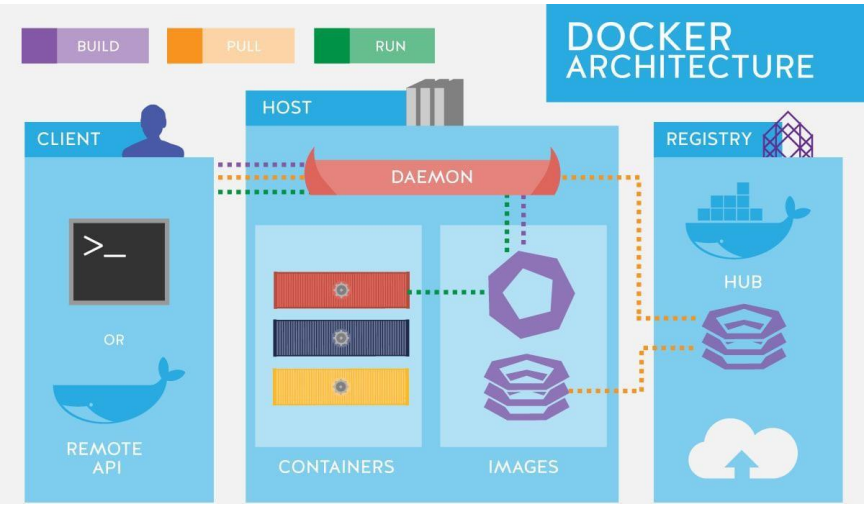

Рис. 2.7. Архітектура Docker

- *Docker-зображення.*

Зображення – це те, з чого запускаються контейнери, що є частиною життєвого циклу Docker. Іншими словами, можна сказати, що зображення – це вихідний код для контейнерів.

- *Реєстри.*

Реєстри – це те, де Docker зберігає зображення. Реєстри можуть бути публічними, або ж приватними. Зображення можна зберігати як приватно, так і публічно в Docker Hub, що є публічним реєстром для зображень від самого Docker Inc. і містить у собі більше 10 тисяч зображень. З іншої ж сторони, можна створити власний приватний реєстр і користуватися ним.

- *Docker-контейнери.*

Docker-контейнер – це, по суті формат зображень, перелік стандартних операцій і середовище виконання, всередині якого зберігаються пакети із застосунками чи сервісами. Ідея роботи Docker-контейнерів подібна до контейнерів на справжньому кораблі – різницею лише є те, що замість доставки продукції, Docker доставляє ПЗ. Для Docker немає ніякого значення склад контейнеру, тобто контейнер може містити базу даних, застосунок, веб-сервер тощо. Переносити контейнери можна на фізичний або віртуальний сервер, на кластер тощо.

Таким чином, Docker є потужною та універсальною програмною платформою, що використовується для різних цілей, наприклад, для створення і тестування складних застосунків та інших завдань. В контексті даної роботи, ми будемо використовувати Docker як віртуальне середовище, де буде запускатися Redis. Це позбавляє нас необхідності налаштування Redis окремо на кожній машині, та спрощує процес роботи із Redis в цілому.

2.5. Redis

Redis (**RE**mote **DI**ctionary **S**erver) – це NoSQL сховище пар ключ-значення структури даних у пам’яті із відкритим кодом, що використовується у якості бази даних, брокеру повідомлень, кешу, потокових рушіїв тощо.

Redis застосовують для кешування даних, обмін повідомленнями, створення таблиць лідерів у режимі реального часу у іграх, сховище сесій, машинного навчання, аналітики у режимі реального часу, і багатьох інших завдань.

Рис. 2.8. Логотип Redis

Головною властивістю Redis є зберігання усіх даних у пам’яті, завдяки чому досягається висока продуктивність.

Серед переваг Redis слід визначити:

- Простота;
- Зручність;
- Масштабованість;
- Асинхронна реплікація і постійне зберігання.

Redis підтримує більшість найпопулярніших мов програмування, включаючи C#, Java, Go, Python і інші.

Зазвичай Redis застосовується на ОС сімейства Unix, наприклад, Linux або macOS, а також як Docker контейнер на Windows.

Redis складається із наступних компонентів:

- Сервер та інтерфейс командної стрічки – Redis запускається як серверне ПЗ. Інтерфейс командного рядку (CLI) надає можливість прямої взаємодії із даними на сервері.
- Клієнт та драйвера – через них здійснюється взаємодія із даними, що знайдені на сервері Redis. Здійснюється це із використанням широкого переліку клієнтських бібліотек, що підтримують різноманітні мови програмування.
- Бази даних – створення бази не вимагає якихось формальних кроків, як і створення таблиці. Дані зберігаються в пам'яті випадкового доступу (RAM) на сервері Redis.

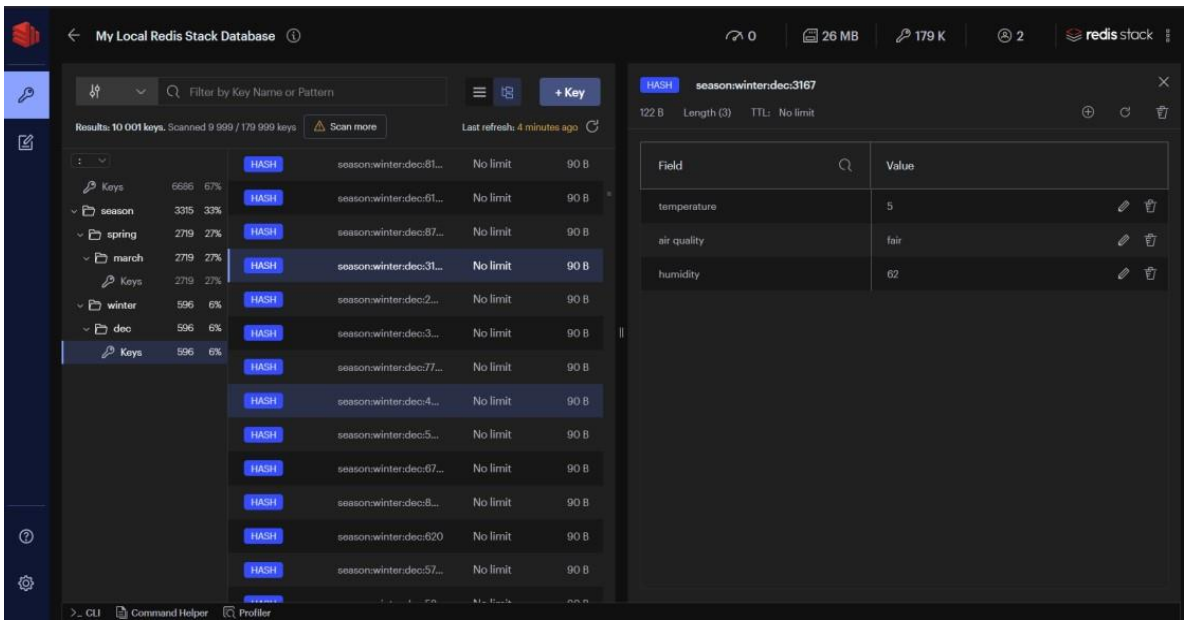

Рис. 2.9. Інтерфейс Redis

Отже, Redis є універсальним і потужним сховищем із відкритим кодом, що підтримує різноманітні функції, бібліотеки і мови програмування. В контексті даної роботи, ми будемо використовувати Redis у якості бази даних для синхронізації даних.

2.6. Висновки до розділу 2

У цьому розділі були розглянуті ключові аспекти, пов'язані з тестуванням, переваги автоматизованого тестування перед ручним, здійснено аналіз недоліків та переваг мов програмування для роботи зі Spark і середовища розробки для створення бібліотеки для системного тестування, а також описано інші програмні засоби, що використовуються в роботі – Docker і noSQL базу даних Redis.

Вивчення основних аспектів тестування було важливим етапом, щоб забезпечити якісне та надійне тестування розробленої бібліотеки. Описуючи основні поняття у сфері тестування, були визначені переваги та недоліки кожного рівня тестування. Також було розглянуто основні відмінності між автоматизованим і ручним тестуванням і обґрунтовано необхідність створення автоматизованих тестів для системного рівня тестування такого багатошарового застосунку як Apache Spark, що включає у собі різноманітні застосунки та бібліотеки. Автоматизоване тестування

сприяє зниження ризику помилок та забезпеченню надійності результатів і є хорошим вибором для тестування Apache Spark.

Вибір мови програмування є важливим аспектом розробки бібліотеки. У цій роботі було обрано мову програмування Java, яка є широко використовуваною, має потужні бібліотеки та фреймворки для розробки, а також є сумісною з Apache Spark і Scala, що є «рідною» мовою фреймворку. Використання Java дозволяє реалізувати потрібну функціональність та забезпечити зручність для користувачів бібліотеки.

Вибір середовища розробки також відіграє важливу роль у продуктивності та зручності розробки бібліотеки. В даній роботі було обрано IntelliJ IDEA як основне середовище розробки. IntelliJ IDEA надає багато корисних функцій, таких як автодоповнення, налагоджувач, система контролю версій, що сприяють швидкій і ефективній розробці.

Також було розглянуто поняття підсистеми Windows для Linux і причини її використання для роботи із фреймворком і запуску тестів, а також сховище Redis, що буде використовуватися у якості середовища спільного доступу для синхронізації складових застосунку і Docker, що в контексті даної роботи використовується для запуску у його контейнері Redis.

РОЗДІЛ 3
РОЗРОБКА БІБЛІОТЕКИ ДЛЯ ТЕСТУВАННЯ

3.1. Структура бібліотеки.

Бібліотека для тестування Spark буде складатися із трьох основних модулів:

1) Core – являє собою ядро проекту. Це, по суті, тестова сторона застосунку, де імплементуються різноманітні функції, основним завданням яких є організація роботи тестів.

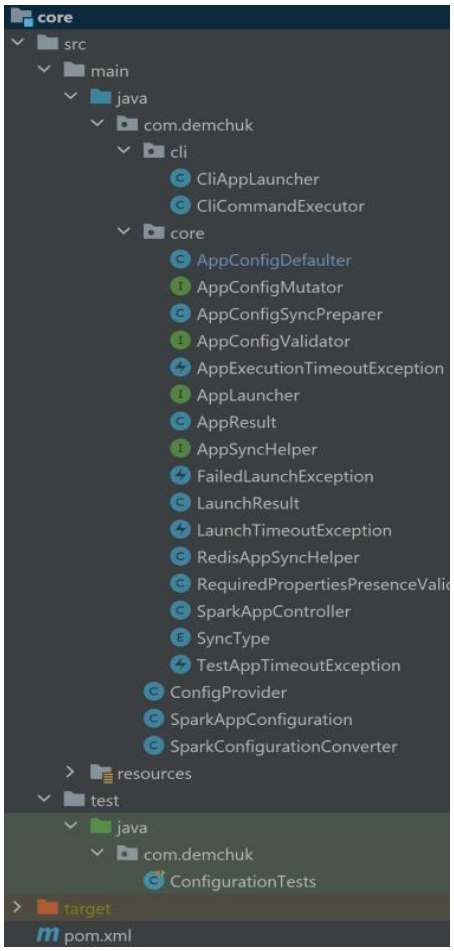

Рис. 3.1. Структура модулю core

2) Client – це сторона самого застосунку Spark. Включає у собі абстрактний клас, що визначає методи та функціональності для написання класів і клас для роботи із середовищем спільного доступу, що використовується для синхронізації.

