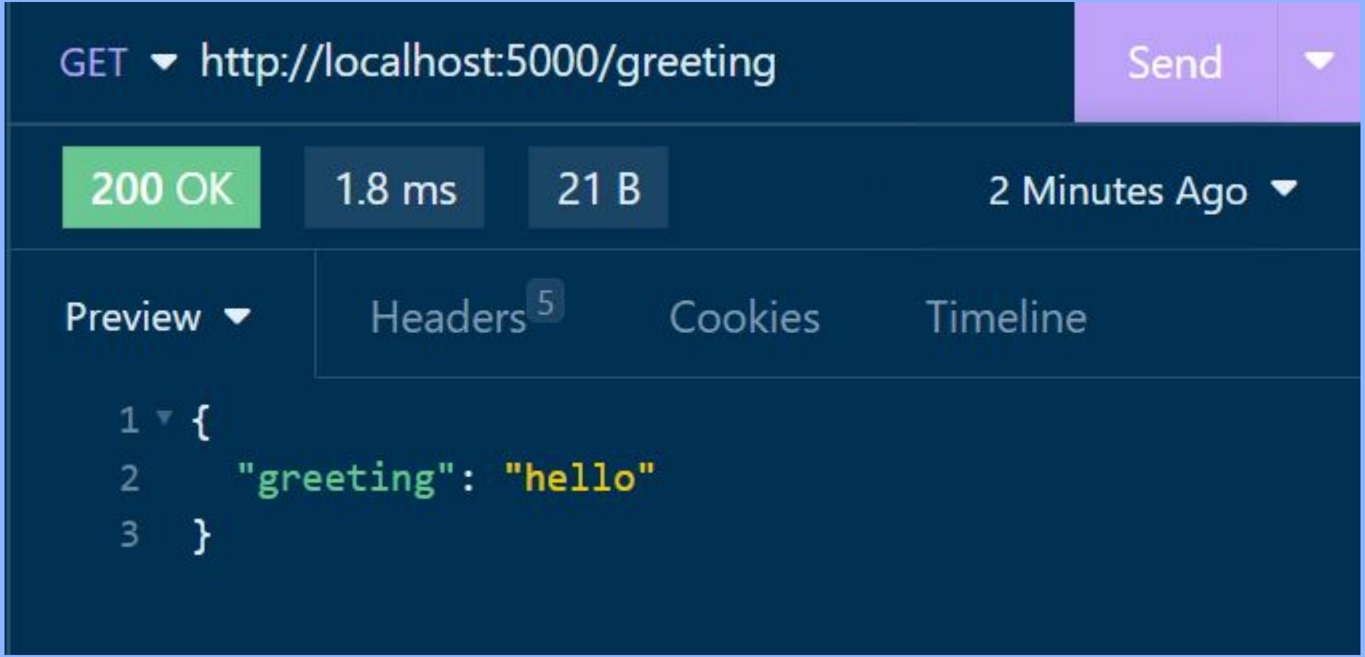
Building Federated GraphQL APIs using Flask

Adarsh Divakaran

REST vs GraphQL

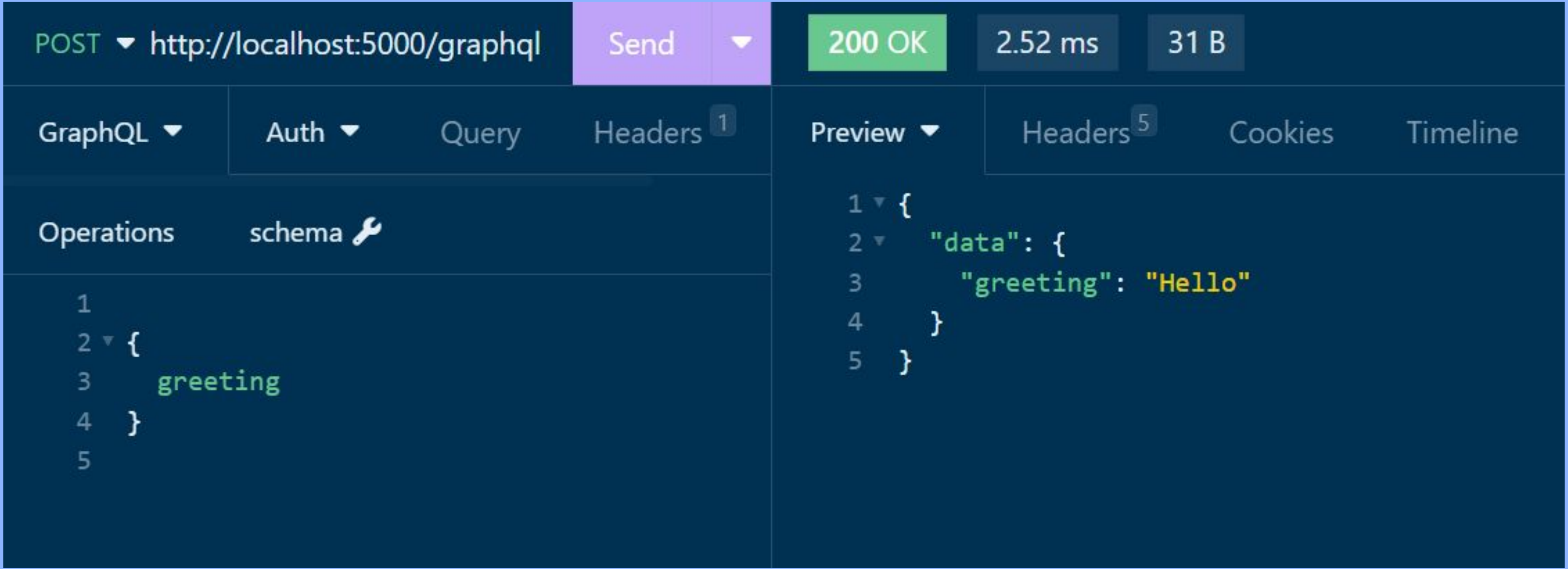
REST



```
GET http://localhost:5000/greeting
200 OK   1.8 ms   21 B   2 Minutes Ago
```

```json
{
  "greeting": "hello"
}
```

