💧 Mayim: Bring Your Own Query

Intro to the non-ORM SQL client

Adam Hopkins

```
start = datetime(2022, 9, 6, 13, 40, 0, tzinfo=ZoneInfo(key="Asia/Jerusalem"))
end = start + timedelta(minutes=30)
```

```python
class Adam:

    def __init__(self):
        self.work = PacketFabric("Director of Software Engineering")
        self.oss = Sanic("Core Maintainer")
        self.home = Israel("Negev")

    async def run(self, inputs: Union[Pretzels, Coffee]) -> None:
        while True:
            await self.work.do(inputs)
            await self.oss.do(inputs)

    def sleep(self):
        raise NotImplemented
```

- PacketFabric - Network-as-a-Service platform; private access to the cloud; secure connectivity between data centers
- Sanic Framework - Python 3.7+ `asyncio` enabled framework and server. Build fast. Run fast.
- GitHub - /ahopkins
- Twitter - @admhpkns

💧 Mayim: Is it an ORM?

... Yes ⛔

... No ⛔

... Kind of? 🤷‍♂️

```
>>> await executor.select_all_foo()
[<Foo one>, <Foo two>, <Foo three>]
```

What is an ORM?

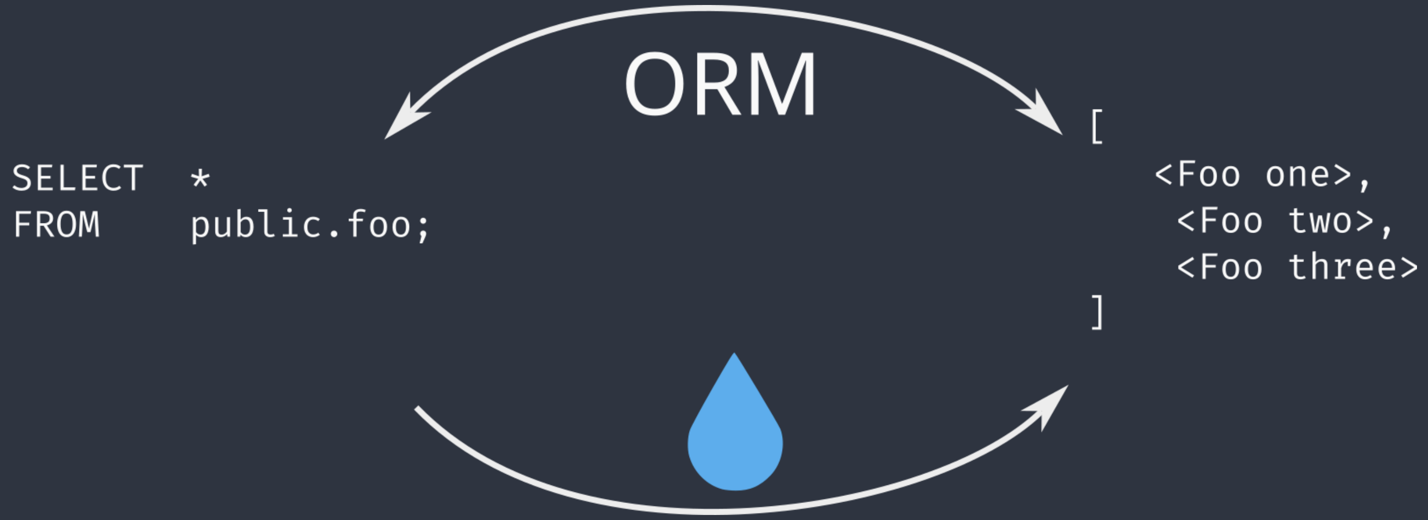
	ORM	Mayim
Connect to remote datasource	✅	
Create connection pool	✅	
Execute queries	✅	
Transaction support	✅	
Maps DB data to Python objects	✅	
Maps Python objects to DB queries	✅	

Why is Mayim NOT an ORM?