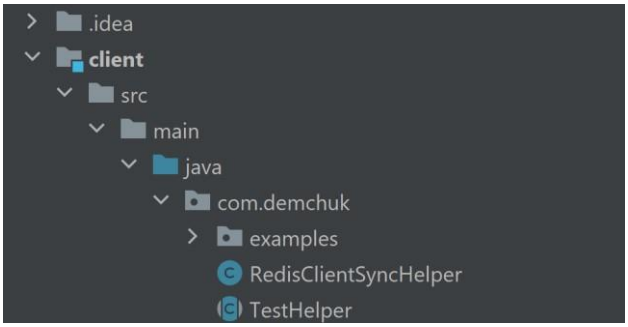

Рис. 3.2. Структура модулю client

3) Demo – модуль для написання тестів, або ж виконання тестових сценаріїв.

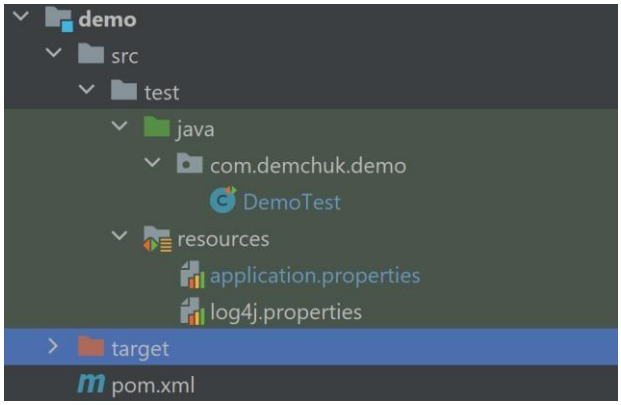

Рис. 3.3. Структура модулю demo

Важливим є зазначити, що по суті, у розробленої бібліотеки є дві сторони: сторона тестових процесів, що включає у собі класи на функціональність, щодопомагають у процесі тестування, і сторона самого застосунку, що відповідає за запуск і виконання застосунку.

3.2. Огляд архітектури бібліотеки

Для глибшого розуміння архітектури застосунку, побудуємо діаграму послідовностей (рис. 3.4), що проілюструє взаємодію тестового середовища(тестового процесу), середовища спільного доступу(в нашому випадку це Redis) і застосунку Spark.

Так як тестовий процес і застосунок Spark – два окремі модулі, постає необхідність синхронізації, тобто організації взаємодії сторін бібліотеки, зберігання станів та результатів і обміну інформацією. У якості середовища спільного доступу може виступати будь-яка база даних.

Існує багато варіантів, що може використовуватися у якості середовища спільного доступу, наприклад, Zookeeper, Kafka, Cassandra та інші середовища.

В даному випадку, у програмі імплементовано лише одне середовище спільного доступу – Redis, проте за бажання користувач може розширити бібліотеку залежно від середовища тестування, вимог та потреб.

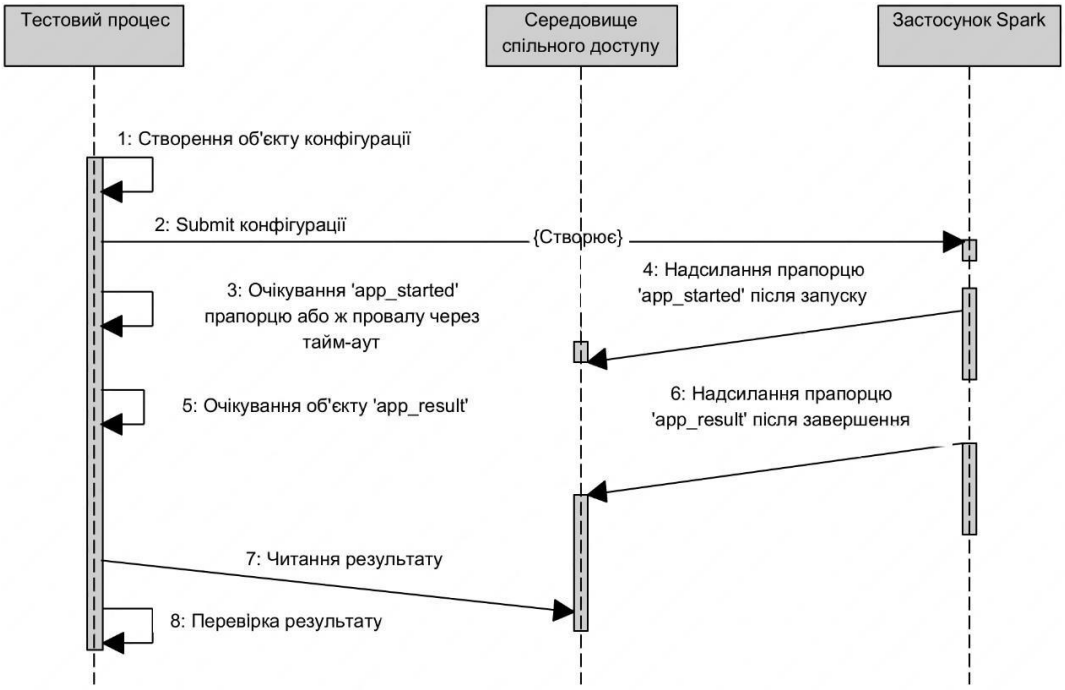

Рис. 3.4. Діаграма послідовностей для бібліотеки для тестування

Як можна побачити, взаємодія між основними компонентами застосунку полягає у наступному:

1. На початку у тестовому процесі створюється об'єкт конфігурації, з якою на другому етапі здійснюється submit.
2. В результаті запиту submit, створюється Spark застосунок.
3. Тим часом тестовий процес починає очікувати прапорцю, що означає, що застосунок було запущено, або провалу через тайм-аут.
4. Коли застосунок запускається, в середовище спільного доступу надсилається прапорець, що застосунок було запущено.
5. Далі тестовий процес починає очікувати завершення застосунку.
6. Застосунок завершується, в середовище спільного доступу надсилається результат.
7. Тестовий процес зчитує результат із середовища спільного доступу і перевіряє його.

Діаграма послідовностей демонструє процес роботи бібліотеки за умови, коли застосунок вже було зібрано, скомпільовано.

Отже, щойно було розглянуто послідовність роботи бібліотеки і зображено принцип взаємодії основних її складових: тестового процесу, середовища спільного доступу і застосунку Spark.

3.3. Клас конфігураційних параметрів

Точкою входу в будь-якому Spark застосунку, як правило, є команда spark-submit. Spark-submit – це скрипт в каталозі Spark bin, який запускає програми у кластері. Іншими словами, це інтерфейс командного рядка, що постачається разом із Spark, і надає користувачеві можливість надсилати та запускати застосунки Spark на кластері.

Основними задачами spark-submit є:

- Упаковка застосунків;
- Запуск застосунків на кластері, або в локальному режимі;

- Управління конфігурацією;
- Управління залежностями;
- Розподіл ресурсів;
- Моніторинг застосунків.

Отже, мінімальний spark-submit запит виглядає наступним чином:

--class <main-class> --master <master-url> deploy-mode <deploy-mode> -conf <key>=<value> <application-jar> [application-args]

Тут використовуються наступні параметри:

- --class: точка входу для програми;
- --master: головна URL адреса для кластера (наприклад:
- k8s://xx.yy.zz.ww:443);
- --deploy-mode: режим запуску, що визначає, де буде запускатися застосунок (наприклад, на кластері – тоді значення буде cluster, або локально – client). По замовчанню використовується клієнтський режим;
- --conf: властивості конфігурації;
- Application-jar: шлях до пакету jar застосунку, що запускається
- Application-args: аргументи, що передаються в основний метод основного класу.

Також слід визначити параметри, що часто використовуються у запитах (таблиця 2.1).

Окрім цих команд, існує широкий перелік інших команд, що є специфічним для певного режиму роботи, Spark-майстра тощо.

Для спрощення роботи із тестами, ми розробимо класи, що нададуть можливість програмно налаштовувати spark-submit команду. Простіше кажучи, замість того, щоб при кожному тесті писати вручну spark-submit, ми автоматизуємо його задання.

Таблиця 2.1

Параметри spark-submit

Команда	Визначення
--name	Вказує назву застосунку Spark.
Команди, що задають додаткові файли	
--jars	Вказує список JAR-файлів, які мають бути включені в клас шляху додатку.
--packages	Вказує список пакетів Maven, які мають бути автоматично завантажені для додатку.
--py-files	Вказує список Python-файлів, які мають бути включені в клас шляху додатку.
--files	Вказує список файлів, які мають бути включені як ресурси і доступні для додатку.
Команди, що задають параметри драйвера та виконавця	
--driver-memory	Налаштування пам’яті драйвера
--driver-javaoptions	Налаштування додаткових параметрів Java
--driver-library-path	Налаштування додаткового шляху до бібліотеки для переходу до драйвера
--driver-class-path	Налаштування додаткового класу для переходу до драйвера
--executor-memory	Налаштування пам’яті виконавця

Для цього створимо клас SparkAppConfiguration, що буде зберігати в собі інформацію, що задаватиметься в тестах.

Здійснюємо ми це з використанням бібліотеки Lombok, що спрощує створення коду, генеруючи його. Таким чином, ми зменшуємо об'єм коду, робимо його чистим і зручним, а також позбавляємося необхідності витрачення часу на генерування стандартних методів.

Для того, щоб використати її, необхідно застосувати анотацію @Data над класом.

Параметри, що мають одне значення, наприклад, «--master yarn», визначаються як рядки.

```
// -- master
private String master;

// --deploy-mode
private String deployMode;
```

Параметри, які можуть мати перелік значень, такі як списки пакетів, файлів, jar-файлів, аргументи та інші, визначаються як списки.

```
@Builder.Default
private List<String> jars = new ArrayList<>();

// --packages
@Builder.Default
private List<String> packages = new ArrayList<>();
```

Параметр conf, що у запиті записується у форматі «--conf <key>=<value>», визначаємо як мапу.

```
// --conf
@Builder.Default
private Map<String, String> conf = new HashMap<>();
```

Параметри, в які у запиті передаються числа, такі як кількість ядер, кількість виконавців тощо, визначаються як змінні типу Integer.

```
// --executor-cores     private Integer executorCores;
```

Параметри, в які не передаються ніякі значення, що використовуються для виклику певних дій чи функцій, наприклад, команда –help, визначаються як Boolean.

```
// --help
private boolean help;
```

Також визначаємо змінні startTimeout і executionTimeout – це час, що буде задаватися для тестів.

```
// test timeouts
private Long startTimeout;
private Long executionTimeout;
```

У цьому класі окрім анотації @Data, ми також використовуємо наступні анотації:

- *@NoArgsConstructor* - генерує конструктор без аргументів (пустий конструктор) для класу. Це дає змогу ініціалізувати об'єкт класу без передачі аргументів у конструктор.