GraphQL



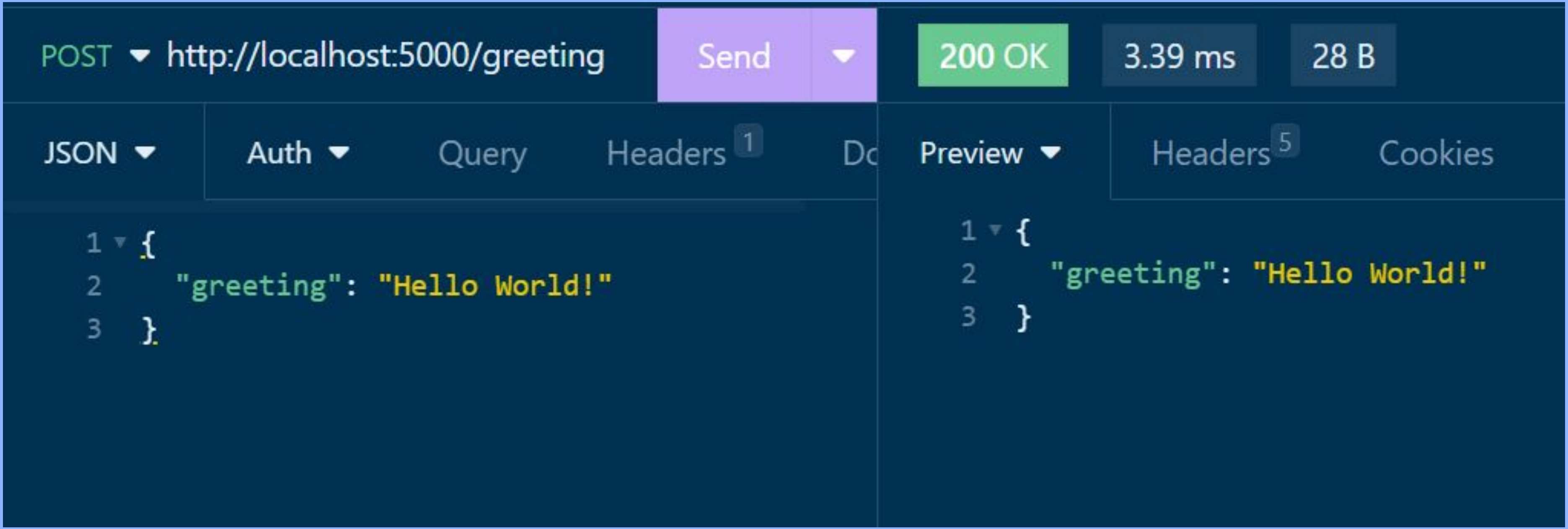
```

{
  greeting
}

```

```
{
  "data": {
    "greeting": "Hello"
  }
}
```

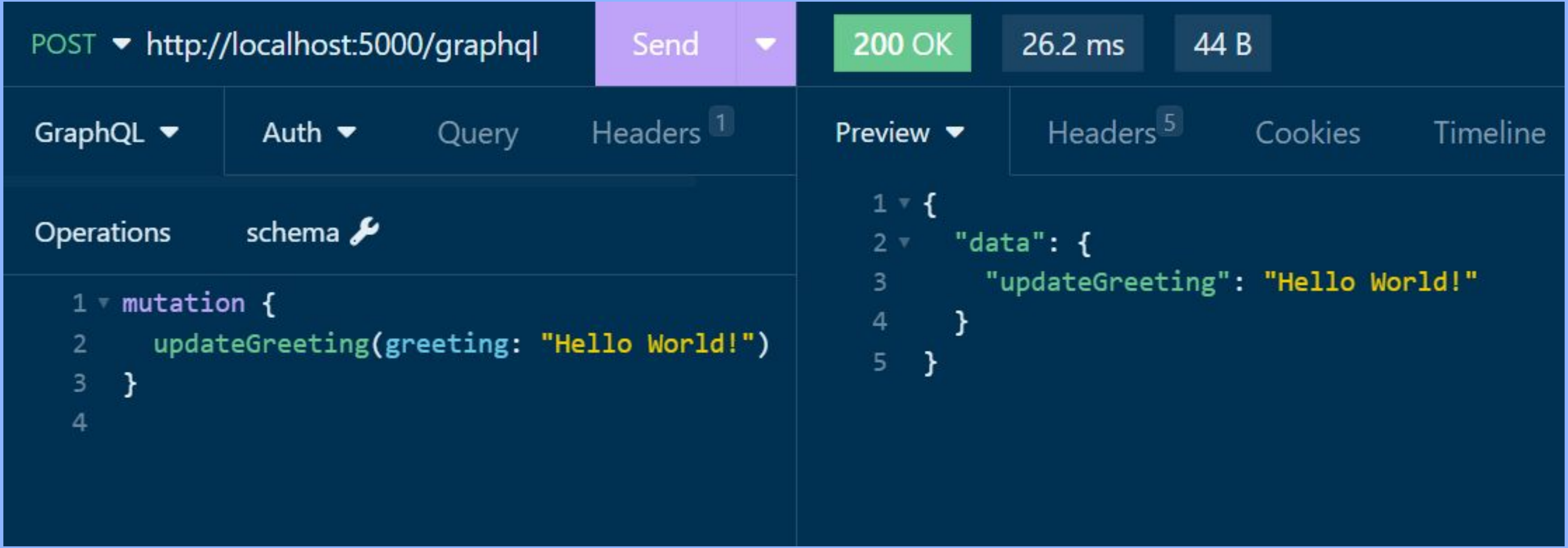
REST



```
{
   "greeting": "Hello World!"
}
```

```
{
   "greeting": "Hello World!"
}
```

