PiterPy

TECHNICAL CONFERENCE FOR HARDCORE PYTHON DEVELOPERS

SAINT PETERSBURG
2019 NOVEMBER 1

PiterPy

GraphQL + Python today.
Build Public API over GraphQL

Sergey Khaletskiy
@sierjkhaletski

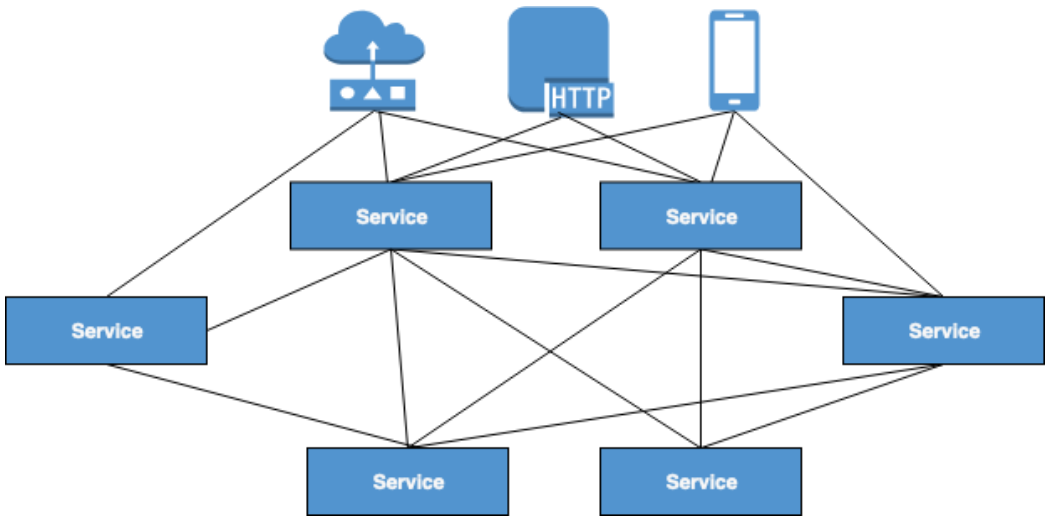
EPAM
Lead Software Engineer

Agenda

- Where are we now?
- GraphQL at a Glance
- Frameworks & UI
- P~~ython~~ Django + GraphQL. Public API
- Django Batteries
- Performance & Security
- API Versioning
- File uploading

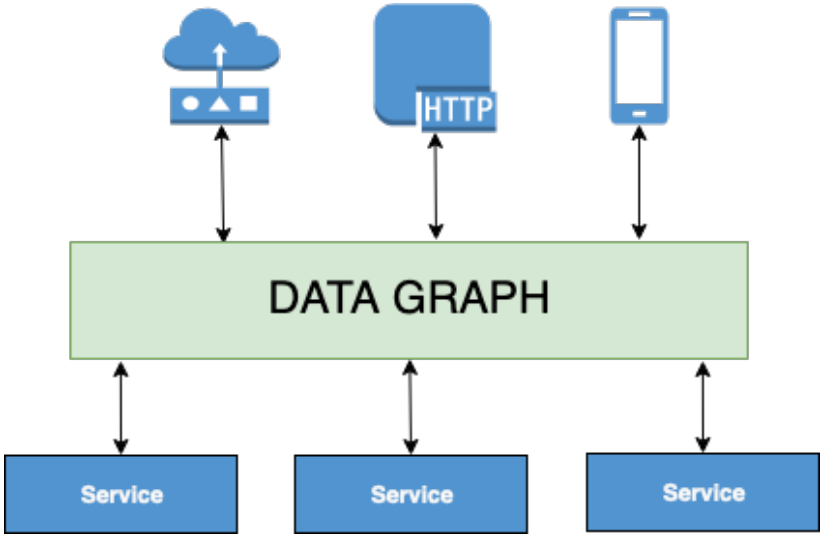
PiterPy

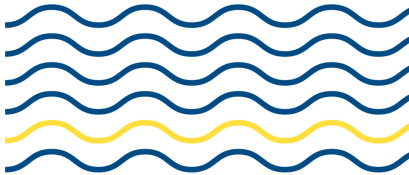
Where are we now?



PiterPy

Where are we going?

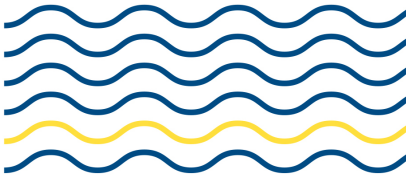




GraphQL at a Glance

- GraphQL is a contract
  REST - architectural style, GraphQL - query language that has been defined to ensure its consistency
- Schema is a core
  Each request to the server must be defined in the schema
- Single endpoint
  Only one entry point, e.g. /api/graphql/
- Query
  Message to the server to request certain data. The language itself loosely resembles JSON
- Mutation
  Unlike query, mutation is used to mutate the data
- POSTge
  Communicate only against POST requests

PiterPy

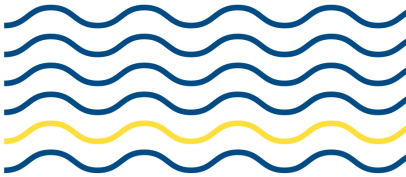


Principal GraphQL

https://principledgraphql.com

- One Graph
- Federated Implementation
- Track Schema in a Registry
- Abstract, Demand-Oriented Schema
- Use an Agile Approach to Schema Development
- Iteratively Improve Performance
- Use Graph Metadata to Empower Developers
- Access and Demand Control
- Structured Logging
- Separate the GraphQL Layer from the Service Layer

PiterPy

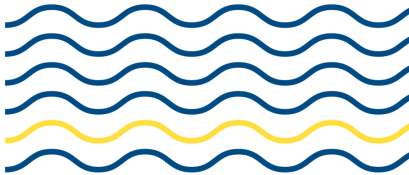


Principal GraphQL

https://principledgraphql.com

- One Graph
- Federated Implementation
- Track Schema in a Registry
- Abstract, Demand-Oriented Schema
- Use an Agile Approach to Schema Development
- Iteratively Improve Performance
- Use Graph Metadata to Empower Developers
- Access and Demand Control
- Structured Logging
- Separate the GraphQL Layer from the Service Layer

PiterPy



What do we use?