- *@AllArgsConstructor* - генерує конструктор, який приймає всі поля класу як аргументи. Це дозволяє створювати об'єкти класу з встановленням значень для всіх полів одразу.
- *@Builder(toBuilder = true)* - генерує паттерн "Builder" для класу, що дозволяє зручно створювати об'єкти з багатьма параметрами.

В останню чергу в цьому класі ми реалізуємо метод submit(), що викликається для надсилання submit запиту на виконання. Створюючи екземпляр SparkAppController, ми ініціалізуємо його з посиланням на поточний об’єкт, передаючи як аргумент методу create(). Таким чином SparkAppController, діяльність якого буде детальніше описуватися пізніше, отримує доступ до даних і налаштувань.

Далі викликається метод submit(), який відправляє застосунок на виконання. Метод повертає результат виконання запиту.

```
public AppResult submit() {
      SparkAppController controller = SparkAppController.create(this);
 return controller.submit();
   }
```

Усі параметри, значення яких будуть передаватися у тестах, треба конвертувати у стрічку. Для цього нам потрібно описати це перетворення заданих об’єктів у стрічку та їх об’єднання, що буде виконуватися у класі SparkAppConverter.

В цьому класі ми використовуємо анотацію @Slf4j (простий фасад логування для Java) з бібліотеки Lombok для генерації коду логування – вивід певної інформації, що спрощує пошук точки, в якій виник дефект.

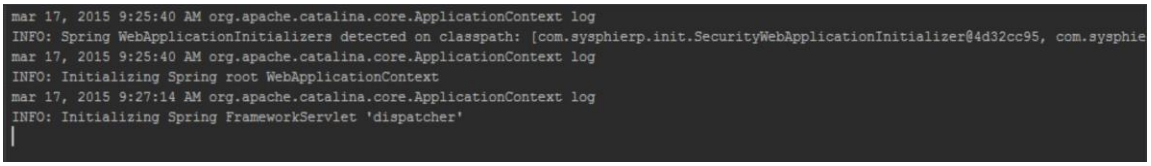

Рис. 3.5. Приклад логування

Логування відіграє важливу роль у процесі тестування, надаючи можливість покращення його ефективності. Таким чином, ми можемо відстежувати дані про стан програми, значення змінних тощо.

log.debug(result);

Тут ми здійснюємо логування значення змінної result на рівні DEBUG. Коли цей рядок коду виконується, значення змінної result логується в на рівні DEBUG. Це дозволяє відстежувати і контролювати значення командного рядка spark-submit, згенерованого з об'єкта SparkAppConfiguration.

Ідея роботи класу SparkAppConverter полягає у наступному: в основному методі класу convert, об’єкт SparkAppConfiguration конвертується у рядок, що містить у собі параметри та їх значення для команди spark-submit. Конвертація рядку здійснюється із використанням об’єкту StringBuilder, завдяки якому ми будуємо рядок запиту, перебираючи всі властивості SparkAppConfiguration. Отже, у разі наявності заданого параметру –name до рядку додається «--name» із відповідним значенням.

Деякі параметри мають спеціальні умови для додавання. Наприклад, параметр --deploy-mode додається тільки якщо має значення і не є порожнім рядком.

```
public String convert(SparkAppConfiguration configuration) {
    StringBuilder builder = new StringBuilder();
    // --class
    String mainClassString = configuration.getMainClass();
        builder.append(" --class " + mainClassString);
    // --master
    String masterString = configuration.getMaster();
         builder.append(" --master " + masterString);
    // --deploy-mode
    String deployModeString = configuration.getDeployMode();
  if (deployModeString != null && !deployModeString.isEmpty()) {
```

```
            builder.append(" --deploy-mode " + deployModeString);
        }
        // --supervise
      if (configuration.isSupervise()) {
        builder.append(" --supervise");
        }
        // --name
        String name = configuration.getName();
      if (name != null && !name.isEmpty()) {
         builder.append(" --name \"" + name + "\"");
        }
```

Параметр --supervise та інші параметри, тип яких Boolean, додається, якщо властивість supervise має значення true.

```
if (configuration.isHelp()) {
  builder.append("--help ");
}
```

Деякі параметри, тип яких мапа(Map<String,String>), таких як --conf, -packages і --exclude-packages, спочатку будуть перетворюватися за допомогою операції потоку, представлятися у форматі «ключ=значення» і далі додаватися до builder за допомогою методу forEach.

```
// --conf, --properties
Map<String, String> conf = configuration.getConf();
 if (!conf.isEmpty()) {
  conf.entrySet()
        .stream()
        .map(entry -> " --conf " + entry.getKey() + "=" + entry.getValue())
```

.forEach(builder::append);

}

Для параметрів, що є списками, ми використовуємо метод String.join для об'єднання значень у рядок з відповідним роздільником.

List<String> args = configuration.getArgs(); String argsString = String.join(" ", args); builder.append(" " + argsString);

Таким чином, ми розробили класи SparkAppConfiguration і SparkConfigurationConverterer, основна мета яких полягає у автоматизації формування spark-submit запитів. У результаті ми позбавляємося необхідності щоразу вводити вручну параметри при запуску застосунків.

3.4. Завантаження конфігураційних параметрів

Наступним кроком ми створимо клас ConfigProvider, основною задачею якого буде завантаження конфігураційних параметрів із файлу application.properties (рис. 3.6).

```
spark.master = yarn
spark.home = /home/user/spark

test.launcher.type = cli

test.sync.type = redis
test.sync.redis.host = localhost
test.sync.redis.port = 6379

test.timeouts.start = 30000
test.timeouts.execution = 30000
```

Рис. 3.6. Зовнішній вигляд файлу application.properties

Файл application.properties в даному випадку використовується для того, аби зробити нашу конфігурацію програми гнучкішою і позбавитися потреби внесення змін у сам код програми. Іншими словами, для того, щоб не хардкодити значення ключових параметрів у самому коді, ми зазначимо їх у application.properties.

Отже, у application.properties ми визначаємо наступні параметри:

1) *Spark майстер.* По замовчанню ми будемо використовувати yarn.

2) *Spark home.* Визначає шлях до директорії, у якій встановлений Spark, на машині, де запускають тести.

3) *Тип лаунчера для тестування.* Те, через що буде запускатися програма. В даному випадку в нас є лише один тип запуску – через командну стрічку, проте є можливість розширення проекту і додання інших типів.

4) *Тип синхронізатора.* Середовище спільного доступу, що буде використовуватися.

5) *Конфігурація синхронізатора.* Місцезнаходження синхронізатора.

6) *Тайм-аути.* Це час запуску і виконання, що використовується у тестах.

Ми використовуємо наступні публічні методи для отримання значень цих параметрів:

1) sparkMaster();

2) sparkHome();

3) launcherType();

4) syncType();

5) redisHost();

6) redisPort();

7) defaultAppStartTimeoutMs();

8) defaultAppExecutionTimeout().

Клас реалізований за допомогою паттерну Singleton - щоб гарантувати, що клас має лише один екземпляр. Вже існуючий екземпляр повертає метод get(), а якщо його немає, то він створюється.

Отже, ми створюємо приватний статичний екземпляр класу:

```
private static ConfigProvider instance;
```

Наступним кроком описуємо метод доступу:

```
public static ConfigProvider get() {
  if (instance == null) {
 instance = new ConfigProvider();
    }
   return instance;
}
```

Тут ми використовуємо так звану «ліниву ініціалізацію». Спочатку перевіряється, чи існує вже екземпляр класу ConfigProvider (instance). Якщо він є null, це означає, що екземпляр ще не був створений, і тоді виконується наступний код.

Для отримання значень параметрів з файлу application properties для початку ми створюємо об’єкт Properties. Далі відкривається поток вводу, і завантажуємо властивості в цей потік за допомогою методу load() у форматі ключ-значення, а також логуємо їх на рівні DEBUG.

```
@SneakyThrows
private ConfigProvider() {
this.properties = new Properties();
InputStream stream=
ConfigProvider.class.getClassLoader().getResourceAsStream("application.properties");
  properties.load(stream);
  properties.entrySet().stream().forEach(keyValue -> log.debug(keyValue.getKey() + "=" + keyValue.getValue()));
```

```
}
```

Як і в минулих класах, тут ми користуємося бібліотекою Lombok, застосовуючи анотації @Slf4j і @SneakyThrows з нею. @Slf4j як вже зазначалося раніше, використовується для логування. В свою чергу, @SneakyThrows – це анотація для обробки винятків. Ця анотація додає блок «try-catch», позбавляючи нас необхідності написання додаткового коду.

Також описуємо методи доступу до конфігураційних параметрів.

```
public String sparkMaster() {
    return properties.getProperty("spark.master");
  }

  public SyncType syncType() {
     String type = properties.getProperty("test.sync.type", "redis");
  switch (type) {
   case "redis":
          return SyncType.REDIS;
default:
          throw new RuntimeException("Unsupported sync type: " + type);
     }
  }
```

Отже, щойно було розроблено клас ConfigProvider, завдяки якому ми отримуємо доступ до конфігураційних параметрів, які були описані у файлі application.properties. Клас було реалізовано за допомогою Singleton-паттерна, що запобігає переініціалізації і повторному виконанню коду.

3.5. Основний функціонал бібліотеки

Усі розроблені класи, які входять до складу бібліотеки, об'єднуються навколо головного компонента - SparkAppController. Цей клас виступає у якості ключового, центрального пристрою управління та контролю виконання застосунків Spark. Він має важливу роль у забезпеченні безперебійної та ефективної роботи додатків, надаючи розробникам потужні інструменти та функціонал.

Таким чином, SparkAppController використовується як центральний механізм для зв'язування всіх розроблених класів і компонентів. Він формує основу для взаємодії між цими класами, об’єднуючи їх функціональність і забезпечуючи їх взаємодію в правильному порядку.

Серед завдань, що стоять перед SparkAppController, слід визначити:

- Ініціалізація конфігурації;
- Встановлення налаштувань за замовчуванням;
- Зміна конфігурації;
- Валідація конфігурації;
- Запуск застосунку;
- Синхронізація та взаємодія із зовнішніми ресурсами;
- Контроль тайм-аутів;
- Очищення ресурсів.

Таким чином, описавши роль класу SparkAppController, необхідним є детально розглянути його структуру.