GraphQL



```graphql
mutation {
  updateGreeting(greeting: "Hello World!")
}

```

```json
{
  "data": {
    "updateGreeting": "Hello World!"
  }
}
```

GraphQL Features

Single Endpoint

REST

```python
# Typical routes in Flask REST apps

app = Flask(__name__)

app.add_url_rule('/greeting',
view_func=GreetingView.as_view('greeting'))

app.add_url_rule('/goodbye',
view_func=GoodByeView.as_view('goodbye'))

@app.route('/welcome', methods=['GET'])
def welcome():
    ...
```

GraphQL

```python
# Route in Flask GraphQL apps

app = Flask(__name__)

app.add_url_rule(
    "/graphql",

 view_func=GraphQLView.as_view("graphql_view",
schema=schema),
)
```

Operations

REST

GET: Fetch data

POST, PUT, PATCH, DELETE: Add, edit, modify and delete operations

GraphQL

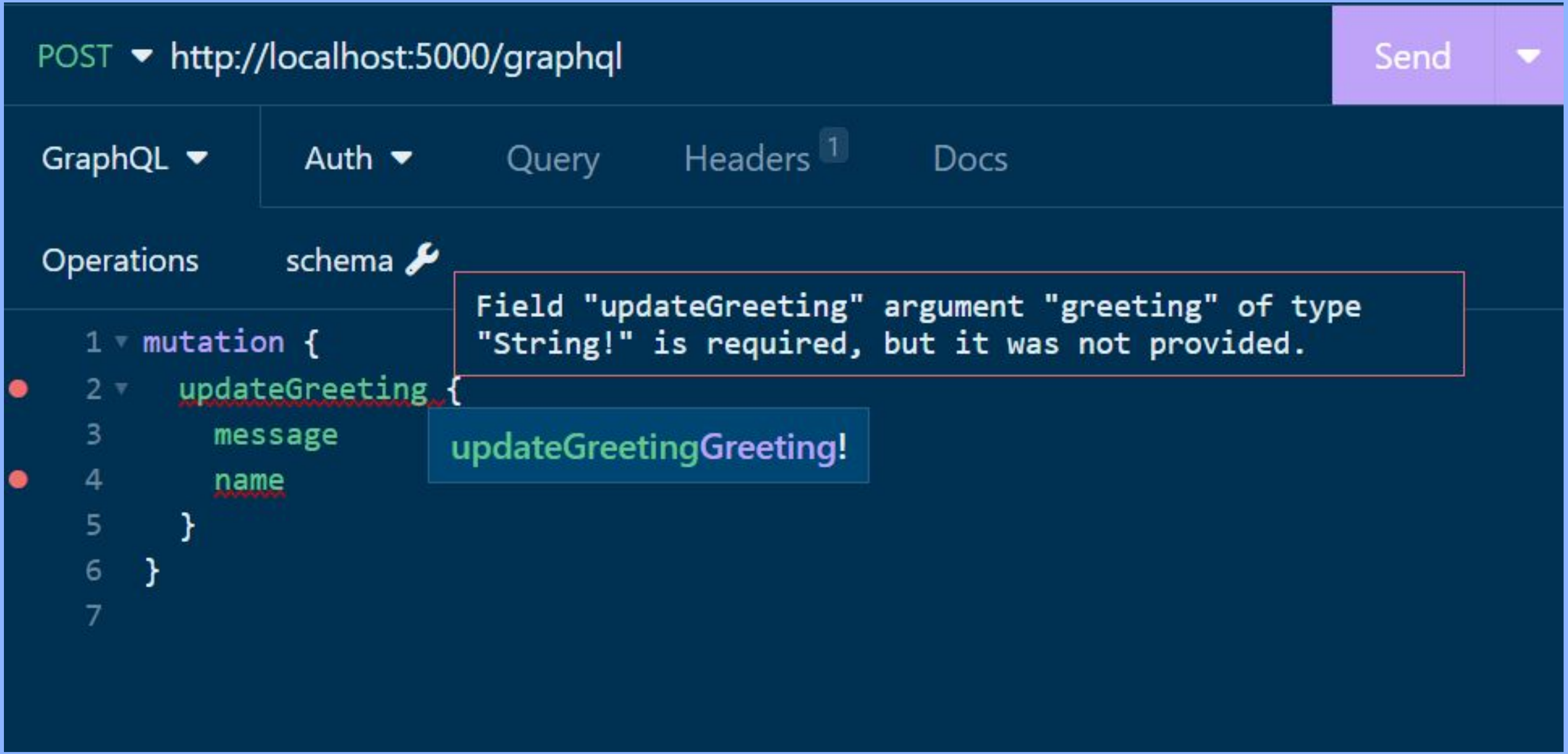
HTTP Method used is POST always.