	ORM	Mayim
Connect to remote datasource	✅	✅
Create connection pool	✅	✅
Execute queries	✅	✅
Transaction support	✅	✅
Maps DB data to Python objects	✅	🏆
Maps Python objects to DB queries	✅	⛔

💧 Mayim: Is it an ORM?

> Mayim is a BYOQ, NOT ORM query runner and hydrator



```
SELECT  *
FROM    public.foo;
```

```
[
    <Foo one>,
     <Foo two>,
     <Foo three>
]
```

What does it look like?

```
CREATE TABLE department (
  department_id NUMBER NOT NULL PRIMARY KEY,
)
CREATE TABLE employee (
  employee_id NUMBER NOT NULL PRIMARY KEY,
  name VARCHAR NOT NULL,
  department NUMBER NOT NULL FOREIGN KEY REFERENCES department(id)
)
```

What does it look like?

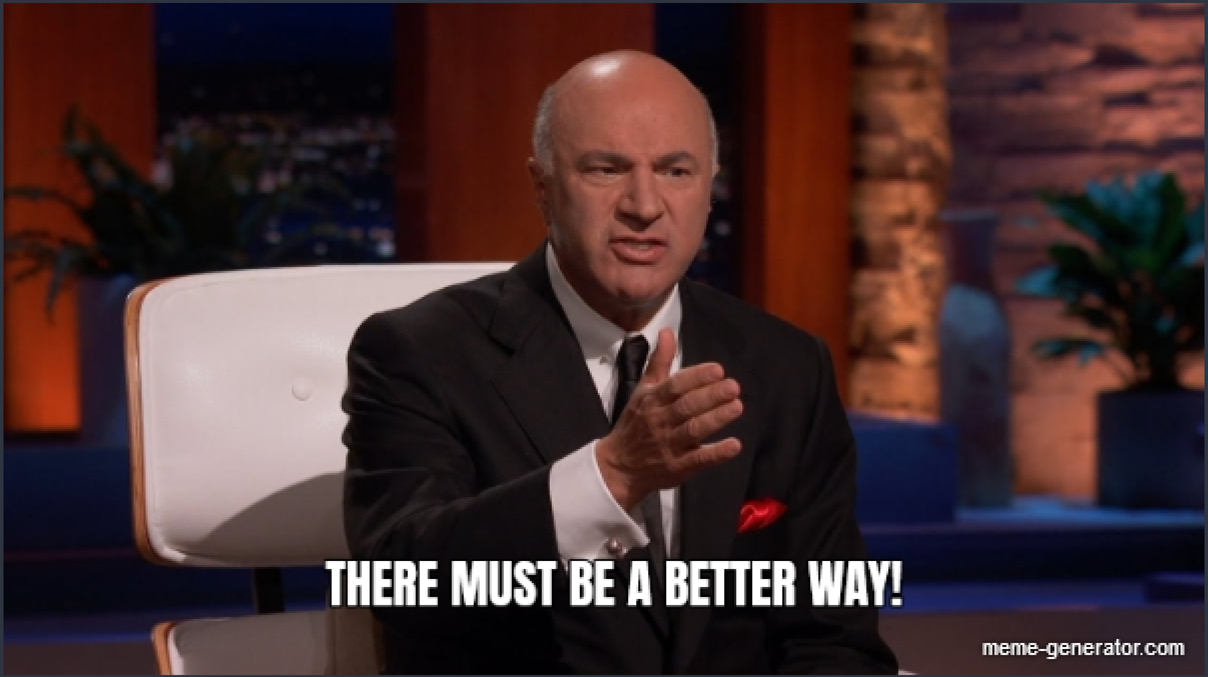
```
CREATE TABLE department (
  department_id NUMBER NOT NULL PRIMARY KEY,
)
CREATE TABLE employee (
  employee_id NUMBER NOT NULL PRIMARY KEY,
  name VARCHAR NOT NULL,
  department NUMBER NOT NULL FOREIGN KEY REFERENCES department(id)
)
```

```
class Department:
    department_id: int
    employees: List[Employee]

    def get(self, department_id): ...


class Employee {
    employee_id: int
    name: str
    department: Department

    def get(self, employee_id): ...
```

```python
import asyncio
from mayim import Mayim, SQLiteExecutor, query
from dataclasses import dataclass


@dataclass
class Person:
    name: str


class PersonExecutor(SQLiteExecutor):
    @query("SELECT $name as name")
    async def select_person(self, name: str) -> Person:
        ...


async def run():
    executor = PersonExecutor()
    Mayim(db_path="./example.db")
    print(await executor.select_person(name="Adam"))


asyncio.run(run())
```

