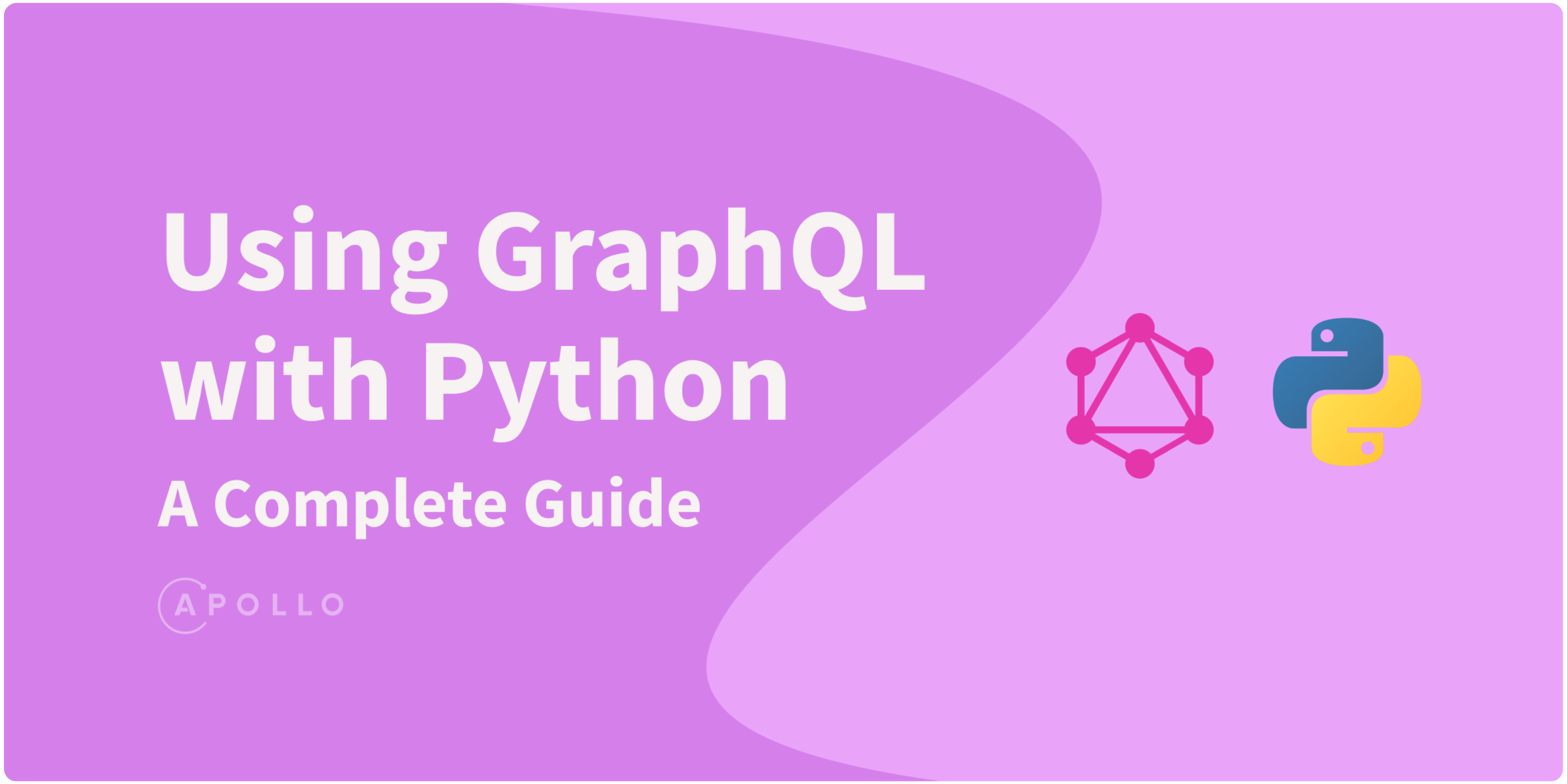
MAY 11, 2021

# Using GraphQL with Python – A Complete Guide

Shadid Haque

BACKEND HOW-TO



Known for its ease of use and simplicity, Python is one of the most beloved general-purpose programming languages. And GraphQL, a declarative query language for APIs and server runtimes, pairs quite nicely with Python. Unfortunately, there are very few comprehensive learning materials out there that give you a step-by-step breakdown of how to use GraphQL with Python. This article will go over everything you need to know to get up and running with GraphQL API using Python, Flask, and Ariadne.

> You can find the complete code for this post on GitHub.

## Learning objectives

By the end of the article, you should know how to:

- Set up a Python web server with Flask
- Use the Ariadne library to implement GraphQL
- Compose a GraphQL Schema
- Perform queries and mutations against a Python GraphQL API

> **GraphQL vs REST: What problem does GraphQL solve?** If you are completely new to GraphQL and want to know how it differs from a traditional REST API, I recommend reading "What is GraphQL? GraphQL introduction".

## Setting up GraphQL with Python (Flask)

Let's dive into creating our very own GraphQL API with Python. For this demo, we will be using the Flask web server. If you are more accustomed to other frameworks such as Django, you can adapt this codebase to your framework. The basic concepts of GraphQL and Python are more or less the same across various frameworks.

### Creating a new python virtual environment

First of all, let's create a new project and change the directory to the project folder.

```
mkdir graphql-python-api
cd graphql-python-api
```

In Python, best practices are to use a *virtual environment*. We can create a new virtual environment by running the following command.

```
python3 -m venv myapp
```

Next, we have to activate the virtual environment. If you are on a Linux or Mac machine you can run the `source` command with the path of the activate script like shown below.

```
source myapp/bin/activate
```

And if you’re on a windows machine, you can run the following command to activate the virtual environment.

```
myapp/bin/activate.bat
```

### Installing dependencies

Our application relies on the following dependencies:

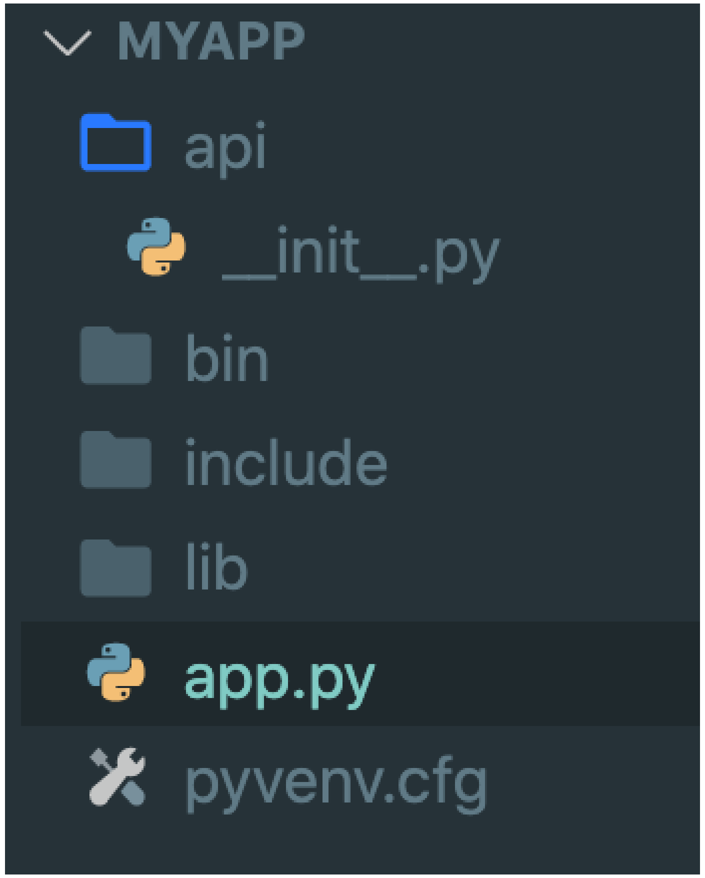
- Flask — this is the web server that we'll use
- Flask-SQLAlchemy — an ORM that makes it easier for us to communicate with our SQL database
- Ariande — a library for GraphQL python integration
- Flask-Cors — an extension for Cross Origin Resource Sharing

You can install them all using a single command:

```
pip install flask ariadne flask-sqlalchemy flask-cors
```

## Up and running with a simple Flask app

We will make the following directory structure. The first file we'll start working with is the `api/__init__.py` file, which will hold all the API-related configuration code.



```
MYAPP
  api
    __init__.py
  bin
  include
  lib
  app.py
  pyvenv.cfg
```

For now, let's populate the `api/__init__.py` with the following code.