QUERY: Fetch data

MUTATION: Modify, update, delete, etc.

SUBSCRIPTION: Realtime persistent operations

Strong Typing



```
mutation {
  updateGreeting {
    message
    name
  }
}
```

```
Field "updateGreeting" argument "greeting" of type
"String!" is required, but it was not provided.
```

updateGreetingGreeting!

Scalars & Types in GraphQL

• Int	A signed 32-bit integer.
• Float	A signed double-precision floating-point value.
• String	A UTF-8 character sequence.
• Boolean	true or false.
• ID	The ID scalar type represents a unique identifier

Other types/containers: List, NonNull, Enum, Union, Interface

(source: graphql.org)

Scalars & Types in GraphQL

Enumeration

```
enum Episode {
 NEWHOPE
 EMPIRE
 JEDI
}
```

Custom type

```
type Character {
 name: String!
 appearsIn: [Episode]!
}
```

Union type

```
union SearchResult =
Human | Droid | Starship
```

Advantages of GraphQL

Overfetching Prevention

Initial API Version

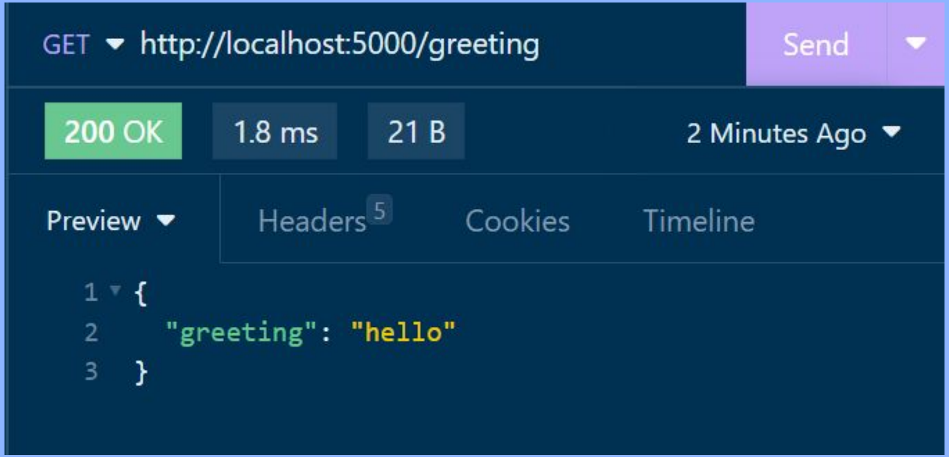
Return a greeting message

New Requirement

For desktop web clients, return a greeting image along with the greeting message

Overfetching Prevention

REST - Initial Version



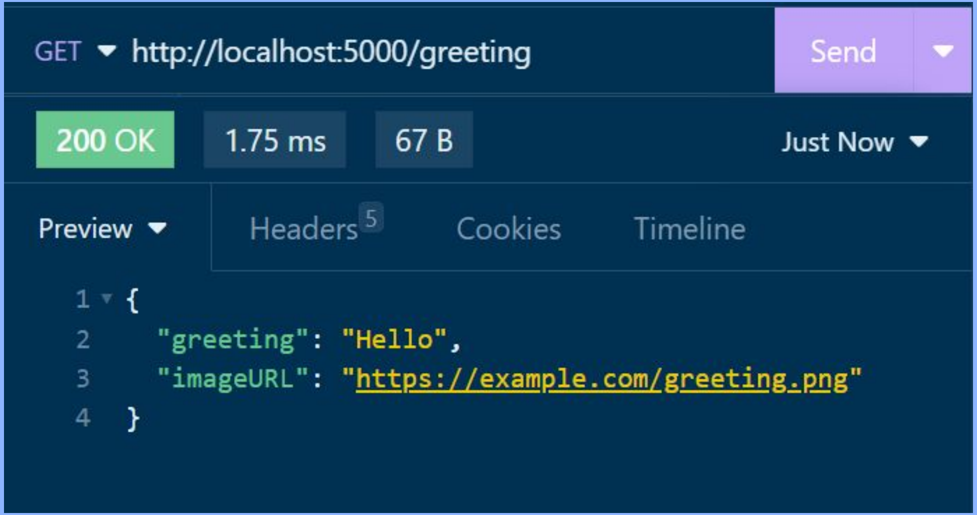
```
http://localhost:5000/greeting
```

```
200 OK    1.8 ms    21 B    2 Minutes Ago
```

```
{
  "greeting": "hello"
}
```

