OPIS ROZWIAZANIA
NAZWA: Parser Logów
AUTORZY:

- **Grupa 1: Hubert Makowski**
  Marcin Burczyk
  Paweł Kaczmarek
- **Grupa 2 : Piotr Falkiewicz**
  Agnieszka Gontarek
  Filip Nienartowicz
  Joanna Sołomiewicz

DATA UTWORZENIA: 25.03.2018

Historia dokumentu:

Wersja /status	***Data***	***Opis zmiany***	***Przyczyna zmiany***	***Dokonujący zmiany***
0.1	25.03	Wprowadzenie	Zainicjowanie projektu	Hubert Makowski
0.1	26.03	Wprowadzenie - uzupełnienie	Przyrost 1	Filip Nienartowicz
0.2	04.04	Etap II - uzupełnienie przetwarzania	Przyrost 2	Marcin Burczyk
0.3	09.04	Etap III - przetwarzanie złożonych planów	Przyrost 3	Hubert Makowski
0.2	10.04	Etap II - zaawansowane przetwarzanie	Przyrost 2	Piotr Falkiewicz
0.3	10.04	Etap III - detekcja błędów w logach	Przyrost 3	Piotr Falkiewicz
0.4	19.04	Etap IV - integracja	Przyrost 4	Hubert Makowski
0.4	9.05	Etap IV - integracja	Przyrost 4	Agnieszka Gontarek

Spis treści

1 PRZEDMIOT OPRACOWANIA

Opis rozwiązania problemu analizy logów z planów i ich wykonań z platformy Abinitio. Realizacja projektu zakłada napisanie oprogramowania w języku PYTHON. Podział na grupy odzwierciedla przydział tematów:

- Grupa numer 1: Plany wykonania.
- Grupa numer 2: Logi z wykonań konkretnych zadań

2 PROJEKT FUNKCJONALNY

2.1 Opis proponowanego rozwiązania

Projekt będzie podzielony na kilka etapów odpowiedzialnych za:

