Marketing (Objective Platform)

This project implements necessary services and API containing endpoints for providing marketing measurements and insights to the front-end application

Content

Running the application

Pre-requisites

This application relies on two technologies that need to be installed locally to be executed:

- Docker - Allows us to run the application and database containerized and within the docker network.
- Taskfile - Task runner that allows us to run commands in a more organized way, making it easier to interact with the application either for building it, executing it, running tests, etc.

Run

To run the application, you need to execute the following command:

```
task run
```

This command will first build the application (which can be also done by running `task build-project` before `task run`) and then run it. The application will be available at `http://localhost:8000`.

:warning: When running the project for the first time, execution might take a bit longer as it needs to download the necessary docker images and build the application, plus perform the initial database migrations (among which inserting the demo data records is included)

Requests

Authentication The service implements token authentication (for demonstration purposes), so in order to perform requests to the API, you need to obtain a token to consume the marketing endpoints.

Obtain a token by performing a request to the `GET /api/auth/token/` (Basic Auth). The service implements sample credentials username: `marketing_op` and password: `marketing_op_supersecret` which you can use for this (base64 encoded).

```
curl --location --request GET 'http://localhost:8000/api/auth/token/' \
--header 'Authorization: Basic bWFya2V0aW5nX29wOm1hcmtldGluZ19vcF9zdXBlcnNlY3JldA=='
```

The response will contain the token:

```
{
    "data": {
        "token": "<token>"
    }
}
```

Performing requests Use the obtained token in previous step to perform requests to the marketing endpoints (Bearer Auth).

For example, to get the weekly sales per channel:

```
curl --location --request GET 'http://localhost:8000/api/marketing/stats/channel-weekly-sale
```

Response:

```
{
    "data": [
        {
            "channel": "facebook",
            "year": 2020,
            "week": 1,
            "sales": 5333.516215408056
        },
        {
            "channel": "facebook",
            "year": 2020,
            "week": 2,
            "sales": 13225.911516370496
        },
        {
            "channel": "facebook",
            "year": 2020,
            "week": 3,
            "sales": 12036.479334585327
        },
        {
            "channel": "facebook",
            "year": 2020,
            "week": 4,
            "sales": 9708.314594927879
        },
        {
            "channel": "facebook",
            "year": 2020,
            "week": 5,
            "sales": 8593.207280276212
        }
    ]
}
```

Tests

To run the tests, you can execute the following command:

```
task test
```

Project Overview

The application is built in Python. Given my recent experience and in order to make it faster to develop, the application is built using `Django`. The assignment stated that it was preferable to do it in `FastAPI` or `Flask`, so considering that I've chosen to implement it using Django-Ninja, which is a web framework for building APIs with Django and, as described in their documentation, *"heavily inspired in FastAPI"*, sharing a lot of its concepts and structure (easy to use and intuitive, typed, high performant, based in standards, etc.) as you will be able to see in the source code.

Architecture

The implementation is approached with a Service Layer pattern, where the business logic is separated from the Django models and the API endpoints implementation. This allows to encapsulate the business logic, providing a clear separation of concerns. It's basically an interface to the domain model, which allows to better test and maintain the code.

Application counts with two Django apps:

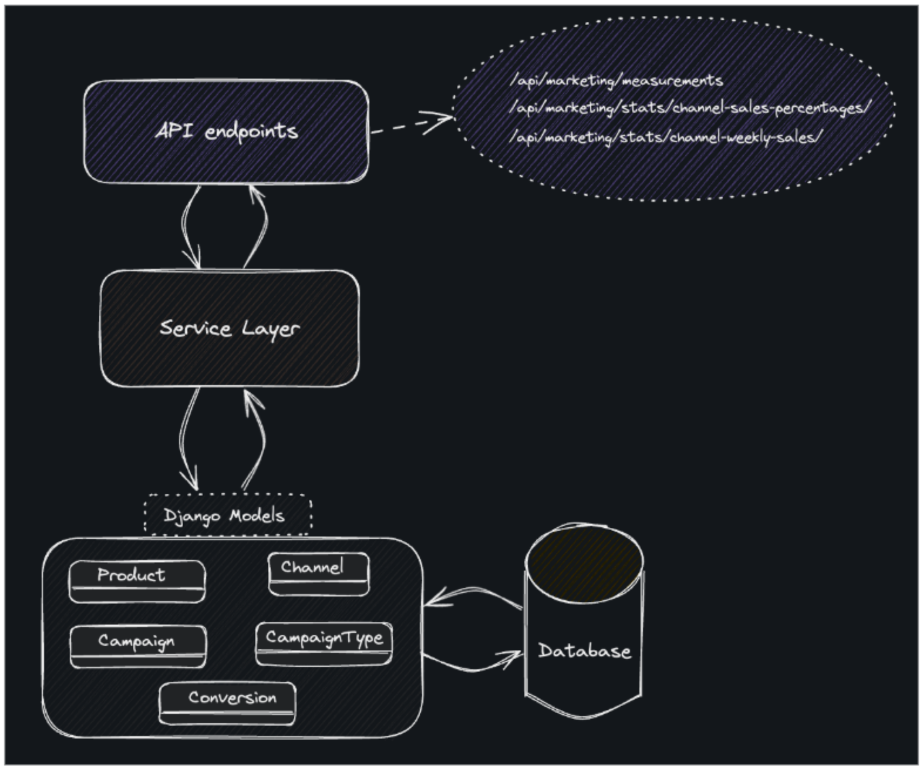

API endpoints
/api/marketing/measurements
/api/marketing/stats/channel-sales-percentages/
/api/marketing/stats/channel-weekly-sales/
Service Layer
Django Models
Product
Channel
Campaign
CampaignType
Conversion
Database

Figure 1: architecture

- `core`: contains the models. In this case with just count with one additional application but, the reason to centralize models in this separate application is to allow re-usability and keep consistency in case we scale the project with more apps.
- `api`: contains the API endpoints and the service layer with all the logic.

`marketing_op_api.py` contains the exposed endpoints implementation, which calls the service layer (`api/services`) modules to obtain the necessary data, without interacting directly with the database and simplifying the code.

Database

The database used is a PostgreSQL database, containing the following tables:

Through the Django ORM, the models are defined in the `core` app, and the migrations are created and applied to the database. The database is initialized with some demo data, which is inserted through the migrations. Also, queries for the different use cases have been implemented with the Django ORM, which allows to interact with the database in a more Pythonic way, but also used raw queries can be seen by using the `query` property of a Django query.

For example, the query used for obtaining the weekly sales per channel is translated as:

```sql
SELECT "channel"."name",
       DATE_TRUNC(WEEK, "conversion"."date") AS "week",
       SUM("conversion"."conversions") AS "net_sales"
FROM "conversion"
INNER JOIN "channel" ON ("conversion"."channel_id" = "channel"."id")
WHERE "channel"."name" IN (radio,
                           tv,
                           facebook,
                           instagram)
GROUP BY "channel"."name",
         2
ORDER BY 2 ASC,
         "channel"."name" ASC
```

*Filtering for channels `radio`, `tv`, `facebook`, `instagram`.

API

Documentation

The OpenAPI documentation (`yaml`) can be found in the openapi.yaml file. And there is also a corresponding html version generated within the same directory.

> :information_source: Even though Django-Ninja generates its own docs, I've decided to also manually create an OpenAPI documentation to customize it a bit and provide some additional