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

@app.route('/')
def hello():
    return 'My First API !!'
```

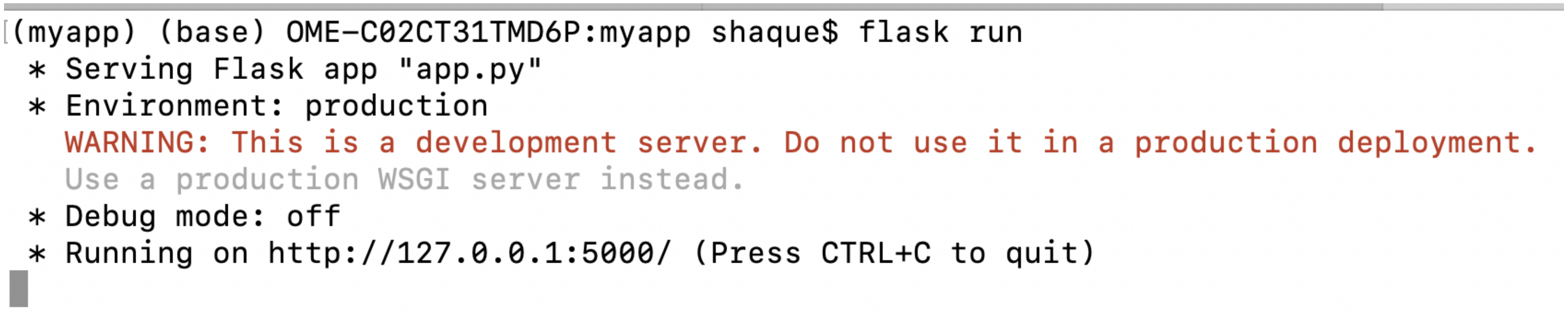
Our `app.py` file is what's responsible for actually starting the flask app. Let's import the flask API instance using the following code:

```
from api import app
```

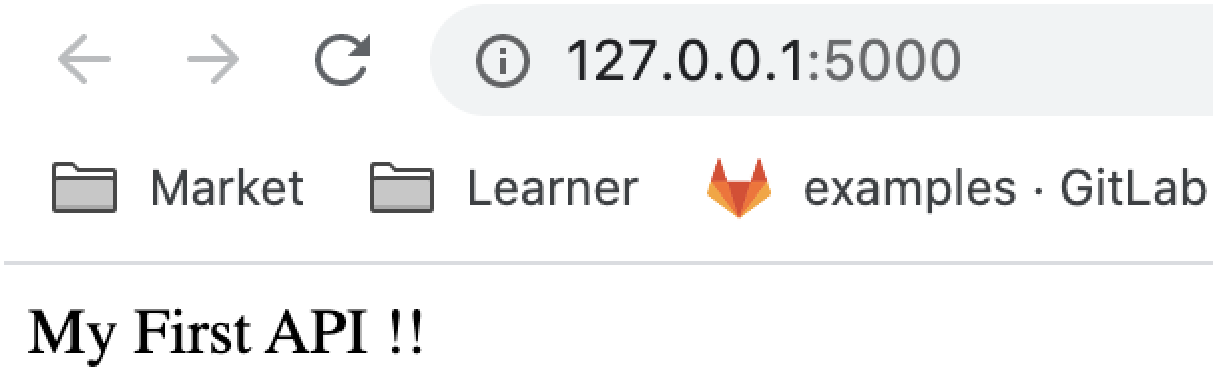
Next, we tell Flask to start the application by looking at our `app.py` file. In the command line, we can accomplish this by setting the `FLASK_APP` environment variable.

```
export FLASK_APP=app.py
```

Finally, we run the app by running the `flask run` command.



```
(myapp) (base) OME-C02CT31TMD6P:myapp shaque$ flask run
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

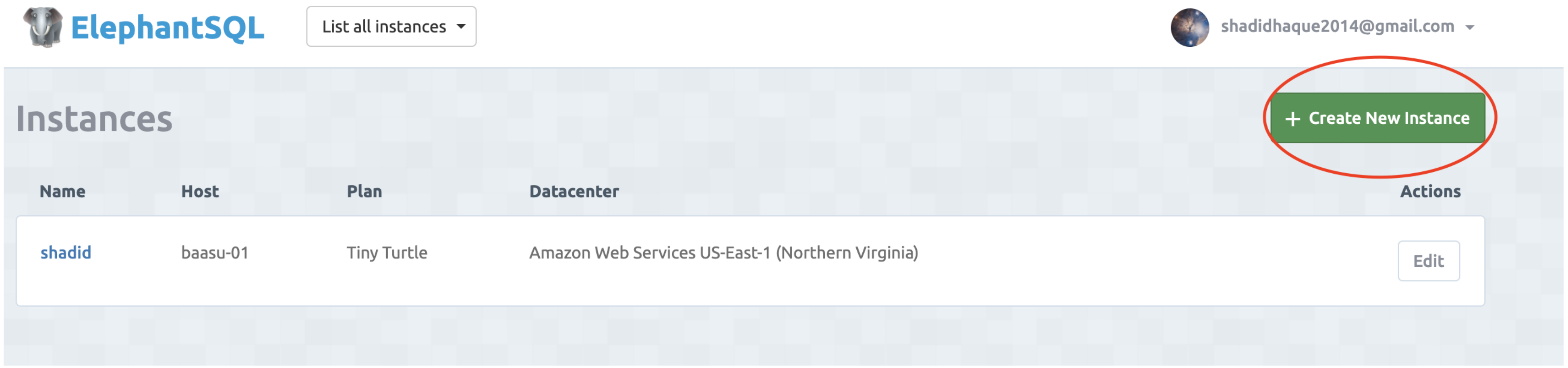


My First API !!

Great! We can see our Flask app up and running. Before we enable it to use GraphQL, lets hook up a database and define some tables.