Python

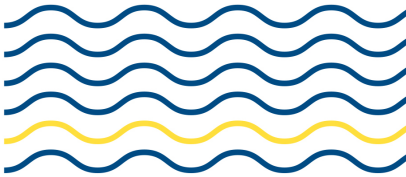
ariadne

Cannula

GraphQL for people who like Python

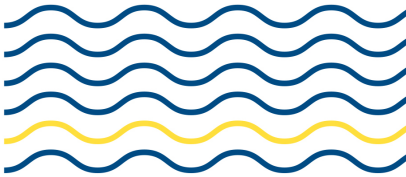
Starlette

GraphQL Playground



Why not REST?

- Over-fetching problem
- Not clear how to deprecate obsolete data
- Responses for a few devices could be different
- GraphQL query is pretty similar for response
- GraphQL can provide us exhaustive analytics for each kind of data



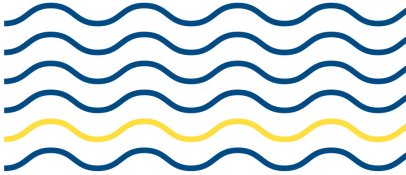
Where is a practice?

?!

PiterPy

Believe in the Magic

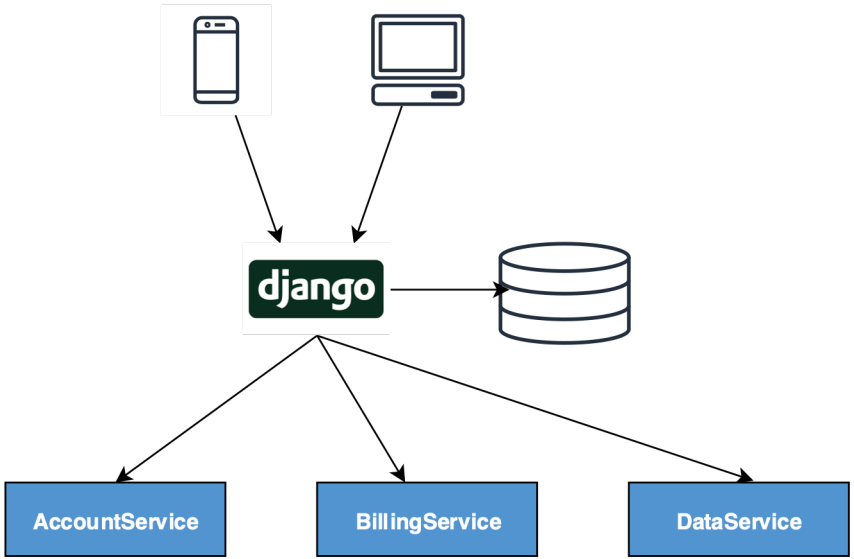
PiterPy

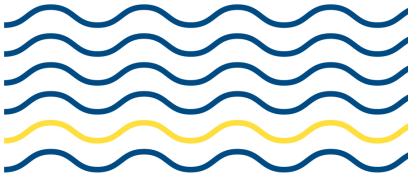


Oops...

PiterPy

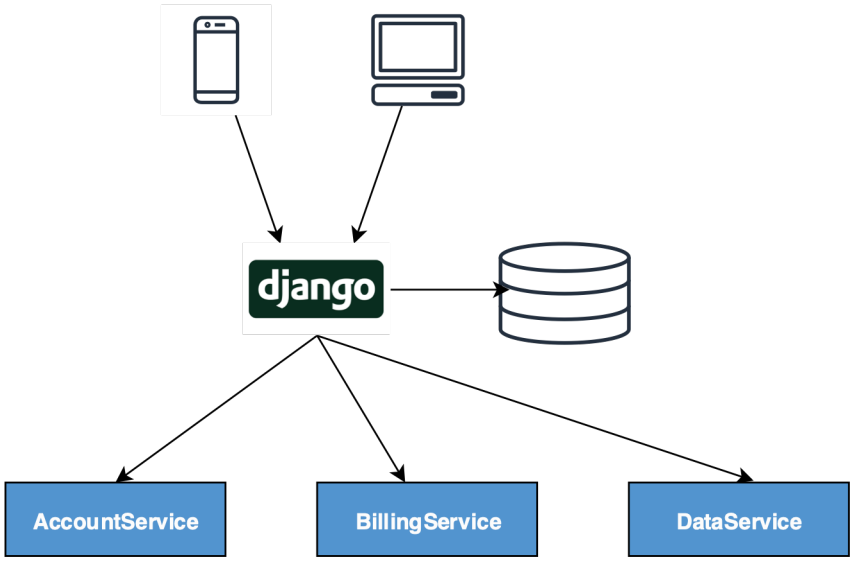
Where were we...

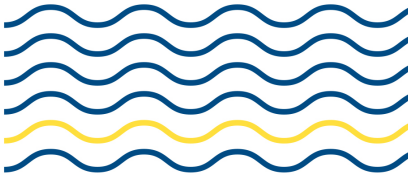




… and what did we need?

