Graphene Documentation

Release 1.0.dev

Syrus Akbary

May 14, 2023

Contents

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Examples . . . 3

2 Inheritance Examples **7**
- 2.1 Create interfaces from inheritance relationships . . . 7
- 2.2 Eager Loading & Using with AsyncSession . . . 9

3 Relay **11**

4 Tips **13**
- 4.1 Querying . . . 13
- 4.2 Sorting . . . 13

5 Schema Examples **15**
- 5.1 Search all Models with Union . . . 15

6 SQLAlchemy + Flask Tutorial **17**
- 6.1 Setup the Project . . . 17
- 6.2 Defining our models . . . 17
- 6.3 Schema . . . 18
- 6.4 Creating GraphQL and GraphiQL views in Flask . . . 19
- 6.5 Creating some data . . . 20
- 6.6 Testing our GraphQL schema . . . 20

7 API Reference **21**
- 7.1 SQLAlchemyObjectType . . . 21
- 7.2 SQLAlchemyInterface . . . 21
- 7.3 ORMField . . . 21
- 7.4 SQLAlchemyConnectionField . . . 21

Contents:

CHAPTER 1

Getting Started

Welcome to the graphene-sqlalchemy documentation! Graphene is a powerful Python library for building GraphQL APIs, and SQLAlchemy is a popular ORM (Object-Relational Mapping) tool for working with databases. When combined, graphene-sqlalchemy allows developers to quickly and easily create a GraphQL API that seamlessly interacts with a SQLAlchemy-managed database. It is fully compatible with SQLAlchemy 1.4 and 2.0. This documentation provides detailed instructions on how to get started with graphene-sqlalchemy, including installation, setup, and usage examples.

1.1 Installation

To install `graphene-sqlalchemy`, just run this command in your shell:

```
pip install --pre "graphene-sqlalchemy"
```

1.2 Examples

Here is a simple SQLAlchemy model:

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class UserModel(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    last_name = Column(String)
```

To create a GraphQL schema for it, you simply have to write the following:

```
import graphene
from graphene_sqlalchemy import SQLAlchemyObjectType

class User(SQLAlchemyObjectType):
    class Meta:
        model = UserModel
        # use `only_fields` to only expose specific fields ie "name"
        # only_fields = ("name",)
        # use `exclude_fields` to exclude specific fields ie "last_name"
        # exclude_fields = ("last_name",)

class Query(graphene.ObjectType):
    users = graphene.List(User)

    def resolve_users(self, info):
        query = User.get_query(info)  # SQLAlchemy query
        return query.all()

schema = graphene.Schema(query=Query)
```

Then you can simply query the schema:

```
query = '''
    query {
      users {
        name,
        lastName
      }
    }
'''
result = schema.execute(query, context_value={'session': db_session})
```

It is important to provide a session for graphene-sqlalchemy to resolve the models. In this example, it is provided using the GraphQL context. See *Tips* for other ways to implement this.

You may also subclass SQLAlchemyObjectType by providing `abstract = True` in your subclasses Meta:

```
from graphene_sqlalchemy import SQLAlchemyObjectType

class ActiveSQLAlchemyObjectType(SQLAlchemyObjectType):
    class Meta:
        abstract = True

    @classmethod
    def get_node(cls, info, id):
        return cls.get_query(info).filter(
            and_(cls._meta.model.deleted_at==None,
                 cls._meta.model.id==id)
            ).first()

class User(ActiveSQLAlchemyObjectType):
    class Meta:
        model = UserModel

class Query(graphene.ObjectType):
    users = graphene.List(User)

    def resolve_users(self, info):
```

(continues on next page)

(continued from previous page)

```
        query = User.get_query(info)  # SQLAlchemy query
        return query.all()

schema = graphene.Schema(query=Query)
```

More complex inhertiance using SQLAlchemy's polymorphic models is also supported. You can check out *Inheritance Examples* for a guide.

CHAPTER 2

Inheritance Examples

2.1 Create interfaces from inheritance relationships

Note: If you're using *AsyncSession*, please check the chapter *Eager Loading & Using with AsyncSession*.

SQLAlchemy has excellent support for class inheritance hierarchies. These hierarchies can be represented in your GraphQL schema by means of interfaces. Much like ObjectTypes, Interfaces in Graphene-SQLAlchemy are able to infer their fields and relationships from the attributes of their underlying SQLAlchemy model:

```
from sqlalchemy import Column, Date, Integer, String
from sqlalchemy.ext.declarative import declarative_base

import graphene
from graphene import relay
from graphene_sqlalchemy import SQLAlchemyInterface, SQLAlchemyObjectType

Base = declarative_base()

class Person(Base):
    id = Column(Integer(), primary_key=True)
    type = Column(String())
    name = Column(String())
    birth_date = Column(Date())

    __tablename__ = "person"
    __mapper_args__ = {
        "polymorphic_on": type,
    }

class Employee(Person):
    hire_date = Column(Date())
```

(continues on next page)

(continued from previous page)

```
    __mapper_args__ = {
        "polymorphic_identity": "employee",
    }

class Customer(Person):
    first_purchase_date = Column(Date())

    __mapper_args__ = {
        "polymorphic_identity": "customer",
    }

class PersonType(SQLAlchemyInterface):
    class Meta:
        model = Person

class EmployeeType(SQLAlchemyObjectType):
    class Meta:
        model = Employee
        interfaces = (relay.Node, PersonType)

class CustomerType(SQLAlchemyObjectType):
    class Meta:
        model = Customer
        interfaces = (relay.Node, PersonType)
```

Keep in mind that *PersonType* is a *SQLAlchemyInterface*. Interfaces must be linked to an abstract Model that does not specify a *polymorphic_identity*, because we cannot return instances of interfaces from a GraphQL query. If Person specified a *polymorphic_identity*, instances of Person could be inserted into and returned by the database, potentially causing Persons to be returned to the resolvers.

When querying on the base type, you can refer directly to common fields, and fields on concrete implementations using the ... *on* syntax:

```
people {
    name
    birthDate
    ... on EmployeeType {
        hireDate
    }
    ... on CustomerType {
        firstPurchaseDate
    }
}
```

Danger: When using joined table inheritance, this style of querying may lead to unbatched implicit IO with negative performance implications. See the chapter *Eager Loading & Using with AsyncSession* for more information on eager loading all possible types of a *SQLAlchemyInterface*.

Please note that by default, the "polymorphic_on" column is *not* generated as a field on types that use polymorphic inheritance, as this is considered an implementation detail. The idiomatic way to retrieve the concrete GraphQL type of an object is to query for the *__typename* field. To override this behavior, an *ORMField* needs to be created for the custom type field on the corresponding *SQLAlchemyInterface*. This is *not recommended* as it promotes abiguous schema design

If your SQLAlchemy model only specifies a relationship to the base type, you will need to explicitly pass your concrete

implementation class to the Schema constructor via the *types*= argument:

```
schema = graphene.Schema(..., types=[PersonType, EmployeeType, CustomerType])
```

See also: Graphene Interfaces

2.2 Eager Loading & Using with AsyncSession

When querying the base type in multi-table inheritance or joined table inheritance, you can only directly refer to polymorphic fields when they are loaded eagerly. This restricting is in place because AsyncSessions don't allow implicit async operations such as the loads of the joined tables. To load the polymorphic fields eagerly, you can use the *with_polymorphic* attribute of the mapper args in the base model:

```
class Person(Base):
    id = Column(Integer(), primary_key=True)
    type = Column(String())
    name = Column(String())
    birth_date = Column(Date())

    __tablename__ = "person"
    __mapper_args__ = {
        "polymorphic_on": type,
        "with_polymorphic": "*", # needed for eager loading in async session
    }
```

Alternatively, the specific polymorphic fields can be loaded explicitly in resolvers:

```
class Query(graphene.ObjectType):
    people = graphene.Field(graphene.List(PersonType))

    async def resolve_people(self, _info):
        return (await session.scalars(with_polymorphic(Person, [Engineer,␣
↪Customer]))).all()
```

Dynamic batching of the types based on the query to avoid eager is currently not supported, but could be implemented in a future PR.

For more information on loading techniques for polymorphic models, please check out the SQLAlchemy docs.

CHAPTER 3

Relay

`graphene-sqlalchemy` comes with pre-defined connection fields to quickly create a functioning relay API. Using the `SQLAlchemyConnectionField`, you have access to relay pagination, sorting and filtering (filtering is coming soon!).

To be used in a relay connection, your `SQLAlchemyObjectType` must implement the `Node` interface from `graphene.relay`. This handles the creation of the `Connection` and `Edge` types automatically.

The following example creates a relay-paginated connection:

```
class Pet(Base):
    __tablename__ = 'pets'
    id = Column(Integer(), primary_key=True)
    name = Column(String(30))
    pet_kind = Column(Enum('cat', 'dog', name='pet_kind'), nullable=False)


class PetNode(SQLAlchemyObjectType):
    class Meta:
        model = Pet
        interfaces=(Node,)


class Query(ObjectType):
    all_pets = SQLAlchemyConnectionField(PetNode.connection)
```

To disable sorting on the connection, you can set `sort` to `None` the `SQLAlchemyConnectionField`:

```
class Query(ObjectType):
    all_pets = SQLAlchemyConnectionField(PetNode.connection, sort=None)
```

CHAPTER 4

Tips

4.1 Querying

In order to make querying against the database work, there are two alternatives:

- Set the db session when you do the execution:

```
schema = graphene.Schema()
schema.execute(context_value={'session': session})
```

- Create a query for the models.

```
Base = declarative_base()
Base.query = db_session.query_property()

class MyModel(Base):
    # ...
```

If you don't specify any, the following error will be displayed:

```
A query in the model Base or a session in the schema is required for querying.
```

4.2 Sorting

By default the SQLAlchemyConnectionField sorts the result elements over the primary key(s). The query has a *sort* argument which allows to sort over a different column(s)

Given the model

```
class Pet(Base):
    __tablename__ = 'pets'
    id = Column(Integer(), primary_key=True)
    name = Column(String(30))
```

(continues on next page)

(continued from previous page)

```
    pet_kind = Column(Enum('cat', 'dog', name='pet_kind'), nullable=False)


class PetNode(SQLAlchemyObjectType):
    class Meta:
        model = Pet


class Query(ObjectType):
    allPets = SQLAlchemyConnectionField(PetNode.connection)
```

some of the allowed queries are

- Sort in ascending order over the *name* column

```
allPets(sort: name_asc){
    edges {
        node {
            name
        }
    }
}
```

- Sort in descending order over the *per_kind* column and in ascending order over the *name* column

```
allPets(sort: [pet_kind_desc, name_asc]) {
    edges {
        node {
            name
            petKind
        }
    }
}
```

CHAPTER 5

Schema Examples

5.1 Search all Models with Union

```
class Book(SQLAlchemyObjectType):
    class Meta:
        model = BookModel
        interfaces = (relay.Node,)


class Author(SQLAlchemyObjectType):
    class Meta:
        model = AuthorModel
        interfaces = (relay.Node,)


class SearchResult(graphene.Union):
    class Meta:
        types = (Book, Author)


class Query(graphene.ObjectType):
    node = relay.Node.Field()
    search = graphene.List(SearchResult, q=graphene.String())  # List field for␣
↪search results

    # Normal Fields
    all_books = SQLAlchemyConnectionField(Book.connection)
    all_authors = SQLAlchemyConnectionField(Author.connection)

    def resolve_search(self, info, **args):
        q = args.get("q")  # Search query

        # Get queries
        bookdata_query = BookData.get_query(info)
```

(continues on next page)

(continued from previous page)

```
        author_query = Author.get_query(info)

        # Query Books
        books = bookdata_query.filter((BookModel.title.contains(q)) |
                                      (BookModel.isbn.contains(q)) |
                                      (BookModel.authors.any(AuthorModel.name.
↪contains(q)))).all()

        # Query Authors
        authors = author_query.filter(AuthorModel.name.contains(q)).all()

        return authors + books  # Combine lists

schema = graphene.Schema(query=Query, types=[Book, Author, SearchResult])
```

Example GraphQL query

```
book(id: "Qm9vazow") {
    id
    title
}
search(q: "Making Games") {
    __typename
    ... on Author {
        fname
        lname
    }
    ... on Book {
        title
        isbn
    }
}
```

CHAPTER 6

SQLAlchemy + Flask Tutorial

Graphene comes with builtin support to SQLAlchemy, which makes quite easy to operate with your current models.

Note: The code in this tutorial is pulled from the Flask SQLAlchemy example app.

6.1 Setup the Project

We will setup the project, execute the following:

```
# Create the project directory
mkdir flask_sqlalchemy
cd flask_sqlalchemy

# Create a virtualenv to isolate our package dependencies locally
virtualenv env
source env/bin/activate  # On Windows use `env\Scripts\activate`

# SQLAlchemy and Graphene with SQLAlchemy support
pip install SQLAlchemy
pip install graphene_sqlalchemy

# Install Flask and GraphQL Flask for exposing the schema through HTTP
pip install Flask
pip install Flask-GraphQL
```

6.2 Defining our models

Let's get started with these models:

```
# flask_sqlalchemy/models.py
from sqlalchemy import *
from sqlalchemy.orm import (scoped_session, sessionmaker, relationship,
                            backref)
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///database.sqlite3', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

Base = declarative_base()
# We will need this for querying
Base.query = db_session.query_property()


class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String)


class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    hired_on = Column(DateTime, default=func.now())
    department_id = Column(Integer, ForeignKey('department.id'))
    department = relationship(
        Department,
        backref=backref('employees',
                        uselist=True,
                        cascade='delete,all'))
```

6.3 Schema

GraphQL presents your objects to the world as a graph structure rather than a more hierarchical structure to which you may be accustomed. In order to create this representation, Graphene needs to know about each *type* of object which will appear in the graph.

This graph also has a *root type* through which all access begins. This is the `Query` class below. In this example, we provide the ability to list all employees via `all_employees`, and the ability to obtain a specific node via `node`.

Create `flask_sqlalchemy/schema.py` and type the following:

```
# flask_sqlalchemy/schema.py
import graphene
from graphene import relay
from graphene_sqlalchemy import SQLAlchemyObjectType, SQLAlchemyConnectionField
from .models import db_session, Department as DepartmentModel, Employee as␣
↪EmployeeModel


class Department(SQLAlchemyObjectType):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
        model = DepartmentModel
        interfaces = (relay.Node, )


class Employee(SQLAlchemyObjectType):
    class Meta:
        model = EmployeeModel
        interfaces = (relay.Node, )


class Query(graphene.ObjectType):
    node = relay.Node.Field()
    # Allows sorting over multiple columns, by default over the primary key
    all_employees = SQLAlchemyConnectionField(Employee.connection)
    # Disable sorting over this field
    all_departments = SQLAlchemyConnectionField(Department.connection, sort=None)

schema = graphene.Schema(query=Query)
```

6.4 Creating GraphQL and GraphiQL views in Flask

Unlike a RESTful API, there is only a single URL from which GraphQL is accessed.

We are going to use Flask to create a server that expose the GraphQL schema under `/graphql` and a interface for querying it easily: GraphiQL (also under `/graphql` when accessed by a browser).

Fortunately for us, the library `Flask-GraphQL` that we previously installed makes this task quite easy.

```
# flask_sqlalchemy/app.py
from flask import Flask
from flask_graphql import GraphQLView

from .models import db_session
from .schema import schema, Department

app = Flask(__name__)
app.debug = True

app.add_url_rule(
    '/graphql',
    view_func=GraphQLView.as_view(
        'graphql',
        schema=schema,
        graphiql=True # for having the GraphiQL interface
    )
)

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()

if __name__ == '__main__':
    app.run()
```

6.5 Creating some data

```
$ python

>>> from .models import engine, db_session, Base, Department, Employee
>>> Base.metadata.create_all(bind=engine)

>>> # Fill the tables with some data
>>> engineering = Department(name='Engineering')
>>> db_session.add(engineering)
>>> hr = Department(name='Human Resources')
>>> db_session.add(hr)

>>> peter = Employee(name='Peter', department=engineering)
>>> db_session.add(peter)
>>> roy = Employee(name='Roy', department=engineering)
>>> db_session.add(roy)
>>> tracy = Employee(name='Tracy', department=hr)
>>> db_session.add(tracy)
>>> db_session.commit()
```

6.6 Testing our GraphQL schema

We're now ready to test the API we've built. Let's fire up the server from the command line.

```
$ python ./app.py

 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Go to localhost:5000/graphql and type your first query!

```
{
  allEmployees {
    edges {
      node {
        id
        name
        department {
          name
        }
      }
    }
  }
}
```

CHAPTER 7

API Reference

7.1 SQLAlchemyObjectType

7.2 SQLAlchemyInterface

7.3 ORMField

7.4 SQLAlchemyConnectionField