

Offer Rider

Projekt na Podstawy Teleinformatyki

Grupa 3

Jakub Riegel, Wojciech Kasperski, Maciej Stosik

Spis Treści

1. DOSTĘP DO PROJEKTU.....	2
WITRYNA INTERNETOWA.....	2
REPOZYTORIUM.....	2
2. OPIS PROJEKTU	2
PROBLEM	2
PROPONOWANE ROZWIĄZANIE	3
FUNKCJONALNOŚCI APLIKACJI	3
TERMINY DOMENOWE	3
3. ARCHITEKTURA ROZWIĄZANIA.....	4
ZASTOSOWANY WZORZEC.....	4
DANE W SYSTEMIE	6
4. MODUŁY SYSTEMU	7
SCRAPER (<i>SCRAPER</i>)	7
SERWIS WYSZUKIWAŃ (<i>SEARCH-SERVICE</i>)	12
APLIKACJA UŻYTKOWNIKA (<i>FRONTEND</i>)	20
5. WDRAŻANIE SYSTEMU	22
SPOSÓB WDRAŻANIA.....	22
WYBÓR TECHNOLOGII	23
6. INSTRUKCJA UŻYTKOWNIKA.....	24
SEARCH	24
TASKS.....	26

1. Dostęp do projektu

Witryna internetowa

Aplikacja jest dostępna pod adresem: <http://offer-rider.jrie.eu/>

Ze względu na możliwości sprzętowe serwera witryna może nie być dostępna przez 100% czasu.

Repozytorium

<https://github.com/jakubriegel/scrapper-search-service>

2. Opis projektu

Problem

Handel w Internecie staje się coraz popularniejszym sposobem robienia zakupów. Dla wielu jest to wręcz domyślny sposób zaopatrywania się w dobra. Tak jak w przypadku tradycyjnych metod kupowania, zakupy przedmiotów różnego typu wymagają odmiennego podejścia klienta. Szczególnym przypadkiem jest rynek używanych samochodów osobowych. W prawdzie w tym obszarze same transakcje zakupu rzadko odbywają się online, to jednak proces wyszukiwania, porównywania i decyzji o zakupie wybranego samochodu odbywa się głównie w Internecie. W Polsce istnieje kilka liczących się platform, służących do sprzedawania i kupowania pojazdów. Z jednej strony daje to dużą swobodę konsumentowi, z drugiej utrudnia mu szybkie znalezienie właściwego ogłoszenia.

Konsument chcący nabyć samochód o określonych parametrach (takich jak zakres roku produkcji czy dodatkowe wyposażenie), dysponujący określonym budżetem z łatwością może wiele ofert, spełniających te kryteria. Aczkolwiek należy tutaj zauważyć, że decyzji o zakupie samochodu *zwykle* nie podejmuje się z marszu. Potencjalny klient może chcieć obserwować rynek jakiś czas, tak aby sprawdzić czy nie pojawiły się nowe (potencjalnie korzystniejsze) oferty. Takie monitorowanie rynku zostałoby znacząco ułatwione przez usługę automatycznie przeszukującą portale w poszukiwaniu nowych ofert spełniających kryteria.

Kolejną przeszkodą w podjęciu dobrej decyzji o zakupie jest fakt, że sprzedający mogą edytować swoje ogłoszenia. Co oznacza, że rzeczy takie jak cena czy warunki finansowania danego pojazdu z biegiem czasu mogą stawać się mniej lub bardziej korzystnie. Monitorowanie takich zmian jest trudne, ponieważ większość portali nie udostępnia publicznie archiwalnych stanów ofert.

Proponowane rozwiązanie

Offer Rider jest produktem, którego celem jest rozwiązanie opisanych wyżej problemów. Jest to aplikacja pozwalająca na śledzenie ofert samochodów odpowiadających na konkretne wyszukiwania w portalu *OTOMOTO*¹. Dostęp do rozwiązania odbywa się poprzez przeglądarkę internetową, zarówno na urządzeniu mobilnym, jak na komputerze stacjonarnym.

Funkcjonalności aplikacji

Aplikacja *Offer Rider* dostarcza użytkownikowi następujących możliwości:

- Definiowanie nowych wyszukiwań,
- Przeglądanie wyników wyszukiwań dla konkretnych dat,
- Podgląd wyników unikalnych dla konkretnej daty i godziny,
- Filtrowanie wyników wyszukiwań,
- Aktywowanie i deaktywowanie tworzenia nowych zleceń dla danego wyszukiwania.

Terminy domenowe

Organizacja pracy nad serwisem wymagała ujednolicenia poniższych terminów:

- Wyszukiwanie (*Search*) – zdefiniowane przez użytkownika kryteria poszukiwanych ofert,
- Zadanie (*Task*) – pojedyncze zlecenie dotyczące wykonania scrapowania dla konkretnego wyszukiwania,
- Wynik (*Result*) – pojedyncza oferta znaleziona dla danego zadania.

¹ <https://www.otomoto.pl/>

3. Architektura rozwiązania

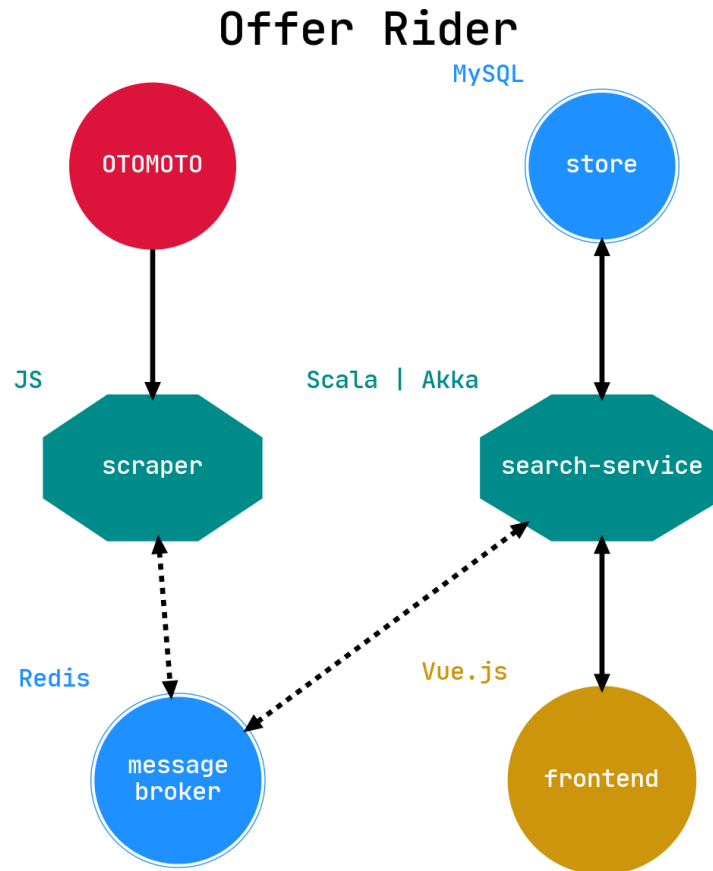
Zastosowany wzorzec

Offer Rider został zaimplementowany zgodnie z architekturą mikrousług. Za taką architekturą przemawiały dwa argumenty. Po pierwsze świat mikrousług jest w stanie dobrze odwzorować przepływ danych w systemie. Po drugie pracę nad tworzeniem takich usług można łatwo wykonywać współbieżnie.

O architekturze *Offer Rider* należy myśleć jako o trzech niezależnych modułach. Są to *scraper*, serwis wyszukiwań i aplikacja frontendowa. Każdy z tych elementów ma inne zadanie. Sprzężenie występuje tylko na poziomie zasilania aplikacji danymi. *Scraper* do działania potrzebuje informacji o zadaniach do wykonania, które są przechowywane w *search-service*. Po wykonaniu *tasków* przesyła on do serwisu znalezione wyniki. Z kolei *frontend* jest interfejsem przez który użytkownik definiuje wyszukiwania oraz ma możliwość przeglądania otrzymanych wyników. *Frontend* wymienia dane tylko z *search-service*. Takie rozwiązanie zapewnia odpowiedni poziom abstrakcji dla każdej z usług. Dzięki temu *scraper* można w ogólności opisać jako algorytm, przyjmujący na wejście dane wyszukiwania i podający na wyjście jego wyniki. Nie musi mieć on świadomości sposobu przechowywania tych danych czy istnienia warstwy użytkownika. Tak samo usługa wyszukiwań nie tylko *nie wie* nic tym jak przebiega *scrapowania*, ten moduł nie musi nawet posiadać informacji o tym, że istnieje *scraper*. Jedyne co powinien robić to przesyłanie informacji o nowym zadaniu i oczekiwanie na wyniki. Podobną regułę można by przedstawić dla komunikacji pomiędzy *frontendem* i *search-service*.

