Uniwersytet Ekonomiczny w Katowicach

Kierunek: Informatyka

Przedmiot: Technologie semantyczne

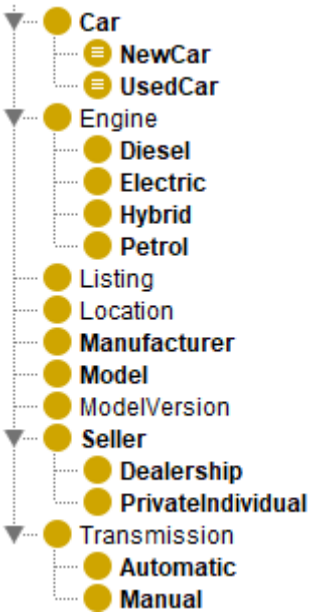
2024/2025

Dokumentacja projektu cars_ontology

Jan Bielski
Paweł Krawczyk

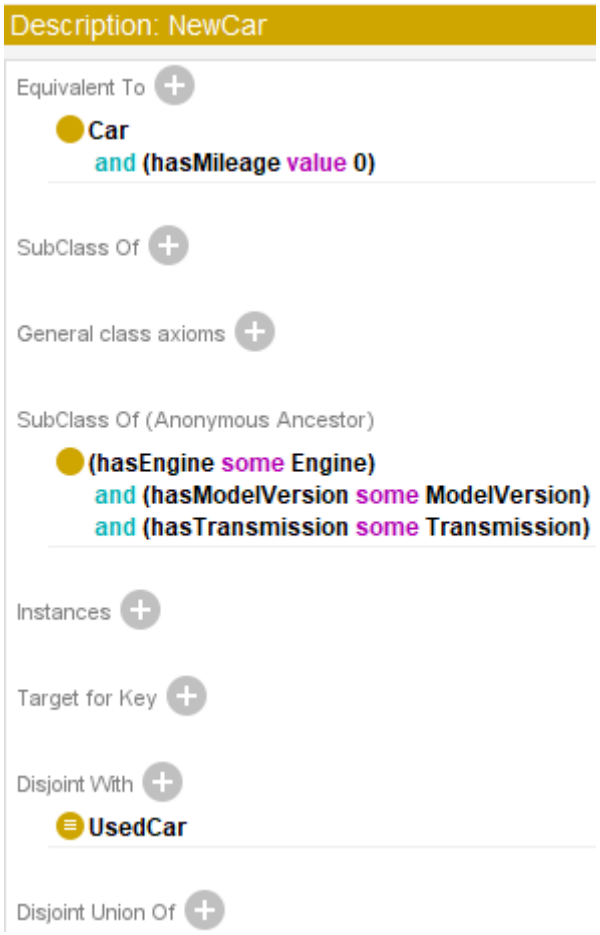
Opis ontologii

Ontologia opisuje proces sprzedaży samochodu. Jest ona luźnie inspirowana popularnymi stronami internetowymi z ogłoszeniami samochodów na sprzedaż. W projekcie nie wykorzystano innych, istniejących już ontologii. Ontologia zapisana jest w formacie RDF/OWL.



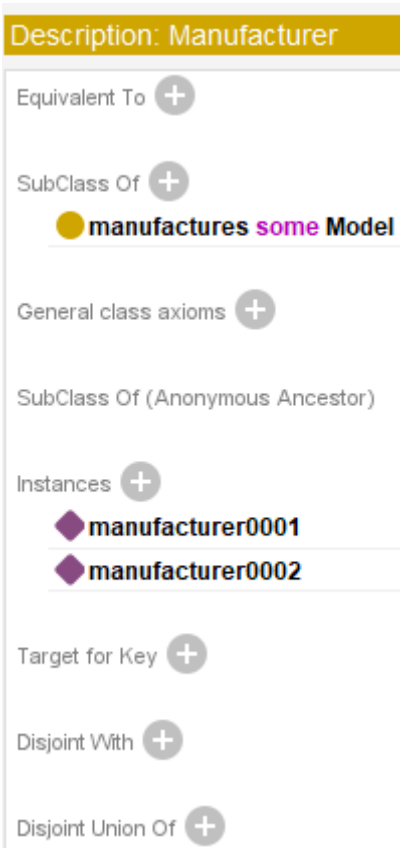
Lista klas

Ontologia definiuje samochody oraz ich atrybuty, a także ogłoszenia i sprzedawców. Niektóre klasy posiadają podklasy, które odzwierciedlają szczególne przypadki dla danych klas.