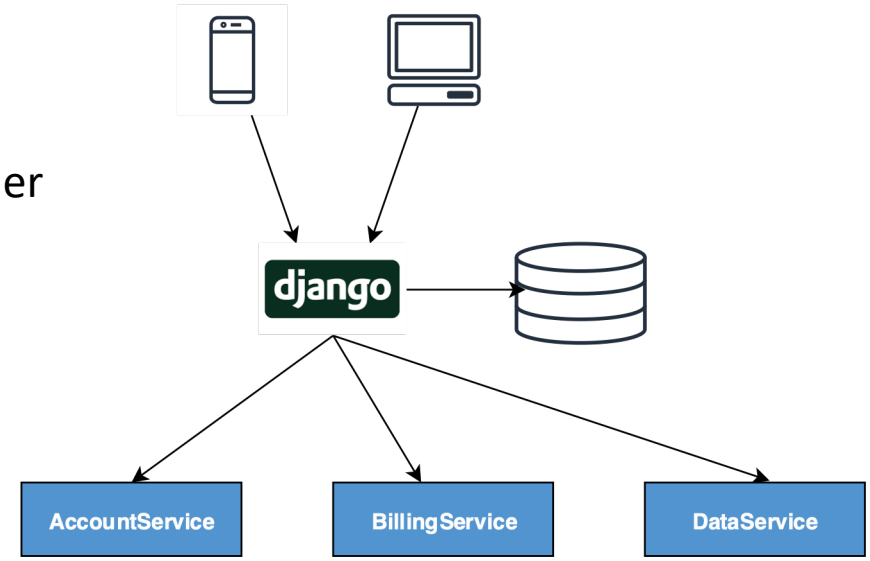
- Public API





… and what did we need?

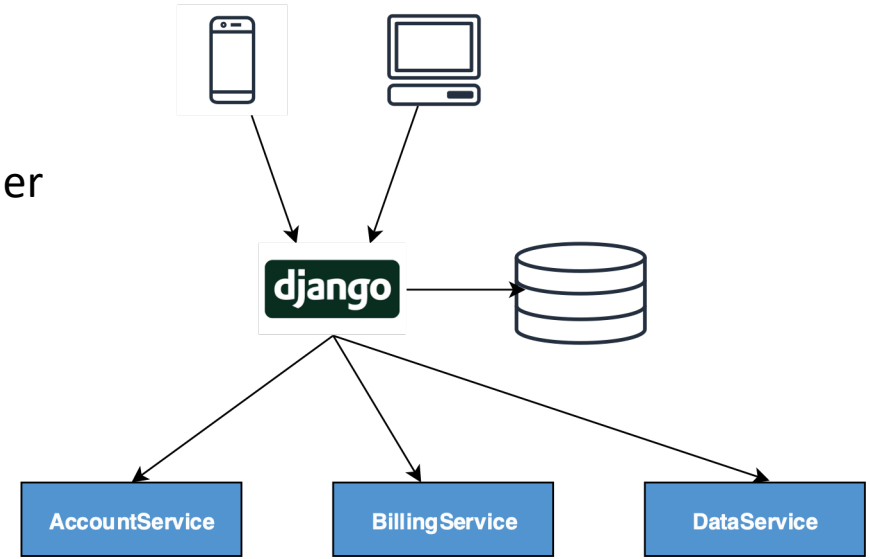
- Public API
- Request side services in the same manner

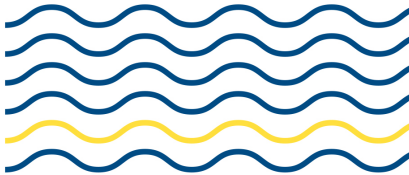


PiterPy

… and what did we need?

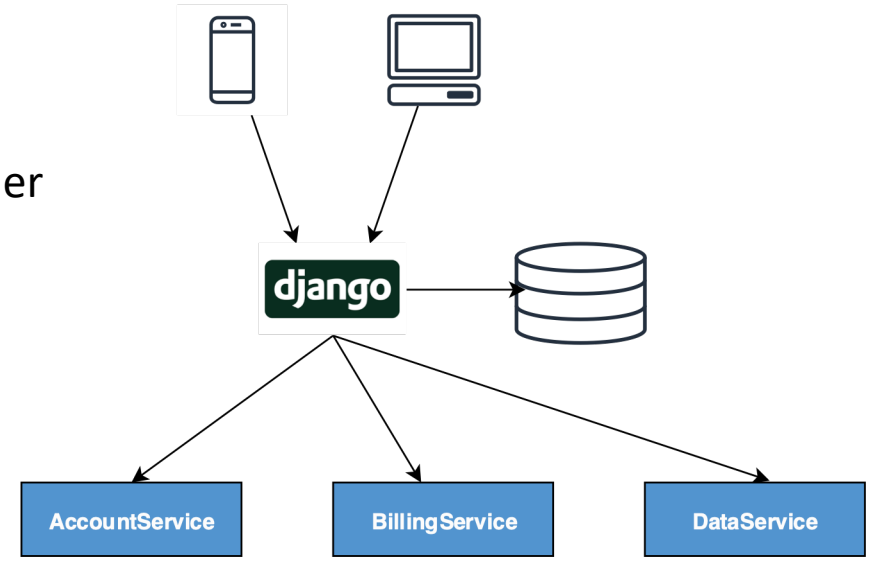
- Public API
- Request side services in the same manner
- Same data as for internal services

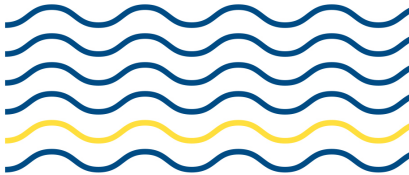




… and what did we need?