## Adding a database

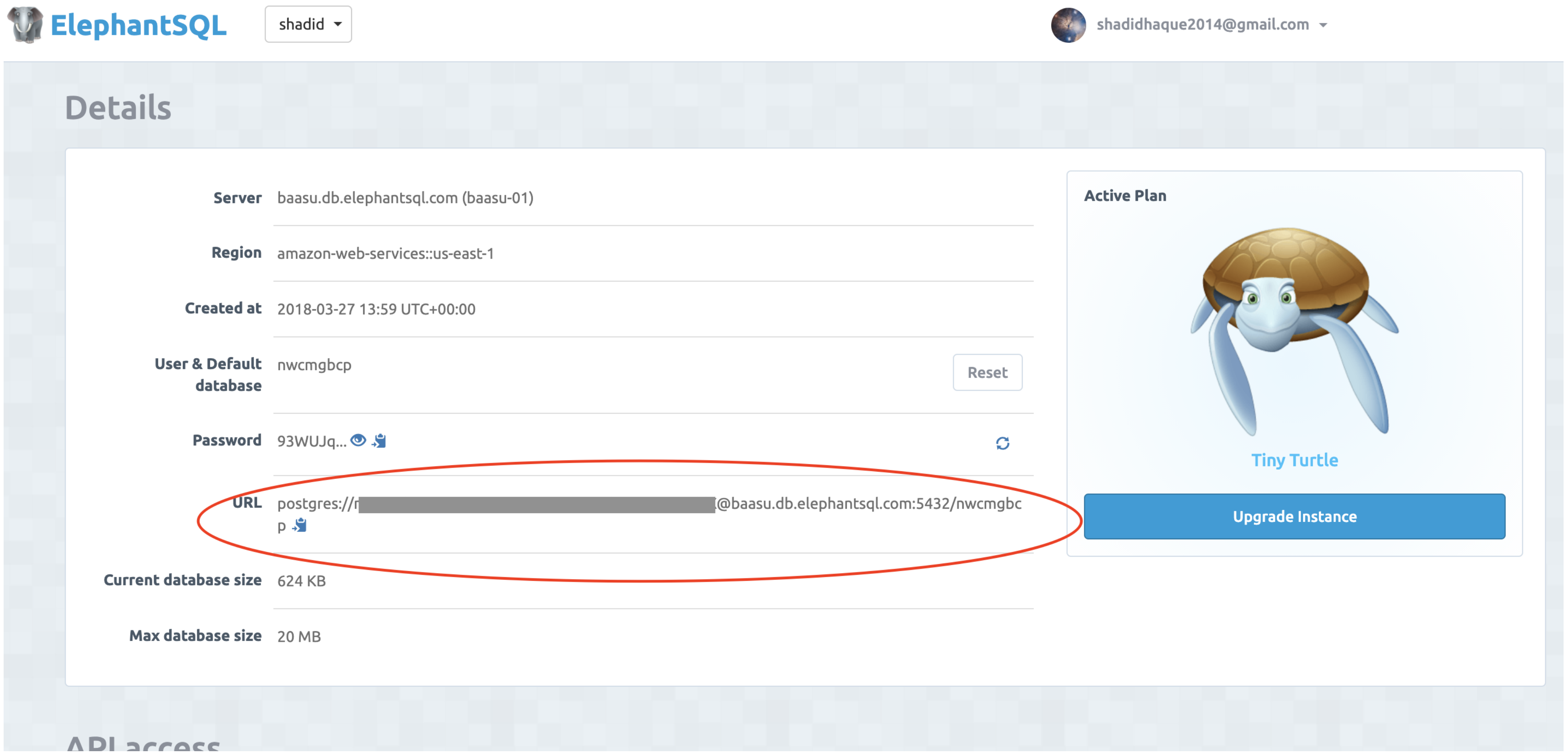
For this example, we are going to be using a Postgres DB instance. I like ElephantSQL, a hosted SQL database, but you can use any SQL database you like.



Instances

| Name | Host | Plan | Datacenter | Actions |
|---|---|---|---|---|
| shadid | baasu-01 | Tiny Turtle | Amazon Web Services US-East-1 (Northern Virginia) | Edit |

In ElephantSQL, once the instance is provisioned on the cloud, we can see the database server information. If you're using ElephantSQL, copy the DB URL as shown below, otherwise, copy the URL to where your SQL database is – whether it's running locally on your machine or with another hosted SQL database service.


ElephantSQL shadid shadidhaque2014@gmail.com

Details

| | |
|---|---|
| Server | baasu.db.elephantsql.com (baasu-01) |
| Region | amazon-web-services::us-east-1 |
| Created at | 2018-03-27 13:59 UTC+00:00 |
| User & Default database | nwcmgbcp |
| Password | 93WUJq... |
| URL | postgres://[redacted]@baasu.db.elephantsql.com:5432/nwcmgbcp |
| Current database size | 624 KB |
| Max database size | 20 MB |

We can now add this database url to the __**init**__.py as shown below.

```
from flask import Flask
from flask_cors import CORS
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
CORS(app)

app.config["SQLALCHEMY_DATABASE_URI"] = "postgresql://mycreds.db.elephantsql.com:543
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db = SQLAlchemy(app)

@app.route('/')
def hello():
    return 'My First API !!'
```

Restart the server and make sure everything is working as usual.

## Creating a model

Next, let’s create our first model. In our database, we are going to have a `Post` table. A `Post` will have a unique `id`, a `title`, `description`, and the `date` it was created. Create an `api/models.py` file and create a new class called `Post` as shown below.

```
from app import db

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String)
    description = db.Column(db.String)
    created_at = db.Column(db.Date)
    def to_dict(self):
        return {
            "id": self.id,
            "title": self.title,
            "description": self.description,
            "created_at": str(self.created_at.strftime('%d-%m-%Y'))
        }
```

We can update our app.py file to include the current models and database settings.

```
from api import app, db
from api import models
```

At this point, we can use the Python interactive terminal to create our table and add some records to it. Let’s do that.

First, let’s open the Python terminal by running the following command.

```
python
```

Once inside the Python terminal, run the following command to create our table.

```
>>> from app import db
>>> db.create_all()
```

On the first line, we import the database instance. On line 2, we run the `create_all()` method to create related tables based on the model we specified earlier.

> Troubleshoot: If you are using Mac for development, you might run into an issue where python can not find `psycopg`. To resolve this, run `pip install psycopg2-binary` within your virtual environment.

To verify whether the table got created or not, hop into the `psql` database console and run the following `SQL` query to get the name of all available tables.

```
SELECT table_name
  FROM information_schema.tables
 WHERE table_schema='public'
   AND table_type='BASE TABLE';
```

Let’s add a few posts to the Post table directly from Python command prompt.

```
>>> from datetime import datetime
>>> from api.models import Post
>>> current_date = datetime.today().date()
>>> new_post = Post(title="A new morning", description="A new morning details", crea
>>> db.session.add(new_post)
>>> db.session.commit()
```

With a working web API connected to the database, we're ready to integrate GraphQL into the server.

## Writing the GraphQL Schema

A schema in GraphQL describes the shape of our data graph. It is the core of any GraphQL server implementation. It defines the functionality available to the client applications that consumes the API. GraphQL has its own language (GraphQL Schema Definition Language) that is used to write the schema. The schema determines what resources the clients can query and update.

`Learn more about GraphQL schemas here.`

Let’s go ahead and create a new file called `schema.graphql` in our root directory. Copy and paste the following code in the file.

```graphql
schema {
    query: Query
}
type Post {
    id: ID!
    title: String!
    description: String!
    created_at: String!
}
type PostResult {
    success: Boolean!
    errors: [String]
    post: Post
}
type PostsResult {
    success: Boolean!
    errors: [String]
    post: [Post]
}
type Query {
    listPosts: PostsResult!
    getPost(id: ID!): PostResult!
}
```

First of all, we have a schema type defined in the top. This determines what type of operations we can perform from our clients. For now, we can only do a `Query` operation.

Next, observe the `Post` type. You will notice that the structure of the `Post` type is identical to our `Post` model that we defined earlier. The `PostsResult` type defines the structure of the response object when we query for all the posts in the database.

Similarly, `PostResult` represents the response when we query for one post in the database.

Finally, we have the type `Query.` This type defines the query operations that our clients can perform. Currently, we have two queries: a `listPosts` query to grab all the posts from the database, and a `getPost` query to get a particular post by its `id`.

## Wiring up Flask server and GraphQL with Ariadne library

Thus far, we have our Flask server up and running, we connected to a database, and created our first GraphQL schema. Next, we need to wire up our server with GraphQL, so that we can start using the queries/mutations defined in the schema. We will be using the Ariadne library to do this.

Ariadne is a lightweight Python library that lets you get up and running with GraphQL quickly. Ariadne is framework agnostic (which means you can use it with Flask, Django, or any other framework of your choice) and it uses a ***schema first*** approach to GraphQL API development. In this approach, we define our schema first (as we did for this demo app) and write the business logic based on our schema.

Another popular pattern is to use ***code first approach*** while designing GraphQL APIs. Graphene is a popular library that uses this approach. In ***code first approach*** we start by writing the resolvers first. Then from the resolvers we can generate our schema.

> Interested in learning more about schema first approach vs code first approach? we recommend you give this article a read.