3.5.1. Структура SparkAppController

Для початку, ми ініціалізуємо усі змінні класу, що будуть використовуватися надалі, такі як configProvider, configMutators, configValidators та інші.

private static ConfigProvider configProvider = ConfigProvider.get();

private SparkAppConfiguration configuration; private String identifier;

private List<AppConfigMutator> configMutators = new ArrayList<>();
private List<AppConfigValidator> configValidators = new ArrayList<>();
private AppLauncher launcher;
private AppSyncHelper syncHelper;

private Duration appStartTimeout;
private Duration appExecutionTimeout;

Головною точкою входу є spark-submit. Метод submit() надсилає застосунок на виконання. Створюємо екземпляр класу SparkAppController і налаштовуємо початкову конфігурацію. Присвоюємо їй унікальний ідентифікатор, а також присвоюємо полю configuration об'єкт конфігурації з методу toBuilder().

public static SparkAppController create(SparkAppConfiguration userConfig) {
SparkAppController controller = new SparkAppController();
String identifier = UUID.randomUUID().toString();
// copy user config
controller.configuration = userConfig.toBuilder().build(); controller.identifier = identifier;

Після цього конфігурація має пройти певні мутації. Для цього ми використовуємо мутатори, що будуть отримувати конфігурацію та повертати змінену конфігурацію.

Всього тут ми використовуємо два мутатори:

- AppConfigDefaulter – використовується для того, щоб задавати значення по замовчанню у випадку, якщо, наприклад, Spark-майстер не був заданий користувачем. У такому випадку конфігурацію доповнюється.
- AppConfigSyncPreparer – використовуємо для задання властивостей для внутрішніх потреб. Цей мутатор допомагає підготувати конфігурацію для роботи із синхронізатором.

Наступним етапом є валідація. Валідація є необхідною для виявлення некоректної конфігурації на ранніх етапах, що покращує надійність і якість роботи. Для валідації ми використовуємо клас RequiredPropertiesPresenceValidator, що імплементує інтерфейс AppConfigValidator. Таким чином, ми маємо можливість перевірити різні аспекти конфігурації, і, найважливіше – наявність обов'язкових spark-submit параметрів.

Після валідації створюється лаунчер, що запускає застосунок. В лаунчер передається тип лаунчера.

В нашому випадку, у нас наявний лише один тип лаунчера – командна стрічка. Проте, в залежності від специфічних потреб, є потенціал розширення бібліотеки шляхом додавання інших лаунчерів, наприклад, для K8s.

Здійснюємо ініціалізацію SyncHelper, основним завданням якого є допомога в синхронізації і отриманні результатів. В SyncHelper задається інформація про Redis сервер.

```
controller.launcher = AppLauncher.create();

// init sync helper
RedisAppSyncHelper redisSyncHelper = new RedisAppSyncHelper(identifier);
controller.syncHelper = redisSyncHelper;
controller.configMutators.add(redisSyncHelper);
```

Наступним кроком ініціалізуємо тайм-аути для запуску застосунку та його виконання. Іншими словами, тайм-аути визначають максимальний час очікування запуску і виконання застосунку.

```
if (userConfig.getStartTimeout() != null) {
    controller.appStartTimeout = Duration.of(userConfig.getStartTimeout(),
ChronoUnit.MILLIS);
  } else {
    controller.appStartTimeout =
Duration.of(configProvider.defaultAppStartTimeoutMs(), ChronoUnit.MILLIS);
  }

  if (userConfig.getExecutionTimeout() != null) {
controller.appExecutionTimeout =
Duration.of(userConfig.getExecutionTimeout(), ChronoUnit.MILLIS);
  } else {
    controller.appExecutionTimeout =
Duration.of(configProvider.defaultAppExecutionTimeout(), ChronoUnit.MILLIS);
  }

  log.info("Created controller for app " + userConfig.getName() + " with id " +
controller.identifier);   return controller;
}
```

Залежно від того, чи вказаний тайм-аут користувачем, значення, що отримують обидва тайм-аути, можуть бути або по замовчанню з ConfigProvider, або відповідно встановлені користувачем. В результаті повертається створенний об’єкт controller, а також логується інформація про його створення.

Метод waitFor, код якого наведений нижче, відповідає за очікування запуску застосунку. Виконання методу закінчується, як тільки функція повертає значення true.

На початку методу обчислюється deadline(крайній термін), що визначає, протягом якого обсягу часу буде виконуватися метод, намагаючись викликати дію action.

```
@SneakyThrows
 private void waitFor(Callable<Boolean> action, Duration timeout) throws
TestAppTimeoutException {
    long deadline = System.currentTimeMillis() + timeout.toMillis();
Exception exception = null;
  while (System.currentTimeMillis() < deadline) {
      try {
        if (Boolean.TRUE.equals(action.call())) {
   return;
        } else {
          Thread.sleep(500L);
        }
      } catch (InterruptedException e) {
   throw e;
      } catch (Exception e) {
  log.trace("Exception while waiting: " + e);
   exception = e;
      }
    }

    if (exception != null) {
  throw new TestAppTimeoutException(exception);
    } else {
      throw new TestAppTimeoutException();
    }
  }
}
```

У випадку, якщо результат виклику action.call() дорівнює true, це означає, що очікування було успішним, і метод waitFor завершується. Якщо крайній термін наступає, то в такому випадку логується виключення. Якщо ж немає збереженого виключення, створюється порожній об'єкт TestAppTimeoutException і викидається.

За допомогою SyncHelper ми перевіряємо, що застосунок було успішно запущено. У випадку, якщо ми отримали виключення – тест завершується, помилка логується, виконуємо cleanup(). В результаті виводиться виключення «тайм-ауту запуску», що означає, що застосунок не запустився.

```
try {
log.info("Application " + identifier + " launched, waiting for start");
  waitFor(() -> syncHelper.isRunning(), appStartTimeout);
} catch (TestAppTimeoutException e) {
  log.error("App " + identifier + " failed to start during " +
appStartTimeout.getSeconds() + " seconds");
 cleanup();
 throw new LaunchTimeoutException(e);
}
```

У випадку, якщо запуск було здійснено успішно, по аналогії із тайм-аутом запуску чекаємо, поки застосунок не завершиться. Якщо застосунок не завершується в крайній термін, то виводиться виключення, помилка логується.

```
try {
log.info("Application " + identifier + " started, waiting for completion");
waitFor(() -> syncHelper.isResultAvailable(), appExecutionTimeout);
}
 catch (TestAppTimeoutException e) {
 log.error("App " + identifier + " failed to complete during " +
appExecutionTimeout.getSeconds() + " seconds");
  cleanup();
```

throw new AppExecutionTimeoutException(e);

}

У разі успішного старту застосунку, його виконання і завершення, ми отримуємо результат, що зчитується і повертається.

String result = syncHelper.getResultString();

// cleanup cleanup();

Отже, таким чином щойно було розглянуто структуру класу, що лежить у основі усієї бібліотеки. SparkAppController об'єднує у собі усі розроблені бібліотеки, використовуючи їх функції і можливості, контролює процес запуску і виконання застосунку.

3.6. Огляд допоміжних класів бібліотеки

Значну роль у діяльності основного класу відіграють допоміжні класи, завдяки яким SparkAppController має можливість запуску і контролю роботи застосунку, що тестується.

Ці класи відповідають за різні аспекти функціонування бібліотеки і надають додаткові функції і можливості для керування і синхронізації застосунків.

Серед таких допоміжних класів слід визначити:

- RedisAppSyncHelper;
- AppConfigDefauler;
- CliAppLauncher;
- CliAppExecutor;
- RedisClientSyncHelper в модулі client;
- TestHelper в модулі client.

Ці класи відповідають за різні задачі, що спрощують роботу із бібліотекою у процесі тестування, тож необхідним є розглянути кожен детальніше.

3.6.1. Використання Redis для синхронізації

Для зберігання даних про стан ми використовуємо Redis, що запускається у контейнері Docker. Клас RedisAppSyncHelper визначає роль Redis при роботі із застосунком і функції, що він виконує і є реалізацією інтерфейсів AppSyncHelper і AppConfigMutator.

```
public interface AppSyncHelper {
   void init();
 void cleanup();
boolean isRunning();
boolean isResultAvailable();
 String getResultString();
}
```

В цьому інтерфейсі описуються методи ініціалізації, очищення, перевірки роботи застосунку і наявності результату виконання застосунку.

```
public interface AppConfigMutator {
   SparkAppConfiguration mutate(SparkAppConfiguration original);
}
```

В цьому інтерфейсі описується метод зміни(мутації) конфігурації.

Серед основних функцій класу RedisAppSyncHelper слід визначити:

- Ініціалізація;
- Очищення;
- Перевірка стану;
- Перевірка результату;
- Отримання результату;

- Зміна конфігурації Spark.

Розглянемо структуру даного класу детальніше. Перш за все, важливим є зазначити, що тут, як і у багатьох інших класах, ми використовуємо анотацію Slf4j.

private static final String SPARK_CONF_REDIS_HOST_OPTION =

"spark.test.sync.redis.host"; private static final String SPARK_CONF_REDIS_PORT_OPTION =

"spark.test.sync.redis.port"; private static final String APP_IS_RUNNING_KEY = "running"; private static final String APP_RESULT_KEY = "result";

private final JedisPool jedisPool;

В першу чергу у класі визначаються наступні статичні поля:

- *SPARK_CONF_REDIS_HOST_OPTION.* Використовується для встановлення хосту Redis – машини, на якій запускається Redis.
- *SPARK_CONF_REDIS_PORT_OPTION.* Використовується для встановлення порту Redis, що відповідає за оброблення запитів клієнту.
- *APP_IS_RUNNING_KEY.* Використовується для визначення стану застосунку і перевірки того, чи він запустився.
- *APP_RESULT_KEY.* Використовується для зберігання результатів застосунку в Redis.
- *configProvider.* Об’єкт ConfigProvider, через який ми отримуємо конфігураційні параметри.
- *jedisPooled.* Об’єкт JedisPool, що представляє пул підключень.

Ініціалізуємо об’єкт JedisPool за допомогою методу initJedisPool(). Таким чином ми отримуємо пул підключень, який надалі буде використовуватися для отримання з’єднань із Redis сервером.

private static JedisPool initJedisPool() {

```
  String redisHost = configProvider.redisHost();
    Integer redisPort = configProvider.redisPort();

    return new JedisPool(redisHost, redisPort);
}
```

Далі створюємо конструктор RedisAppSyncHelper, що приймає ідентифікатор конкретного застосунку, ініціалізує це поле і отримує з'єднання із Redis з пулу JedisPool за допомогою методу getResource().

```
public RedisAppSyncHelper(final String appId) {
this.appId = appId;
 this.redisClient = jedisPooled.getResource();
}
```

Далі ми створюємо методи роботи із Redis, такі як метод init(), cleanup(), isRunning(), isResultAvailable(), getResultString(), що здійснюють відповідні операції.

```
@Override

public void init() {
    Map<String,String> data = new HashMap<>();
   data.put("initialized", "true");
  redisClient.hset(appId, data);
  log.debug("Initialized redis app sync helper for appId: " + appId); }

@Override
public void cleanup() {
   redisClient.del(appId);
 redisClient.close();
 log.debug("Cleaned redis items for appId: " + appId);
```

```
    }

    @Override
    public boolean isRunning() {
        Map<String,String> data = redisClient.hgetAll(appId);     return
Boolean.parseBoolean(data.getOrDefault(APP_IS_RUNNING_KEY,
    "false"));
    }

    @Override
    public boolean isResultAvailable() {
        Map<String,String> data = redisClient.hgetAll(appId);     return
data.containsKey(APP_RESULT_KEY);
    }

    @Override
    public String getResultString() {
        Map<String,String> data = redisClient.hgetAll(appId);     return
data.get(APP_RESULT_KEY);
    }
```

Як бачимо, метод init() встановлює ключ "initialized" зі значенням "true" для ідентифікатора застосунку appId в Redis. Те, як виглядають передані дані, проілюстровано на рис. 3.7.