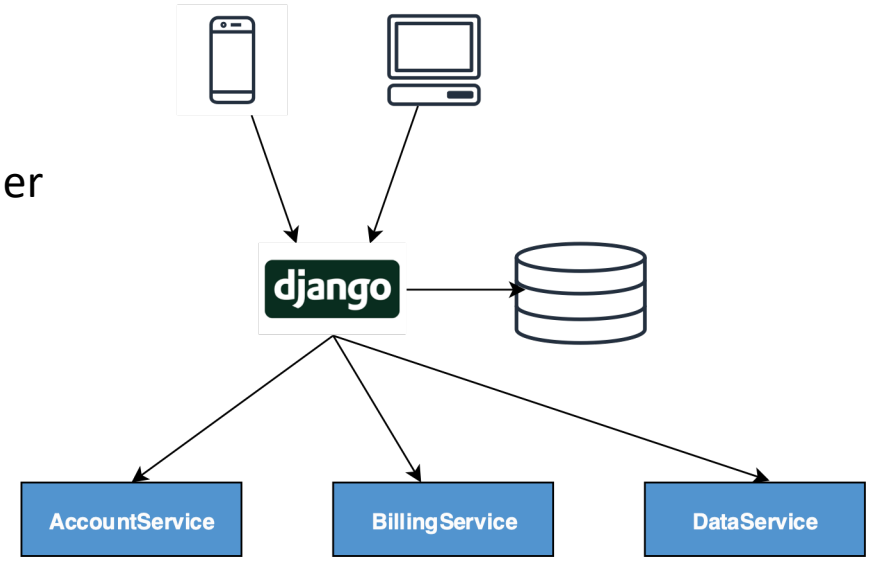
- Public API
- Request side services in the same manner
- Same data as for internal services
- Analytics over clients requests

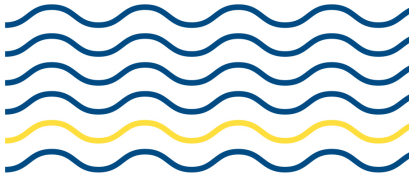




… and what did we need?

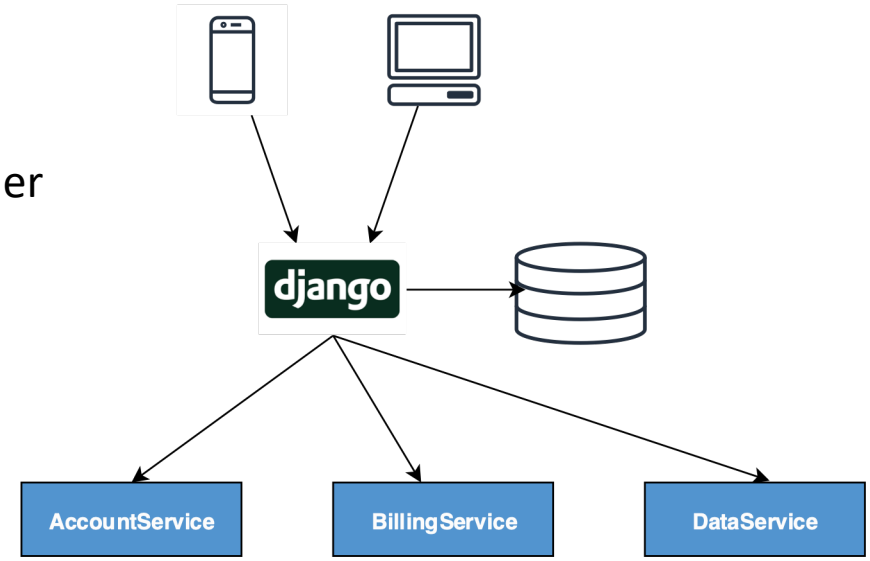
- Public API
- Request side services in the same manner
- Same data as for internal services
- Analytics over clients requests
- As little as possible codebase changes





… and what did we need?

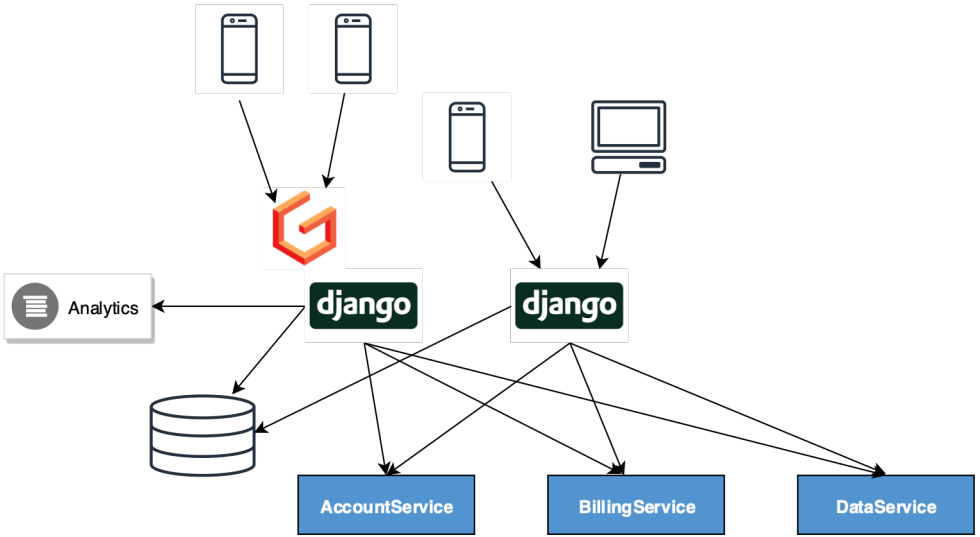
- Public API
- Request side services in the same manner
- Same data as for internal services
- Analytics over clients requests
- As little as possible codebase changes
- Make some money ☺