Let’s go and make the following changes in our `app.py` file.

```python
from api import app, db
from ariadne import load_schema_from_path, make_executable_schema, \
    graphql_sync, snake_case_fallback_resolvers, ObjectType
from ariadne.constants import PLAYGROUND_HTML
from flask import request, jsonify

type_defs = load_schema_from_path("schema.graphql")
schema = make_executable_schema(
    type_defs, snake_case_fallback_resolvers
)

@app.route("/graphql", methods=["GET"])
def graphql_playground():
    return PLAYGROUND_HTML, 200

@app.route("/graphql", methods=["POST"])
def graphql_server():
    data = request.get_json()
    success, result = graphql_sync(
        schema,
        data,
        context_value=request,
        debug=app.debug
    )
    status_code = 200 if success else 400
    return jsonify(result), status_code
```

On lines 2 ~ 4 we import a couple functions from the Ariadne library. On line 7, we import the types from our GraphQL schema. Then, we call the `make_executable_schema` method from Ariadne. We pass the type definitions as the first argument. The second argument `snake_case_fallback_resolvers` is a Bindable; these are special types from Ariadne library that is used to bind python methods to GraphQL schema.

> Learn more about Bindables.

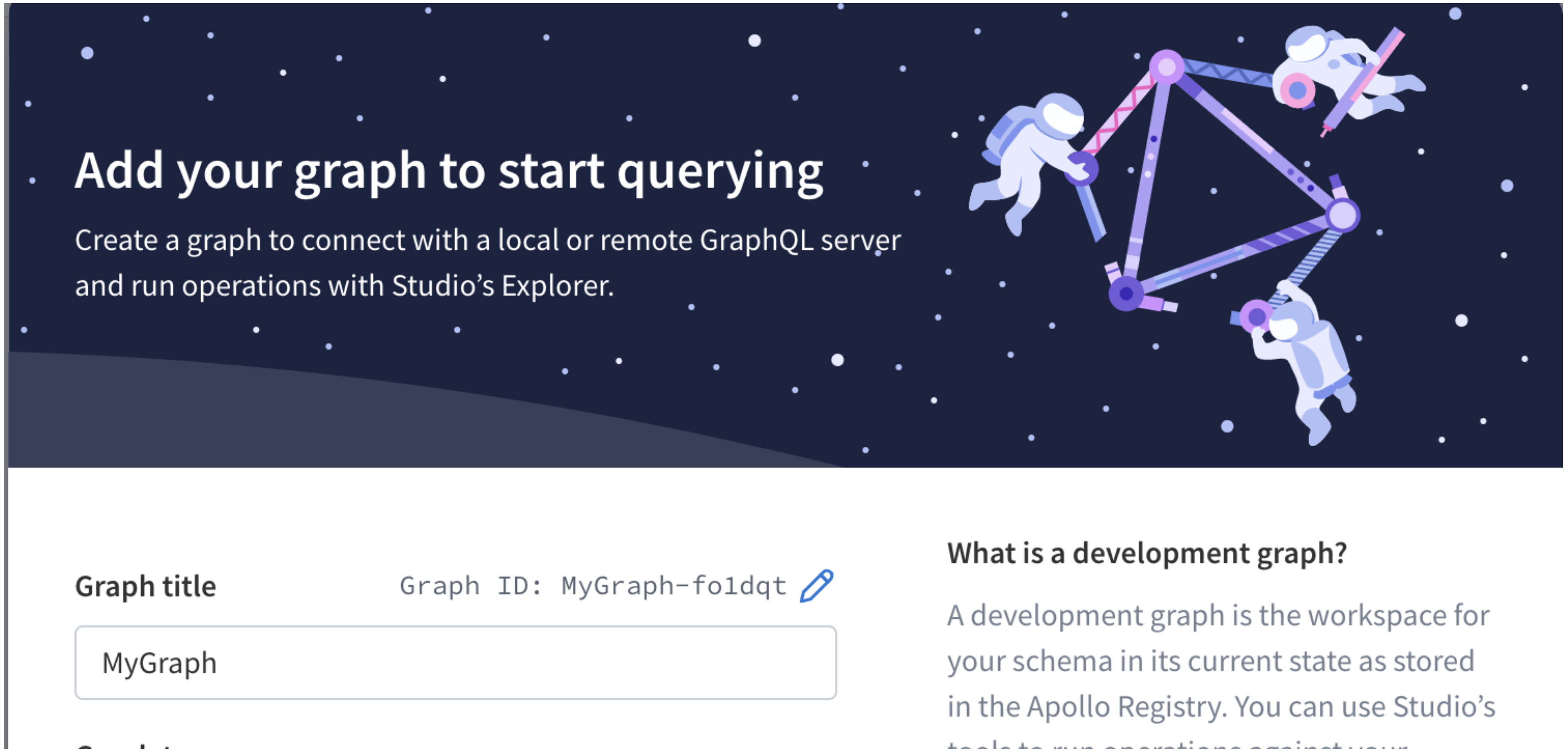
Next, we have two methods. The first method will load up the GraphQL user interface for us. The second method is a `POST` method. This endpoint is will be used by our clients to run queries and mutations.

## Testing our server

We can run the application by running `flask run`. Next, we need a GraphQL IDE to build our queries, explore schemas and test the API functionality. We will be using the Apollo Explorer IDE. It is a ***free to use*** GraphQL IDE built specifically for improving the developer experience around creating GraphQL APIs.

To get started, head over to studio.apollographql.com/dev and create an account (using either GitHub or your email). Choose a name for our graph, and select the `development` option as the graph type.

We will add our localhost endpoint `http://localhost:5000/graphql` in the endpoint field and click create graph.

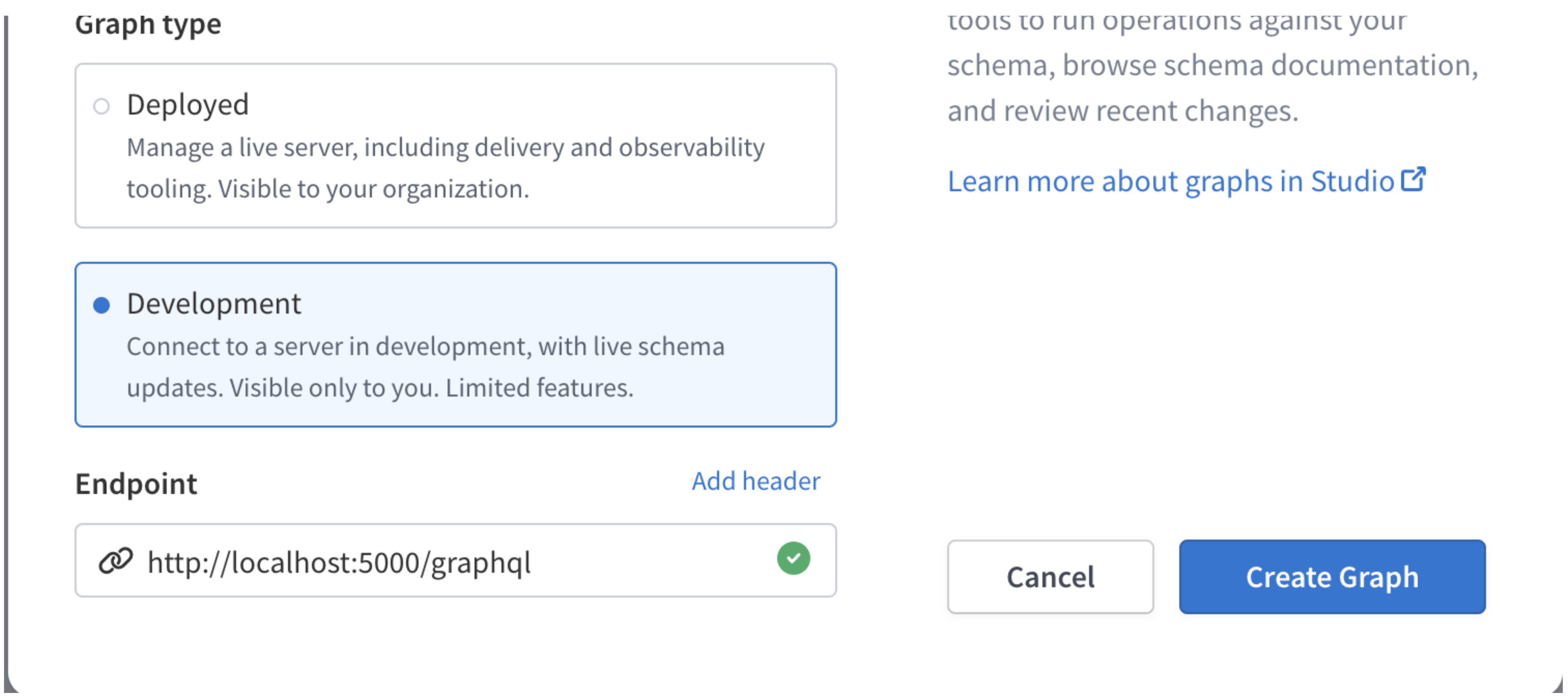

Add your graph to start querying
Create a graph to connect with a local or remote GraphQL server and run operations with Studio's Explorer.