But … writing SQL in strings is not fun 😫

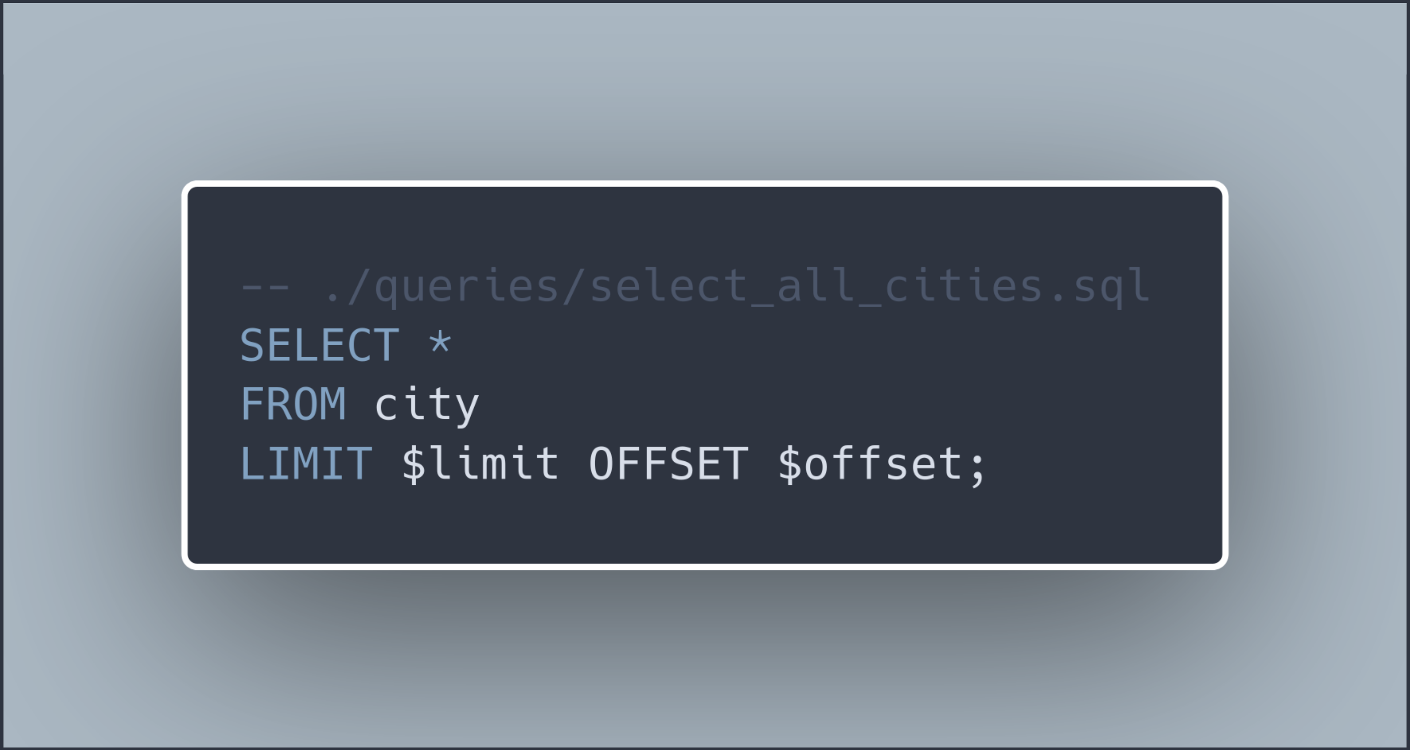
```python
class PersonExecutor(SQLiteExecutor):
    @query("SELECT $name as name")
    async def select_person(self, name: str) -> Person:
        ...
```