Mikrousługi są też praktyczne z punktu widzenia pracy nad produktem. Z założenia w tej architekturze nie występuje silne sprzężenie pomiędzy usługami. Są one połączone jedynie niezależnymi od implementacji interfejsami komunikacyjnymi (jak np. [REST API](#)). Dzięki temu każda usługa może być implementowana niezależnie, w różnych technologiach i różnym czasie. Jest to czynnik zdecydowanie ułatwiający pracę zespołu. Dobrze wpisuje się to charakterystykę prac wykonywanych przez studentów. Zwykle nie są one wykonywane podczas wspólnych sesji, ale oddzielnie i są synchronizowane na cyklicznych spotkaniach.

Pierwsze tygodnie prac udowodniły, że rozwój systemu idzie sprawnie. Na proces nie wpłynęła negatywnie pandemia *Covid-19*. Pozwoliło to też na szybkie uzyskanie *MVP*², co z kolei ułatwiło wczesne wykrywanie wad i niedomówień w koncepcji systemu.

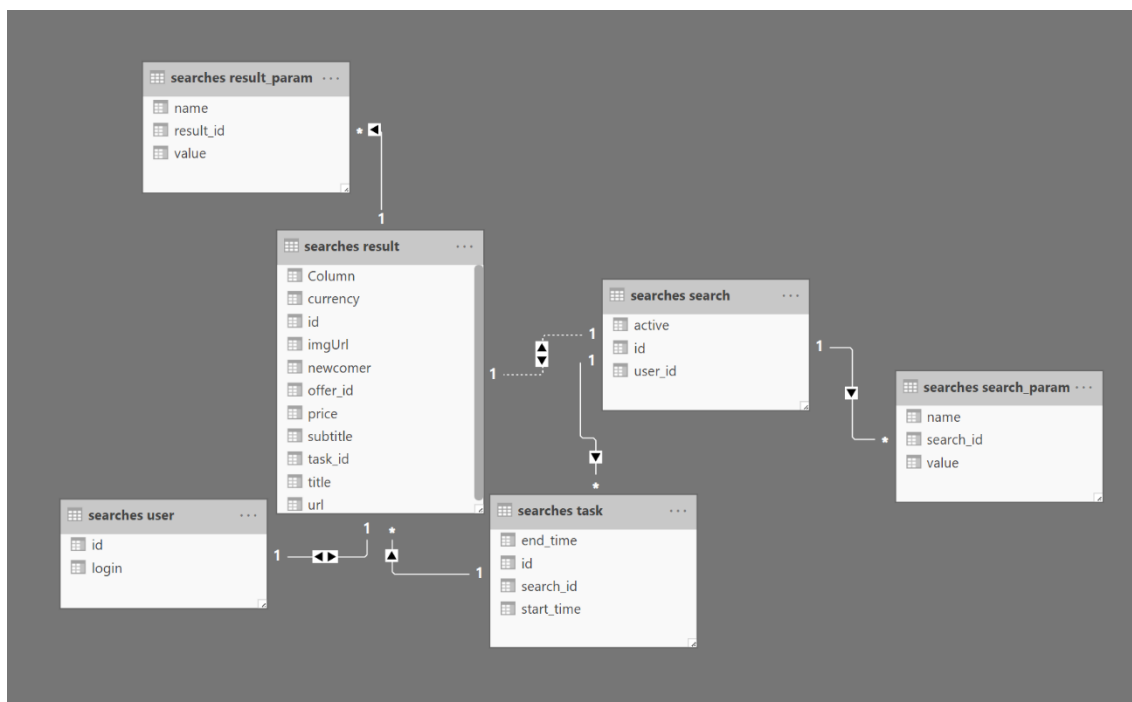


Rysunek 1 Schemat systemu. Kolorem zielonym oznaczono usługi backendowe, niebieskim moduły przechowywania danych, ciemnożółtym interfejs użytkownika, a czerwonym portal OTOMOTO. Linia ciągłą oznaczoną komunikację synchroniczną, a przerywaną asynchroniczną. Schemat pokazuje kierunki przepływu danych w systemie. Definicje wyszukiwań mają swoje źródło na *frontendzie*, skąd poprzez *search-service* trafiają do bazy danych i asynchronicznie są przekazywane skraperowi. Wyniki wyszukiwań pochodzą z *OTOMOTO*, skąd są pobierane przez *scraper*. Następnie są asynchronicznie przekazywane do bazy danych i na życzenie wyświetlane użytkownikowi. Na diagramie każdy element systemu został dodatkowo opisany przez używaną technologię. Dyskusja na temat wyboru technologii ma miejsce w rozdziale 4.

² Ang. *Minimal Value Product*, działający produkt posiadający minimum funkcjonalności do tego, żeby został wykorzystany zgodnie z przeznaczeniem

Dane w systemie

Wszystkie dane systemu są przechowywane w relacyjnej bazie danych podłączonej do usługi *search-service*. Rozważania na temat wyboru konkretnych technologii bazodanowych są opisane w rozdziale 4. Za zastosowaniem bazy relacyjnej przemawiał fakt, że dane w omawianej domenie mają relacyjną naturę. Ponadto skoro wszystkie są zarządzane przez jedną usługę, która dodatkowo na bieżąco wykonuje operacje typu *join*, korzystanie z baz *NoSQL* wprowadziłoby tylko nowe komplikacje i obniżenie wydajności.



Rysunek 2 Schemat bazy danych. Głównymi bytami są reprezentowane przez tabele search, task i result. Dzięki wydzieleniu w domenie encji zadania możliwe jest określenie czasu w jakim oferta z danego wyniku znajdowała się w serwisie oraz porównanie jej z innymi wynikami z tej daty. Tabela task posiada również informacje o czasie początku i zakończenia zadania. Dzięki temu w przypadku awarii można łatwo stwierdzić, kiedy proces wyszukiwania zawiódł oraz czy problem leży po stronie którejś z usług czy może uszkodzone jest medium komunikacji.

Pewnym wyzwaniem było dostosowanie bazy do zwinnej pracy nad rozwojem projektu oraz nad założeniem, że parametry wyników mogą być różne dla różnych krotek i że w trakcie życia platformy ma być możliwe dodawanie nowych typów parametrów. Teoretycznie oznacza to konieczność łatwej zmiany modelu, co jest

mocną stroną baz NoSQL³. Innym rozwiązaniem jest skorzystanie z bazy, która posiada słownikowy⁴ typ pola. Tradycyjnym podejściem wykorzystywanym w bazach relacyjnych byłoby stworzenie kolumny dla każdego typu parametru. To niestety spowodowałoby konieczność utrzymywania dużej ilości kolumn, w których w wielu krotkach znajdowałyby się *null*. A każda zmiana w ilości przechwytywanych parametrów skutkowałaby ciężką dla bazy operacją *ALTER* i koniecznością zmiany implementacji w *search-service*. Ostatecznie wypracowane rozwiązanie możliwe do realizacji w tradycyjnej bazie relacyjnej, a jednocześnie na tyle elastyczne, żeby zmiany i braki parametrów nie powodowały konieczności zmian w bazie czy kodzie aplikacji. Parametry ofert, które nie identyfikują ich oraz nie zawsze są podawane w każdym wyniku są przechowywane w oddzielnej tabeli *results_param*. Każda krotka tej tabeli składa się z *id* wyniku, nazwy parametru i jego wartości zapisanej jako tekst. Tworzy to relację jeden do wielu z tabelą wyników. Zalety takiego to rozwiązania to przede wszystkim (1) brak konieczności rozróżniania konkretnych parametrów w implementacji *search-service* ani *frontendu*. (2) Przechowywanie parametrów nie jest wrażliwe na ich typ. (3) Schemat bazy danych nie musi uwzględniać zmian w parametrach zbieranych przez *scraper*.