Graph title Graph ID: MyGraph-fo1dqt

MyGraph

What is a development graph?

A development graph is the workspace for your schema in its current state as stored in the Apollo Registry. You can use Studio's



Graph type

○ Deployed
Manage a live server, including delivery and observability tooling. Visible to your organization.

● Development
Connect to a server in development, with live schema updates. Visible only to you. Limited features.

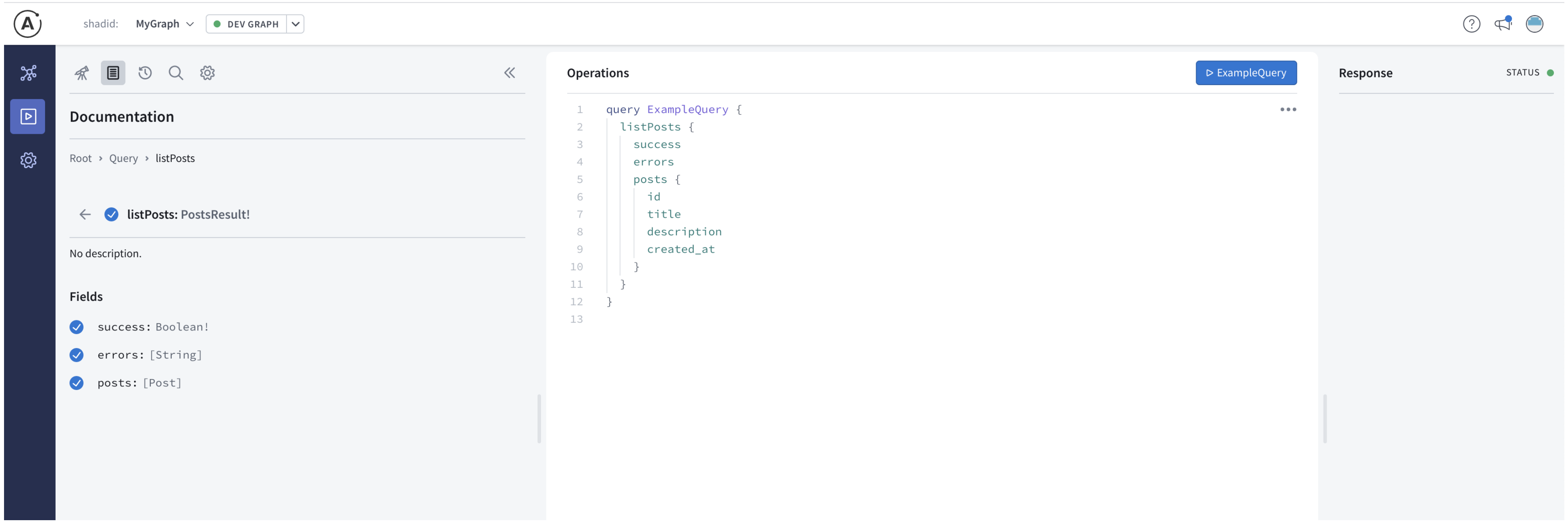
Endpoint

http://localhost:5000/graphql

tools to run operations against your schema, browse schema documentation, and review recent changes.

Learn more about graphs in Studio

Once the setup is done we will see that the GraphQL IDE will load up in our browser.



listPosts: PostsResult!

No description.

Fields

success: Boolean!
errors: [String]
posts: [Post]

```
query ExampleQuery {
  listPosts {
    success
    errors
    posts {
      id
      title
      description
      created_at
    }
  }
}
```

## Query all posts

We are still not able to run queries. Let’s change that. We will write our first query resolver that will return all the posts in the database.

## Writing our first Resolver

We can create a new file called `api/queries.py` and write the following resolver method as shown below.

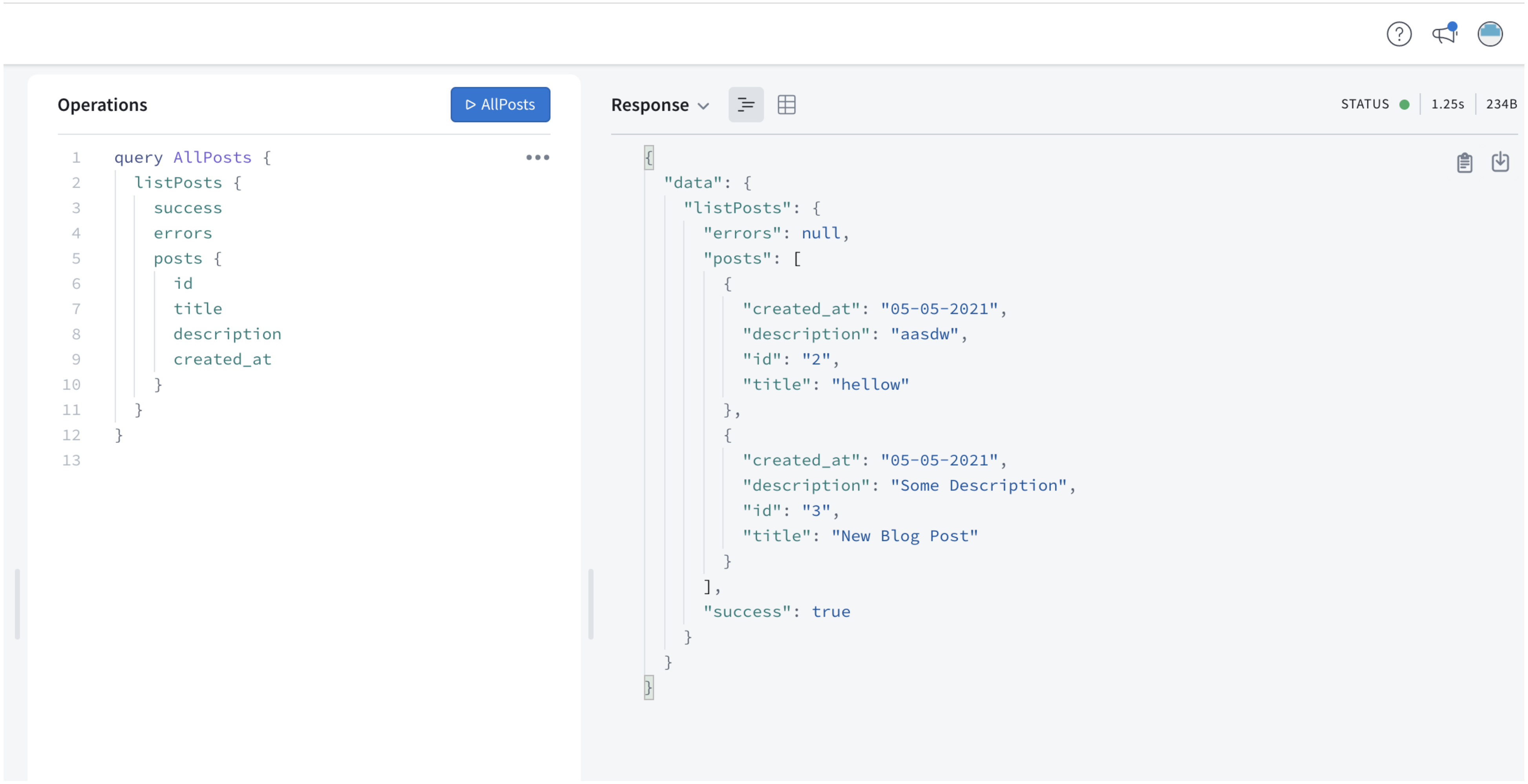
```
from .models import Post
def listPosts_resolver(obj, info):
    try:
        posts = [post.to_dict() for post in Post.query.all()]
        print(posts)
        payload = {
            "success": True,
            "posts": posts
        }
    except Exception as error:
        payload = {
            "success": False,
            "errors": [str(error)]
        }
    return payload
```