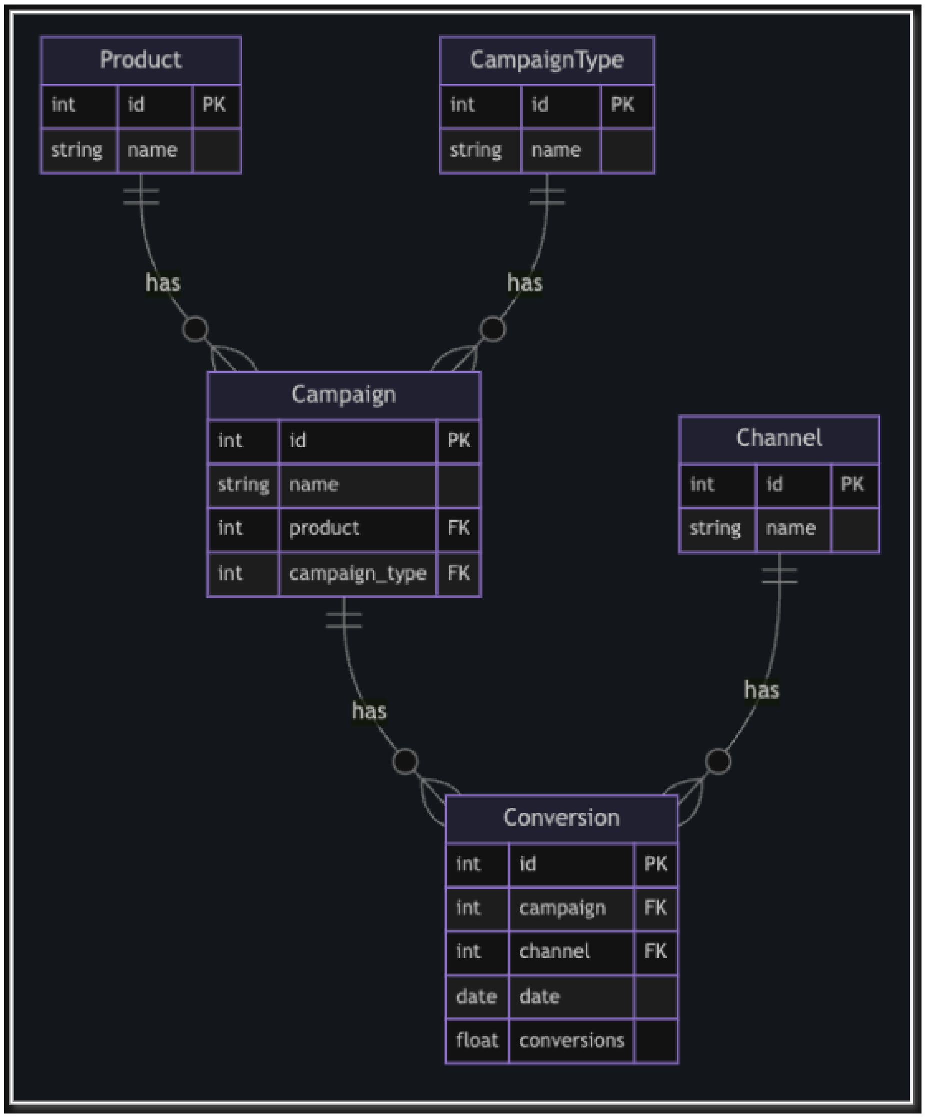
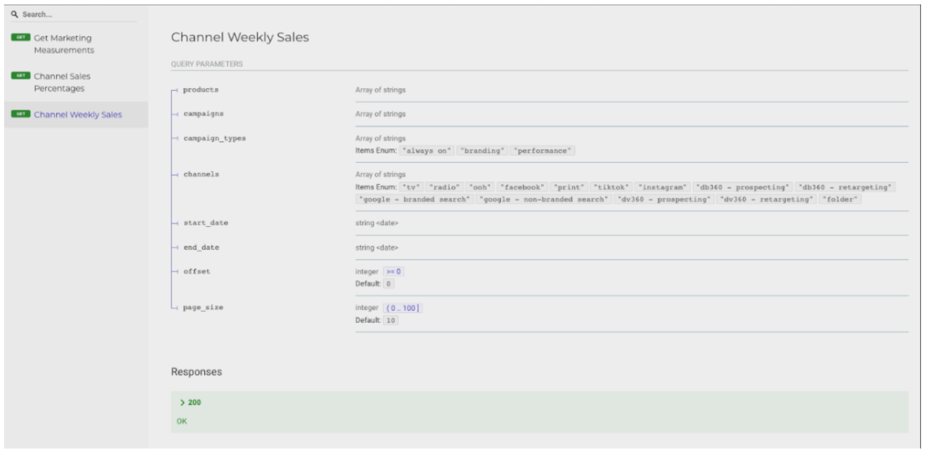


Figure 2: database



Channel Weekly Sales

QUERY PARAMETERS

```
products        Array of strings
campaigns       Array of strings
campaign_types  Array of strings
                Items Enum: "always on" "branding" "performance"
channels        Array of strings
                Items Enum: "tv" "radio" "ooh" "facebook" "print" "tiktok" "instagram" "db360 - prospecting" "db360 - retargeting"
                "google - branded search" "google - non-branded search" "dv360 - prospecting" "dv360 - retargeting" "folder"
start_date      string <date>
end_date        string <date>
offset          integer >= 0
                Default: 0
page_size       integer [ 0 .. 100 ]
                Default: 10
```

Responses

> 200
OK

Figure 3: api_docs_html

> context. The auto-generated docs can in any case be accessed in http://localhost:8000/api/docs

Endpoints

The API has four endpoints for the requested use cases, which details can be found in the previously mentioned OpenAPI documentation:

- `/api/token/`: returns a token to authenticate the user in the rest of the endpoints.
- `/api/marketing/measurements/`: returns the marketing measurements for a given date range and channel.
- `/api/marketing/stats/channels-sales-percentages`: returns the net sales attributed to media channels.
- `/api/marketing/stats/channel-weekly-sales`: returns the weekly sales per channel.

Authentication

As briefly mentioned in the previous section, the API endpoints related to the marketing measurements require authentication, which is implemented with demonstration purposes. This is done through token-based authentication, where the user needs to obtain a token by providing a username and password (Basic Auth) to the `/api/token/` endpoint. Then the token can be used in the Authorization header (Bearer Auth) to authenticate the user in the rest of the endpoints.