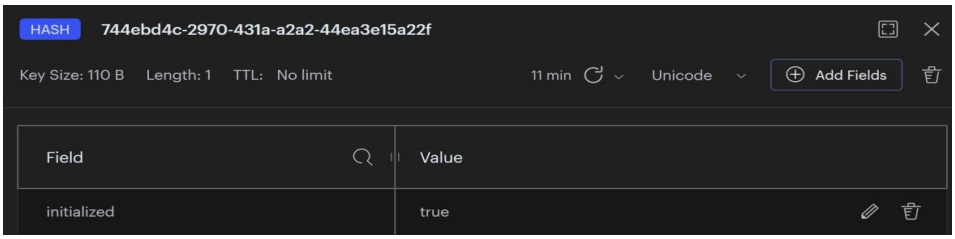

Рис. 3.7. Ключ initialized, встановлений методом init()

Також в цьому класі ми створюємо метод-мутатор, що відповідає за перетворення оригінальної конфігурації у нову конфігурацію із додаванням опцій Redis хоста і порту. Принцип роботи цього методу полягає у отриманні значення хоста і порту із configProvider, додання їх до конфігурації newConfig, і власне повернення нової конфігурації.

```
@Override
public SparkAppConfiguration mutate(SparkAppConfiguration original) {
 Map<String,String> newConfig = new HashMap<>(original.getConf());
newConfig.put(SPARK_CONF_REDIS_HOST_OPTION,
configProvider.redisHost());
 newConfig.put(SPARK_CONF_REDIS_PORT_OPTION,
String.valueOf(configProvider.redisPort()));
  return original.toBuilder()
        .conf(newConfig)
        .build();
}
```

На стороні самого застосунку також присутній RedisSyncHelper, або RedisClientSyncHelper. Ідея цього класу аналогічна RedisSyncHelper, проте присутні певні відмінності.

3.6.2. Допоміжний клас для написання тестів

Клас, що відповідає за запис інформації про запуск застосунку і запуск результату тестів є TestHelper. Цей клас використовуватиметься для написання тестів. Об’єкт цього класу має створюватися у тесті, куди буде передаватися конфігурація. Розглянемо структуру цього класу детальніше.

TestHelper – це абстрактний клас, що означає, що цей клас по суті визначає загальну поведінку допоміжних класів для тестування.

В першу чергу у цьому класі створюються статичні поля, що визначають параметри конфігурації Spark, а саме - SPARK_CONF_IDENTITY_OPTION і SPARK_CONF_SYNC_TYPE_OPTION.

Далі визначаються поля класу, такі як SparkConf sparkConf і String appId, що зберігають посилання на об'єкт SparkConf, який містить конфігурацію Spark,

та ідентифікатор додатку (appId), який отримується з конфігурації Spark.

Створюємо статичний метод create(), основним завданням якого є створення відповідного екземпляра підкласу TestHelper в залежності від типу синхронізації, вказаного у переданій конфігурації. Як вже зазначалося раніше, тут ми використовуємо лише тип синхронізатора Redis, проте є можливість розширення бібліотеки і додавання інших синхронізаторів.

У випадку, якщо маємо тип синхронізатора Redis, створюється і повертається новий екземпляр RedisClientSyncHelper. Так як ми не маємо іншого типу синхронізатора, то в усіх інших випадках виводиться виключення з повідомленням про непідтримуваний тип синхронізатора.

```
public static TestHelper create(final SparkConf sparkConf) {
   String syncType = sparkConf.get(SPARK_CONF_SYNC_TYPE_OPTION);
switch (syncType) {      case "redis":
         return new RedisClientSyncHelper(sparkConf);      default:
         throw new RuntimeException("Unsupported sync type: " + syncType);
   }
}
```

Отже, маємо метод що дозволяє динамічно створювати екземпляр TestHelper, завдяки чому маємо можливість розширення функціональності тестів.

Далі створюється конструктор TestHelper(, що приймає об’єкт SparkConf у якості параметру. Ідея цього конструктору полягає у наступному: полю sparkCong присвоюється параметр sparkConf, що дозволяє зберігати конфігурцію для

подальшого використання, а також отримання значення параметра із ключем SPARK_CONF_IDENTITY_OPTION і присвоєння цього значення полю appId.

Таким чином ми отримуємо унікальний ідентифікатор для поточної програми.

Також визначаємо методи notifyAppStarted() і writeResult(final String result) – абстрактні методи, тобто реалізація цих методів залежить від конкретного підкласу TestHelper, який використовується. Перший метод, notifyAppStarted(), відповідає за повідомлення про старт програми. Другий же метод, writeResult(final String result), відповідає за запис результату виконання застосунку.

3.6.3. Реалізація класу для здійснення мутації

Одним із перших етапів, що виконує SparkAppController є здійснення мутації. Всього використовується два мутатори, один із яких повертає конфігурацію із параметрами за замовчанням, у випадку, якщо вони не були вказані у початковій конфігурації. За здійснення мутації відповідає клас

AppConfigDefaulter, що являє собою імплементацію інтерфейсу AppConfigMutator. Розглянемо структуру цього класу детальніше.

В першу чергу, у класі створюється приватне поле provider, що використовуватиметься для отримання конфігураційних значень.

private ConfigProvider provider = ConfigProvider.get();

Наступним кроком, створюється метод, що визначений інтерфейсом AppConfigMutator і відповідає за мутацію конфігурації. Іншими словами, метод отримує початку конфігурація як параметр і повертає змінену.

```
@Override
 public SparkAppConfiguration mutate(SparkAppConfiguration original) {
SparkAppConfiguration.SparkAppConfigurationBuilder builder =
original.toBuilder();
```

```
    if (original.getMaster() == null) {        builder.master(provider.sparkMaster());
    }
    return builder.build();
}
```

В методі створюється новий об’єкт, що використовуватиметься для зміни конфігурації. Далі за допомогою оператора if, ми перевіряємо, чи присутнє значення у параметра master. Якщо значення null, то в такому випадку присвоюється provider для отримання значення sparkMaster(). Змінна builder встановлює нове значення master за допомогою методу master(). У результаті повертається змінена конфігурація.

В даному випадку у якості параметра, який задається по замовчанню, виступає лише Spark майстер. Проте у разі необхідності, цей клас можна доповнювати, адже завдяки ньому ми позбавляємося необхідності вводити кожний параметр вручну, що є особливо незручним у процесі тестування.

Отже, клас AppConfigDefaulter дозволяє легко налаштувати значення за замовчуванням для певних полів конфігурації SparkAppConfiguration і використовувати їх у випадку, якщо вони не були визначені в початковій конфігурації, завдяки чому робота з конфігурацією стає простішою.

3.6.4. Реалізація класів для взаємодії із CLI

Існують різні варіанти запуску застосунків Spark. Одним із найрозповсюдженіших серед них є запуск через командну стрічку. Саме тому було розроблено класи для роботи із командною стрічкою, проте у разі необхідності бібліотеку можна розширити і додати інші варіанти запуску застосунків.

Для визначення лаунчера ми розробили інтерфейс AppLauncher, імплементації для командної стрічки якого використовуються у SparkAppController.

```
public interface AppLauncher {
    CompletableFuture<LaunchResult>
```

```
    launch(SparkAppConfiguration configuration) throws FailedLaunchException;
void terminate();
        static AppLauncher create() {
            ConfigProvider provider = ConfigProvider.get();
            String launcherType = provider.launcherType();
        switch (launcherType) {
        case "cli":
            return new CliAppLauncher();
       default:
                    throw new RuntimeException("Unsupported launcher type " +
launcherType);
            }
        }
    }
```

Основна ідея роботи інтерфейсу полягає у наступному: інтерфейс приймає об’єкт SparkAppConfiguration, на основі якого запускає застосунок Spark, і також він може зупиняти його за допомогою методу terminate().

Імплементацією цього інтерфейсу для запуску застосунку за допомогою командної стрічки є клас CliAppLauncher. Розглянемо структуру цього класу детальніше.

Серед змінних, що використовуються у класі, слід визначити:

- *configuration* – об’єкт типу SparkAppConfiguration, що містить конфігурацію Spark;
- *converterer* – об’єкт типу SparkConfigurationConverter, що конвертує конфігурацію у відповідний формат для передачі у командну стрічку;
- *configProvider* – об’єкт типу ConfigProvider, що використовується для отримання певних значень конфігурації.
- *launchProcess* – об’єкт типу Process, що представляє запущений процес застосунку Spark.

Далі у програмному коді створюється конструктор, що ініціалізує змінну converter об'єктом типу SparkConfigurationConverter.

```
 public CliAppLauncher() {
   this.converter = new SparkConfigurationConverter();
}
```

Далі перевизначаємо методи launch(SparkAppConfiguration configuration), terminate().

```
 @Override
   public void terminate() {
      if (launchProcess != null && launchProcess.isAlive()) {
launchProcess.destroyForcibly();
      }
   }
   private void logOutput(Process p) {
      log.info("App " + this.configuration.getName() + " exit code: " + p.exitValue());
log.debug("App " + this.configuration.getName() + " std output");
      BufferedReader reader = new BufferedReader(new
InputStreamReader(p.getInputStream())); reader.lines().forEach(log::debug);
   }
}
```

Таким чином, щойно було розглянуто структуру класу CliAppLauncher, що відповідає за роботу із застосунком за допомогою командної стрічки.

3.7. Приклад Spark-застосунку для тесту

Продемонструємо роботу розробленої роботи на прикладі Spark-застосунку, що обчислює число π. Приклад коду тексту наведений у модулі demo, в той час як код самого застосунку, що буде тестуватися – в модулі client у пакеті examples.

Ідея тестованого застосунку подібна до прикладу стандартного Sparkзастосунку із документації фреймворку, але з певними змінами.

```
public static void main(String[] args) throws Exception {
  SparkSession spark = SparkSession.builder().getOrCreate();
  JavaSparkContext jsc = new JavaSparkContext(spark.sparkContext());
TestHelper helper = TestHelper.create(jsc.getConf());
  helper.notifyAppStarted();
```

В цьому уривку коду тесту застосунку Spark ми в першу чергу створюємо Spark-сесію, викликаємо конструктор JavaSparkContext, що є обгорткою SparkContext і створюємо об’єкт TestHelper, завдяки якому організовується послідовність дій виконання тесту.