THERE MUST BE A BETTER WAY!
meme-generator.com

Sample project structure

```
./project
├── queries
│   └── select_all_cities.sql
└── basic.py
```

First, the SQL



```sql
-- ./queries/select_all_cities.sql
SELECT *
FROM city
LIMIT $limit OFFSET $offset;
```

Second, create a model

```
from dataclasses import dataclass


@dataclass
class City:
    id: int
    name: str
    countrycode: str
    district: str
    population: int
```

Third, define an executor

```python
from mayim import PostgresExecutor


class CityExecutor(PostgresExecutor):
    async def select_all_cities(
        self, limit: int = 4, offset: int = 0
    ) -> List[City]:
        ...
```

Third, define an executor, and run it

```
from mayim import PostgresExecutor


class CityExecutor(PostgresExecutor):
    async def select_all_cities(
        self, limit: int = 4, offset: int = 0
    ) -> List[City]:
        ...
```

```
async def run():
    executor = CityExecutor()
    Mayim(dsn="postgres://postgres:postgres@localhost:5432/world")
    print(await executor.select_all_cities())

asyncio.run(run())
```

```
[
    City(id=1, name='Kabul', countrycode='AFG', district='Kabol', pop
    City(id=2, name='Qandahar', countrycode='AFG', district='Qandahar
    City(id=3, name='Herat', countrycode='AFG', district='Herat', pop
    City(id=4, name='Mazar-e-Sharif', countrycode='AFG', district='Ba
]
```

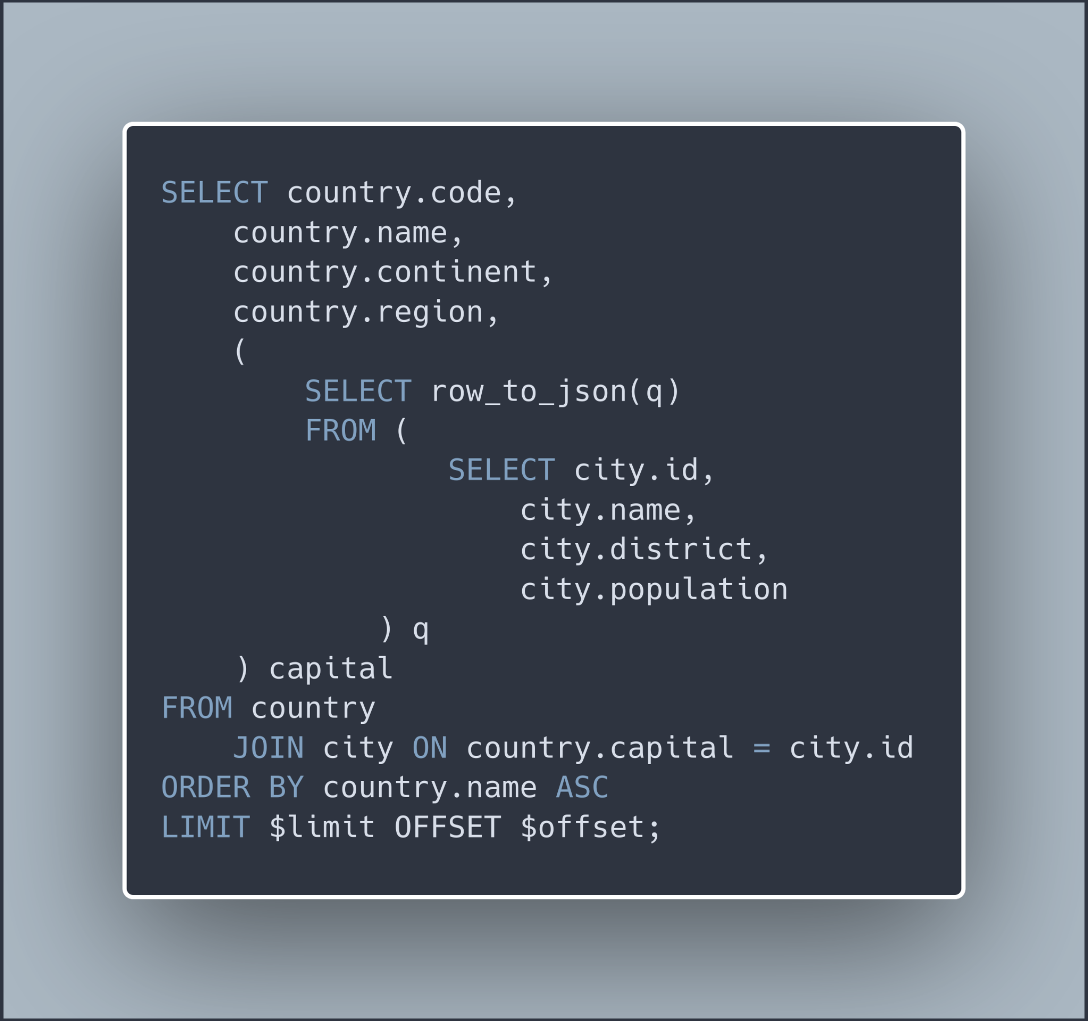
Why write my own SQL?