This resolver method is very self explanatory. We are trying to query all the Posts from the database and return them as a Payload dictionary.  We have to reference this resolver in our `app.py` file.  Let’s make the following changes to `app.py` file.

```python
from api import app, db
from ariadne import load_schema_from_path, make_executable_schema, \
    graphql_sync, snake_case_fallback_resolvers, ObjectType
from ariadne.constants import PLAYGROUND_HTML
from flask import request, jsonify
from api.queries import listPosts_resolver

query = ObjectType("Query")
query.set_field("listPosts", listPosts_resolver)

type_defs = load_schema_from_path("schema.graphql")
schema = make_executable_schema(
    type_defs, query, snake_case_fallback_resolvers
)
@app.route("/graphql", methods=["GET"])
def graphql_playground():
    return PLAYGROUND_HTML, 200

@app.route("/graphql", methods=["POST"])
def graphql_server():
    data = request.get_json()
    success, result = graphql_sync(
        schema,
        data,
        context_value=request,
        debug=app.debug
    )
    status_code = 200 if success else 400
    return jsonify(result), status_code
```

On line 6, we are importing the resolver. We are then creating a query instance and specifying the query field and the corresponding resolver (line 9). Finally, we are adding the query instance to the make_executable_schema method call as a parameter. Restart the server, go back to the GraphQL playground and you will be able to run the following query.

```
query AllPosts {
  listPosts {
    success
    errors
    posts {
      id
      title
      description
      created_at
    }
  }
}
```



```
query AllPosts {
  listPosts {
    success
    errors
    posts {
      id
      title
      description
      created_at
    }
  }
}
```

```
{
  "data": {
    "listPosts": {
      "errors": null,
      "posts": [
        {
          "created_at": "05-05-2021",
          "description": "aasdw",
          "id": "2",
          "title": "hellow"
        },
        {
          "created_at": "05-05-2021",
          "description": "Some Description",
          "id": "3",
          "title": "New Blog Post"
        }
      ],
      "success": true
    }
  }
}
```

## Querying a single post by id

Next, we will take a look at how we can query a single item by a property. For this example, we will query a `Post` by its id.

We will create a new resolver method inside our `queries.py` file.

```python
from ariadne import convert_kwargs_to_snake_case
...

@convert_kwargs_to_snake_case
def getPost_resolver(obj, info, id):
    try:
        post = Post.query.get(id)
        payload = {
            "success": True,
            "post": post.to_dict()
        }
    except AttributeError:  # todo not found
        payload = {
            "success": False,
            "errors": ["Post item matching {id} not found"]
        }
    return payload
```

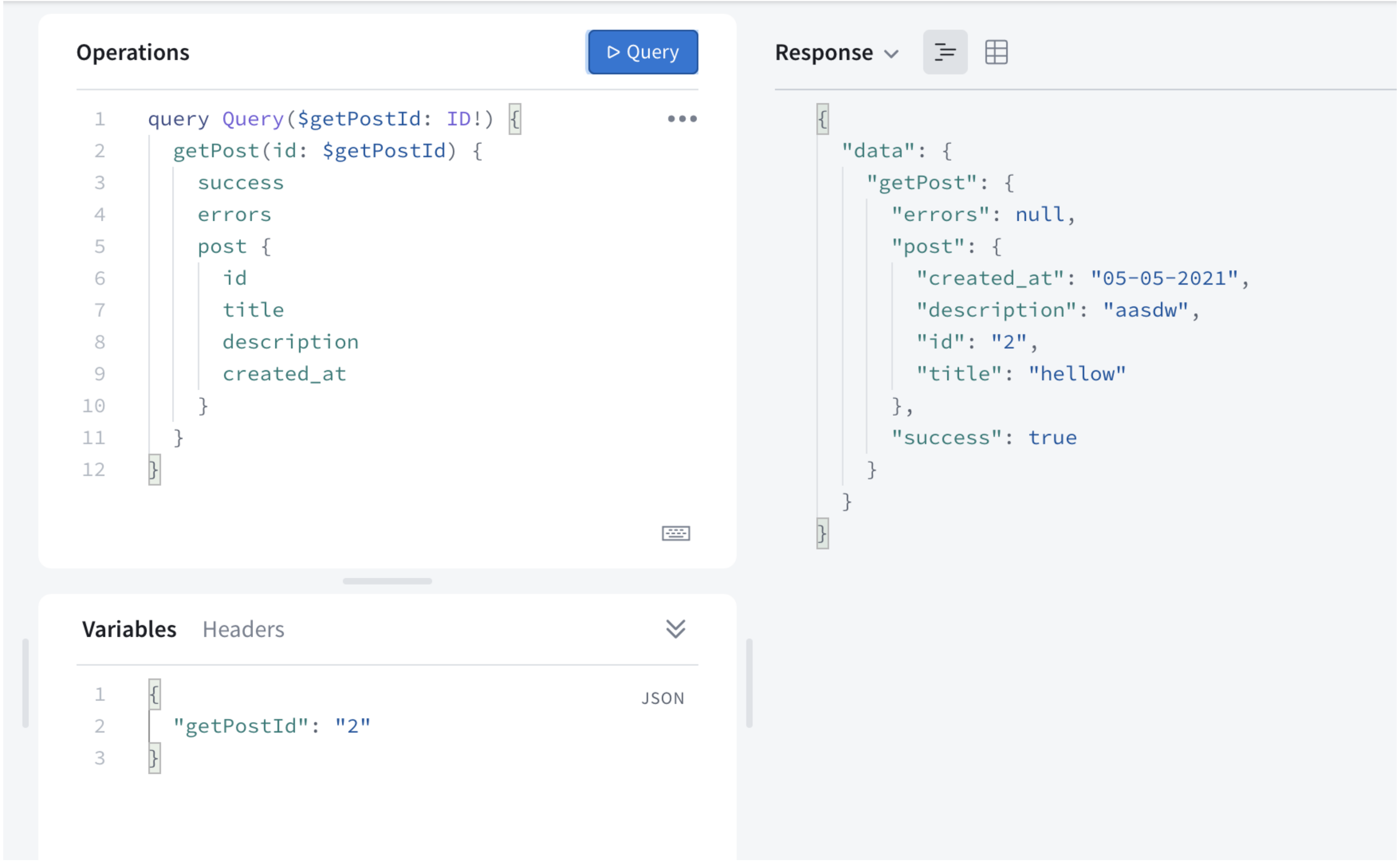
We imported a decorator called `convert_kwargs_to_snake_case` from Ariadne. This decorator converts the method arguments from camel case to snake case.  Let's update the `app.py` file to include the latest resolver

```
...
from api.queries import listPosts_resolver, getPost_resolver
query = ObjectType("Query")
query.set_field("listPosts", listPosts_resolver)
query.set_field("getPost", getPost_resolver)
...
```

We can run the query and verify if everything is working as expected.

```
query GetPost {
  getPost(id: "1") {
    post {
      id
      title
      description
    }
```

```
      success
      errors
    }
  }
```



```
query Query($getPostId: ID!) {
  getPost(id: $getPostId) {
    success
    errors
    post {
      id
      title
      description
      created_at
    }
  }
}
```

```
{
  "getPostId": "2"
}
```

```
{
  "data": {
    "getPost": {
      "errors": null,
      "post": {
        "created_at": "05-05-2021",
        "description": "aasdw",
        "id": "2",
        "title": "hellow"
      },
      "success": true
    }
  }
}
```

## Mutation

Mutations are used to create, update or delete records from the database. Let’s set up our first mutation.

