botify

Python/API Challenge

Duration: 4 hours

Don't spend more than 4 hours on the test and send what you have done in these 4 hours.

The test is the same for junior and senior techs, and we don't expect every candidate to fulfill everything.

Go as far as you can (not completing every steps but polishing one will be appreciated too).

Steps are given in order of importance - please try to complete Step 1 as well as possible before starting Step 2.

Please consider that every aspect of the test is judged, from Git history to READMEs, from code design and comments to the actual working result.

Source-code management and hosting:

You should use **Git** to manage your code using iterative commits.

Each step is a new branch.

Please try to submit a clean commit history, as you will be judged on the presentation of your code.

Either send us your project by email (via git bundle) or push your local git repository on a private GitHub repo and send us the link!

When complete, send your challenge to robin@botify.com (VP of Engineering), jb@botify.com (VP of Engineering) and david@botify.com (API Engineer), or invite their Github users (@eisenbergrobin, @jbbarth, and @David-Wobrock) to your private repo.

Goal:

The goal is to create an almost-shippable API that allows the requester to access information on French population data, using **Django** (https://www.djangoproject.com/) with **Django Rest Framework** (http://www.django-rest-framework.org/) to build the API.

The first part of the challenge requires you to **build a read-only REST API** for reading the data from the database.

The second part of the challenge requires you to **implement a small DSL** to query the data from the database.

The test consists of 5 steps.

- Implement an endpoint that lists all towns in the database (/towns)
- Implement an endpoint that allows for aggregation of the towns in the database (/aggs)
- Implement a small DSL to allow queries that perform fields selections and filters
- Improve your DSL to allow for compounding filters
- [BONUS] Implement a Dockerfile for your project

The data you will be providing an interface to is a list of French cities and towns, with some basic information about each town.
The data is located here:
https://docs.google.com/spreadsheets/d/1UNLhH3NNoXqmX_ibgBufhv8cni2SFyGmh4mXOjNP1uM

This data can be downloaded and added in your code locally.

Explore Django Rest Framework and all its concepts, you'll see that you'll be able to implement all the following requirements by using existing bricks.

Step 1: Read list of Towns

Implement a JSON REST read endpoint to list all of the towns in the database.

/towns - Returns a list of towns.

- Each town should be a JSON object with department name, town code, department code, population as key/value.
- Results should be paginated and the pagination mechanism should be reusable across other -future- read endpoints.
- Offer options for:
    - Sort (eg: "I want the towns sorted by population size")
    - Filtering (eg: "I want only the towns with a population that exceeds 15k")

Scripts:
The project should add at least 2 scripts:
- **script/install**: Install all dependencies needed
- **script/run**: Start the app, frontend app can be accessed on *localhost:8000*.

Tips:
- Split your code (actions, views, components, utils..) into different files and folders.
- Feel free to use any additional libraries. We are always interested in new ideas.

botify

Step 2 - Simple Aggregation

In a new branch, add a new endpoint to offer the possibility to aggregate the Towns in the database by Region Code or by Dept Code.

/aggs - Returns aggregation results on towns' populations

- Asks for a key to aggregate by (eg: department code or region code)
- Returns for each key:
    - Number of towns
    - min/max/avg population for each resulting key

Tips:

- Feel free to use any additional libraries. We are always interested in new ideas.

Step 3 - Implement a small DSL

In this step, we want you to reimplement some concepts of the Botify Query Language (BQL, https://developers.botify.com/api/bql/) consisting of transforming a JSON query (in a custom DSL) into a SQLite Query.

This step is not plugged into the previous steps for now, the goal is just to implement the DSL and its corresponding query.
For testing purposes, implement a "**/query**" endpoint that takes the DSL queries as POST inputs, and responds with the string of the generated SQLite query.

We want to implement :

- Fields selectors
- Filters selectors with predicates and operators (contains, greater than, etc…)
- Query Validation

botify

Ex of queries :

DSL Query	SQLite Query
{“fields”: [“name”, “population”]}	SELECT name, population FROM towns

{
“fields”: [“name”],
“filters”: {
“field”: “distr_code”, “value”: 1
}
}
	SELECT name
	FROM towns
	WHERE distr_code = 1

{
“fields”: [“name”],
“filters”: {
“field”: “population”,
“value”: 10000,
“predicate”: “gt”
}
}
	SELECT name
	FROM towns
	WHERE population > 10000

You don’t need to validate that the fields exist in the following schema, which represents the dataset in the CSV file:

```
[
  { name: 'name', type: str },
  { name: 'code', type: int },
  { name: 'population', type: int },
  { name: 'average_age', type: float },
  { name: 'distr_code', type: int },
  { name: 'dept_code', type: int },
  { name: 'region_name', type: str },
  { name: 'region_code', type: int },
];
```

You do, however, **need to validate the query given as input.**

The DSL should implement:

- Schema instantiation (according to the format above)
- Fields selection
- Filters conditions :
    - It can be directly if field condition : {“field”: “xx”, “value”: yy}
    - The field condition can be completed by a predicate :
        - equal

        - gt
        - lt
        - contains
- Validation :
    - You must validate that the query is rightly formatted
    - You must validate that the field exist in the schema
    - You must validate that the value is permitted by the field type
    - Returned exceptions must be as precise as possible when the query is not valid.

Step 4: Add filters compounding to your DSL

Amend your DSL so it can compound filters together using “and” or “or” conditions, **recursively**.

DSL Query	**SQLite Query**

{
“fields”: [“name”],
“filters”: {“and”: [
{
“field”: “population”,
“value”: 10000,
“predicate”: “gt”
},
{
“field”: “region_name”,
“value”: “Hauts-de”,
“predicate”: “contains”
}
]
}
}
	SELECT name
	FROM towns
	WHERE population > 10000 AND region_name LIKE
	‘%Hauts-de%’

This filter compounding should work recursively.

[BONUS] Step 5: Dockerfile

Do not start this step unless you have a complete and polished result for all other steps and only if you have time remaining on the 4-hour allowance.

Create a Dockerfile that runs your project server. At this point your server should serve three endpoints on :8000 :

- **/towns -** Returns a list of towns.
- **/aggs** - Returns aggregation results on towns’ populations
- **/query** - Returns the generated SQLite query for the JSON query given as input