Отже, дана частина коду є початковою точкою запуску Spark-застосунку і включає необхідні ініціалізаційні кроки для налагодження взаємодії з Sparkкластером та іншими компонентами системи.

Далі ми описуємо логіку самого застосунку. Тут здійснюється обчислення числа π методом Монте-Карло.

На початку відбувається ініціалізація змінних. Далі створюється і заповнюється список. Створюється пустий список ‘l’, який буде містити послідовні числа від 0 до ‘n-1’ і за допомогою циклу for елементи від 0 до ‘n-1’ додаються до списку ‘l’.

```
 int slices = (args.length == 1) ? Integer.parseInt(args[0]) : 2; int n = 100000 *
slices;
List<Integer> l = new ArrayList<>(n); for (int i = 0; i < n; i++) {
```

```
    l.add(i);
}
```

Наступним кроком є створення RDD за допомогою виклику методу parallelize() на об’єкті jsc. В результаті створюється RDD зі списку ‘1’. Параметр ‘slices’ вказує на кількість частин, на які буде розбито RDD для розподіленого обчислення.

```
JavaRDD<Integer> dataSet = jsc.parallelize(l, slices);
```

Далі виконується обчислення шляхом використання методу map(), що застосовується до кожного елементу RDD. Функція генерує випадкові координати `(x, y)` і повертає 1, якщо точка потрапляє в коло з радіусом 1, і 0 - в протилежному випадку. Після використовується метод reduce(), який зводить всі значення до одного, шляхом додавання. Отримане значення count представляє загальну кількість точок, які потрапили в коло.

```
int count = dataSet.map(integer -> {     double x = Math.random() * 2 - 1;
 double y = Math.random() * 2 - 1;
 return (x * x + y * y <= 1) ? 1 : 0;
}).reduce((integer, integer2) -> integer + integer2);
```

Після цього здійснюється обчислення результату та запис у TestHelper і зупинка контексту Spark через метод stop().

```
String result = String.valueOf(4.0 * count / n); helper.writeResult(result);
```

```
jsc.stop();
```

3.8. Написання тесту і його запуск

Сам тест, на прикладі якого ми перевірятимемо роботу розробленої бібліотеки виглядає наступним чином:

```
public void sparkPiTest() {
    SparkAppConfiguration application = SparkAppConfiguration.builder()
            .name("SparkPiTest")
            .mainClass("com.demchuk.examples.SparkPiExample")
    .appJar("/mnt/c/users/Sonic/IdeaProjects/Spark_demo/client/target/SparkTestClient.jar")
            .args(List.of("100"))
            .build();
    AppResult result = application.submit();
    Assertions.assertThat(result.getResult()).contains("3.14");
}
```

Тут відбувається наступна послідовній дій:

1. Створюємо конфігурацію застосунку і передаємо властивості конфігурації.
2. Далі викликаємо метод submit() для запуску застосунку за заданою конфігурацією.
3. Наступним кроком результат перевіряється – спочатку ми отримуємо результат застосунку у форматі рядка і перевіряємо, чи містить результат рядок «3.14».

Запуск тесту відбувається через WSL систему. Для того, щоб передати WSL застосунок, що знаходиться у файлах на Windows-системі, використовуємо команду mnt.

Перед запуском тестів необхідно створити redis-сесію на контейнері docker за допомогою команди.

docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redisstack:latest

Якщо ми відкриємо шлях localhost:8001 у браузері, ми перейдемо до сторінки користувацького інтерфейсу нашого Redis-сервісу.

Також перед запуском тестів, усі JAR-файли необхідно зібрати, щоб система використовувала артефакти із локального репозиторію. Робимо це за допомогою команди в директорії застосунку (переходимо в директорію за допомогою команди cd [необхідна директорія або шлях до неї]):

mvn clean package -D skipTests

Наступним кроком можна запускати самі тести. Для цього необхідно зайти в директорію із застосунком, у модуль із тестами (в нашому випадку це demo).

Тести запускаються за допомогою команди mvn test (рис. 3.8.).

Рис. 3.8.

Після запуску тестів отримуємо наступний результат:

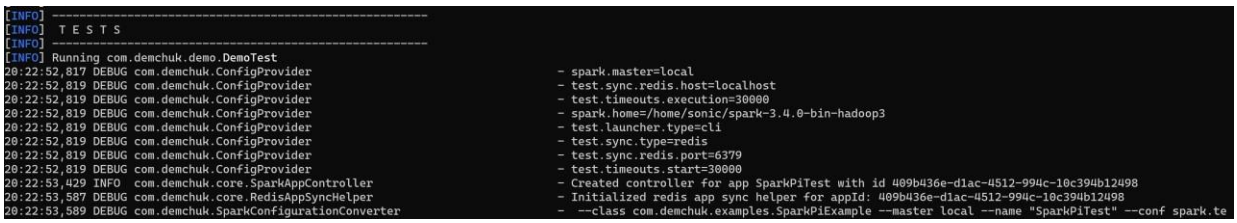

Рис. 3.9. Логування даних про конфігурацію

В першу чергу ми бачимо, що система логує дані про конфігурацію, що включає у собі різні параметри для запиту. Готовий submit-запит також виводиться, що проілюстровано на рис. 21.

Рис. 3.10. Spark-submit запит

Уся послідовність дій запуску та обробки застосунку також логується.

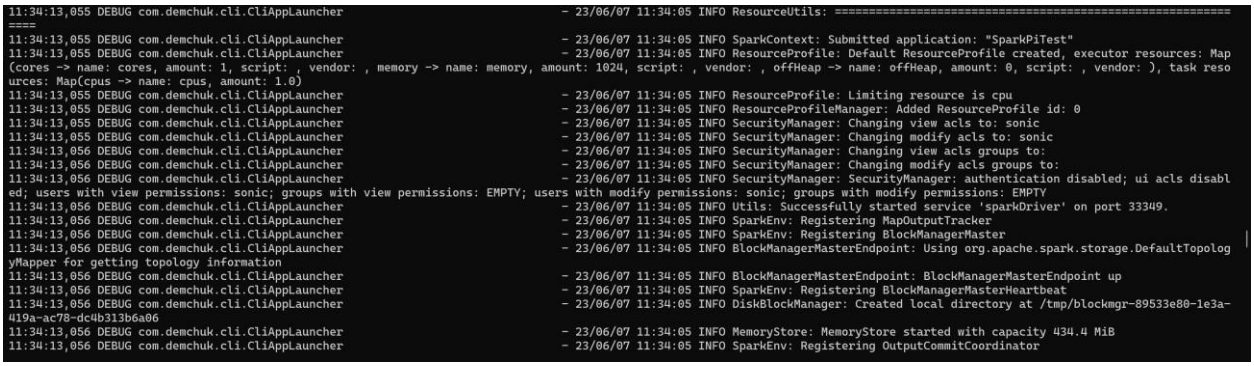

Рис. 3.11.

І наприкінці виведеного результату ми можемо побачити інформацію про те, чи вдало було виконано тести і за який проміжок часу. Якщо фактичний результат відповідає очікуваному, то в такому випадку виводиться інформація про те, що тест було пройдено успішно.

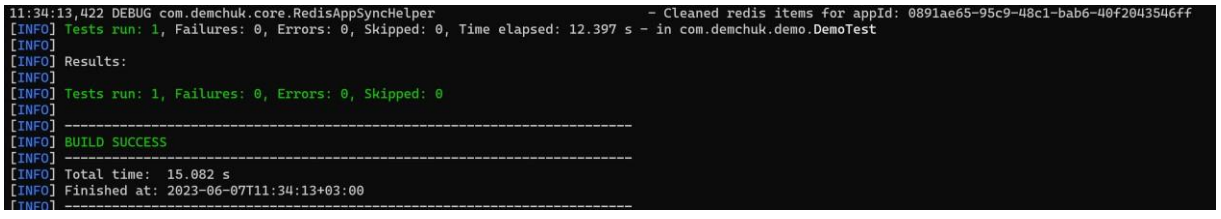

Рис. 3.12.

Спробуємо передати некоректні дані в тест. Наприклад, замість числа передамо слово number.

Assertions.assertThat(result.getResult()).contains("number");

Результат, що ми отримуємо проілюстрований на рис. 3.13.

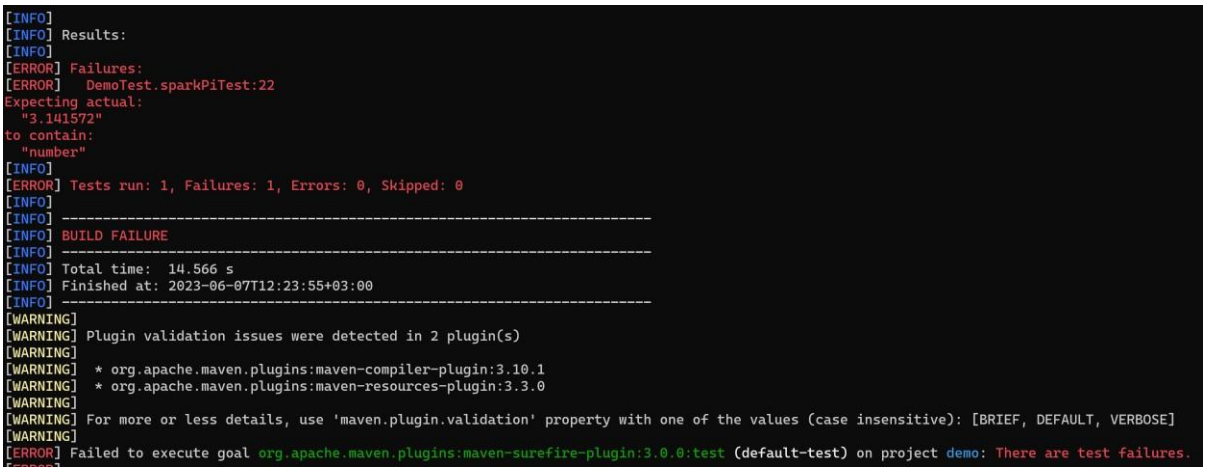

Рис. 3.13.

Як і очікувалося, тест не пройшов, а на екрані можна побачити повідомлення про помилку, а саме про те, що фактичний результат не містить у собі слова “number”.

Таким чином, ми перевірили роботу бібліотеки на прикладі тесту із коректно заданими даними і поведінку бібліотеки у випадку, якщо дані для перевірки задані некоректно. Той факт, що тест не пройшов і у результаті запуску ми отримали повідомлення про помилку із описом причини, вказує на те, що бібліотека правильно працює в контексті некоректних даних, що є важливим аспектом, щоб не допустити неточностей у процесі виконання тестів.

3.9. Висновки до розділу 3

У цьому розділі ми провели розробку тестової бібліотеки для валідації та ефективного тестування Apache Spark та клієнтських бібліотек, створених на основі фреймворку. Метою нашої роботи було створення гнучкого інструменту, який допоможе розробникам та тестувальникам перевіряти правильність роботи своїх Spark-застосунків шляхом здійснення системного тестування.