### Creating a new post

First of all, in our schema, we need to define the type of mutation we are trying to add. In our case, we want to create a new post. Therefore, we will make a mutation called createPost.

```
// schema.graphql
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    createPost(title: String!, description: String!, created_at: String): PostResult
}
...
```

We updated our schema.graphql file accordingly as shown above. We add a new **Mutation** type. We specify the mutation name, required parameters and finally update schema type to include Mutation type. Updating the schema itself will not do much. We need a resolver to correspond to the `createPost` mutation in the schema.

We will create a new file called `api/mutations.py`. All our mutation resolvers will live in this file.

```
# mutations.py
from datetime import date
from ariadne import convert_kwargs_to_snake_case
from api import db
from api.models import Post

@convert_kwargs_to_snake_case
def create_post_resolver(obj, info, title, description):
    try:
        today = date.today()
        post = Post(
            title=title, description=description, created_at=today.strftime("%b-%d-%
        )
        db.session.add(post)
        db.session.commit()
        payload = {
                "success": True,
                "post": post.to_dict()
            }
    except ValueError:  # date format errors
        payload = {
            "success": False,
            "errors": [f"Incorrect date format provided. Date should be in "
                       f"the format dd-mm-yyyy"]
        }
    return payload
```

The resolver method is pretty self-explanatory. We are here trying to create and save a new instance of a `Post`. On success, we return the post. We also need to bind this new mutation resolver in our app.py.

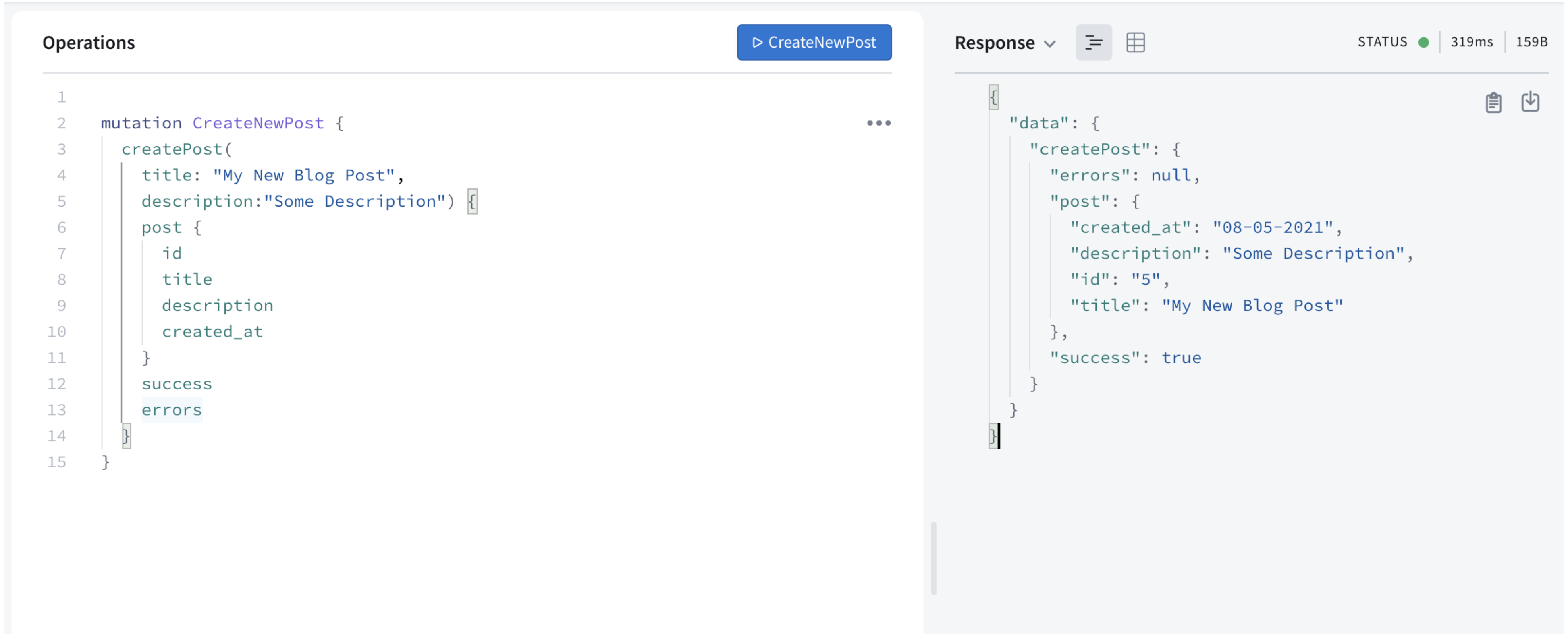
```
...
from api.queries import listPosts_resolver, getPost_resolver
from api.mutations import create_post_resolver
query = ObjectType("Query")
mutation = ObjectType("Mutation")
query.set_field("listPosts", listPosts_resolver)
query.set_field("getPost", getPost_resolver)
mutation.set_field("createPost", create_post_resolver)
```

```
type_defs = load_schema_from_path("schema.graphql")
schema = make_executable_schema(
    type_defs, query, mutation, snake_case_fallback_resolvers
)
..
```

As you can see from the code example above, importing and binding the mutation follows the same pattern as importing and binding queries that we have done previously.  we can now hop into the GraphQL playground and try to execute this new mutation.

```
mutation CreateNewPost {
  createPost(
    title: "New Blog Post",
    description:"Some Description") {
    post {
      id
      title
      description
      created_at
    }
    success
```

```
      errors
    }
}
```



```
mutation CreateNewPost {
  createPost(
    title: "My New Blog Post",
    description:"Some Description") {
    post {
      id
      title
      description
      created_at
    }
    success
    errors
  }
}
```

```
{
  "data": {
    "createPost": {
      "errors": null,
      "post": {
        "created_at": "08-05-2021",
        "description": "Some Description",
        "id": "5",
        "title": "My New Blog Post"
      },
      "success": true
    }
  }
}
```

## Updating a post

Next, we will be looking at updating a post. To do so we will follow the same pattern. First, we will update the schema and add a new mutation called `updatePost`.

```
type Mutation {
    createPost(title: String!, description: String!, created_at: String): PostResult
    updatePost(id: ID!, title: String, description: String): PostResult!
}
```

`updatePost` takes in a mandatory parameter id and optional parameters title and description. Now we can create a resolver for this mutation.

```
# mutations.py
...
@convert_kwargs_to_snake_case
def update_post_resolver(obj, info, id, title, description):
    try:
        post = Post.query.get(id)
        if post:
            post.title = title
            post.description = description
        db.session.add(post)
        db.session.commit()
        payload = {
            "success": True,
            "post": post.to_dict()
        }
    except AttributeError:  # todo not found
        payload = {
            "success": False,
            "errors": ["item matching id {id} not found"]
        }
    return payload
```