Przykład klasy spełniającej warunek konieczny i wystarczający w ontologii (a także przykład klasy rozłącznej)

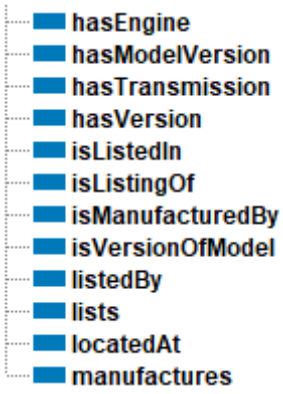
Na zrzucie ekranu widać, jak została zdefiniowana klasa NewCar oznaczająca nowe auto. Warunek konieczny i wystarczający jest taki, że obiekt klasy NewCar jest jednocześnie obiektem klasy Car i musi mieć wartość relacji hasMileage równą 0. Oznacza to, że samochód ma przebieg równy 0. Ponadto klasa NewCar jest klasą rozłączną z klasą UsedCar oznaczającą auto używane.



Przykład klasy spełniającej warunek konieczny

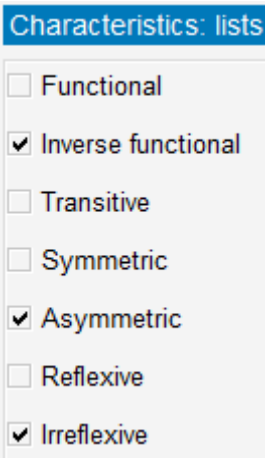
Na zrzucie ekranu widać, jak została zdefiniowana klasa Manufacturer oznaczająca producenta. Warunek konieczny i wystarczający jest taki, że obiekt klasy Manufacturer ma relację manufactures some Model, co oznacza, że producent produkuje jakiś model (samochodu).

Również niektóre klasy mają zdefiniowane warunki konieczne oraz warunki konieczne i wystarczające.



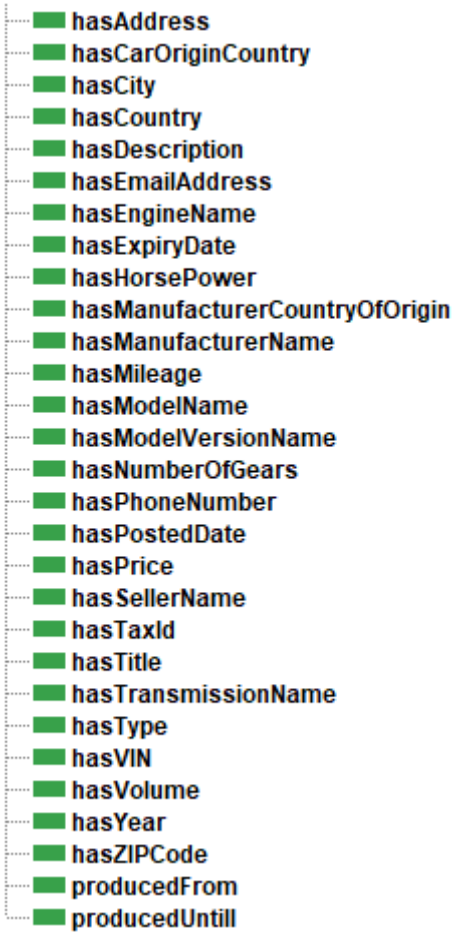
Lista relacji między klasami

Relacje w tej ontologii dotyczy głównie samochodów. Choć w większości trójek nie pojawia się bezpośrednio klasa „Car” to jednak wszystkie powiązania prowadzą do aut.



