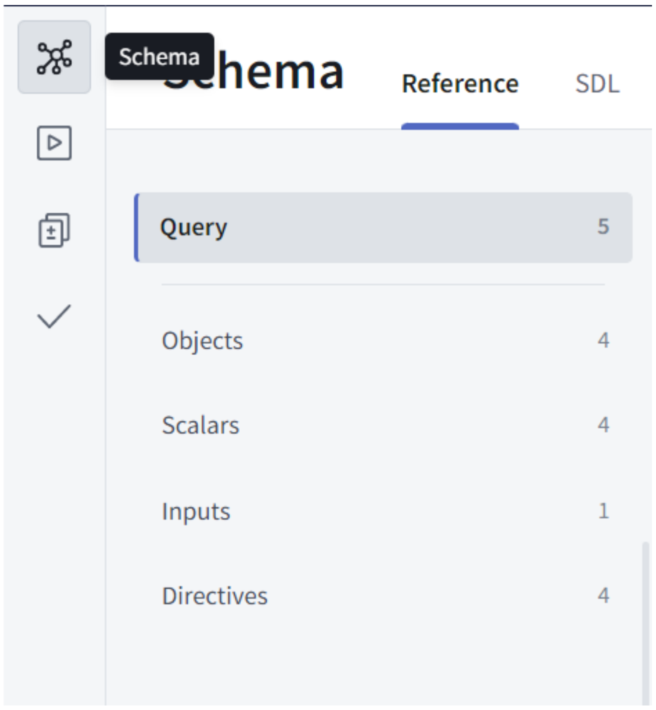
Guide to using the GraphQL Server

As part of this short guide we will not be describing how to actually format and send GraphQL requests to the server (because that is one of the actual integration tasks), or get too in depth into the structure of the API (because it's based on the database file you've sent us, you know how the data looks).
Since GraphQL allows for querying in very specific formats this guide should get you up and running, making the queries you need for your client website. The GraphQL endpoint itself can be found at the following URL: https://si-products.azurewebsites.net/graphql