> :information_source: Possible improvements here would be: - Implementation of token expiration or refresh mechanism. - Implementing a more secure authentication method (e.g. OAuth2, JWT).

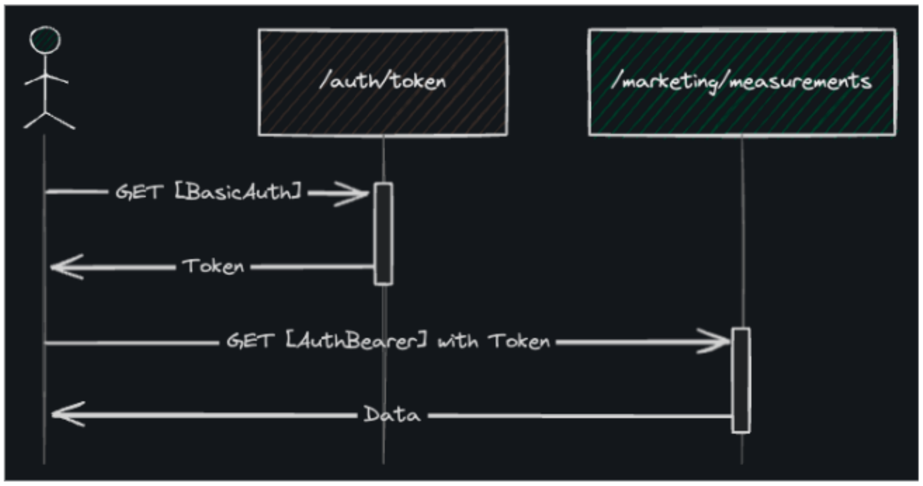


Figure 4: auth

Pagination

The API endpoints implement a basic pagination, which you will be able to see in the OpenAPI schema, managed through the query parameters `offset` and `page_size` (default is `10`).

> :information_source: Possible improvements here would be: - Implementation of pagination information within Link header or in the response body.

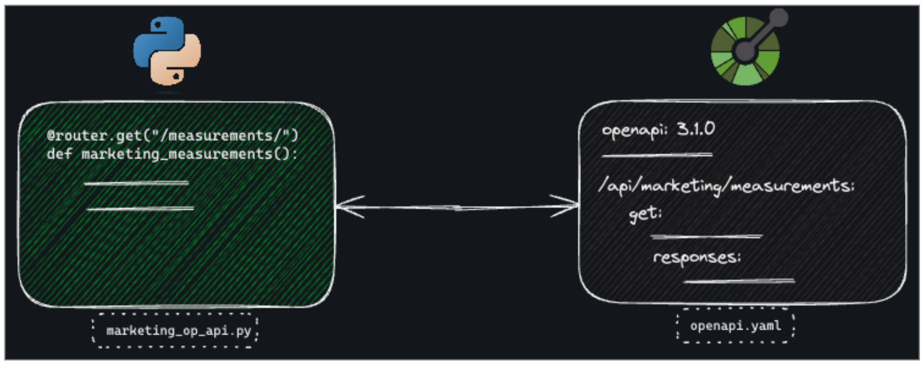
Testing

There are included in the code some unit tests and also API tests, separated from the service functions for example (advantage of having a service layer). The tests can be run with the command `task test`.

Factories For demo data creation (and as demonstration) I've used factoryboy to implement factories, which allow to easily create objects for testing purposes, and is also compatible with Django models.

Contract Testing Tests also implement contract testing for all the API tests, this is done using django-contract-tester library, which implements a client, that allow us to easily override our test clients and check in every existing tests, apart from the functional validations also if they match the written documentation/design.

This is a useful features for all APIs, but specially in public APIs, to keep consistency and make sure your API is always aligned with that the consumer



```
@router.get("/measurements/")
def marketing_measurements():
```

```
openapi: 3.1.0
/api/marketing/measurements:
  get:
    responses:
```

Figure 5: contract_testing

expects. The clients are instantiated as fixtures, and how it works can be checked for example modifying the openapi schema (removing for example a response from an endpoint) and re-running the tests, they should fail stating an undocumented response error.

BDD Behaviour driven development is a software development process which includes natural-language constructs and encourages collaboration among developers. Within the tests suite there is an example of using pytest-bdd to implement automation of tests, using Gherkin Language and facilitating behavioral driven development.

```
Scenario: Retrieve marketing measurements for specific channel
  Given a set of existing conversions for marketing campaigns
  When requesting marketing measurements through API for a specific channel
  Then I should receive a list of measurements for that channel
```