Бібліотеку було розроблено мовою програмування Java у IDE IntelliJ IDEA з використанням усіх переваг та функціональностей, що спрощують розробку програмного коду і тестування. Роботу бібліотеки було продемонстровано на прикладі стандартного Spark-застосунку, що обчислює число π методом Монте-Карло. Результати підтвердили, що розроблена бібліотека є ефективним і надійним інструментом тестування Spark.

Дана бібліотека без розширення може знайти застосування для тестування Spark-застосунків написаних мовою Java та Scala, що обумовлено сумісністю Scala із JVM.

Завдяки розробленій бібліотеці, розробники та тестувальники можуть забезпечити високу якість своїх Spark-застосунків, та забезпечити їх надійність та ефективність.

У розробленій бібліотеці імплементовано лише один варіант запуску застосунку – через командний рядок, що дозволяє користувачам легко вказувати параметри і налаштування для своїх Spark-застосунків, і одне середовище спільного доступу для синхронізації – база даних типу «ключ-значення» Redis.

Однак важливо зауважити, що розроблена бібліотека має потенціал для розширення та налаштування залежно від потреб користувача. Завдяки гнучкій архітектурі та організованим інтерфейсам, розробникам можна додавати нові варіанти запуску застосунків, розширювати функціональні можливості та підключати різноманітні середовища спільного доступу для синхронізації даних. Це дає можливість користувачам адаптувати бібліотеку для своїх конкретних потреб і забезпечує гнучкість та розширюваність для подальшого розвитку.

ВИСНОВКИ

Отже, в даній роботі було розроблено бібліотеку для тестування Spark, його застосунків та клієнтських бібліотек. Головною метою, що стояла при виконанні роботи, було розробити бібліотеку, що надасть можливість написання автоматизованих тестів для системного рівня тестування Spark і можливість легко і просто фіксувати баги та помилки.

Розроблена бібліотека реалізовує задану мету завдяки використанню логування та методів, що можуть зафіксувати некоректну конфігурацію ще на ранніх етапах, користувач має можливість перевіряти роботу застосунку і у разі виникнення дефекту отримувати чітку інформацію про точку застосунку, де виникли проблеми, щоб виправити їх.

Розробка бібліотеки здійснювалася в IDE IntelliJ IDEA, що є чудовим варіантом при роботі із мовою програмування Java та фреймворком Spark. Завдяки сумісності Java і Scala, нативної мови фреймворку, бібліотека підійде для написання тестів на системному рівні як для застосунків мовою Java, так і Scala.

В ході роботи у якості середовища спільного доступу, що використовується для синхронізації компонентів бібліотеки та отримання результатів, було використано noSQL базу даних Redis, що запускалася в контейнері Docker, щоб позбавитися необхідності встановлювати Redis.

Отже, Redis забезпечує швидкий доступ до даних та можливість розподіленого кешування, що покращує продуктивність тестування. Використання Docker контейнера для запуску Redis спрощує процес налаштування і установки, забезпечуючи зручне середовище для використання бази даних.

Отже, у першому розділі роботи було розглянуто фреймворк Apache Spark, його основні бібліотеки та архітектуру, а також принцип роботи, визначено його переваги над іншими фреймворками, а також недоліками. Дослідження його роботи та архітектури є важливим етапом для формування розуміння того, як необхідно розроблювати бібліотеку і які етапи запуску застосунку можливо автоматизувати.

У другому розділі було розглянуто поняття тестування, його рівні, а також обґрунтовано необхідність автоматизації процесу тестування Spark. Spark є складним і багатошаровим фреймворком, так само як його застосунки зазвичай є дуже складними для ручного тестування.

У третьому розділі був описаний процес розробки бібліотеки, його основні модулі та архітектура. Бібліотека складається з трьох основних модулів: core, client, demo. В процесі розробки для зручності виконання тестів було автоматизовано процес відправки spark-submit запиту, що запускає застосунок. Бібліотека імплементує різні варіанти запуску застосунку і середовищ спільного доступу, такі як командна стрічка і Redis база даних. Користувач має можливість розширювати бібліотеку, додаючи інші варіанти запуску та середовища спільного доступу залежно від власних потреб та вимог.

Виконання Spark-застосунку та його тестування здійснювалося локально, проте бібліотека може використовуватися і для тестування розподілених застосунків. Для використання бібліотеки для системного тестування власних бібліотек користувач повинен імпортувати JAR-файли з модулем core і demo, а також встановити відповідні залежності.

Роботу бібліотеки було продемонстровано на прикладі стандартного Spark-застосунку, що обчислює число π методом Монте-Карло. Було продемонстровано поведінку бібліотеки у разі введення коректних і некоректних даних, і результати показали, що розроблена бібліотека працює правильно і не пропускає помилок, надає детальну інформацію про запуск і роботу застосунку, дані конфігурації і місце, де виникла помилка.

Отже, розроблена бібліотека працює правильно і надає надійну підтримку для системного тестування Spark-застосунків. Вона здатна виявляти помилки, та надавати корисну інформацію про запуск застосунку, дані конфігурації та точку, де виникла помилка. Бібліотека надає інструменти для автоматизованого тестування системного рівня, що спрощує процес перевірки працездатності та відповідності очікуваним результатам.

Зокрема, бібліотека дозволяє перевіряти правильність встановлення конфігурації Spark, виявляти проблеми з підключенням до середовища спільного доступу, контролювати виконання задач та перевіряти очікувані результати. При виявленні помилки, бібліотека надає детальну інформацію, що допомагає знайти та виправити проблему.

Таким чином, розроблена бібліотека для системного тестування Spark демонструє свою ефективність і корисність, надаючи розробникам засоби для автоматизованого тестування, полегшуючи виявлення та виправлення помилок, та забезпечуючи надійну підтримку при розробці та підтримці Spark-проектів.

Розроблена бібліотека може бути корисною для розробників, що працюють зі Spark-застосунками чи над клієнтськими бібліотеками Spark, незалежно від їх рівня досвіду. Вона надає автоматизацію складання spark-submit запитів, а також можливість складання автоматизованих тестів, що допомагають зекономити час та зусилля, спрощують процес виявлення помилок і забезпечують більш швидке виправлення. Вона може значно підвищити продуктивність розробників, забезпечити стабільність та якість розроблюваного ПЗ та сприяти покращенню спільної роботи в команді.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 30+ Incredible Big Data Statistics [Electronic recourse]. Access mode: https://explodingtopics.com/blog/big-data-stats (lastaccess: 16.05.2023). – Title from the screen.
2. What is big data? [Electronic recourse]. Access mode: https://www.oracle.com/big-data/what-is-big-data/ (lastaccess: 16.05.2023). – Title from the screen.
3. Bill Chambers. Spark: The Definitive Guide / Bill Chambers, Matei Zaharia – O'Reilly Media, Inc., 2020. – 624p.
4. Spark Overview [Electronic recourse]. Access mode: https://spark.apache.org/docs/latest/ (lastaccess: 18.05.2023). – Title from the screen.
5. Mastering Apache Spark [Electronic recourse]. Access mode: https://bjpcjp.github.io/pdfs/tools/spark-mastery.pdf (lastaccess 15.05.2023). – Title from the screen.
6. Holden Karau. High Performance Spark / Holden Karau, Rachel Warren. – O'Reilly Media, Inc., 2017. – 358p.
7. Raul Estrada. Big Data SMACK / Raul Estrada, Isaac Ruiz. - Apress Berkeley, CA, 2016. – 289p.
8. Hien Lu. Beginning Apache Spark 2 / Hien Lu – Apress Berkeley, CA, 2018. – 404p.
9. The difference between deployments and releases [Electronic resource]. Access mode: https://octopus.com/devops/continuous-delivery/deployments-vs-releases/#technical-and-business-tension (last access: 30.05.2023). – Title from the screen.
10. Dorothy Graham. Foundations of Software Testing ISTQB Certification / Dorothy Graham, Erik van Veenendaal, Dorothy Graham. – O'Reilly Media, Inc., 2019. – 242p.
11. Aruna Deoskar. Software Testing / Aruna Deoskar, Ms. Jyoti J. Malhotra, Vikas S. Tayade. – Nirali Prakashan, 2018. – 189p.

12. Unit Testing Tutorial – What is, Types & Test Example [Electronic resource]. Access mode: https://www.guru99.com/unit-testing-guide.html (lastaccess: 01.06.2023). – Title from the screen.

13. What is Integration Testing: Learn with Integration Testing Examples [Electronic resource]. – Access mode: https://www.softwaretestinghelp.com/what-is-integration-testing/ (lastaccess: 30.05.2023). – Title from the screen.

14. Роберт Сесіл Мартін. Чистий код: створення і рефакторінг за допомогою Agile» / Роберт Сесіл Мартін. – Вид-во «Фабула» , 2019. – 464с.

15. Automated and Manual Testing [Electronic resource]. Access mode: https://www.browserstack.com/guide/manual-vs-automatedtesting-differences (lastaccess: 01.06.2023). – Title from the screen.

16. Automated Testing [Electronic resource]. Access mode: https://www.techtarget.com/searchsoftwarequality/definition/automated-software-testing (lastaccess: 01.06.2023). – Title from the screen.

17. Which language is better for Spark & Why? [Electronic resource]. Access mode: https://www.knowledgehut.com/blog/programming/scala-vs-python-vs-r-vs-java (lastacess: 01.06.2023). – Title from the screen.

18. Олексій Васильєв. Програмування мовою Java / Олексій Васильєв. – Вид-во «Навчальна книга – Богдан», 2020. – 696с.

19. Arnon Axelrod. Complete Guide to Test Automation Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects / Arnon Axelrod. — Apress, 2018. – 542p.

20. Docker documentation [Electronic resource]. Access mode: https://docs.docker.com/get-started/ (last access: 05.06.2023) – Title from the screen.

21. Redis documentation [Electronic resource]. Access mode: https://redis.io/docs/about/ (last access: 05.06.2023) – Title from the screen.