Przykład zastosowanych własności relacji

Relacja lists (tutaj oznacza „wystawia ogłoszenie”) jest odwrotnie funkcjonalna (takie jest założenie tej ontologii), asymetryczna i przeciwzwrotna. Wszystkie relacje ontologii są asymetryczne i przeciwzwrotne, ale nie wszystkie są odwrotnie funkcjonalne, ponieważ mniej więcej połowa relacji jest funkcjonalna.



Lista własności klas

Własności klas są bardziej zróżnicowane od relacji i pozwalają na określenie szczegółów opisujących sprzedawanych samochodów, ogłoszeń i sprzedających. Zdecydowana większość własności jest funkcjonalna.

Przykładowa instancja klasy Car

Oprócz tego, jak wygląda przykład instacji auta, zrzut ekranu pokazuje, że reasoner znalazł relacje, które wynikają z ontologii, a które nie zostały jawnie określone. Relacje te zostały zdefiniowane poprawne.

Opis aplikacji

Ta aplikacja służy do interakcji z bazą danych zawierającą oferty sprzedaży samochodów, opartą na ontologii. Umożliwia użytkownikom filtrowanie, pobieranie i odpytywanie ogłoszeń sprzedaży aut i powiązanych metadanych za pomocą zapytań SPARQL wykonywanych na serwerze Apache Jena Fuseki. Aplikacja zapewnia interfejsy API dla różnych funkcji, takich jak pobieranie ofert samochodów, filtrowanie danych według atrybutów i określanie przedziałów cenowych.

Podstawowym język programowania zastosowanym w kodzie aplikacji jest Python. Aplikacja wykorzystuje framework Django do zarządzania ustawieniami, w szczególności do przechowywania endpointu Fuseki. Apache Jena Fuseki jest serwerem hostującym ontologię i umożliwiający wysyłanie zapytań w języku SPARQL. SPARQL to język zapytań pozwalający na interakcję z ontologią. Ważną funkcję pełni biblioteka SPARQLWrapper, która służy do wykonywania zapytań SPARQL na serwerze Fuseki. Zapewnia metody konstruowania zapytań i formatowania wyników.

Zapytania są konstruowane dynamicznie przy użyciu filtrów. Prefiksy ontologii są dodawane automatycznie. Zapytania są wysyłane do endpointu Fuseki przy użyciu biblioteki SPARQLWrapper. Wyniki są pobierane w formacie JSON, a następnie przetwarzane w celu wyodrębnienia odpowiednich pól. Dane są formatowane do słowników lub list Pythona w celu łatwego wykorzystania.

Opis kluczowych funkcji:

1. execute_sparql_query

```python
def execute_sparql_query(query):
    """
    Helper function to execute a SPARQL query against the Fuseki server.
    """
    sparql = SPARQLWrapper(settings.FUSEKI_SPARQL_ENDPOINT)
    prefix = """PREFIX : <http://www.semanticweb.org/janbi/ontologies/2025/0/cars-ontology/>\n
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>"""
    query = prefix + "\n" + query
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    return sparql.query().convert()
```

Jest to funkcja pomocnicza, która wykonuje zapytania SPARQL na serwerze Fuseki, dodaje prefiksy ontologii do zapytania oraz zwraca wyniki zapytania w formacie JSON.