Overfetching Prevention

REST - Option 1



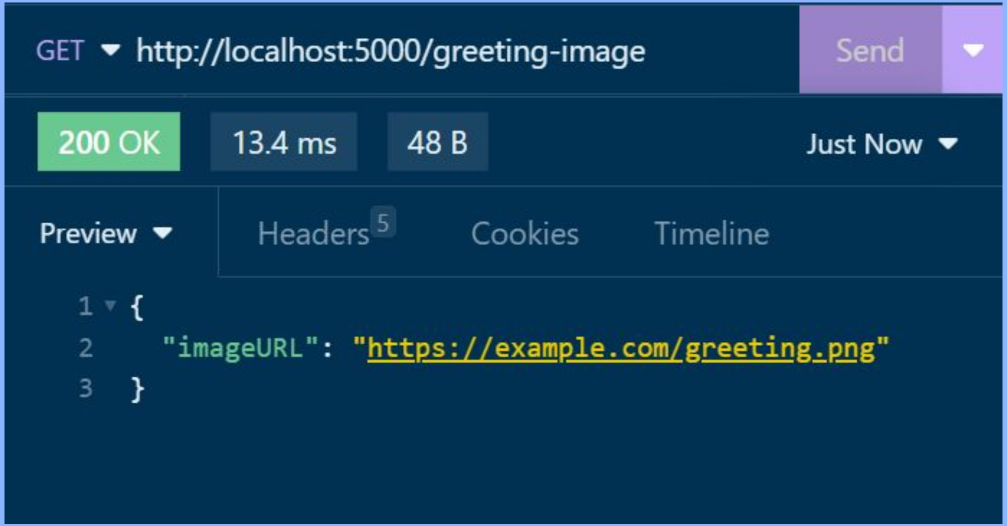
```
GET http://localhost:5000/greeting
200 OK   1.75 ms   67 B   Just Now
```

```
{
  "greeting": "Hello",
  "imageURL": "https://example.com/greeting.png"
}
```

Disadvantage: For mobile clients, an extra unused field is returned with the response

Overfetching Prevention

REST - Option 2 - Adding separate endpoint

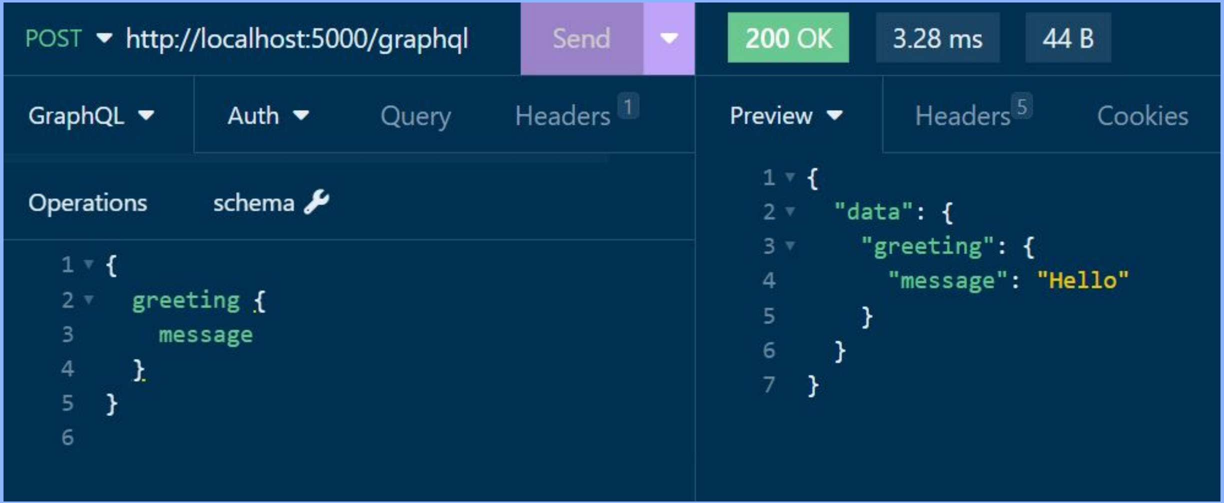


```
{
  "imageURL": "https://example.com/greeting.png"
}
```