4. Moduły systemu

Scraper (*scraper*)

Podstawowe informacje

Przeznaczenie: scrapowanie ofert ze strony otomoto.pl, publikowanie wyników na kolejce w Redis

Główne technologie: JavaScript, Node.js, Cherio.js, Redis

Autor: Wojciech Kasperski

Opis działania

Scraper jest główną funkcjonalnością opracowywanego serwisu. Odpowiada za pobieranie danych ofertowych z serwisu OTOMOTO i przekazywanie ich za pomocą

³ Np. w *MongoDB* możliwe jest, aby każdy dokument w kolekcji składał się z innych pól

⁴ Słownik w niektórych implementacjach jest nazywana *mapą* lub strukturą *klucz-wartość*

mechanizmu PubSub⁵ do magazynu danych jakim jest Redis⁶. Natomiast same zlecenia wykonania scrapowania, w których znajdują się parametry jakie są brane pod uwagę podczas scrapowania również przekazywane są przez Redisa, w tej samej postaci, ale w innym strumieniu. Na początku zlecenia scrapowania przesyłane są na kolejkę pod kluczem *pt-scrapers-search-tasks*, a wyniki publikowane są pod kluczem *pt-scrapers-results*. Dodatkowo cały mechanizm działa nieblokująco, co oznacza, że można wykonywać kilkanaście zadań scrapowania jednocześnie. Całość konfiguracji scrapera jest przechowywana w folderze *config*, skąd można skonfigurować połączenie z Redisem oraz sposób działania scrapera.

Wybrane technologie i motywacja

Decydując się na technologie jakie wybrano do budowy scrapera, zwracano uwagę przede wszystkim na łatwość czytania danych ze strony oraz przechodzenia między kolejnymi stronami danej listy ofertowej. Pierwotnym założeniem dla scrapera była uniwersalność i możliwość prostego dostosowania go do innych serwisów, niż OTOMOTO. Jednocześnie odrzucono pomysł pisania go w oparciu o znane już, z serwisu GitHub, implementacje scrapersów tego serwisu w języku *Python*. Dlatego postawiono na dość szybką i łatwą w modyfikacji implementację w języku JavaScript na serwerze Node.js.

Wybrane aspekty implementacji

Na początku rozpoczęto od implementacji prostego scrapowania jednej oferty i poszczególnych jej elementów. Okazuje się, że implementacja serwerowa biblioteki *jQuery*, pod nazwą *Cherio.js* świetnie nadaje się do tego typu celów. Wszystkie oferty z danej strony pod postacią HTMLa zostały wyciągnięte do tablicy za pomocą selektorów wyznaczających znaczniki HTMLowe *<article>*. Następnie przetworzono pojedynczo zawartość każdego z tych znaczników, w których znajdowała się oferta. Wyciąganie parametrów (np. liczby przejechanych mil) następowało poprzez zdefiniowanie selektora w następujący sposób:

⁵ <https://redis.io/topics/pubsub>

⁶ <https://redis.io/>


```
const mileage = $(el).find('li[data-code=mileage] span').text();
```

Blok kodu 1 Selektor odpowiedzialny za pobieranie danych określających ilość przejechanych przez samochód mil.

Powyższy blok kodu odpowiada za wyciągnięcie danych z listy, której element posiada dodatkowy atrybut *data-code*. Z tak przygotowanych danych oferty nastąpiło budowanie jej modelu i następnie przekazywanie, go na kolejkę *PubSub* w magazynie danych jakim jest Redis.

Model oferty w postaci obiektu JavaScript:

```
const metadata = {
  result: {
    taskId,
    offerId,
    title,
    subtitle,
    price: parseInt(price),
    currency,
    url,
    imgUrl: image,
    params: {
      year: parseFloat(year),
      mileage: mileage,
      engine_capacity,
      fuel_type,
      town,
      region
    }
  },
  last
};
```

Blok kodu 2 Model oferty w postaci obiektu języka JavaScript.

Powyższy model to gotowy obiekt JSON, natomiast każde jego pole np. *taskId*, czy *imgUrl*, to zmienna z przechowywaną zescrapowaną wartością (przykład wcześniej z wyciąganiem danych).

Samo przechodzenie między stronami polega na rekurencyjnym wywołaniu funkcji scrapującej stronę z nowym *URL*, który można uzyskać za pomocą Cherio.js, scrapując go ze znacznika listingu tzw. *Next Page*, gdy owy znacznik nie zostanie odnaleziony, oznacza to, że program dotarł do ostatniej strony i dane zadanie zostaje zakończone, a do ostatniego wyniku zostaje przekazana flaga z informacją o zakończeniu scrapowania.

Napotkane problemy

Heap out of memory przy zbyt dużej ilości jednocześnie wykonywanych zadań

Jednym z problemów jakie napotkano podczas testowania implementacji samego scrapera była przeciążająca się sverta, na której kończyła się pamięć przy jednoczesnym scrapowaniu równolegle 15 zadań, co było spowodowane nieprawidłowym wykorzystywaniem funkcji asynchronicznych języka Java Script. Można to rozważyć na przykładzie funkcji odbierającej zlecenia wykonania zadania przedstawionej poniżej:

```
subscriber.on("message", async (_, job) => {
  const url = buildUrlFromParams(taskData.params);
  const parsedResults = [];

  if (appConfig.isSavingData) {
    await getWebsiteContent(url, taskData.taskId, publisher, parsedResults);

    await exportResults(parsedResults);
  } else getWebsiteContent(url, taskData.taskId, publisher, parsedResults);
});
```

Blok kodu 3 Funkcja odpowiedzialna, za nasłuchiwanie na dane PubSub Redis'a oraz wywołująca funkcję odpowiedzialną za ekstrakcję danych ze stron listy.

W powyższej funkcji subscriber.on jest ustawiane nasłuchiwanie na wiadomość z kolejki PubSub. Następnie z przekazanych w zleceniu (lub inaczej *Tasku*) parametrów jest budowany link do żądanej strony listy ofert samochodowych i przekazywany do asynchronicznej funkcji *getWebsiteContent* w zależności od tego, czy dane mają zostać zapisane lub nie. W *Bloku kodu 3* można dostrzec dwa takie wywołania funkcji. Skąd one się biorą i czym się różnią? Pierwsze z nich dzieje się w przypadku chęci zapisania dodatkowo zscrapowanych danych do pliku z

rozszerzeniem `.json`, co jest przydane np. w debugowaniu. Dodatkowo wywołanie tej funkcji posiada przedrostek `await`⁷ co oznacza, że oczekuje się tutaj na wykonanie do końca tej funkcji, zanim zostanie wywołana następna linia kodu z eksportem wyników do pliku. Z kolei w normalnym trybie działania, gdzie tylko przekazujemy dane do kolejki na Redisie przedrostek `await` nie jest dodawany.

W czym w takim razie tkwi konkretnie problem i czym różnią się te dwa wywołania?

Funkcja `getWebsiteContent`, jest funkcją rekurencyjną, a to oznacza, że jeśli zostanie dodany tam przedrostek `await`, to oczekuje się również na wykonanie jej wszystkich zagnieżdżonych wywołań. W praktyce oznacza to tyle, że wszystkie obiekty z danymi ofert trzymane są w pamięci na sterckie do zakończenia scrapowania danej oferty. Gdy jednocześnie odbiera się przykładowo 10 zadań, z których każde ma hipotetycznie 500 ofert, w pamięci jest przechowywanych wówczas 5000 obiektów, które nie zostaną usunięte do momentu ich eksportu do pliku. Wobec czego następuje przeciążenie sterty, która w zależności od systemu operacyjnego na którym wykonywany jest kod – 32bit/64bit – ma odpowiednio 700MB/1400MB, po czym aplikacja rzuca wyjątek i przestaje działać.

Jak więc tego uniknąć?

Rozwiązaniem problemu okazało się usunięcie przedrostka `await`, wtedy każde następne wywołanie rekurencyjne funkcji `getWebsiteContent` odczepiało ją od głównego wątku i czyściło pamięć po każdym zakończonym jej wywołaniu. Dodatkowo warto też wiedzieć, że jeśli istnieje funkcja asynchroniczna i dodany zostaje do niej przedrostek `await`, to takie wywołanie tej funkcji będzie działać w sposób synchroniczny. Sam ten przedrostek został dodany do języka JavaScript na potrzeby wywoływania funkcji asynchronicznych synchronicznie, co jest przydatne przykładowo w testach jednostkowych, czy przy stosowaniu zasady DRY (*Don't-Repeat-Yourself*) podczas wytwarzania oprogramowania.

⁷ https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Polecenia/funkcja_async

Serwis wyszukiwań (*search-service*)

Podstawowe informacje

Przeznaczenie: zarządzanie wyszukiwaniami, zlecenie scrapowania oraz przechowywanie danych

Główne technologie: Scala, Akka, MySQL, Redis

Autor: Jakub Riegel

Opis działania

Search-service jest sercem systemu, odpowiada za zarządzanie i kontrolę przepływu danych. Dzięki asynchronicznemu API komunikują się ze scraperem, wysyłając mu zlecenia wyszukiwania (w domenie *Tasks*) oraz odbierając ich wyniki (w domenie *Results*). Serwis ten udostępnia również *REST API*, z którego korzysta aplikacja użytkownika. Poprzez *REST* możliwe jest stworzenie wyszukiwania (w domenie *Search*), zarządzanie nim oraz podgląd wyników z poszczególnych zleceń.