ДОДАТКИ

Додаток А

Діаграма класів основних компонентів модуля core

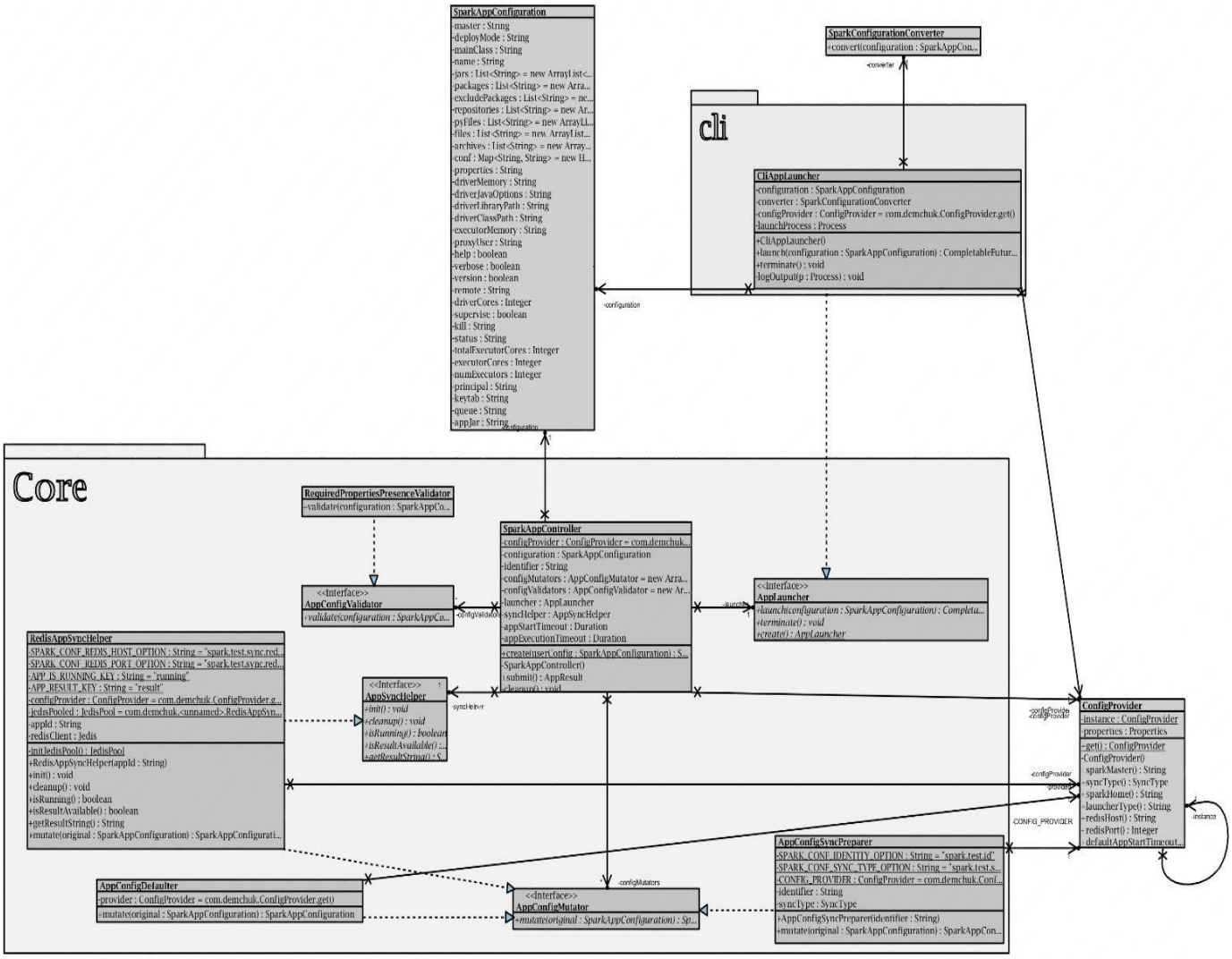

Рис. 1. Діаграма основних клачів модуля core

Повний код головного класу застосунку

```
package com.demchuk.core;

import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import com.demchuk.ConfigProvider;
import com.demchuk.SparkAppConfiguration;

import java.time.Duration;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import java.util.concurrent.Callable;
@Slf4j
public class SparkAppController {

    private static ConfigProvider configProvider = ConfigProvider.get();

    private SparkAppConfiguration configuration;
    private String identifier;

    private List<AppConfigMutator> configMutators = new ArrayList<>();
    private List<AppConfigValidator> configValidators = new ArrayList<>();
    private AppLauncher launcher;
    private AppSyncHelper syncHelper;
    private Duration appStartTimeout;
```

```
private Duration appExecutionTimeout;

    public static SparkAppController create(SparkAppConfiguration userConfig) {
        SparkAppController controller = new SparkAppController();
        String identifier = UUID.randomUUID().toString();

        // copy user config
        controller.configuration = userConfig.toBuilder().build();
        controller.identifier = identifier;

        controller.configMutators.add(new AppConfigDefaulter());
        controller.configMutators.add(new AppConfigSyncPreparer(identifier));

        // add validators
        RequiredPropertiesPresenceValidator validator = new
RequiredPropertiesPresenceValidator();
        controller.configValidators.add(validator);

        // init launcher
        controller.launcher = AppLauncher.create();

        // init sync helper
        RedisAppSyncHelper redisSyncHelper = new RedisAppSyncHelper(identifier);
        controller.syncHelper = redisSyncHelper;
        controller.configMutators.add(redisSyncHelper);

        // init timeouts
        if (userConfig.getStartTimeout() != null) {
```

```
        controller.appStartTimeout = Duration.of(userConfig.getStartTimeout(),
ChronoUnit.MILLIS);
      }
else {
        controller.appStartTimeout =
Duration.of(configProvider.defaultAppStartTimeoutMs(), ChronoUnit.MILLIS);
      }

      if (userConfig.getExecutionTimeout() != null) {
        controller.appExecutionTimeout = Duration.of(userConfig.getExecutionTimeout(),
ChronoUnit.MILLIS);
      } else {
        controller.appExecutionTimeout =
Duration.of(configProvider.defaultAppExecutionTimeout(), ChronoUnit.MILLIS);
      }

      log.info("Created controller for app " + userConfig.getName() + " with id " +
controller.identifier);
      return controller;
    }

    private SparkAppController() {}

    public AppResult submit() {
      // update configuration: set defaults
      // update configuration: set test identifiers
      for (AppConfigMutator mutator : configMutators) {
        configuration = mutator.mutate(configuration);
```

```
        }

// validate configuration

  configValidators.forEach(validator -> validator.validate(this.configuration));
        // launch application
        syncHelper.init();
        launcher.launch(configuration);

        // wait for app to start or timeout
        try {
            log.info("Application " + identifier + " launched, waiting for start");
            waitFor(() -> syncHelper.isRunning(), appStartTimeout);
        } catch (TestAppTimeoutException e) {
            log.error("App " + identifier + " failed to start during " +
appStartTimeout.getSeconds() + " seconds");
            cleanup();
            throw new LaunchTimeoutException(e);
        }
        // wait for app to complete or timeout
        try {
            log.info("Application " + identifier + " started, waiting for completion");
            waitFor(() -> syncHelper.isResultAvailable(), appExecutionTimeout);
        } catch (TestAppTimeoutException e) {
            log.error("App " + identifier + " failed to complete during " +
appExecutionTimeout.getSeconds() + " seconds");
            cleanup();
            throw new AppExecutionTimeoutException(e);
```

```
        }
        // read result
  String result = syncHelper.getResultString();

// cleanup
        cleanup();
        // return result
        return new AppResult(result);
    }

    private void cleanup() {
        launcher.terminate();
        syncHelper.cleanup();
    }

    @SneakyThrows
    private void waitFor(Callable<Boolean> action, Duration timeout) throws
TestAppTimeoutException {
        long deadline = System.currentTimeMillis() + timeout.toMillis();
        Exception exception = null;
        while (System.currentTimeMillis() < deadline) {
            try {
                if (Boolean.TRUE.equals(action.call())) {
                    return;
                } else {
                    Thread.sleep(500L);
                }
            } catch (InterruptedException e) {
                throw e;
```

```
            } catch (Exception e) {

log.trace("Exception while waiting: " + e);
                exception = e;
            }
        }

        if (exception != null) {
            throw new TestAppTimeoutException(exception);
        } else {
            throw new TestAppTimeoutException();
        }
    }
}
```

Pom.xml файл модулю core

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>com.demchuk</groupId>
        <artifactId>spark_test_framework</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>core</artifactId>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>4.3.1</version>
        </dependency>
    </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <finalName>${core.jar.name}</finalName>
          <appendAssemblyId>false</appendAssemblyId>
          <outputDirectory>${project.build.directory}</outputDirectory>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

</project>
```

Pom.xml файл модулю client

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>com.demchuk</groupId>
        <artifactId>spark_test_framework</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>client</artifactId>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${spark.scala.version}</artifactId>
            <version>${spark.version}</version>
            <scope>provided</scope>
```

```
    </dependency>
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_${spark.scala.version}</artifactId>
        <version>${spark.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>4.3.1</version>
    </dependency>
 </dependencies>

 <build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <finalName>${client.jar.name}</finalName>
                <appendAssemblyId>false</appendAssemblyId>
                <outputDirectory>${project.build.directory}</outputDirectory>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
```

```
            </configuration>
            <executions>

      <execution>
                  <id>make-assembly</id>
                  <phase>package</phase>
                  <goals>
                     <goal>single</goal>
                  </goals>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>

</project>
```

Повний код RedisAppSyncHelper

```
package com.demchuk.core;

import com.demchuk.ConfigProvider;
import lombok.extern.slf4j.Slf4j;
import com.demchuk.SparkAppConfiguration;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

import java.util.HashMap;
import java.util.Map;

@Slf4j
public class RedisAppSyncHelper implements AppSyncHelper, AppConfigMutator {

    private static final String SPARK_CONF_REDIS_HOST_OPTION =
"spark.test.sync.redis.host";
    private static final String SPARK_CONF_REDIS_PORT_OPTION =
"spark.test.sync.redis.port";
    private static final String APP_IS_RUNNING_KEY = "running";
    private static final String APP_RESULT_KEY = "result";

    private static ConfigProvider configProvider = ConfigProvider.get();
    private static JedisPool jedisPooled = initJedisPool();

    private final String appId;
```

```
private final Jedis redisClient;

private static JedisPool initJedisPool() {
    String redisHost = configProvider.redisHost();
    Integer redisPort = configProvider.redisPort();

    return new JedisPool(redisHost, redisPort);
}

public RedisAppSyncHelper(final String appId) {
    this.appId = appId;
    this.redisClient = jedisPooled.getResource();
}

@Override
public void init() {
    Map<String,String> data = new HashMap<>();
    data.put("initialized", "true");
    redisClient.hset(appId, data);
    log.debug("Initialized redis app sync helper for appId: " + appId);
}

@Override
public void cleanup() {
    redisClient.del(appId);
    redisClient.close();
    log.debug("Cleaned redis items for appId: " + appId);
}
```

```
@Override
    public boolean isRunning() {
        Map<String,String> data = redisClient.hgetAll(appId);
        return Boolean.parseBoolean(data.getOrDefault(APP_IS_RUNNING_KEY,
"false"));
    }
    @Override
    public boolean isResultAvailable() {
        Map<String,String> data = redisClient.hgetAll(appId);
        return data.containsKey(APP_RESULT_KEY);
    }
    @Override
    public String getResultString() {
        Map<String,String> data = redisClient.hgetAll(appId);
        return data.get(APP_RESULT_KEY);
    }
    @Override
    public SparkAppConfiguration mutate(SparkAppConfiguration original) {
        Map<String,String> newConfig = new HashMap<>(original.getConf());
        newConfig.put(SPARK_CONF_REDIS_HOST_OPTION,
configProvider.redisHost());
        newConfig.put(SPARK_CONF_REDIS_PORT_OPTION,
String.valueOf(configProvider.redisPort()));
        return original.toBuilder()
                .conf(newConfig)
                .build();
    }
}
```