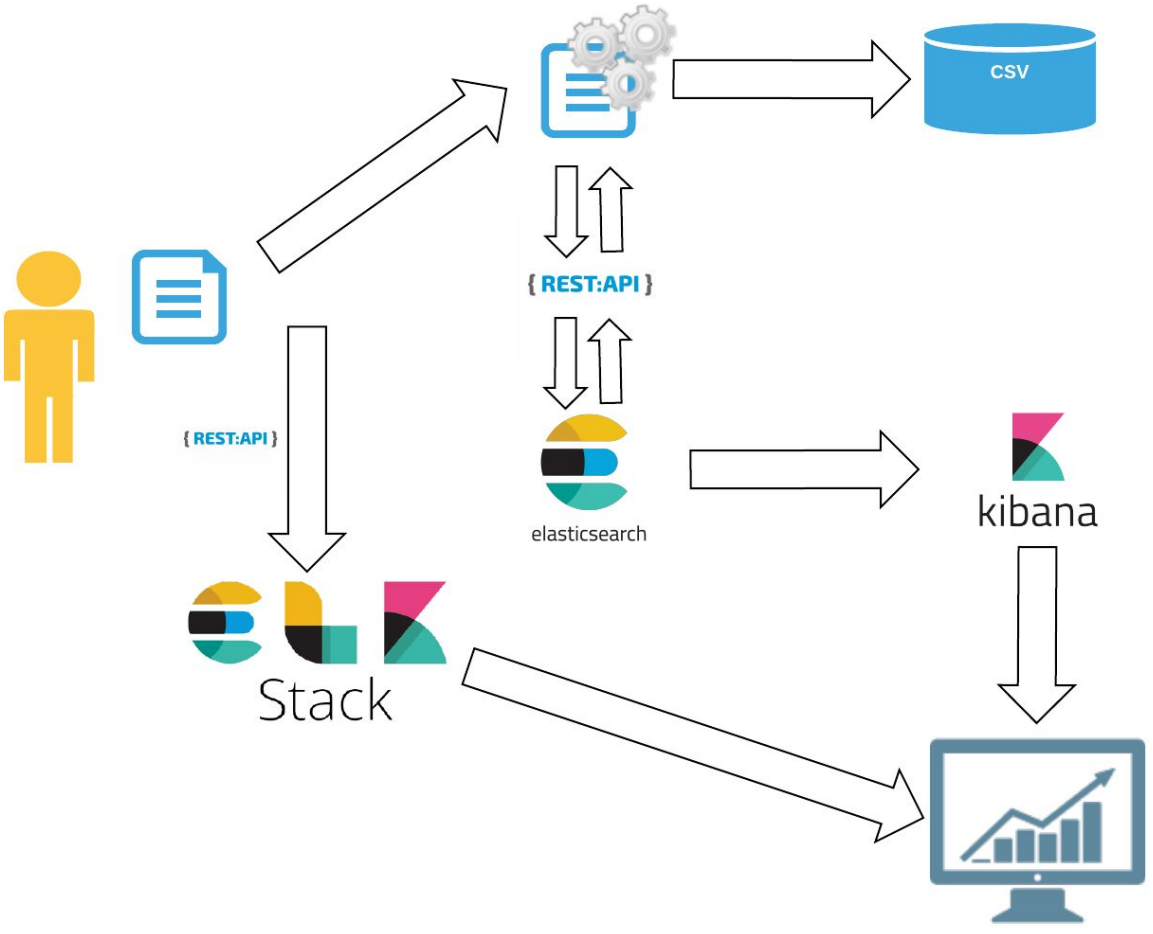
Etap I: Analiza danych

1. wczytanie plików logu.
2. załadowanie potrzebnych informacji do obiektów.
3. wyznaczenie wszystkich wartości niezbędnych do dalszej analizy/wizualizacji.
4. zapis do bazy danych/plik csv.

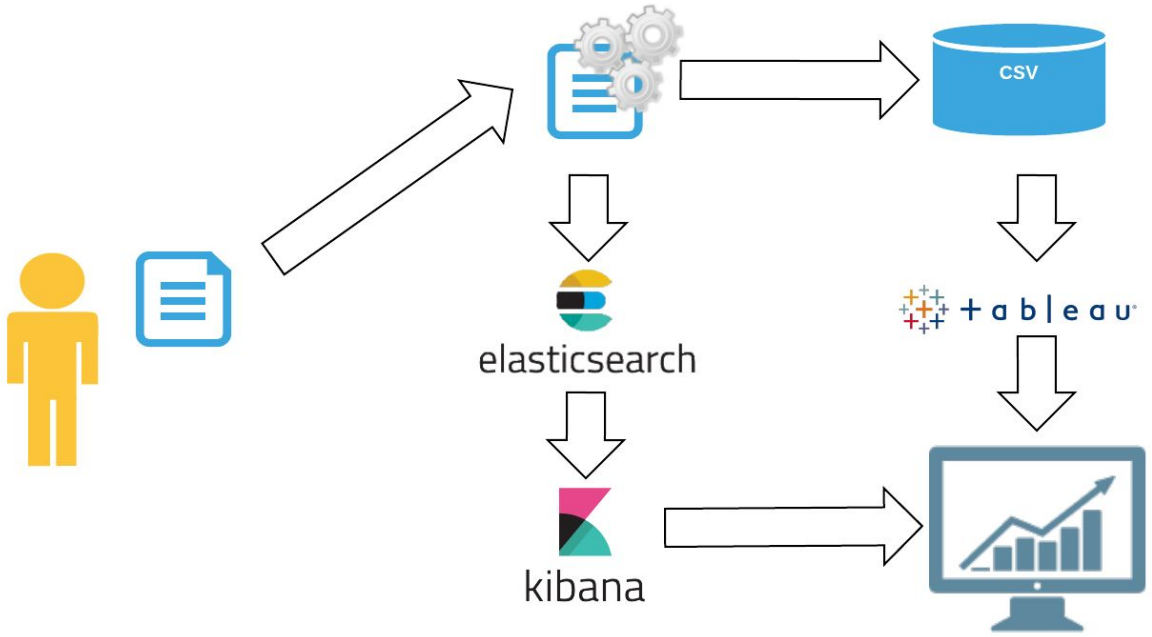
Etap II: Wizualizacja

1. Tableau
2. Kibana
    a. załadowanie danych na serwer Elasticsearch

2.2 Diagram architektury



2.3 Diagram architektury - uzgodniony



2.4 Przetwarzanie

1. Wczytanie plików wsadowych.
2. Załadowanie danych do modeli (klas).
    a. przeszukiwanie pliku na podstawie wzorców
    b. występowanie danych na konkretnych pozycjach
3. Zapisanie danych w bazie danych/plik csv.

Realizacja w kodzie:

```
def splitLineByBrackets(text=""):
   index = text.find("{")
   if index > -1:
       nextOpenIndex = text.find("{", index + 1)
       nextCloseIndex = text.find("}", index + 1)
       if nextOpenIndex < 0 or nextCloseIndex < nextOpenIndex:
           return [text[index + 1:nextCloseIndex]], nextCloseIndex
       endTextIndex = nextOpenIndex
       array = []
       while 0 < nextOpenIndex < nextCloseIndex:
           # Dopisujemy tekst między klamrami
           if len(array) > 0:
               array.append(text[endIndex + 1:nextOpenIndex])
           tmp, endIndex=splitLineByBrackets(text[nextOpenIndex:])
           endIndex += nextOpenIndex
           array.append(tmp)
           nextOpenIndex = text.find("{", endIndex + 1)
           nextCloseIndex = text.find("}", endIndex + 1)
       return [text[index + 1:endTextIndex]] + array +
[text[endIndex + 1:nextCloseIndex]], nextCloseIndex
```

Funkcja rozbija nam plan wykonania na tablicę wielowymiarową.
Przykładowo “{test{2}}” na [“test”,[“2”],””].
Następnie wyszukujemy w niej elementy zawierające parametry które musimy wczytać np. “XXG_task_prior_task” czy “XXGtask”.