Wybrane technologie i motywacja

Usługa jest zaimplementowana w języku *Scala*⁸ z użyciem narzędzia *akka*⁹. Dane są przechowywane w bazie *MySQL*¹⁰, dostęp do danych zapewnia biblioteka *Slick*¹¹ wraz z jej modułem *Alpakka*¹². Do stworzenia *REST API* posłużyła biblioteka *akka http*¹³, która jest rozszerzeniem *akka*. Natomiast komunikacja asynchroniczna odbywa się na platformie *Redis*¹⁴, z którym aplikacja łączy się dzięki *scala-redis*¹⁵.

Wybór języka implementacji był kluczowym elementem, który miał zapewnić wydajną i wygodną pracę nad aplikacją oraz być dobrym środkiem wyrazu dla proponowanej architektury. Odpowiedzialność *search-service* jest dosyć duża i w produkcyjnym środowisku prawdopodobnie byłaby oddana za pomocą 2-3 mikro usług. W tej implementacji ze względu na to charakter projektu studenckiego,

⁸ <https://www.scala-lang.org/>

⁹ <https://akka.io/>

¹⁰ <https://www.mysql.com/>

¹¹ <https://scala-slick.org/>

¹² <https://doc.akka.io/docs/alpakka/current/slick.html>

¹³ <https://doc.akka.io/docs/akka-http/current/index.html>

¹⁴ <https://redis.io/>

¹⁵ <https://github.com/acrosa/scala-redis>

dostępność zasobów ludzkich oraz ramy czasowe zdecydowano się na stworzenie pojedynczej usługi do zarządzania wyszukiwaniami (tzw. monolitu¹⁶). Oznacza to, że potrzebny był język mogący łatwo odwzorować działanie wielu częściowo niezależnych modułów. Sugeruje to zastosowanie współbieżności. Ponadto ze względu na to, że usługa miała działać w sposób ciągły, należało zapewnić jej utrzymywanie wydajności podczas ciągłego uruchomienia. Wybór padł na język *Scala*. *Scala* jest językiem funkcyjnym posiadającym wszystkie cechy klasycznych języków obiektowych. Ponadto posiada potężny i jednocześnie prosty w obsłudze silnik współbieżności. Jest on oparty o obiekty typu *Future*, leniwe (ang. *lazy*) właściwości oraz o oczekiwanie na wyniki obliczeń. Efektywne wykorzystanie tych możliwości do stworzenia modułów aplikacji umożliwia *akka*. Narzędzie to zostanie opisane później w tym podrozdziale. Język *Scala* jest kompilowany do kodu wykonywalnego *Java Virtual Machine*. Oznacza to, że można go uruchomić na każdej platformie. Bardzo ważną cechą *JVM* jest to, że świetnie radzi sobie z długotrwałymi uruchomieniami. Dzieje to dzięki wielu poziomom optymalizacji kompilacji kodu oraz podziałowi pamięci dynamicznej programu. Dodatkową motywacją do wyboru języka funkcyjnego była rosnąca popularność tego typu programowania, a więc była to dobra okazja do zwiększenia swoich umiejętności w tym zakresie.

Do budowania projektu wykorzystywane jest narzędzie *sbt*¹⁷. Pozwala ono na zarządzanie zależnościami oraz dzięki wtyczce *assembly*¹⁸ na budowanie pliku *jar* z wbudowanymi zależnościami (tzw. *fat jar*). Dzięki *sbt* kompilacja i uruchomienie aplikacji stało się szybkie, proste i powtarzalne na każdym urządzeniu. Ponadto *sbt* dobrze współpracuje z *Intelij IDEA*¹⁹, czyli *IDE* wykorzystywanego do pracy nad projektem.

¹⁶ Przykładowa charakterystyka architektury monolitycznej: <https://docs.microsoft.com/pl-pl/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#what-is-a-monolithic-application>

¹⁷ <https://www.scala-sbt.org/>

¹⁸ <https://github.com/sbt/sbt-assembly>

¹⁹ <https://www.jetbrains.com/idea/>

Jako wzorzec dobrze spełniający potrzeby implementowanej aplikacji wybrano model aktorów. Jest klasyczny to model współbieżności, w którym operując na wysokim poziomie abstrakcji można tworzyć wydajne i skalowalne wielomodułowe aplikacje. Uznany nie tylko w świecie *JVM* narzędziem pomagającym na tworzenie takich rozwiązań jest *akka*. Środowisko to umożliwia implementację wzorca aktorów, wykorzystując zarówno funkcyjne jak i obiektowe właściwości tego języka. W dodatku jest dobrze udokumentowane i posiada rozległą społeczność, dzięki czemu jest dobrym kompanem w pracy dewelopera.

Usługa wyszukiwań odpowiada za przechowywanie wszystkich danych w systemie *offer-rider*. Dane te mają charakter relacyjny, ich struktura została opisana w rozdziale 3. Charakter danych oraz projektu oznacza, że była potrzebna baza relacyjna, która jest prosta w użyciu, wydajna i posiada dobre wsparcie. Wybór padł na *MySQL*, ponieważ spełnia powyższe wymagania a jej implementacja *SQL* spełnia klasyczne założenia tego języka.

Dostęp do bazy *MySQL* w językach *JVM* klasycznie realizuje się poprzez *JDBC*²⁰. Jako że *JDBC* zapewnia niskopoziomowe *API* zdecydowano się na zastosowanie pośredniczącej biblioteki *Slick*. Biblioteka ta jest zaimplementowana z myślą o rozwoju oprogramowania w *Scali*. Ponadto w zbiorze narzędzi *Alpakka* znajduje się jej rozszerzenie wspierające *akka streams*²¹. Dzięki temu w warstwie dostępu do bazy danych (klasy **Repository*) możliwe było zachowanie konwencji języka *Scala* i środowiska *akka*.

Serwer *http* potrzebny do realizacji *REST API* musiał spełniać dwa założenia: (1) nie wносить dodatkowego narzutu związanego z wydajnością aplikacji oraz (2) być na poziomie implementacji łatwo integrowalny z resztą projektu. W tej sytuacji naturalnym wyborem była *akka http*. Jest to narzędzie do tworzenia serwerów *http* będące rozszerzeniem *akka*. Dzięki temu zachowuje cechy zarówno modelu

²⁰ <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

²¹ <https://doc.akka.io/docs/akka/current/stream/index.html>

aktorów jak i klasycznych metod tworzenia punktów końcowych (ang. *endpoints*) w serwerze *http*.

Asynchroniczne API *search-service* w założeniu realizuje model producent-konsument w dwóch kierunkach. W pierwszym serwer publikuje *Taski*, które konsumuje *scraper*. Następnie w drugim kierunku *scraper* produkuje *Results*, które konsumuje serwer. Dodatkową założeniem jest to, że rozwiązanie nie może narzucać sposobu ani języka implementacji po żadnej ze stron. Na rynku istnieje wiele dedykowanych rozwiązań pozwalających na realizację tego konceptu, m. in. *Kafka*, *RabbitMQ*, *ZeroMQ* czy *Flink*. Jednak w projekcie zdecydowano się na implementację najbardziej banalną – z wykorzystaniem platformy *Redis* i rozwiązania *pub/sub*²². Jest ono proste do implementacji, uruchomienia i utrzymania. Ponadto wiedza potrzebna to uruchomienia *Redisa* była zawarta w programie przedmiotu WTI²³. *Pub/sub* przekazuje wysłane wiadomości do wszystkich podłączonych klientów, jednak nie przechowuje wiadomości w przypadku, gdy żaden klient nie jest podłączony. Z jednej strony jest to pewne ograniczenie (np. nie jest możliwa retransmisja wiadomości), ale z drugiej strony posiadane przez nas ograniczone zasoby są chronione przed nadmiernym zużyciem pamięci w przypadku awarii jednej z usług.

Jako klienta *Redisa* w *search-service* wykorzystano projekt *scala-redis*. Przemawiało za nim proste, niskopoziomowe API. Dzięki temu biblioteka nie jest dużym narzutem w trakcie wykonywania aplikacji, a do jej zastosowania wystarczy podstawowa wiedza z zakresu obsługi *Redisa*.