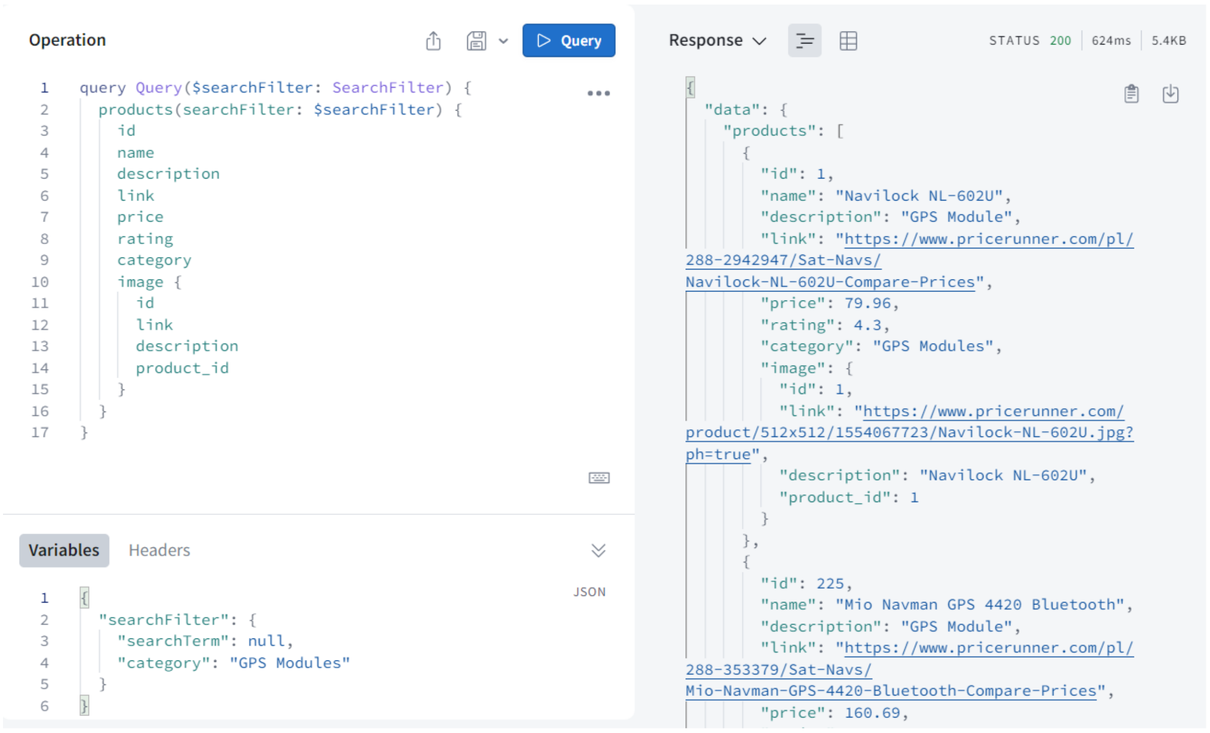
Using the API Sandbox

The sandbox functions both as API documentation and "easy query builder". It can be found at the following URL: https://si-products.azurewebsites.net/

To see the API documentation simply access the "Schema" section of the sandbox found on the left hand side of the sandbox. The schema provides descriptions for all queries, other types, inputs and their individual fields. The entire schema definition language code is also available if needed.



We recommend using the “Explorer” section of the sandbox to easily build queries in different formats, test them out, see the expected query variables format and the response format. After which you can simply “copy and paste” the queries into your client.



```
query Query($searchFilter: SearchFilter) {
  products(searchFilter: $searchFilter) {
    id
    name
    description
    link
    price
    rating
    category
    image {
      id
      link
      description
      product_id
    }
  }
}
```

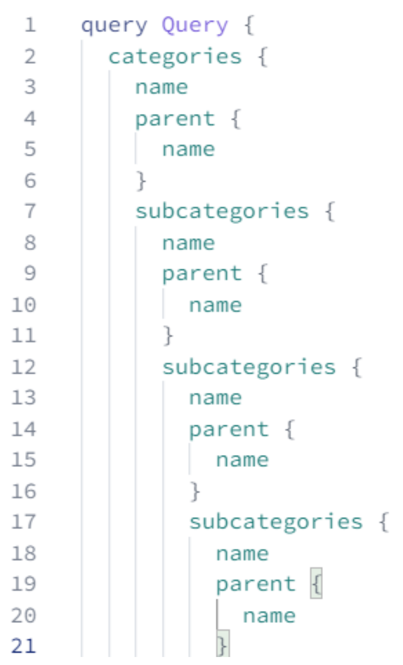
```
{
  "searchFilter": {
    "searchTerm": null,
    "category": "GPS Modules"
  }
}
```

```
{
  "data": {
    "products": [
      {
        "id": 1,
        "name": "Navilock NL-602U",
        "description": "GPS Module",
        "link": "https://www.pricerunner.com/pl/
288-2942947/Sat-Navs/
Navilock-NL-602U-Compare-Prices",
        "price": 79.96,
        "rating": 4.3,
        "category": "GPS Modules",
        "image": {
          "id": 1,
          "link": "https://www.pricerunner.com/
product/512x512/1554067723/Navilock-NL-602U.jpg?
ph=true",
          "description": "Navilock NL-602U",
          "product_id": 1
        }
      },
      {
        "id": 225,
        "name": "Mio Navman GPS 4420 Bluetooth",
        "description": "GPS Module",
        "link": "https://www.pricerunner.com/pl/
288-353379/Sat-Navs/
Mio-Navman-GPS-4420-Bluetooth-Compare-Prices",
        "price": 160.69,
```

Hierarchical categories

In implementing the **Category** type we wanted to provide a somewhat convenient way of getting all categories organised in hierarchical order, or at the very least find out a category’s parent and direct descendents. This is why the type also contains the **parent** and **subcategories** fields.

A simple but not very efficient way of getting all categories in hierarchical order is to keep nesting categories in queries. By knowing the maximum depth of the category tree we can do a query such as the one below



```
query Query {
  categories {
    name
    parent {
      name
    }
    subcategories {
      name
      parent {
        name
      }
      subcategories {
        name
        parent {
          name
        }
        subcategories {
          name
          parent {
            name
          }
```

An unfortunate side effect of the way this query works is that the response object will contain duplicate categories (if we're searching for all categories, if we're searching by id it works fine). This can be solved by running a cleanup line on the first level of the hierarchy after the data is fetched i.e. keeping the categories that do not have a parent (they are at the root level)

```
const hierarchicalCategories = data.categories.filter(
   (category) => category.parent === null
);
```

We are currently working on a more efficient way of providing category data. When it is ready this document will be updated accordingly.