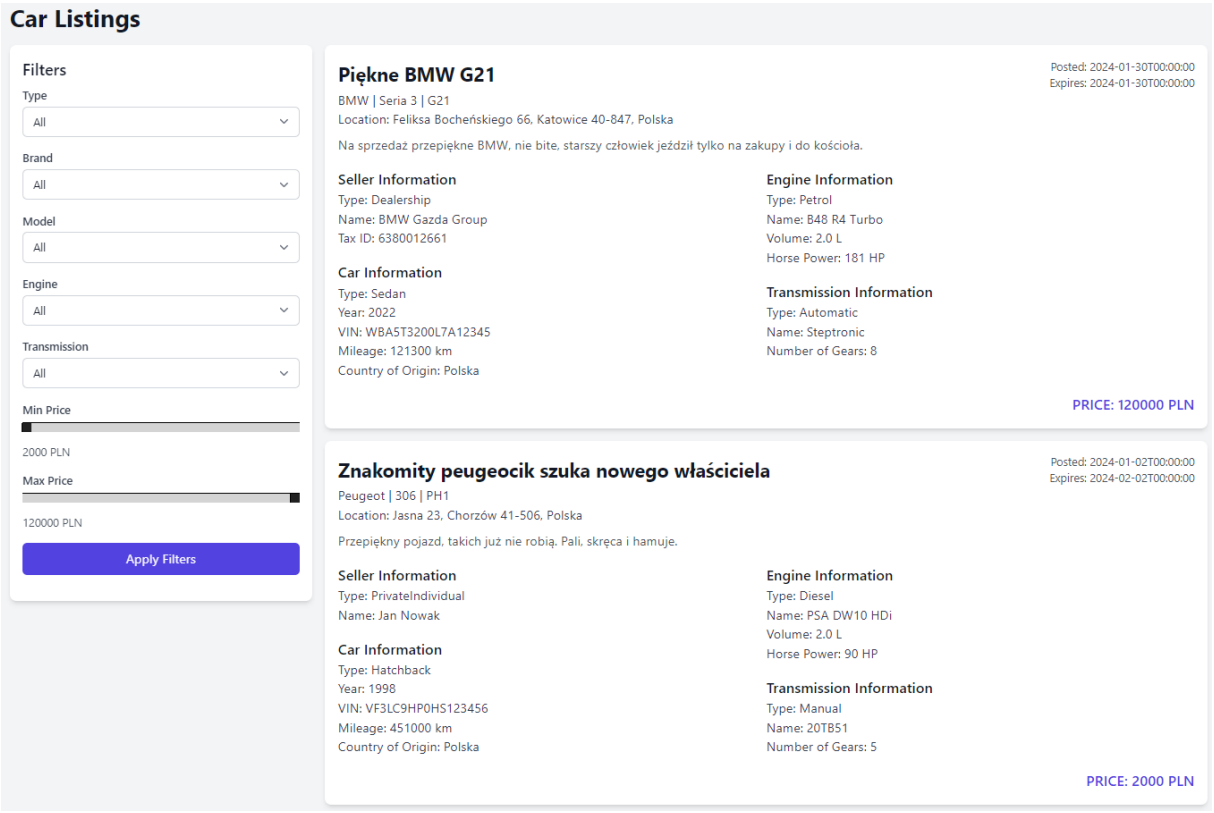
```
query = f"""
    SELECT ?title ?modelVersionName ?modelName ?manufacturerName ?postedDate ?expiryDate ?description ?mileage ?price
           (STRAFTER(STR(?transmissionType), "cars-ontology/") AS ?transmissionTypeName) ?transmissionName ?numberOfGears
           (STRAFTER(STR(?engineType), "cars-ontology/") AS ?engineTypeName) ?engineName ?volume ?horsePower ?countryOfOrigin
           ?vin ?year ?address ?city ?country ?zipCode
           (STRAFTER(STR(?sellerType), "cars-ontology/") AS ?sellerTypeName) ?sellerName ?taxId ?modelVersionType ?carState WHERE {{
        ?listing a :Listing ;
            :hasTitle ?title ;
            :isListingOf ?car ;
            :hasPostedDate ?postedDate ;
            :hasExpiryDate ?expiryDate ;
            :hasDescription ?description ;
            :hasPrice ?price ;
            :locatedAt ?location ;
            :listedBy ?seller .
        ?car :hasModelVersion ?modelVersion ;
             :hasMileage ?mileage ;
             :hasTransmission ?transmission ;
             :hasEngine ?engine ;
             :hasVIN ?vin ;
             :hasYear ?year ;
             :hasCarOriginCountry ?countryOfOrigin .
        ?modelVersion :isVersionOfModel ?model ;
                      :hasModelVersionName ?modelVersionName ;
                      :hasType ?modelVersionType .
        ?model :hasModelName ?modelName ;
               :isManufacturedBy ?manufacturer .
        ?manufacturer :hasManufacturerName ?manufacturerName .
        ?transmission rdf:type ?transmissionType ;
                      :hasTransmissionName ?transmissionName ;
                      :hasNumberOfGears ?numberOfGears .
        ?transmissionType rdfs:subClassOf :Transmission .
        ?engine rdf:type ?engineType ;
                :hasEngineName ?engineName ;
                :hasVolume ?volume ;
                :hasHorsePower ?horsePower .
        ?engineType rdfs:subClassOf :Engine .
        ?location :hasAddress ?address ;
                  :hasCity ?city ;
                  :hasCountry ?country ;
                  :hasZIPCode ?zipCode .
        ?seller rdf:type ?sellerType ;
                :hasSellerName ?sellerName .
        ?sellerType rdfs:subClassOf :Seller .
        OPTIONAL {{
            ?seller a :Dealership ;
                    :hasTaxId ?taxId .
        }}
        BIND(IF(EXISTS {{ ?car a :NewCar }}, "New", "Used") AS ?carState)
        {filter_string}
    }}
"""
```