Wybrane aspekty implementacji

Usługa

Sam serwis jest zaimplementowany jako klasyczna aplikacja *JVM* z domenowym podziałem na pakiety i zależnościami pobieranymi z publicznych repozytoriów *maven*²⁴. Główne aktory systemu to *SearchTaskCreator* – odpowiadający za

²² <https://redis.io/topics/pubsub>

²³ Wybrane Technologie Internetowe, semestr 6, prowadzący dr Andrzej Szwaab

²⁴ <https://mvnrepository.com/>

cykliczne tworzenie nowych zleceń, *Subscriber* – nasłuchujący nowych wyników wyszukiwań oraz *RestApi* – tworzący punkty końcowe aplikacji. Wymienione aktory komunikują się z aktorami niższego rzędu. Są to m. in. repozytoria, odpowiadające za operacje bazodanowe wybranych modeli.

REST API

Restowe API serwisu ma na celu zapewnienie aplikacji frontendowej dostępu do metod pozwalających na przeglądanie oraz zarządzanie wyszukiwaniami oraz wynikami. Implementacja zakłada wykorzystywanie metod oraz statusów *http* zgodnie z normami *RFC*. Z powodu, że omawiane *API* nie jest zasobem publicznym nie znaleziono uzasadnienia dla zastosowania w nim założeń *HATEOAS*.

Dostępne punkty końcowe przedstawia lista poniżej. Szczegóły zasobów oraz środowisko testowa można uzyskać uruchamiając kolekcję postamanową²⁵, znajdującą się na repozytorium pod adresem `/search-service/docs/craper-search-service.postman_collection.json`

- POST `/search`
- GET `/search?userId=<userId>`
- PUT `/search/<searchId>/deactivate`
- PUT `/search/<searchId>/activate`
- GET `/tasks?userId=<userId>&searchId=<searchId>`
- GET `/results?userId=<userId>&searchId=<searchId>&taskId=<taskId>`

Asynchroniczne API

API asynchroniczne umożliwia sprawną komunikację *search-service* i *scraper*. Jest ono oparte o dwa kanały: pierwszy na którym serwer publikuje eventy o nowych *taskach* i drugi na którym *scraper* publikuje wyniki wyszukiwań.

²⁵ Postman – narzędzie do rozwoju i testowania REST API: <https://www.postman.com/>


```
{
  "taskId": "uuid, the id assigned to this task",
  "params": {
    "key1": "textValue",
    "key2": "numberValueAsText"
  }
}
```

Blok kodu 4 Event o nowym Tasku wyszukiwania. Zawiera on identyfikator zlecenia oraz listę parametrów. Identyfikator jest losowo wygenerowanym UUID²⁶, który jednoznacznie wskazuje na konkretne zlecenie i jest dołączany do eventu o nowym wyniku. Lista parametrów jest przekazywana jako słownik String -> String, dzięki czemu jest odporna na zmiany w ilości przesyłanych parametrów.

```
{
  "result": {
    "taskId": "uuid, the id of task this result belongs to",
    "title": "string",
    "subtitle": "string?",
    "url": "string?",
    "imgUrl": "string?",
    "offerId": "string?",
    "price": "string?",
    "currency": "string?",
    "params": {
      "key1": "textValue",
      "key2": "numberValueAsText"
    }
  }
  "last": "bool, true if no further results will be emitted"
}
```

Blok kodu 5 Event o nowym wyniku wyszukiwania. Zawiera on identyfikator zlecenia, dane wyniku oraz flagę last.

Napotkane problemy

Implementacja dostępu do bazy danych

Ze względu na to, że świat języka *Scala* dzieli się na dwie części: (1) programowanie w *Scali* i (2) programowanie w *akka*, często występuje pewne rozdwojenie w dokumentacji oraz wsparcia ze strony społeczności. Dedykowany dla *akka* sposób wysyłania zapytań do bazy danych poprzez bibliotekę *Slick* zakłada podanie komendy SQL i stworzenie na jej podstawie strumienia.

²⁶ <https://tools.ietf.org/html/rfc4122>

```
Slick.source {  
  sql"SELECT * FROM result_param"  
}
```

Blok kodu 6 Zapytanie SQL z wykorzystaniem Alpakka Slick

Jednak *Slick* standardowo posiada zupełnie innym mechanizm, wykorzystujący dziedziczenie klas i przeciążanie operatorów. Niestety spowodowało to niespójność w implementacji i redundantny kod. W miarę coraz lepszego opanowywania tej biblioteki kod był poprawiany. Na chwilę obecną jedyne miejsce, w którym wykorzystywane jest podejście bez użycia strumieni to operacje polegające na zapisie do danych do bazy i odczytaniu *id* zapisanych krotek – nie udało się tego zmigrować.

Zmiany były wprowadzane iteracyjnie wraz z innymi pracami.

Wydajność zapisu wyników do bazy danych

Po wykonaniu testów dla dużego obciążenia systemu okazało się, że szybkość zapisu nowych wyników do bazy danych jest dużo wolniejsza od tempa scrapowania. System działał stabilnie dzięki temu, że niezapisane jeszcze wyniki zostały zakolejkowane w skrzynce odbiorczej²⁷ aktora zajmującego się zapisem. Jednak powodowało długi czas oczekiwania na pojawienie się wyników z zakończonego *Taska*.

Wąskim gardłem okazała się funkcjonalność oznaczająca flagą *newcomer* oferty, które pierwszy raz pojawiły się w wyniku wyszukiwania. Dla każdego dodanego wyniku wykonywane było skomplikowane zapytanie *SQL*. Znacząco obciążało to bazę danych. Problem rozwiązano poprzez zastosowanie *cache*. Wyniki omawianego zapytania są przechowywane przez 30min na *Redisie*. Pozwana to zachowanie dotychczasowej funkcjonalności i jednocześnie zwiększenie wydajności systemu.

²⁷ <https://doc.akka.io/docs/akka/current/typed/mailboxes.html>

```
SELECT offer_id FROM result
WHERE task_id = (
  SELECT id FROM task
  WHERE search_id = (
    SELECT search_id FROM task WHERE id = ${result.taskId}
  )
  AND id != ${result.taskId}
ORDER BY end_time DESC LIMIT 1
);
```

Blok kodu 7 Polecenie SQL wykonywane podczas każdego zapisu nowego wyniku

Warte dodania jest też pochodzenie linii *AND id != \${result.taskId}*. Ze względu na to, że za zamykanie zleceń odpowiedzialny jest inny aktor, zdarzało się, że zlecenie zostawało zamknięte zanim został zapisany ostatni wynik. Powodowało to, że powyższa komenda brała pod uwagę bieżące zlecenie i wszystkie wyniki miały ustawianą flagę *newcomer*.

Mimo takiego usprawnienia zapis wyników nadal był wąskim gardłem. Z tego powodu zdecydowano się na współbieżny zapis do bazy zrealizowany za pomocą *Routers*²⁸. Zamiast jednego aktora zapisującego dane do bazy powstaje pula dziesięciu, z której metodą *round robin* wybierany jest ten, który wykona akcję.

Pull request: <https://github.com/jakubriegel/scrapper-search-service/pull/34>

Wydajność zapisu wyników do bazy danych 2

Niestety po kilku dniach od wdrożenia opisanej wyżej zmiany, okazało się, że nadal występują problemy wydajnościowe. Tym razem wąskim gardłem była sieć dockera połączona z niską wydajnością *Raspberry Pi*. Duża ilość zapytań do *Redisa* powodowała uszkodzanie się pakietów.

Rozwiązanie było bardzo proste – zmniejszenie ilości zapytań do *Redisa*. Remedium okazało się wykorzystanie *akka streams*. Po zmianach przychodzące od *scrapera* wyniki wyszukiwań grupowane są w bloki (ang. *chunks*). Następnie do każdego bloku dodawana jest aktualna lista poprzednich id pobrana z *Redisa*. W następnych krokach dane przetwarzane są bez zmian w porównaniu z poprzednim

²⁸ <https://doc.akka.io/docs/akka/current/typed/routers.html>

podejściem. Dzięki temu udało się nie tylko rozwiązać usterkę, ale też znacząco podnieść wydajność systemu.