PiterPy

Actions

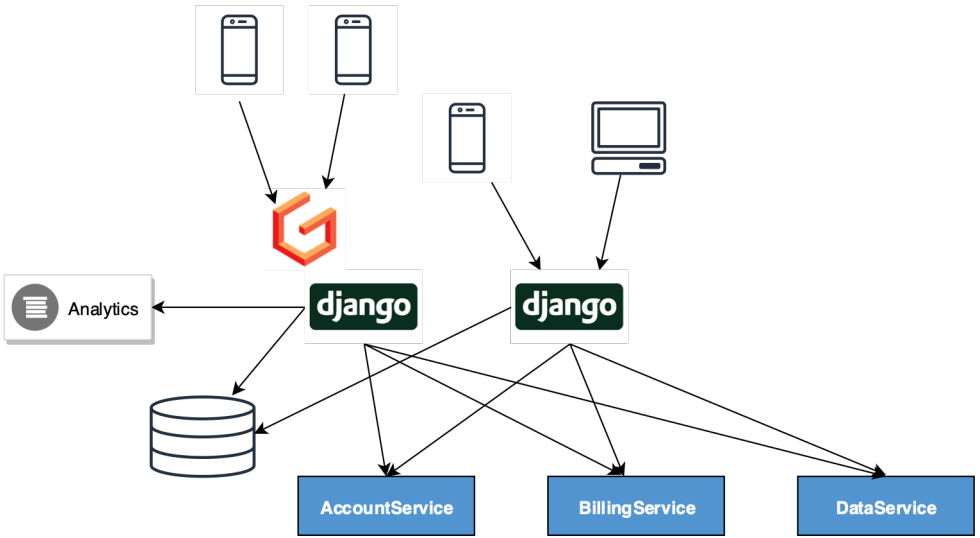
- Plug in *graphene-django* battery



PiterPy

Actions

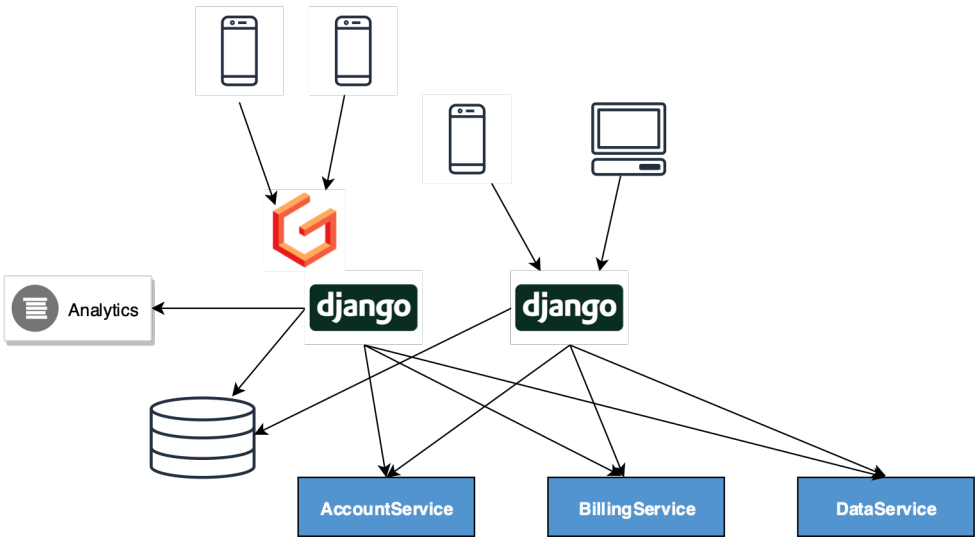
- Plug in *graphene-django* battery
- Use the same models for resolvers



PiterPy

Actions

- Plug in *graphene-django* battery
- Use the same models for resolvers
- Deploy separately Public API app



PiterPy

Graphene Python

schema basic flow

```python
import graphene
from graphene_django.types import DjangoObjectType
from cookbook.ingredients.models import Category, Ingredient

class CategoryType(DjangoObjectType):
    class Meta:
        model = Category

class IngredientType(DjangoObjectType):
    class Meta:
        model = Ingredient

class Query(object):
    category = graphene.Field(CategoryType,
                              id=graphene.Int(),
                              name=graphene.String())
    all_categories = graphene.List(CategoryType)

    ingredient = graphene.Field(IngredientType,
                                id=graphene.Int(),
                                name=graphene.String())
    all_ingredients = graphene.List(IngredientType)

    def resolve_all_categories(self, info, **kwargs):
        return Category.objects.all()

    def resolve_all_ingredients(self, info, **kwargs):
        return Ingredient.objects.all()

    def resolve_category(self, info, **kwargs):
        id = kwargs.get('id')
        name = kwargs.get('name')

        if id is not None:
            return Category.objects.get(pk=id)

        if name is not None:
            return Category.objects.get(name=name)

        return None

    def resolve_ingredient(self, info, **kwargs):
        id = kwargs.get('id')
        name = kwargs.get('name')

        if id is not None:
            return Ingredient.objects.get(pk=id)

        if name is not None:
            return Ingredient.objects.get(name=name)

        return None
```