Disadvantage: Extra network call and complexity for web clients

Overfetching Prevention

GraphQL Solution



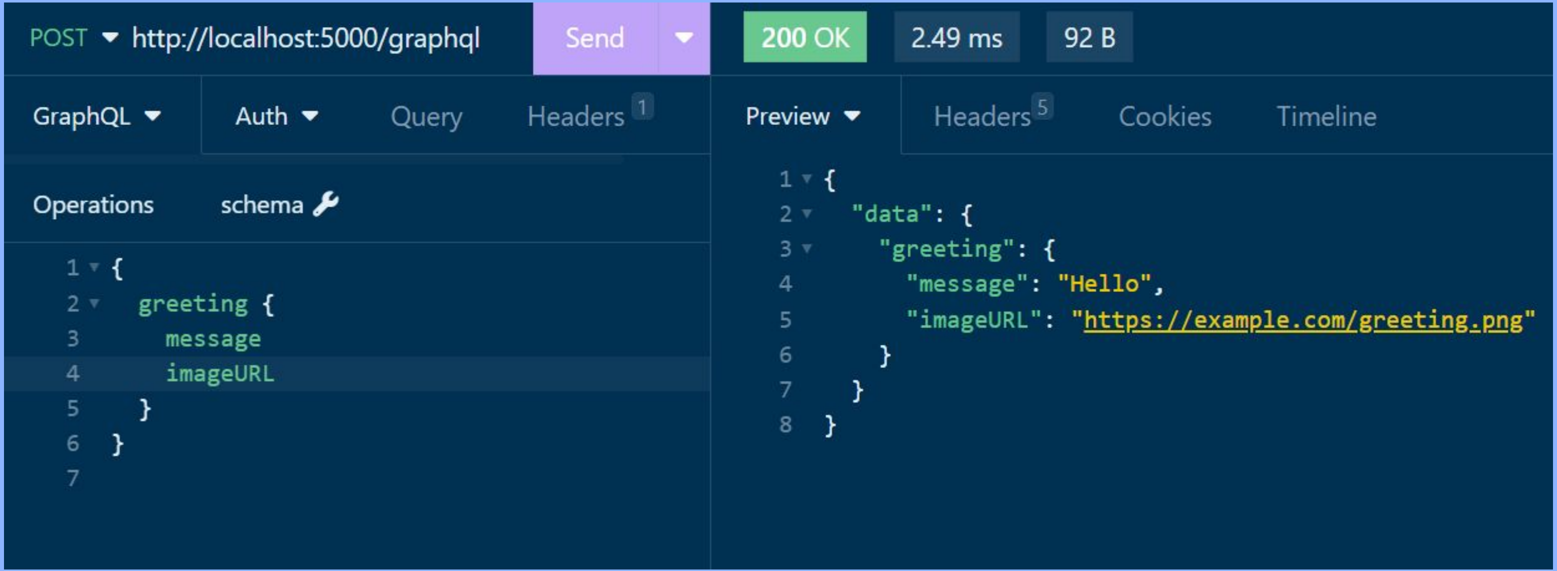
```
{
  greeting {
    message
  }
}
```

```
{
  "data": {
    "greeting": {
      "message": "Hello"
    }
  }
}
```

Query of Mobile client

Overfetching Prevention

GraphQL Solution - Desktop client



```
{
  greeting {
    message
    imageURL
  }
}
```

```
{
  "data": {
    "greeting": {
      "message": "Hello",
      "imageURL": "https://example.com/greeting.png"
    }
  }
}
```

GraphQL allows us to query only the fields we need

Introspection & Type system

- Strongly typed and schema based
- All supported operations by a server are returned by ‘introspection’
- Presence of tooling to auto generate client code
- REST would require additional doc tools - ‘flasgger’ or ‘flask-rest-api’.
- GraphQL development is centered around its schema

Choosing a GraphQL Server

Schema First vs Code First

Schema First

```
from ariadne import QueryType, make_executable_schema

# Ariadne - GraphQL schema definition as Python string
type_defs = """
type User {
    id: ID!
    name: String!
}

type Query {
    getUser(id: ID!): User
}
"""

query = QueryType()

@query.field("getUser")
def resolve_get_user(_, info, id):
    return {"id": id, "name": "John Doe"}

schema = make_executable_schema(type_defs, query)
```

Code First

```
import graphene

# Graphene - Schema expressed using
Python objects
class User(graphene.ObjectType):
    id = graphene.ID(required=True)
    name = graphene.String(required=True)

class Query(graphene.ObjectType):
    get_user = graphene.Field(User,
id=graphene.ID(required=True))

    def resolve_get_user(root, info, id):
        return User(id=id, name="John
Doe")

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

Demo

GraphQL Federation

Apollo Federation

- A standard/spec for combining multiple independent GraphQL schemas
- Combines multiple related schemas from microservices (subgraphs) to a single unified schema (supergraph)
- Abstracts away the microservice design from clients

When to use GraphQL & Federation

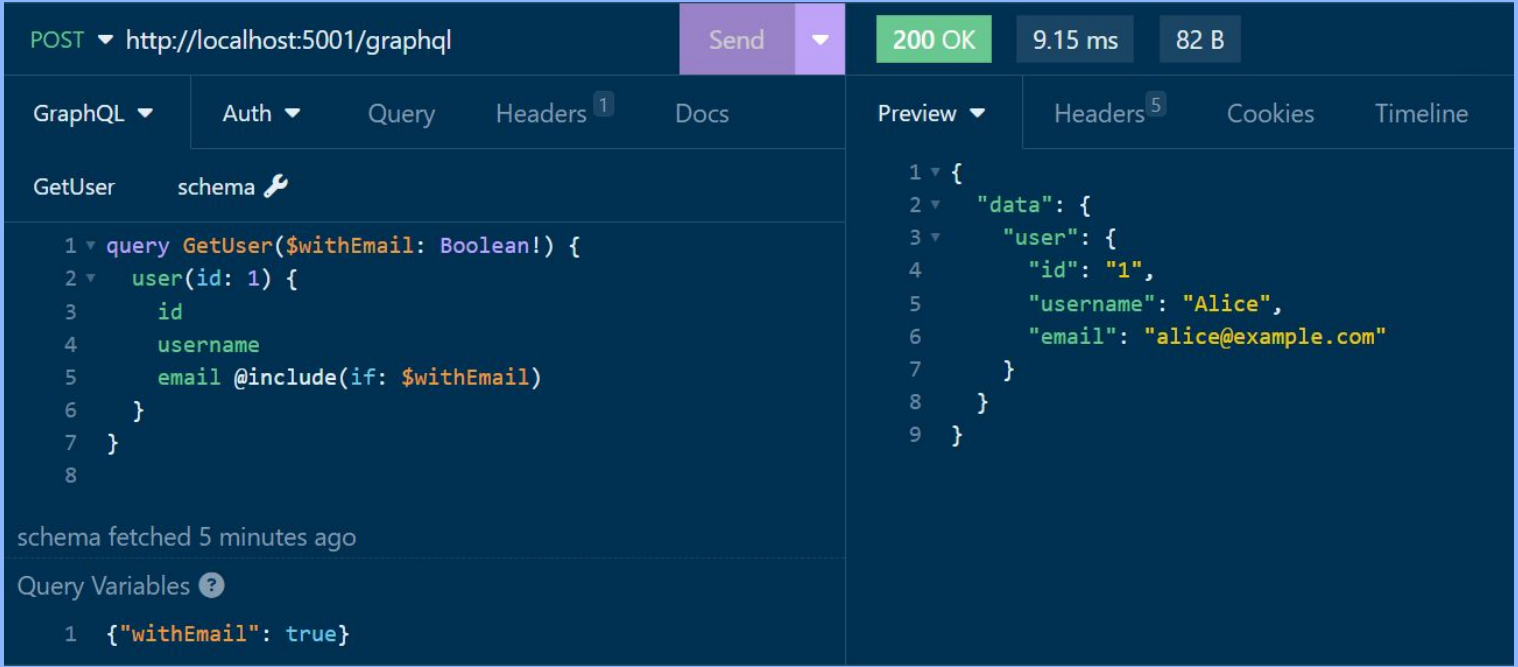
- Use GraphQL where it shines - Example: internal APIs with diverse use cases
- Use GraphQL Federation when it fits your architecture / when a monolith becomes unmanageable

[From “8 Years of GraphQL: Unraveling the Trade-Offs” Talk by Marc-Andre Giroux (GraphQL Conf 2023)]

Federation - Concepts

Directives

A directive decorates part of a GraphQL schema or operation with additional configuration. Denoted using ‘@’ - similar to decorators in Python