- ORMs work well *if* strong object model, but breakdown with higher complexity
- ORMs lack easy insight into *what* is happening under the hood
- Every ORM has its own framework, and patterns to be learned
  - avoiding **N+1**
  - pagination
  - object proxies
  - aggregation strategies
- Do you want?
  - foreign data wrappers
  - stored procedures
  - highly nested and complex operations
  - use built-in functions
  - higher control of data access patterns

Why write my own SQL?

```
from pydantic import BaseModel

class City(BaseModel):
    id: int
    name: str
    district: str
    population: int


class Country(BaseModel):
    code: str
    name: str
    continent: str
    region: str
    capital: City
```

Why write my own SQL?

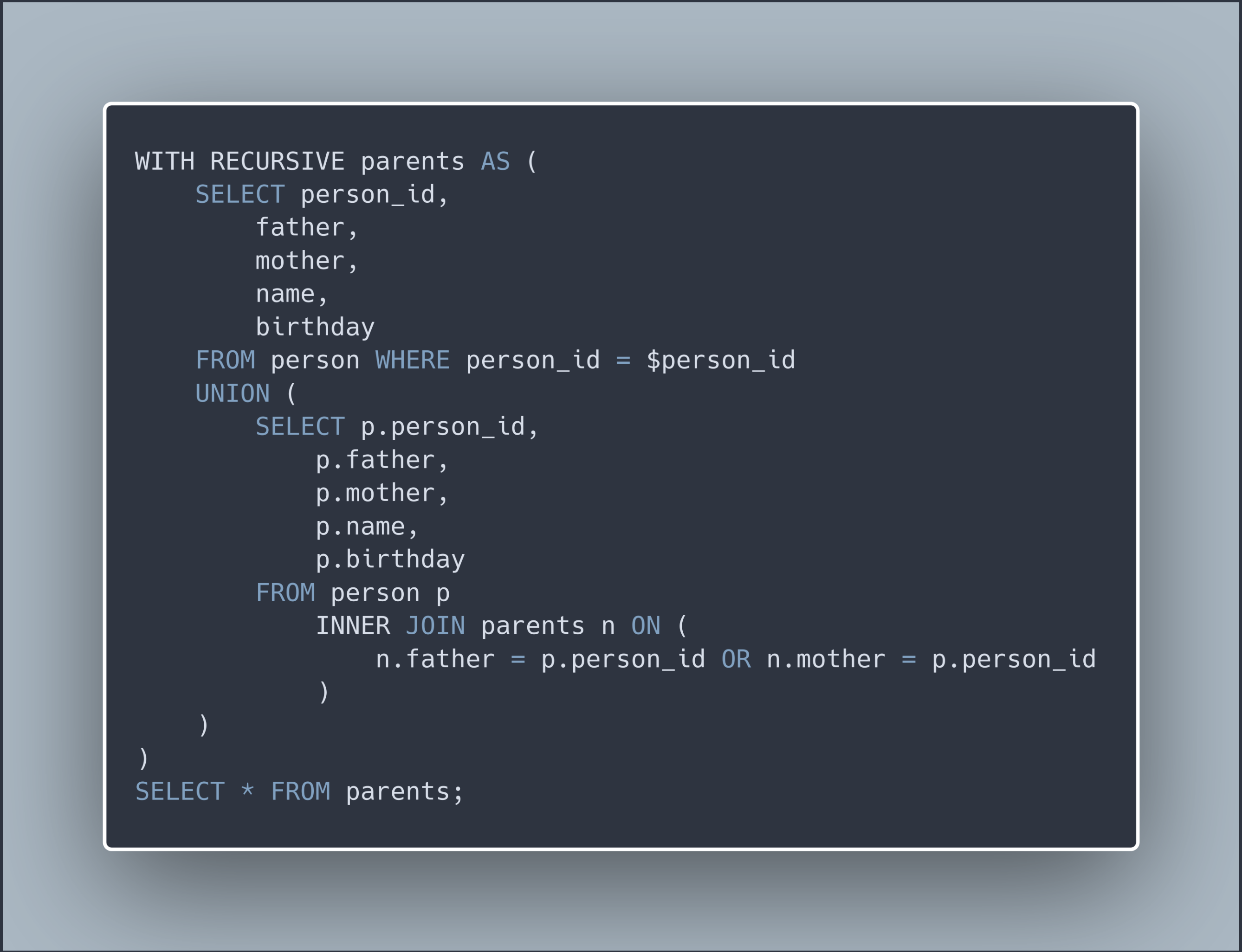


```sql
SELECT country.code,
    country.name,
    country.continent,
    country.region,
    (
        SELECT row_to_json(q)
        FROM (
                SELECT city.id,
                    city.name,
                    city.district,
                    city.population
            ) q
    ) capital
FROM country
    JOIN city ON country.capital = city.id
ORDER BY country.name ASC
LIMIT $limit OFFSET $offset;
```

Why write my own SQL?

```
[
    Country(
        code="AFG",
        name="Afghanistan",
        continent="Asia",
        region="Southern and Central Asia",
        capital=City(id=1, name="Kabul", district="Kabol", population=1780000),
    ),
    Country(
        code="ALB",
        name="Albania",
        continent="Europe",
        region="Southern Europe",
        capital=City(id=34, name="Tirana", district="Tirana", population=270000),
    ),
    Country(
        code="DZA",
        name="Algeria",
        continent="Africa",
        region="Northern Africa",
        capital=City(id=35, name="Alger", district="Alger", population=2168000),
    ),
    ...
]
```

The possibilities are limitless … 😎



```sql
WITH RECURSIVE parents AS (
    SELECT person_id,
        father,
        mother,
        name,
        birthday
    FROM person WHERE person_id = $person_id
    UNION (
        SELECT p.person_id,
            p.father,
            p.mother,
            p.name,
            p.birthday
        FROM person p
            INNER JOIN parents n ON (
                n.father = p.person_id OR n.mother = p.person_id
            )
    )
)
SELECT * FROM parents;
```