PiterPy

Graphene Python

custom scalar

```python
import datetime
from graphene.types import Scalar
from graphql.language import ast

class DateTime(Scalar):
    '''DateTime Scalar Description'''

    @staticmethod
    def serialize(dt):
        return dt.isoformat()

    @staticmethod
    def parse_literal(node):
        if isinstance(node, ast.StringValue):
            return datetime.datetime.strptime(
                node.value, "%Y-%m-%dT%H:%M:%S.%f")

    @staticmethod
    def parse_value(value):
        return datetime.datetime.strptime(value, "%Y-%m-%dT%H:%M:%S.%f")
```

PiterPy

Graphene Python

query example

```
{
  recipe(id: "UmVjaXBlTm9kZTox") {
    id
    instructions
    amounts {
      edges {
        node {
          ingredient {
            id
            name
          }
        }
      }
    }
  }
}
```

```
{
  "data": {
    "recipe": {
      "id": "UmVjaXBlTm9kZTox",
      "instructions": "Do everything on you
taste",
      "amounts": {
        "edges": [
          {
            "node": {
              "ingredient": {
                "id": "SW5ncmVkaWVudE5vZGU6MQ==",
                "name": "Eggs"
              }
            }
          },
          {
            "node": {
              "ingredient": {
                "id": "SW5ncmVkaWVudE5vZGU6NA==",
                "name": "Chicken"
              }
            }
          },
          {
            "node": {
              "ingredient": {
                "id": "SW5ncmVkaWVudE5vZGU6Mg==",
                "name": "Milk"
              }
            }
          }
        ]
      }
    }
  }
}
```

PiterPy

Graphene Python

batching

```
{
  allCategories{
    edges{
      node{
        name
        ingredients{
          edges{
            node{
              name
            }
          }
        }
      }
    }
  }
}
```

```
1 class Query(object):
2     category = Node.Field(CategoryNode)
3     all_categories = DjangoFilterConnectionField(CategoryNode)
4
5     ingredient = Node.Field(IngredientNode)
6     all_ingredients = DjangoFilterConnectionField(IngredientNode)
```

```
1 SELECT * FROM "ingredients" WHERE "ingredients"."category_id" = '1'
2 SELECT * FROM "ingredients" WHERE "ingredients"."category_id" = '2'
3 SELECT * FROM "ingredients" WHERE "ingredients"."category_id" = '3'
4 SELECT * FROM "ingredients" WHERE "ingredients"."category_id" = '4'
5 SELECT * FROM "ingredients" WHERE "ingredients"."category_id" = '5'
```

- Defeats N+1 problem
- Based on DataLoader approach
- Implementation is absolutely on developer flavor
- Works in pair with caching

PiterPy

Graphene Python

batching

```python
from collections import defaultdict
from promise import Promise
from promise.dataloader import DataLoader
from cookbook.ingredients.models import Ingredient


class IngredientsByCategoryIdLoader(DataLoader):
    def batch_load_fn(self, category_ids):
        ingredients_by_cat_id = defaultdict(list)
        for ingredient in Ingredient.objects.filter(category__in=category_ids).iterator():
            ingredients_by_cat_id[ingredient.category_id].append(ingredient)
        return Promise.resolve([ingredients_by_cat_id.get(category_id, [])
                                for category_id in category_ids])

```

PiterPy

Graphene Python

batching

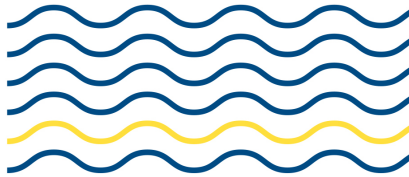
```python
# cookbook/context.py file

from django.utils.functional import cached_property
from cookbook.dataloaders import IngredientsByCategoryIdLoader

class GraphQLContext:
    def __init__(self, request):
        self.request = request

    @cached_property
    def user(self):
        return self.request.user

    @cached_property
    def ingredients_by_category_id_loader(self):
        return IngredientsByCategoryIdLoader()
```

```python
# cookbook/views.py file
from graphene_django.views import GraphQLView
from cookbook.context import GraphQLContext

class GraphQLContextView(GraphQLView):
    def get_context(self, request):
        return GraphQLContext(request)
```

PiterPy

Graphene Django

project structure



- graphql
  - accounts
    - resolvers.py
    - schema.py
    - types.py
    - utils.py
  - billing
    - resolvers.py
    - schema.py
    - types.py
    - utils.py
  - publicapi.py

- Keep all API relations in one folder *graphql*

PiterPy

Graphene Django

project structure

```python
# publicapi.py file

import graphene

from graphene_django.debug import DjangoDebug

from graphql.accounts.schema import Query as AccountsQuery
from graphql.billing.schema import Query as BillingQuery

class Query(
    AccountsQuery,
    BillingQuery,
    graphene.ObjectType
):
    debug = graphene.Field(DjangoDebug, name="_debug")

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

- *publicapi.py* imports all modules form *graphql* and exposes the schema

PiterPy

Graphene Python

Pros

- Easy install & deploy
- Common code base with Django
- Extendable. Easy to create custom scalars
- UI out of the box (GraphiQL)
- Testable

PiterPy

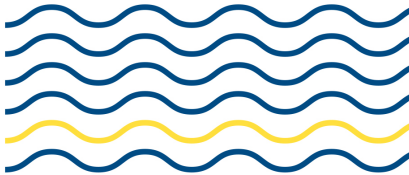
Graphene Python

Pros

- Easy install & deploy
- Common code base with Django
- Extendable. Easy to create custom scalars
- UI out of the box (GraphiQL)
- Testable

Cons

- Query cost is unknown. There is no way to manage query depth
- Not all features are documented. Sources reading
- Caching on your own
- Multiple schemas serving is unsupported
- Require 3rd-party batteries

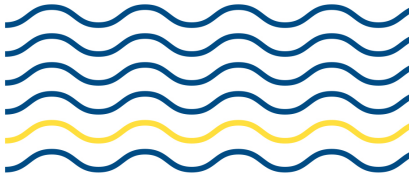


Django Batteries

django-graphql-jwt

Mutation Body