Kodowanie tekstu w bazie danych

W trakcie testowania całości systemu okazało się, że polskie znaki w tekstach zapisywanych w bazie danych były zastąpione znakiem ?. Naprawą błędy było dodanie stosownej opcji do parametrów połączenia z bazą: `characterEncoding=utf8`

Pull request: <https://github.com/jakubriegel/scrapper-search-service/pull/31>

Aplikacja użytkownika (*frontend*)

Podstawowe informacje

Przeznaczenie: umożliwienie użytkownikowi korzystania z aplikacji

Główne technologie: JavaScript, Vue.js, Vuetify

Autor: Maciej Stosik

Opis działania

Frontend odpowiada za wyświetlanie informacji użytkownikowi oraz umożliwienie kontroli i tworzenia zapytań dla *search-service* za pomocą REST API. Możliwe jest utworzenie nowego wyszukiwania, sprawdzenie aktualnych wyszukiwani i ich wyników.

Wybrane technologie i motywacja

Aplikacja stworzona we *Vue*²⁹, które jest frameworkiem do *JavaScript*. Do zarządzania bibliotekami wykorzystano *npm*³⁰. REST API konsumowane jest przez *axios*³¹, a za zarządzanie formatowaniem daty i godziny odpowiada *moment.js*³². Za cały wygląd odpowiada *Vuetify*³³, które jest biblioteką komponentów zgodnych z *Material Design*³⁴.

²⁹ <https://vuejs.org/>

³⁰ <https://www.npmjs.com/>

³¹ <https://github.com/axios/axios>

³² <https://momentjs.com/>

³³ <https://vuetifyjs.com/>

³⁴ <https://material.io/>

Vue zostało wybrane ze względu na rosnącą popularność tego języka, szybkość z jaką można stworzyć projekt oraz przez chęć udoskonalenia swoich umiejętności. Również architektura *Vue* jest bardzo dobrze przystosowana do pisania wszelakich projektów internetowych, poprzez wykorzystanie wzorca Model–view–viewmodel. Każdy komponent składa się z trzech części: kodu HTML, kodu *Vue*, oraz wyglądu w CSS lub jakimś preprocesorze stylu. Umożliwiło to rozdzielenie aplikacji na części, które mogą być łatwo zmienione w razie potrzeby.

Ponieważ *Vue* jest frameworkiem opartym na *JavaScript* wymusza to korzystanie z *npm* do tworzenia projektu, zarządzania zależnościami, budowania i uruchamiania aplikacji. W połączeniu z *Webstorm*³⁵ – IDE do *JavaScript* od firmy *JetBrains*, które zostało użyte do napisania aplikacji, pozwala za pomocą *npm* na uruchomienie lokalnego serwera zintegrowanego z IDE i natychmiastowo reagującego na zmiany w kodzie.

Wybór biblioteki komponentów podyktowany był spójnym wyglądem zgodnym z *Material Design*, jak również bardzo dobrze stworzoną dokumentacją. *Vuetify* pozwala na szybkie prototypowanie wyglądu i późniejsze jego dopracowanie. Rozwiązany jest również problem responsywności, ponieważ stosowanie się do *Material Design* z góry zakłada jednolity wygląd w każdej możliwej konfiguracji urządzeń.

Komunikacja z *search-service* zrealizowana jest za pomocą *axios* – biblioteki opartej na *obietnicach*³⁶. W prosty sposób umożliwia na wysłanie zapytania na dany *endpoint* i uzyskania odpowiedzi w sposób asynchroniczny. Równie proste jest wyświetlenie paska ładowania w przypadku większego zapytania korzystając z funkcji `then()`, która zostanie wywołania po spełnieniu *obietnicy*.

W celu ułatwienia pisania i trzymania jakości kodu użyto *ESLint* i *Prettier*. Narzędzia te analizują i w miarę możliwości poprawiają błędy oraz styl od razu po zapisie pliku, co pozwala na szybką poprawę pozostałych nieprawidłowości.

³⁵ <https://www.jetbrains.com/webstorm/>

³⁶ Ang. *promise* – proxy dla wartości, której wartość niekoniecznie jest jeszcze znana

Wybrane aspekty implementacji

Podział na komponenty

W celu wyświetlenia wyników skorzystano z komponentu `<v-data-table>` od *Vuetify*, który pozwala na wyświetlenie danych w tabeli, jak również na definiowanie własnych pól w przypadku np. flagi oznaczającej nowe oferty, albo rozwijanego pola zawierającego kolejny komponent – w tym przypadku ze szczegółami.

```
<template v-slot:expanded-item="{ headers, item }">
  <td :colspan="headers.length">
    <search-details :item="item" />
  </td>
</template>
```

Blok kodu 8 Fragment tabeli z wynikami odpowiadający za dodanie rozwijanej tabeli ze szczegółami

Autorski `<search-details>` przyjmuje obiekt `item`, który odpowiada jednemu wierszowi tabeli, jako parametr dla pola wewnątrz, również nazwanego `item`. Kod wewnątrz może na nim operować i zająć się jego wyświetlaniem, co pozwoli na pozostawienie komponentu-rodzica przejrzystym.

Napotkane problemy

Częste zapytania do portalu OTOMOTO

W celu uniknięcia odpytywania serwisu *OTOMOTO* za każdym razem jak odświeżona jest strona podjęto decyzję o zmockowaniu danych wypełniających rozwijane listy marek i modeli samochodów. Plik z wartościami zapisano na stałe w projekcie, usuwając przy tym zbędne dane takie jak informacje o lokalizacji.

5. Wdrażanie systemu

Sposób wdrażania

Do wdrożenia systemu na dowolną platformę sprzętową wymagany jest Docker oraz procesor ze wsparciem dla funkcji wirtualizacji. W kontekście architektury sprzętowej również jest tu pewna dowolność, gdyż system zadziała i wdroży się poprawnie dla rodzin procesorów x86, x64 oraz wszystkich procesorów ARM ze wsparciem dla platformy Docker.

System można uruchomić korzystając z terminala, który ma dostęp do demona Dockerowego, poprzez wpisanie komendy `docker-compose up` w katalogu głównym aplikacji. Wówczas na podstawie obrazów z bazy Docker Hub, budowane są środowiska dla poszczególnych części systemu, a następnie wszystkie usługi uruchamiają się w kolejności nadanej przez zależności wpisane w plik `docker-compose.yml` w głównym katalogu repozytorium. Jest to jednocześnie plik z definicją działania całego systemu. Natomiast definicje obrazów do budowania środowiska wykonawczego poszczególnych aplikacji będących częścią systemu znajdują się odpowiednio w katalogach każdej z nich pod postacią tzw. plików *Dockerfile*.

Sposób uruchomienia został również opisany w README aplikacji: <https://github.com/jakubriegel/scrapper-search-service/blob/master/readme.md>