This thing called an `Executor`

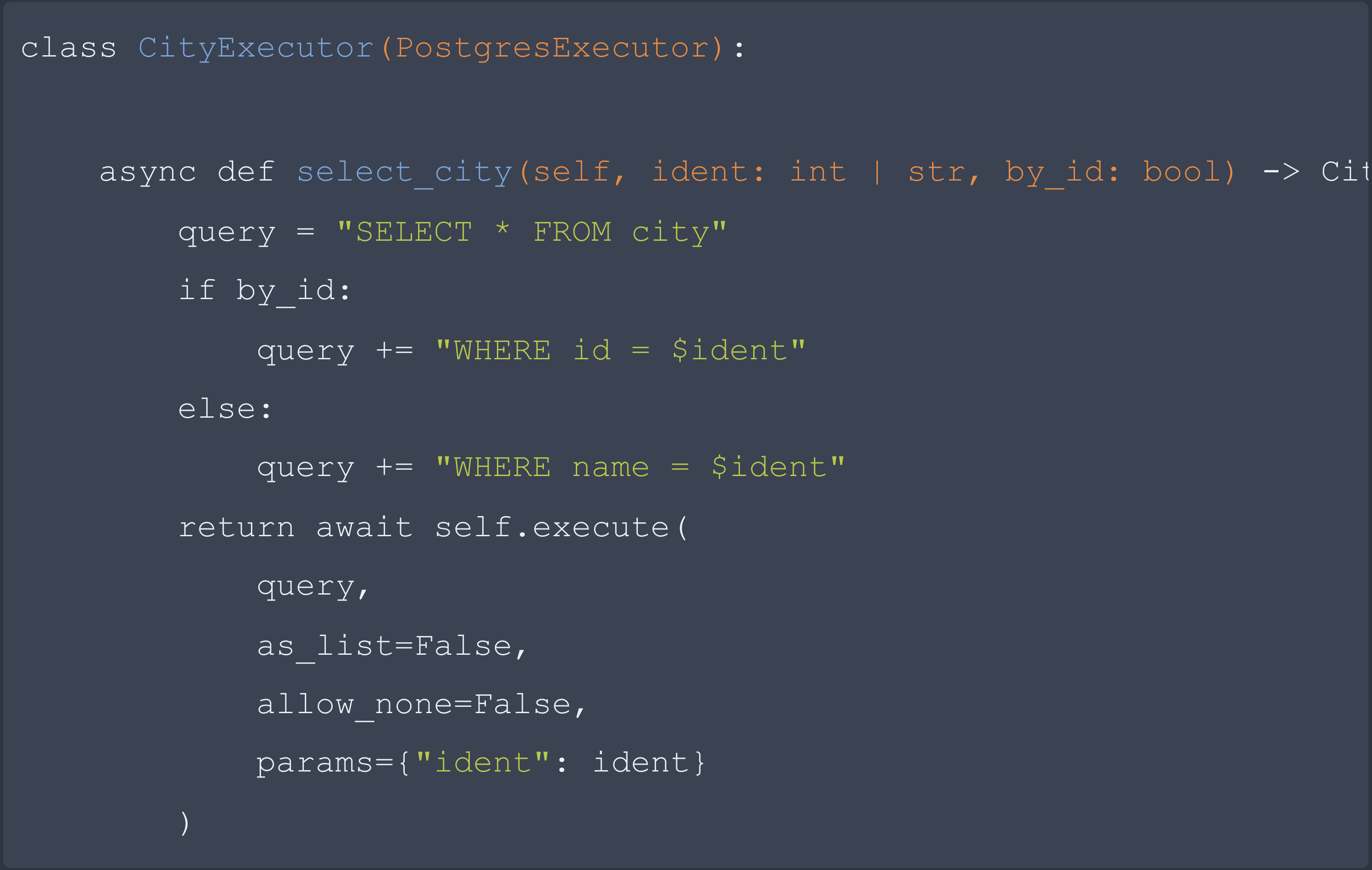
```
from mayim import PostgresExecutor

class CityExecutor(PostgresExecutor):
    async def select_all_cities(
        self, limit: int = 4, offset: int = 0
    ) -> List[City]:
        ...
```

This thing called an `Executor`

```
from mayim import PostgresExecutor

class CityExecutor(PostgresExecutor):
    async def select_all_cities(
        self, limit: int = 4, offset: int = 0
    ) -> List[City]:
        ...
```