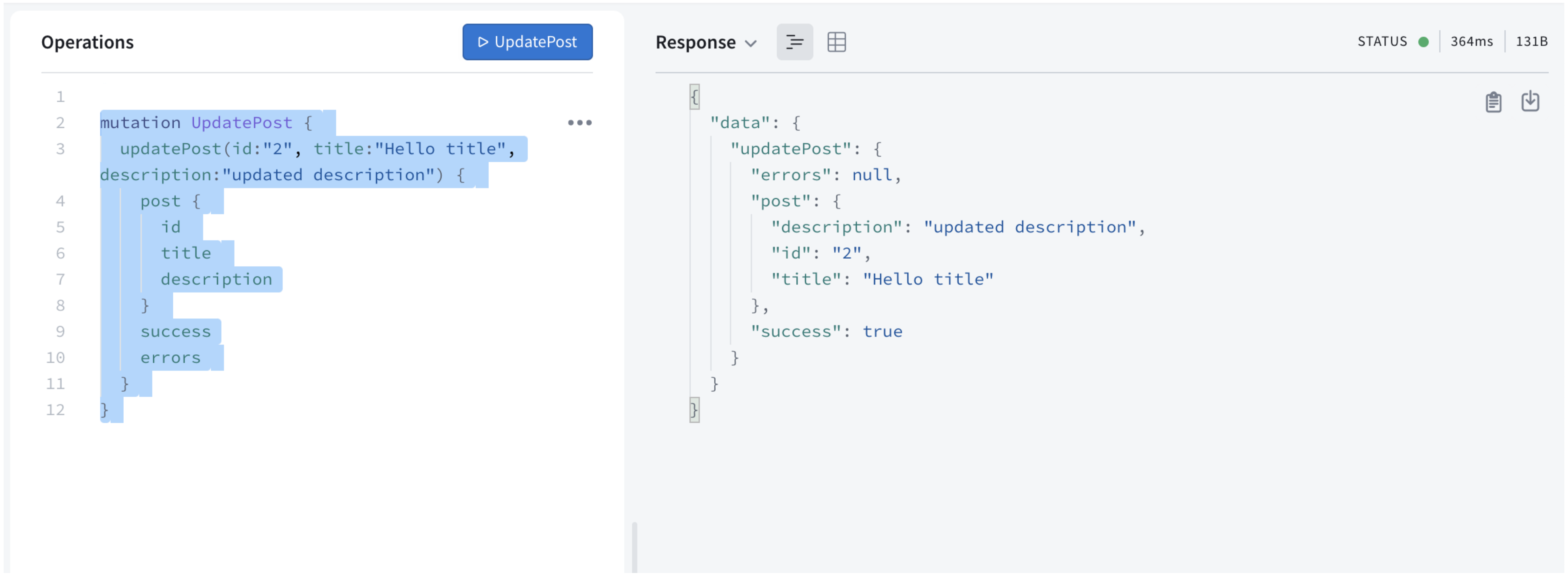
In this method, we are querying the post by id and updating the title and description of the post. We can wire this new resolver up in `app.py` like the previous one.

```
...
from api.mutations import create_post_resolver, update_post_resolver

mutation = ObjectType("Mutation")
```

```
mutation.set_field("createPost", create_post_resolver)
mutation.set_field("updatePost", update_post_resolver)
```

That’s it. We can restart the server and run the `updatePost` mutation now.

```
mutation UpdatePost {
  updatePost(id:"2", title:"Hello title", description:"updated description") {
    post {
      id
      title
      description
    }
    success
    errors
  }
}
```



```
mutation UpdatePost {
  updatePost(id:"2", title:"Hello title", description:"updated description") {
    post {
      id
      title
      description
    }
    success
    errors
  }
}
```

```
{
  "data": {
    "updatePost": {
      "errors": null,
      "post": {
        "description": "updated description",
        "id": "2",
        "title": "Hello title"
      },
      "success": true
    }
  }
}
```

## Deleting a post

Finally, let's take a look how we can delete a post. We are going to exactly the same thing as we did with updatePost mutation. We will first create the deletePost mutation in the schema.

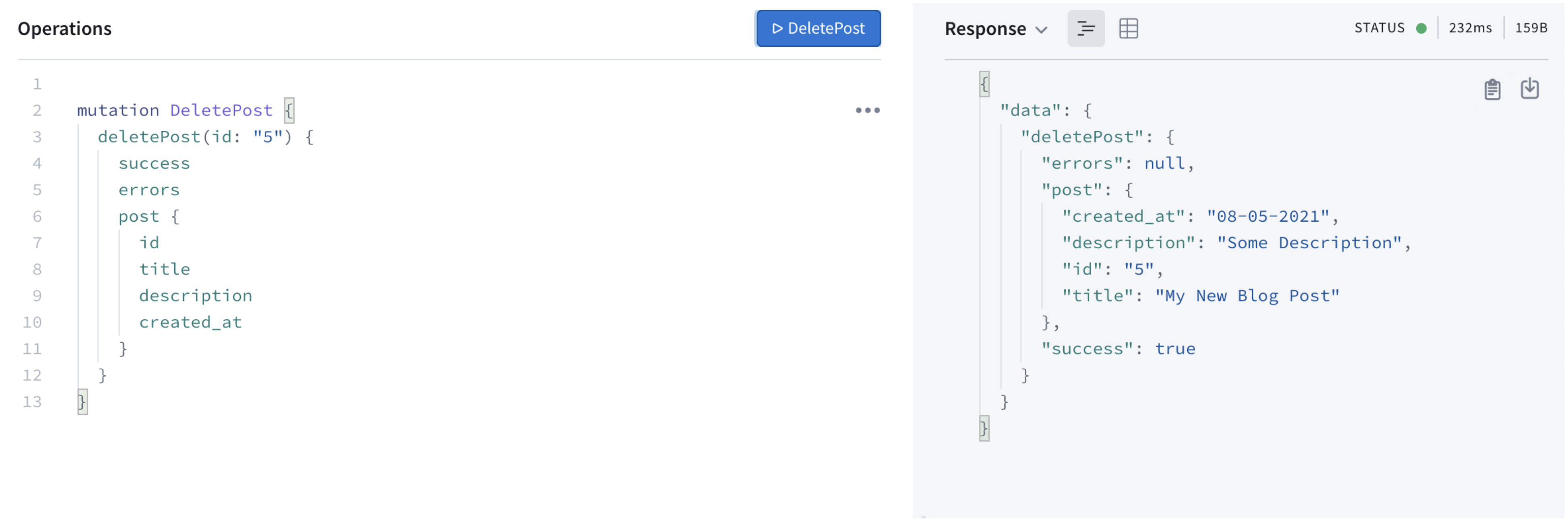
```
type Mutation {
    createPost(title: String!, description: String!, created_at: String): PostResult
    updatePost(id: ID!, title: String, description: String): PostResult!
    deletePost(id: ID): PostResult!
}
```

Once that is done we can create a new resolver for it and reference it in the app.py file.

```
# mutations.py
...
@convert_kwargs_to_snake_case
def delete_post_resolver(obj, info, id):
    try:
        post = Post.query.get(id)
        db.session.delete(post)
        db.session.commit()
        payload = {"success": True, "post": post.to_dict()}
    except AttributeError:
        payload = {
            "success": False,
            "errors": ["Not found"]
        }
    return payload
```

```
# app.py
...
from api.mutations import create_post_resolver, update_post_resolver, delete_post_re
...
mutation.set_field("deletePost", delete_post_resolver)
```

Let’s test the functionality.



```
mutation DeletePost {
  deletePost(id: "5") {
    success
    errors
    post {
      id
      title
      description
      created_at
    }
  }
}
```

```
{
  "data": {
    "deletePost": {
      "errors": null,
      "post": {
        "created_at": "08-05-2021",
        "description": "Some Description",
        "id": "5",
        "title": "My New Blog Post"
      },
      "success": true
    }
  }
}
```

Awesome, we now have our GraphQL and Python API up and running.

## Final thoughts

The main intention of this article was to get you up and running with GraphQL and Python, as well as introduce some widely used patterns and best practices. I hope you found this article informative.

This is just the start. I suggest checking out some of the other posts on the Apollo blog on topics like caching, GraphQL security, and if you're really into Python, checking out the rest of the Ariadne documentation.

That's a wrap! Happy hacking and see you next time.

WRITTEN BY

**Shadid Haque**

Read more by Shadid Haque →

## Stay in our orbit!

Become an Apollo insider and get first access to new features, best practices, and community events. Oh, and no junk mail. Ever.

Enter your email

Subscribe

## Make this article better!

Was this post helpful? Have suggestions? Consider leaving feedback so we can improve it for future readers ✨.

## SIMILAR POSTS

MAY 7, 2021

**Why You Should Disable GraphQL Introspection In Production – GraphQL Security**

by Khalil Stemmler

APRIL 22, 2021

**Testing Apollo Client Applications**

by Khalil Stemmler

APRIL 6, 2021

**Unblocking teams to go faster with Apollo Federation**

by Matt DeBergalis

**Company**

About Us

Platform

Enterprise

Pricing

Customers

Careers

Team

## Community

Blog

Docs

GraphQL Summit

## Help

Get Support

Terms of Service

Privacy Policy