Zmienna query, która funkcja przyjmuje jako argument

Za zmienną query kryje się zapytanie SPARQL. Przykładowe zapytanie (wykorzystywane przez funkcję fetch_filtered_listings), zwracające szczegółowe informacje o ogłoszeniach sprzedaży samochodów, przedstawiono powyżej. Wyniki zapytania prezentują pełen obraz dostępnych ogłoszeń z podziałem na szczegóły samochodu, jego specyfikacji technicznej, lokalizacji i informacji o sprzedawcy.

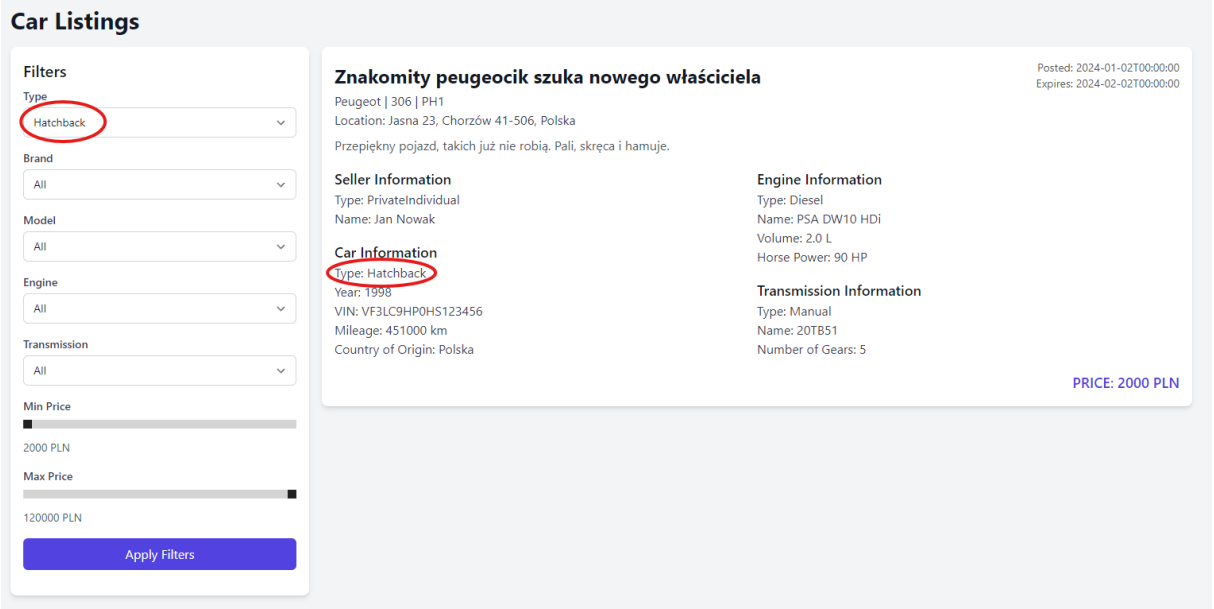
2. fetch_filtered_listings

Ta funkcja pobiera oferty samochodów spełniające określone kryteria filtrowania.