```
mutation {
    jwtCreate(login:"login@email.com", password:"example_password") {
      token
      user {
        id
        username
      }
    }
}
```



Django Batteries

django-graphql-jwt

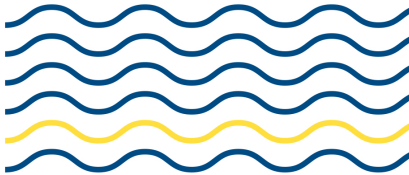
Mutation Body

```
mutation {
    jwtCreate(login:"login@email.com", password:"example_password") {
      token
      user {
        id
        username
      }
    }
}
```

Response Body

```
{
    "data": {
        "jwtCreate": {
            "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
                .eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
                .SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c",
            "user": {
                "id": "da4d5a62-b8f1-4ad7-970f-652fdb5ebede",
                "username": "superduperuser"
            }
        }
    }
}
```

Pass the provided JWT token into Authorization header for subsequent requests

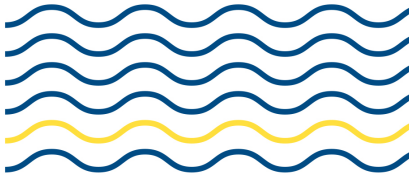


Django Batteries

django-graphql-jwt

- @login_required
- @user_passes_test
- @permission_required
- @staff_member_required
- @superuser_required

```python
import graphene
from graphql_jwt.decorators import login_required


class Query(graphene.ObjectType):
    viewer = graphene.Field(UserType)

    @login_required
    def resolve_viewer(self, info, **kwargs):
        return info.context.user
```

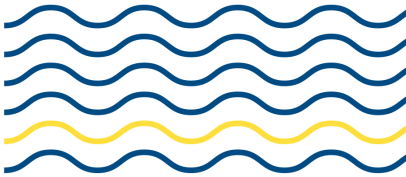


Django Batteries

graphene-django-optimizer

”Optimize queries executed by *graphene-django* automatically,
using ***select_related***, ***prefetch_related*** and ***only*** methods of Django QuerySet”

PiterPy



Django Batteries

graphene-django-optimizer

Request Body

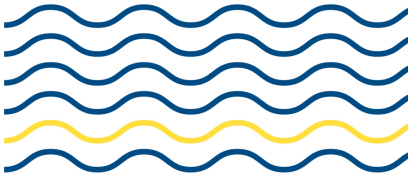
```
{
  all_ingredients {
    id
    name
    category {
        id
        name
    }
  }
}
```

Response Body

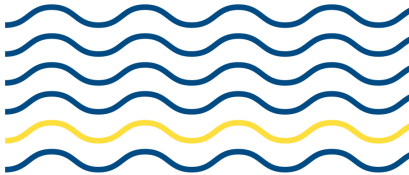
```
# optimized queryset:
ingredients = (
    Ingredient.objects
    .select_related('category')
    .only('id', 'name',
          'category__id', 'category__name')
)
```

PiterPy

What else?



What else?



What else?

Lightweight ASGI framework/toolkit, which is ideal for building high performance asyncio services.

PiterPy

Starlette

- Seriously impressive performance.
- WebSocket support.
- GraphQL support.
- In-process background tasks.
- Startup and shutdown events.
- Test client built on requests.
- CORS, GZip, Static Files, Streaming.
- Session and Cookie support.
- 100% test coverage.
- 100% type annotated codebase.
- Zero hard dependencies.

PiterPy

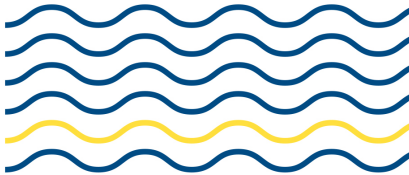
Starlette

- Seriously impressive performance.
- WebSocket support.
- GraphQL support.
- In-process background tasks.
- Startup and shutdown events.
- Test client built on requests.
- CORS, GZip, Static Files, Streaming.
- Session and Cookie support.
- 100% test coverage.
- 100% type annotated codebase.
- Zero hard dependencies.

PiterPy

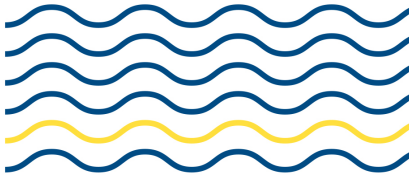
Starlette

- Seriously impressive performance.
- WebSocket support.
- GraphQL support.
- In-process background tasks.
- Startup and shutdown events.
- Test client built on requests.
- CORS, GZip, Static Files, Streaming.
- Session and Cookie support.
- 100% test coverage.
- 100% type annotated codebase.
- Zero hard dependencies.



Starlette: GraphQL

```python
from graphql.execution.executors.asyncio import AsyncioExecutor
from starlette.applications import Starlette
from starlette.graphql import GraphQLApp
import graphene


class Query(graphene.ObjectType):
    hello = graphene.String(name=graphene.String(default_value="stranger"))

    async def resolve_hello(self, info, name):
        # We can make asynchronous network calls here.
        return "Hello " + name


app = Starlette()

# We're using `executor_class=AsyncioExecutor` here.
app.add_route('/', GraphQLApp(schema=graphene.Schema(query=Query), executor_class=AsyncioExecutor))
```



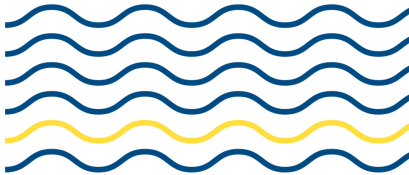
Starlette: Background Tasks