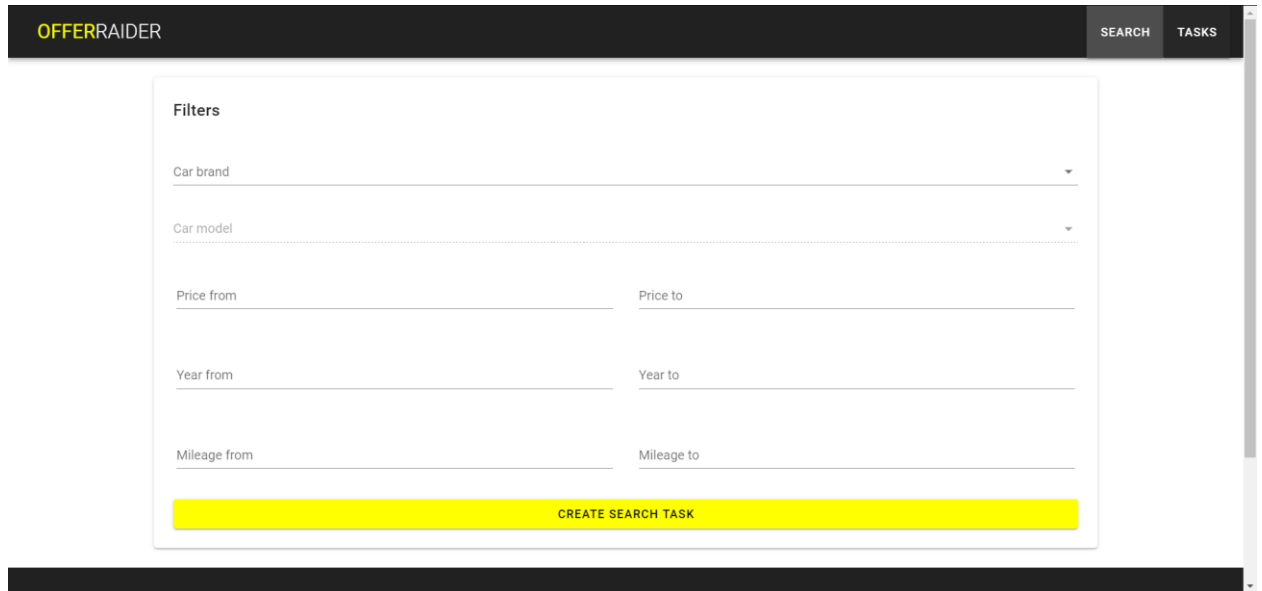
Wybór technologii

Dlaczego Docker jako platforma uruchomieniowa? Przede wszystkim ze względu na prostotę zdefiniowania całego serwisu za pomocą jednego pliku, łatwość konfiguracji pojedynczych usług, bogaty zbiór obrazów, jak i łatwość konfiguracji dla wielu platform sprzętowych oraz systemowych. Alternatywą był również Vagrant, jednak ze względu na to iż jako środowisko testowe podczas rozwijania aplikacji zostało wykorzystane Raspberry Pi, Vagrant w porównaniu do Dockera wydawał się dużo bardziej wymagającą sprzętowo platformą, stąd wybór był w tym wypadku dość oczywisty. Oczywiście obie platformy były rozważane ze względu na możliwości izolacji aplikacji o środowiska systemu operacyjnego maszyny. Sam Docker zapewnia też ruch tylko między zdefiniowanymi połączeniami w swojej sieci wewnętrznej co pozwala w prosty sposób zadbać o aspekty związane z bezpieczeństwem.

6. Instrukcja użytkownika

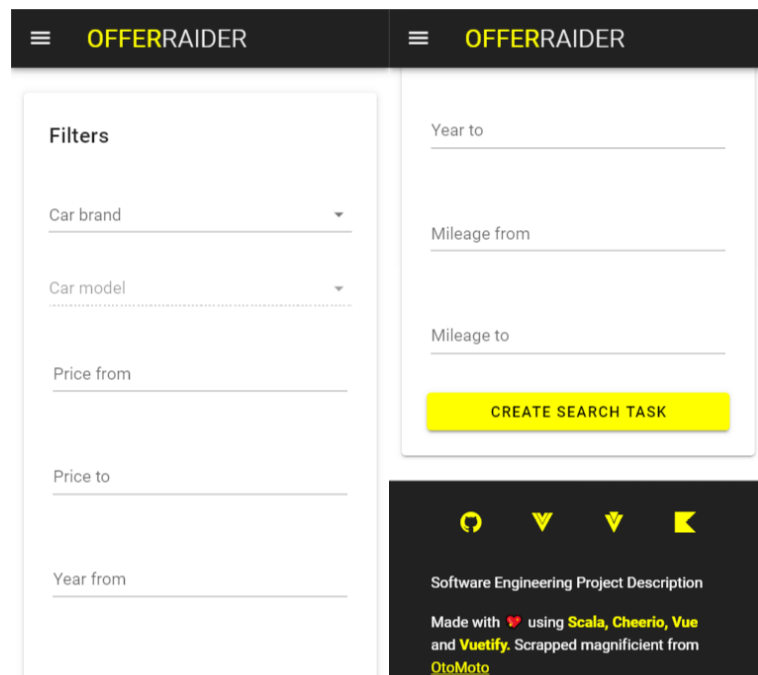
Search

Zakładka *Search* umożliwia definiowanie wyszukiwań, które będą wyświetlane w zakładce *Tasks*.



The screenshot shows the desktop version of the OFFERRAIDER application. At the top, there is a dark header bar with the text "OFFERRAIDER" in yellow on the left and two tabs, "SEARCH" and "TASKS", on the right. Below the header, the main content area is titled "Filters". It contains several input fields: "Car brand" and "Car model" are dropdown menus; "Price from", "Price to", "Year from", "Year to", "Mileage from", and "Mileage to" are text input fields. At the bottom of the filter section, there is a prominent yellow button labeled "CREATE SEARCH TASK".

Zrzut ekranu 1 Widok zakładki search



The screenshot shows the mobile version of the OFFERRAIDER application. It features a dark header bar with a hamburger menu icon on the left and the text "OFFERRAIDER" in yellow. The search filters are displayed in a vertical list on the left side of the screen. On the right side, there are input fields for "Year to", "Mileage from", and "Mileage to", followed by a yellow "CREATE SEARCH TASK" button. At the bottom of the screen, there is a dark footer area containing four yellow icons (a magnifying glass, a downward arrow, a leftward arrow, and a rightward arrow) and text that reads: "Software Engineering Project Description", "Made with ❤️ using Scala, Cheerio, Vue and Vuetify. Scrapped magnificent from OtoMoto".

Zrzut ekranu 2 Widok zakładki search (mobilny)

Z rozwijanej listy *Car brand* można wybrać markę samochodu, a jego konkretny model z pola *Car model*. Zdefiniować można również zakres cenowy, rok produkcji lub przebieg.

Filters

Car brand
Skoda

Car model
Scala

Price from
1000

Price to
70000

Year from
2019

Year to
2020

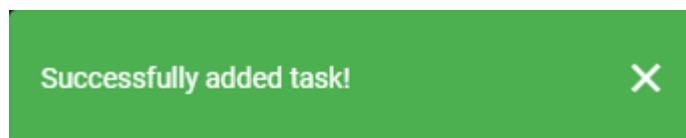
Mileage from
1000

Mileage to
100000

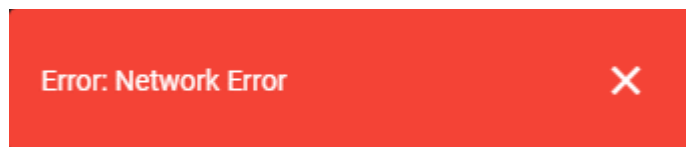
CREATE SEARCH TASK

Zrzut ekranu 3 Przykładowe wypełnienie filtrów wyszukiwania

Po kliknięciu przycisku *Create Search Task* pokazuje się powiadomienie mówiące, czy zadanie zostało utworzone, czy pojawiły się jakieś błędy.



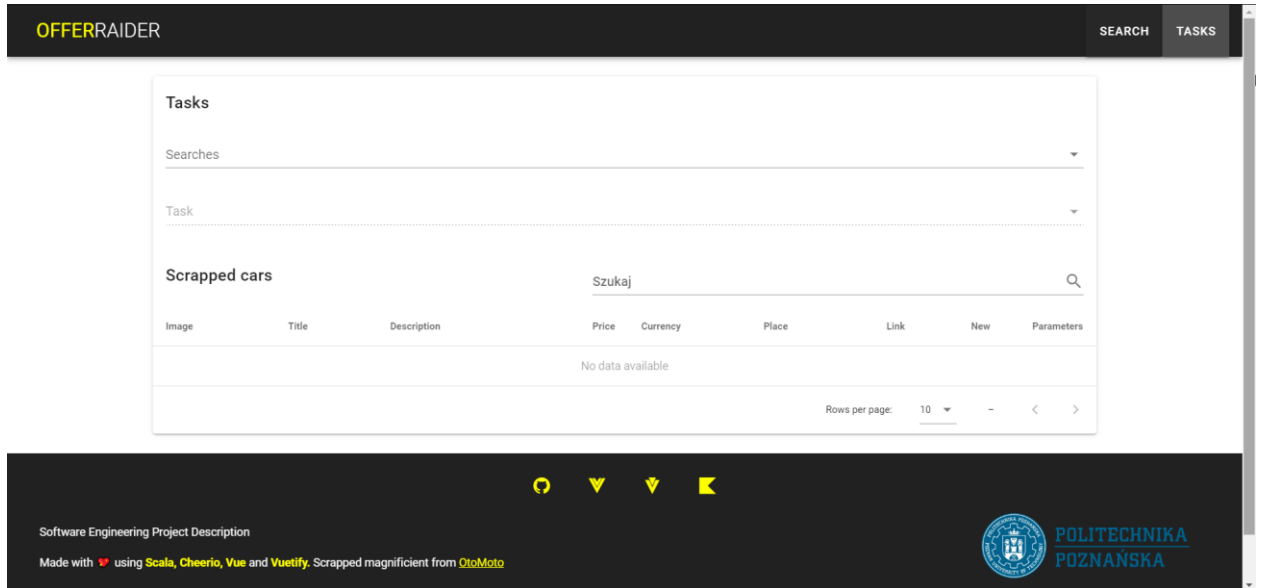
Zrzut ekranu 4 Wyskakujące powiadomienie



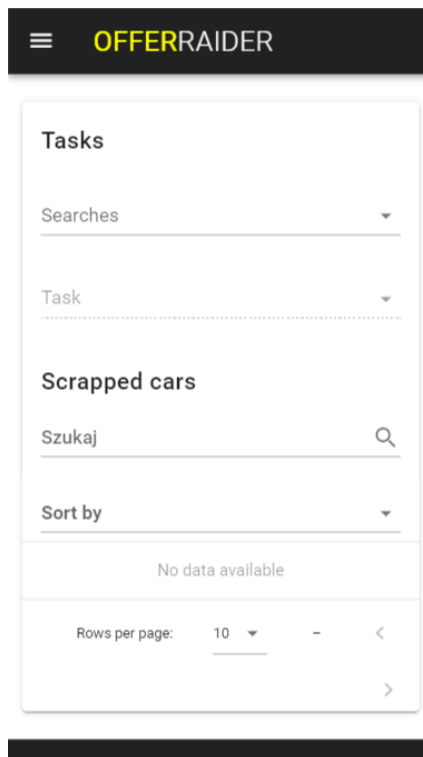
Zrzut ekranu 5 Powiadomienie o błędzie

Tasks

Zakładka *Tasks* pozwala na przeglądanie wyników wyszukiwań, jak również umożliwia zarządzanie zadaniami.