Relacje zapisujemy w pliku csv w formacie:
task, success, failure

2.5 Komentarze i Opis kodu.

2.5.1 Grupa 1

Struktura projektu:

HD_Projekt - główny katalog projektu
- **DOC** - dokumentacja od banku
- **project1** - katalog projektu grupy 1
  - **resources** - dane wejściowe
  - **src** - kody źródłowe
  - **test** - testy jednostkowe

Katalog src:

GraphTask - klasa przechowuje:
- (*)nazwę (name)
- (*)zasoby (resources)
- (*)czy jest wyłączone (disable)
- czy w głównym planie (inMainPlan)
- (*)ścieżka do grafu (path)

Posiada metody:
- setOptions():
  - Wejście: tablica kilku-wymiarowa (wycinek pliku logu)
  - Wyjście: uzupełniony obiekt o dane (*)

Logger - klasa zapisująca logi z wykonań programu

Posiada metody:
- log:
  - Wejście: treść i typ logu
  - Wyjście: Plik logu (log1.log w kalatogu “resultDir - parametr w main”)

searchFunctions - plik zawierający funkcje

Posiada funkcje:
- searchTasks():
  - tablica wielowymiarowa (elementów z splitLineByBrackets) (array)
  - słownik: nazwa zadania (name): instancja klasy GraphTask
- searchRelations():
  - tablica wielowymiarowa (elementów z splitLineByBrackets) (array)
  - tablica tablica 2-wymiarowa relacji (relations)
- splitLineByBrackets():
  - Wejście: treść wiersza logu (text)
  - Wyjście: tablica wielowymiarowa (podział względem “{}”)
- splitByBrackets():
  - Wejście: ścieżka do pliku logu (read_file)
  - Wyjście: tablica wielowymiarowa (elementów z splitLineByBrackets) (array)
- findTasksSubplans():
  - Wejście: tablica tablica 2-wymiarowa relacji (relations) i słownik zadań (tasks)
  - Wyjście: uzupełnione pola “czy w głównym planie (inMainPlan)”
- search():
  - Wejście: tablica tablica 2-wymiarowa relacji (relations), słownik zadań (tasks), nazwa zadania (taskName), kierunek (direction) (<- True -> False)
  - Wyjście: wykonania rekurencyjne (wyszukiwanie kolejnych zagnieżdżeń w planie)

main - plik z metodą main:

- Wejście: nazwa pliku logu i ścieżka do katalogu wynikowego
- Wyjście: wynikowe pliki w “resultDir”

2.5.2 Grupa 2

Skrypt wykorzystuje pliki logów oraz uzyskane (patrz pkt nt 2.4) dane nt zależności między zadaniami. Na tej podstawie wyznaczane są czasy przetwarzania tj. czas startu zadania, czas zakończenia, czas wykonania oraz czas oczekiwania na zasoby. Dodatkowo wyznaczana jest liczba błędów oraz szczegółowe informacje na ich temat.

Kod standaryzujący zadania:

```
def normalTasks(candidates, tasks):
    result = []
    for task in tasks:
        row = []
        start = ''
        end = ''
        for line in candidates:
            if getTypeNameTime(line)[1] == parseTaskName(task):
                if getTypeNameTime(line)[0] == 0:
                    start = int(getTypeNameTime(line)[2])
                elif getTypeNameTime(line)[0] == 1:
                    end = int(getTypeNameTime(line)[2])
        if end != '' and start != '':
            row.append(parseTaskName(task))
            row.append(start)
            row.append(end)
            result.append(row)
```

Struktura projektu:

LogsParser- główny katalog projektu grupy 2
main - kody źródłowe
resources - dane wejściowe
test - testy jednostkowe
resources - dane wejściowe

Katalog main:

WBKErrorDetection.py - plik zawierający funkcje:

- readFile(filename) - wczytuje plik podany jako argument, zwraca listę linii odczytanych z danego pliku
- main() - funkcja wykrywa błędy wykryte w danym wykonaniu planu

WBKLogsMain.py - plik zawierający główną funkcję:

- main(planname, inputDirectory, countOfLogs, relationsFile, outputDirectory) - funkcja realizująca główne założenia projektu, parametry:
  - planname - nazwa planu
  - inputDirectory - ścieżka folderu w którym znajdują się pliki logów
  - countOfLogs - liczba logów do analizy
  - relationsFile - ścieżka wskazująca plik csv z relacjami pomiędzy taskami
  - outputDirectory - ścieżka folderu, w którym będą zapisywane dokumenty wynikowe
- writeErrorsToCSV(`outputDirectory, inputFile, data`) - funkcja zapisuje szczegóły wystąpienia błędów dla danego planu

WBKLogsParser.py - plik zawierający następujące funkcje:

- utilDistinct(listArg) - przetwarza wejściową listę, na listę zawierającą unikalne wartości
- readFile(filename) - wczytuje plik podany jako argument, zwraca listę linii odczytanych z danego pliku
- readCSVFile(filename) - wczytuje plik CSV podany jako argument, zwraca listę odczytanych wierszy
- parseTaskName(taskname) - przetwarza nazwę zadania
- chooseCandidates(lines) - funkcja zwraca tylko takie linie z pliku, które zawierają nazwy zadań
- chooseCandidatesWithTime(lines) - funckja zwraca te linie, które zawierają informację o rozpoczęciu/zakończeniu zadań
- getTypeNameTime(line) - funckja wyodrębnia typ zdarzenia (czyli informację czy zadanie się rozpoczęło czy zakończyło), nazwę zadania oraz czas w którym nastąpiło dane zdarzenie
- parseLinesToNames(candidates) - funkcja zwraca nazwy zadań wyłonione z przekazanych kandydatów
- normalTasks(candidates, tasks) - funkcja przyjmuje linie zawierające zdarzenia rozpoczęcia/zakończenia zadania oraz nazwy zadań, a zwraca tablicę zawierającą informacje (nazwę, czas rozpoczęcia, czas zakończenia) o zadaniach, które się poprawnie wykonały
- reformatRelations(collection) - funkcja wyznacza relację między zadaniami
- calculateTimes(collection, relations) - funkcja wyznacza czas realizacji zadania
- main(inputFile, relationsFile, outputDirectory) - główna funkcja, której argumentami wywołania są wczytywane pliki logów, wyznacza nazwy wykonywanych zadań, czasy przetwarzania oraz oczekiwania na zasoby

3 OGRANICZENIA

- Brak dostępu do oprogramowania Abinitio - testy oparte o przykłady wygenerowane przez kierownika projektu lub tworzone ręcznie (ryzyko błędu)
- W planach (z różnych okresów powstawania) występują różnice w oznaczeniach np. zaznaczenie relacji między Graph_Taskami. W pierwszych dwóch planach było to “G_T1 >> G_T2”, w obecnych planach rzeczywistych z pod planami jest opis, a nie znak “>>”.
- Grupa 1 nie jest w stanie określić, które zadania są przypisane/wykonywane w danych pod-planie. Jednak prawdopodobnie dane te, jest w stanie uzyskać z logów wykonania grupa 2.

4 WARUNKI TESTOWANIA APLIKACJI

4.1 Ogólny opis techniczny

Testowanie aplikacji zostanie przeprowadzone poprzez testy jednostkowe, integracyjne i akceptacyjne.

a) Testy jednostkowe - testowane będą pojedyncze metody. Test będzie składał się z parametrów metody, jej wywołania i sprawdzenia poprawności rezultatów
b) Testy integracyjne - testowane będzie przetworzenie pełnego logu opisującego wywołanie np. planu
c) Testy akceptacyjne - testowane będzie przetworzenie zbioru logów (np. plan i jego logi podrzędne)

Testowanie aplikacji Grupy 2 jest uwarunkowane wygenerowaniem pliku relations.csv zawierającego relację pomiędzy Graph_Taskami.

4.2 Przebieg procesu testowania

Rozwiązanie będzie testowane na kilku etapach

a) Testy pojedynczych metod
b) Testy na podstawie dostarczonych plików logów z oprogramowania Abinitio
c) Testy na podstawie przykładów wygenerowanych przez zespół deweloperski
d) Testy podczas spotkań projektowych w budynku BZ WBK - na podstawie przykładów przygotowanych przez koordynatora projektu

Implementacja oraz testowanie będą miały charakter przyrostowy - każdy etap zwiększy zakres wymagań, które będzie musiało spełnić oprogramowanie. Jego działanie zostanie sprawdzone poprzez wytworzenie bardziej kompleksowych przypadków testowych.

Dodatkowo program będzie logował poprawność wykonanych operacji jak i czasy ich trwania. Zapisywał wynik w pliku: log1.log.

4.3 Wywołanie programu

Program wywoływany jest przy pomocy polecenia w konsoli:

bizon.py -p “plan name” -m (1..4)”

opcje:

- 1 - plan
- 2 - logi
- 3 - zasoby
- 4 - relacje

Domyślnie przetwarzane jest 5 ostatnich plików logu.