```python
class Query(graphene.ObjectType):
    user_agent = graphene.String()

    def resolve_user_agent(self, info):
        """
        Return the User-Agent of the incoming request.
        """
        user_agent = request.headers.get("User-Agent", "<unknown>")
        background = info.context["background"]
        background.add_task(log_user_agent, user_agent=user_agent)
        return user_agent

async def log_user_agent(user_agent):
    ...
```

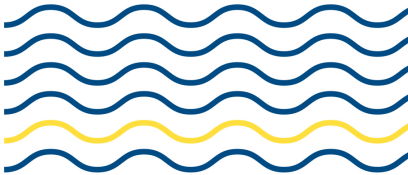
PiterPy

Something else…

PiterPy

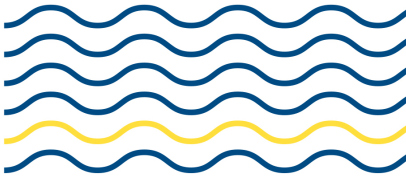


Something else…



Something else…

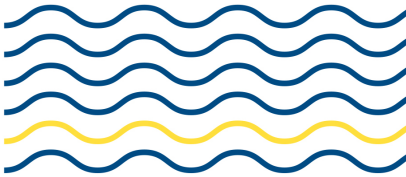
PiterPy



- Asynchronous
- Schema-first approach to the API implementation
- Simple
- Extensible
- Starlette is used for ASGI
- Dev server (synchronous!)
- Easy to implementing GraphQL in existing sites

  (WSGI middleware or Django GraphQL Views)

PiterPy

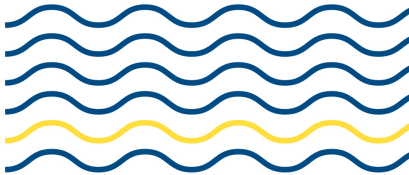


- Compatibility with GraphQL.js version 14.4.0
- Queries, mutations and input types
- Subscriptions
- File uploads
- Custom scalars and enums
- Loading schema from .graphql files
- GraphQL syntax validation via gql() helper function
- …

PiterPy

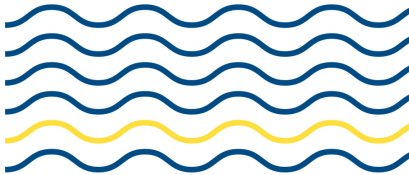
ariadne

```python
from ariadne import ObjectType, QueryType, gql, make_executable_schema
from ariadne.asgi import GraphQL

type_defs = gql("""
    type Query {
        people: [Person!]!
    }

    type Person {
        firstName: String
        lastName: String
        age: Int
        fullName: String
    }
""")

# Map resolver functions to Query fields using QueryType
query = QueryType()

# Resolvers are simple python functions
@query.field("people")
def resolve_people(*_):
    return [
        {"firstName": "John", "lastName": "Doe", "age": 21},
        {"firstName": "Bob", "lastName": "Boberson", "age": 24},
    ]


# Map resolver functions to custom type fields using ObjectType
person = ObjectType("Person")

@person.field("fullName")
def resolve_person_fullname(person, *_):
    return "%s %s" % (person["firstName"], person["lastName"])

# Create executable GraphQL schema
schema = make_executable_schema(type_defs, [query, person])

# Create an ASGI app using the schema, running in debug mode
app = GraphQL(schema, debug=True)
```



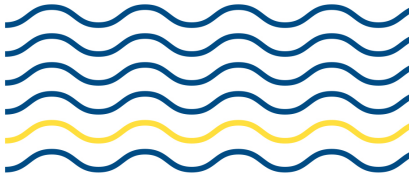
GraphQL: API Versioning

”GraphQL takes a strong opinion on avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema.”



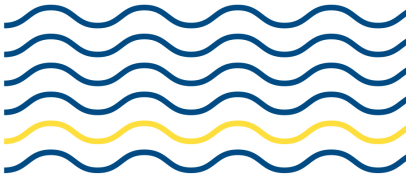
GraphQL: Files Uploading

- REST endpoint in front of GraphQL mutation
  upload via REST and pass a resulted URL into GraphQL mutation.
- Upload files as Base64 Encoded String
  encoded string sends with GraphQL mutation. Resource intensive and it is sometimes fraught with errors.
- Using of an external URL (such as AWS S3, Google Cloud Storage etc.)
  once file uploaded to S3 then generated URL can be used to pass a GraphQL mutation
- Using *graphene-file-upload* battery
  based on GraphQL multipart request specification (https://github.com/jaydenseric/graphql-multipart-request-spec)



Conclusion

- GraphQL Public API in production
  indeed, easy to build Public API on application in production
- All libs are production ready and permanently evaluates
- Don’t worry. Django is close
  all reviewed frameworks have Django wrappers or can combined
- Control requests depth
  try to prevent client outrage
- GraphQL will not solve your API design problems
  “If you suck at providing REST API You’ll suck at providing GraphQL API”
  @apihandyman



Useful Links

- https://github.com/graphql-python/graphene-django
- https://github.com/flavors/django-graphql-jwt
- https://github.com/tfoxy/graphene-django-optimizer
- https://github.com/graphql-python/graphql-core-next
- https://github.com/rmyers/cannula
- https://github.com/mirumee/ariadne
- https://github.com/encode/starlette
- https://github.com/strawberry-graphql/strawberry
- https://github.com/encode/uvicorn
- https://github.com/kensho-technologies/graphql-compiler

Questions