```graphql
query GetUser($withEmail: Boolean!) {
  user(id: 1) {
    id
    username
    email @include(if: $withEmail)
  }
}
```

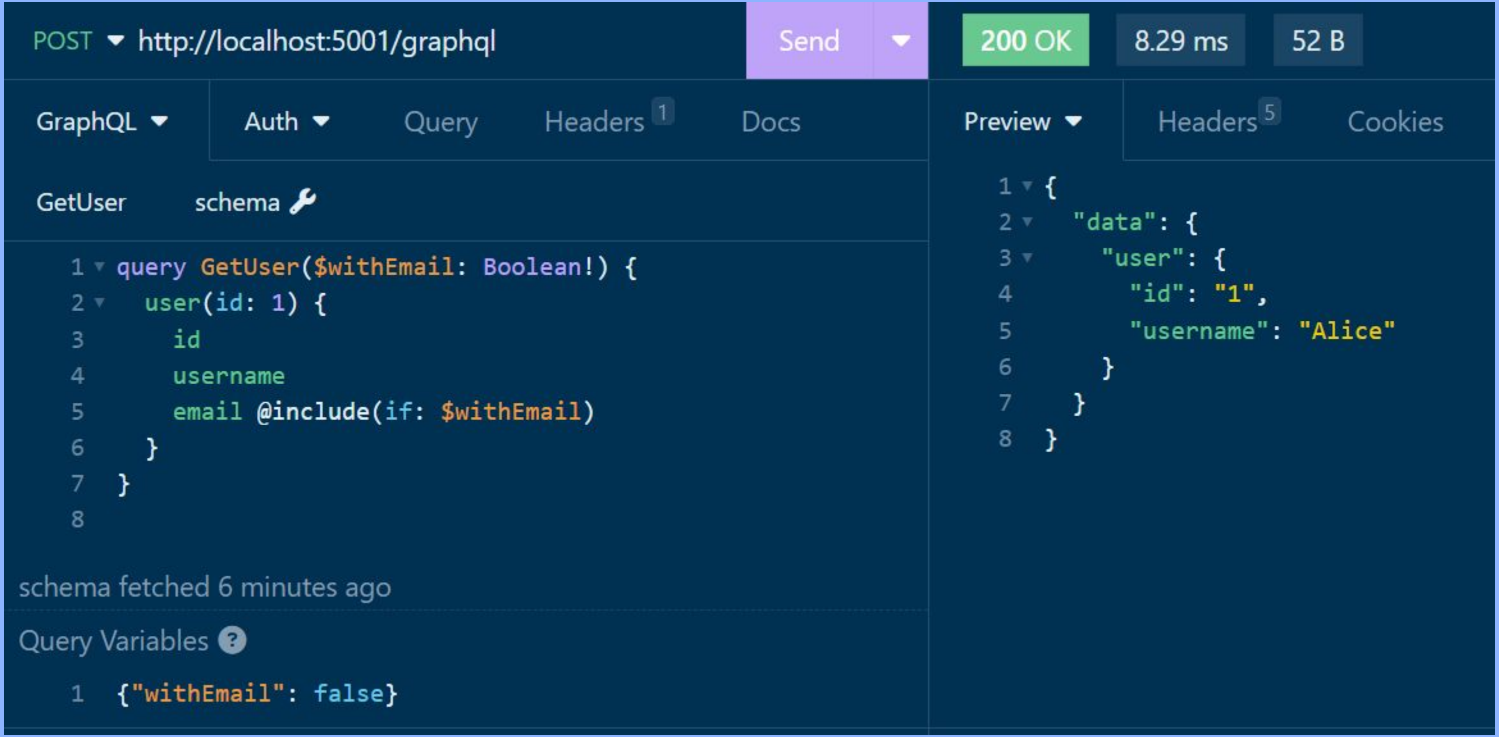
schema fetched 5 minutes ago

Query Variables