Podgląd aplikacji z domyślnymi filtrami

Domyślnie filtry ustawione są tak, by pokazywać wszystkie dostępne ogłoszenia, niezależnie od ich charakterystyki. Funkcja pozwala na dynamiczne konstruowanie zapytań SPARQL w celu zastosowania określonych filtrów.



Przykład wykorzystania funkcji w praktyce - filtrowanie wg typu samochodu

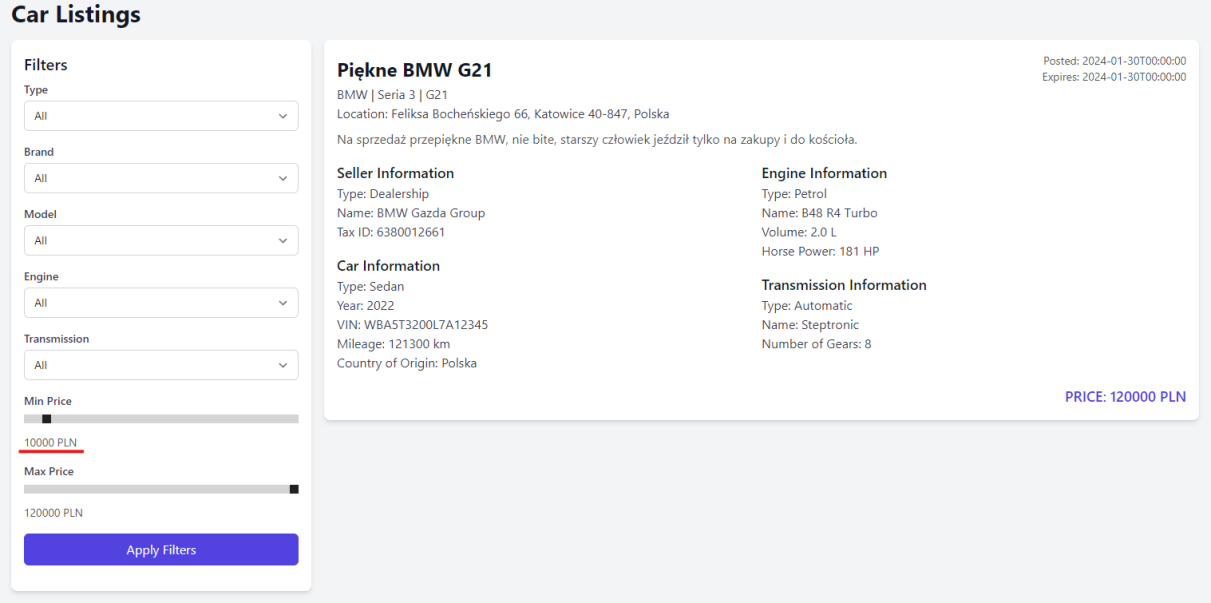
Funkcja zwraca szczegółowe informacje o pasujących ofertach, co pozwala funkcji home na poprawne wyświetlenie zastosowanego filtrowania.

3. fetch_all_listings, fetch_all_manufacturers, fetch_all_models, fetch_all_transmission_types, fetch_all_engine_types, fetch_all_model_version_types

Te funkcje zostały przypisane do tego samego punktu, ponieważ mają podobne zadanie – zwracają wszystkie zdefiniowane w ontologii kolejno: ogłoszenia, producentów, modele, rodzaje skrzyni biegów, rodzaje silników i wersje modeli. Te z kolei są następnie wykorzystywane przez funkcję home.

4. get_price_range

Funkcja zwraca minimalną i maksymalną cenę znajdujące się w ogłoszenia w ontologii. Pozwala to funkcji home na utworzenie suwaków, którymi można manipulować widełkami cenowymi.