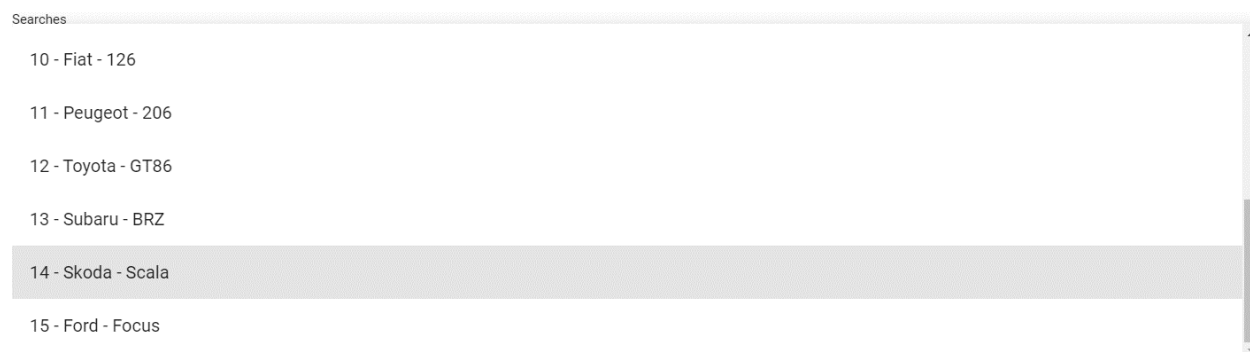


Zrzut ekranu 6 Widok zakładki Tasks



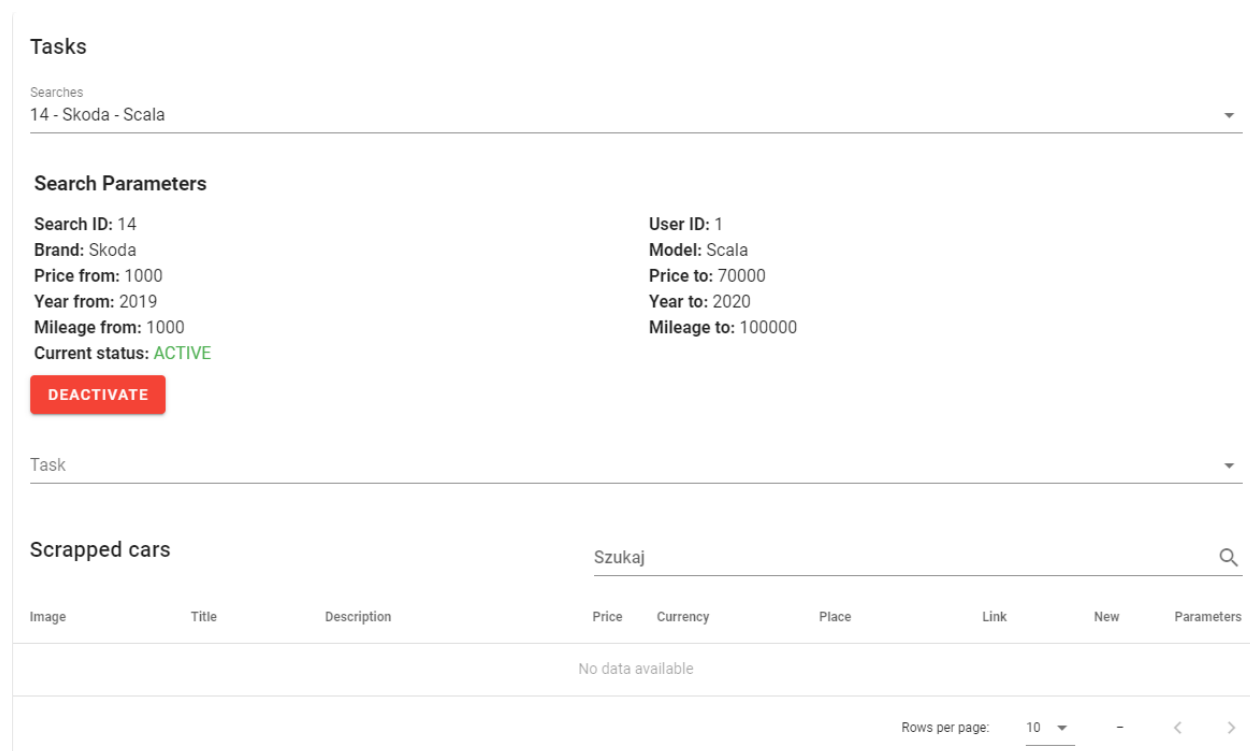
Zrzut ekranu 7 Widok zakładki Tasks (mobilny)

W zakładce *Tasks* można wyświetlić wyniki zadanych wyszukiwań. Pierwsze pole pozwala na wybór zdefiniowanych w zakładce *Search* zadań.



Zrzut ekranu 8 Pole Searches

Po dokonaniu wyboru, pole niżej znajdują się informacje o parametrach poprzednio wybranych przez użytkownika. Widoczny jest również przycisk pozwalający na zatrzymanie/wznowienie działania szukania.
















Zrzut ekranu 9 Widok wcześniej zadanego searcha

Pola *Task* pokazuje którego dnia i o której godzinie zwrócone zostały wyniki dla wybranego wyszukiwania. Każdy kolejny *task* zawiera poprzednie wyniki, a nowe pozycje są oznaczane flagą.




Zrzut ekranu 10 Lista rozwijana z wyborem taska

Po wybraniu taska pojawia się lista wyników, każdy wiersz oznacza jedną ofertę. W widoku ogólnym widać zdjęcie, tytuł, opis, cenę i lokalizację, jak również flagę oznaczającą nowe oferty. Do faktycznego serwisu można przejść za pomocą ikony odnośnika – , lub klikając w zdjęcie.

Task 2020 Jul 10 Friday, 21:59									
Scrapped cars					Szukaj				
Image	Title	Description	Price	Currency	Place	Link	New	Parameters	
	Škoda Scala 1.0	1.0 TSI 115KM STYLE, dostępna od ręki	69900	PLN	Wrocław, Dolnośląskie			▼	
	Škoda Scala	1.0 TSI 115 KM Style Gwarancja Salon PL FV23% ASO Plichta	68617	PLN	Bydgoszcz, Kujawsko-pomorskie			▼	
	Škoda Scala 1.0	Upust 15 tys DEMO Style 1.0 TSI/151 KM Comfort 2019 Full LED	66990	PLN	Wieliczka, Małopolskie			▼	
	Škoda Scala	Ambition 1.0 TSI 115 KM * Już od 487,06 zł/mc	68900	PLN	Olsztyn, Warmińsko-mazurskie			▼	

Zrzut ekranu 11 Widok taska

Po prawej stronie rozwinąć można szczegółowy widok parametrów oferty. Wyświetlony jest rok produkcji, typ paliwa, przejechane kilometry i pojemność silnika.







Škoda Scala

1.0 TSI 115 KM Style Gwarancja Salon PL
FV23% ASO Plichta

68617 PLN

Bydgoszcz,
Kujawsko-
pomorskie





Škoda Scala

1.0 TSI 115 KM Style Gwarancja Salon PL FV23% ASO Plichta

68617 PLN

Bydgoszcz – Kujawsko-pomorskie

Details

Year	2019
Fuel type	Benzyna
Mileage	13 100 km
Engine capacity	999 cm3

Zrzut ekranu 12 Szczegółowy widok oferty