GOAL: *To provide SQL execution with **first-class** treatment* 🥇

```python
from mayim import PostgresExecutor

class SomeExecutor(PostgresExecutor):
    verb_prefixes = ...  # Default: select_,insert_,update_,delete_
    generic_prefix = ...  # Default: mayim_
    path = ...  # Default: ./queries
```



```
class CityExecutor(PostgresExecutor):


    async def select_city(self, ident: int | str, by_id: bool) -> Cit
        query = "SELECT * FROM city"
        if by_id:
            query += "WHERE id = $ident"
        else:
            query += "WHERE name = $ident"
        return await self.execute(
            query,
            as_list=False,
            allow_none=False,
            params={"ident": ident}
        )
```

```
class CityExecutor(PostgresExecutor):
    generic_prefix: str = "fragment_"

    async def select_city(self, ident: int | str, by_id: bool) -> Cit
        query = self.get_query("fragment_select_city")
        if by_id:
            query += self.get_query("fragment_where_id")
        else:
            query += self.get_query("fragment_where_name")
        return await self.execute(
            query,
            as_list=False,
            allow_none=False,
            params={"ident": ident}
        )
```

What is a `Hydrator`? 🥤

> An object that turns a `dict` into a model

What is a `Hydrator`? 🥤

> An object that turns a `dict` into a model

```python
from mayim import Hydrator

class CityHydrator(Hydrator):
    def hydrate(
        self, data: Dict[str, Any], model: Type[object] = City
    ) -> City:
        data["population"] = round(data["population"] / 1_000_000, 2)
        return model(**data)
```

💧 Mayim 💗 loves Pydantic

```
import asyncio
from mayim import Mayim, SQLiteExecutor, query
from pydantic import BaseModel


class Person(BaseModel):
    name: str


class PersonExecutor(SQLiteExecutor):
    @query("SELECT $name as name")
    async def select_person(self, name: str) -> Person:
        ...


async def run():
    executor = PersonExecutor()
    Mayim(db_path="./example.db")
    print(await executor.select_person(name="Adam"))


asyncio.run(run())
```

(This script is complete, it should run "as is")

```python
class MarshmallowHydrator(Hydrator):
    def hydrate(self, data: Dict[str, Any], model: Type[Schema] = ...
        schema = model()
        return schema.load(data)


async def run()
    Mayim(hydrator=MarshmallowHydrator(), ...)
```

```python
from mayim import Mayim, Executor, Hydrator, hydrator

class HydratorA(Hydrator):
    ...

class HydratorB(Hydrator):
    ...

class SomeExecutor(Executor):
    async def select_a(...) -> Something:
        ...

    @hydrator(HydratorB())
    async def select_b(...) -> Something:
        ...

Mayim(executors=[SomeExecutor(hydrator=HydratorA())])
```

That's great, but... *How do I use it?*

With **Sanic** Extensions

```python
from mayim.extensions import SanicMayimExtension
from sanic_ext import Extend

class CityExecutor(Executor):
    async def select_all_cities(
        self, limit: int = 4, offset: int = 0
    ) -> List[City]:
        ...


app = Sanic(__name__)
Extend.register(
    SanicMayimExtension(
        executors=[CityExecutor], dsn="postgres://..."
    )
)


@app.get("/")
async def handler(request: Request, executor: CityExecutor):
    cities = await executor.select_all_cities()
    return json({"cities": [asdict(city) for city in cities]})
```

Also for **Quart** and **Starlette**

```
from quart import Quart
from mayim.extension import QuartMayimExtension

app = Quart(__name__)

QuartMayimExtension(
    executors=[CityExecutor],
    dsn="postgres://postgres:postgres@localhost:5432/world",
).init_app(app)
```

```
from starlette.applications import Starlette
from mayim.extension import StarletteMayimExtension

app = Starlette(routes=some_routes)

StarletteMayimExtension(
    executors=[CityExecutor],
    dsn="postgres://postgres:postgres@localhost:5432/world",
).init_app(app)
```

GitHub - /ahopkins

Twitter - @admhpkns

PacketFabric - packetfabric.com

Mayim homepage - ahopkins.github.io/mayim

Mayim repo - /ahopkins/mayim

Python Web Development with SANIC
An in-depth guide for Python web developers to improve the speed and scalability of web applications
Adam Hopkins

https://sanicbook.com