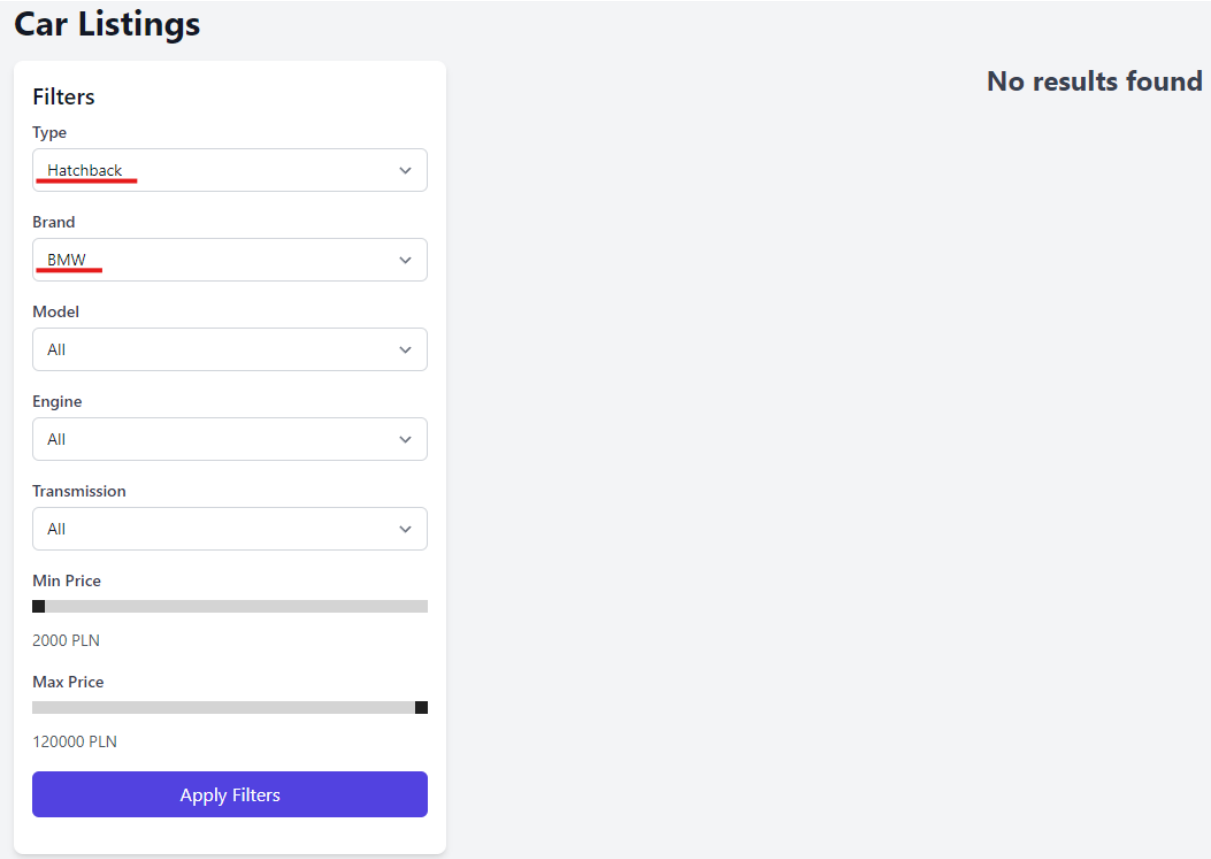


Przykład wykorzystania funkcji w praktyce - filtrowanie wg ceny

W tym przykładzie ustawienie minimalnej ceny na poziomie 10 000 zł powoduje, że wyświetla się tylko jeden samochód, ponieważ druga oferta ma cenę 2 000 zł.

5. home

Ta funkcja powiązana z Django służy za backend dla strony głównej aplikacji. Przetwarza dane modyfikowane przez użytkownika, pobiera odpowiednie dane i przekazuje wyniki do szablonu w celu ich wyświetlenia. Użytkownik podaje filtry za pomocą parametrów zapytania adresu URL. Funkcja home przetwarza te filtry, pobierając odpowiednie listy samochodów z bazy danych za pomocą zapytań SPARQL. Pobiera również listy dostępnych atrybutów, które mogą być wykorzystane w opcjach filtrowania. Na koniec widok wyświetla szablon home.html, przekazując wszystkie niezbędne dane (np. oferty samochodów, opcje filtrowania, zakresy cen) do szablonu w celu wyświetlenia.



Przykład użycia aplikacji - zastosowanie takich filtrów, które nie pokrywają się z żadnym ogłoszeniem w ontologii