```json
{"withEmail": true}
```

```json
{
  "data": {
    "user": {
      "id": "1",
      "username": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

Directives



```graphql
query GetUser($withEmail: Boolean!) {
  user(id: 1) {
    id
    username
    email @include(if: $withEmail)
  }
}
```

schema fetched 6 minutes ago

Query Variables

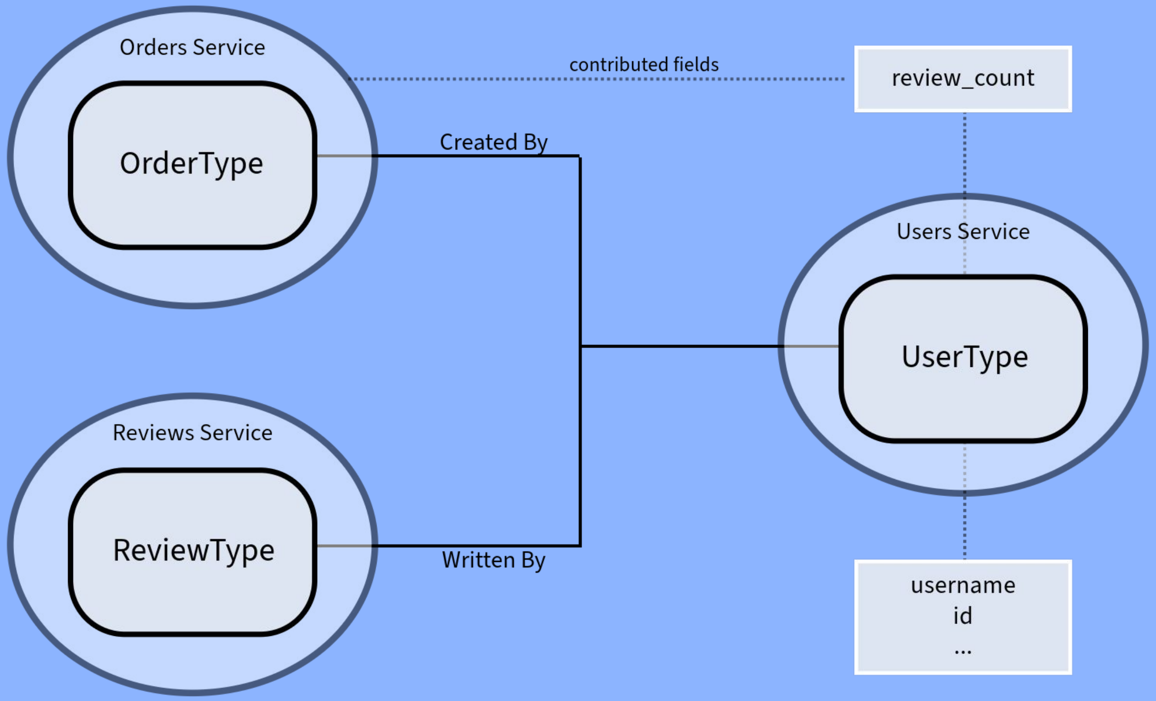
```json
{"withEmail": false}
```

```json
{
  "data": {
    "user": {
      "id": "1",
      "username": "Alice"
    }
  }
}
```

Entity

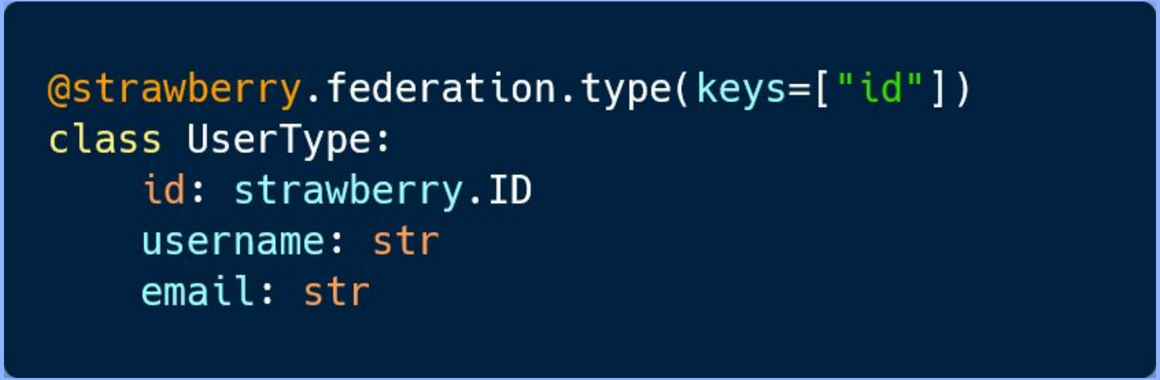
- An Entity in Federation is an object type that can resolve its fields across multiple subgraphs.
- It can be thought of as a GraphQL type which appears across multiple microservice subgraphs.

Entity - UserType



@key directive

- Designates an object type as an entity.
- The @key directive is used to indicate fields that can be used to uniquely identify and fetch an object.



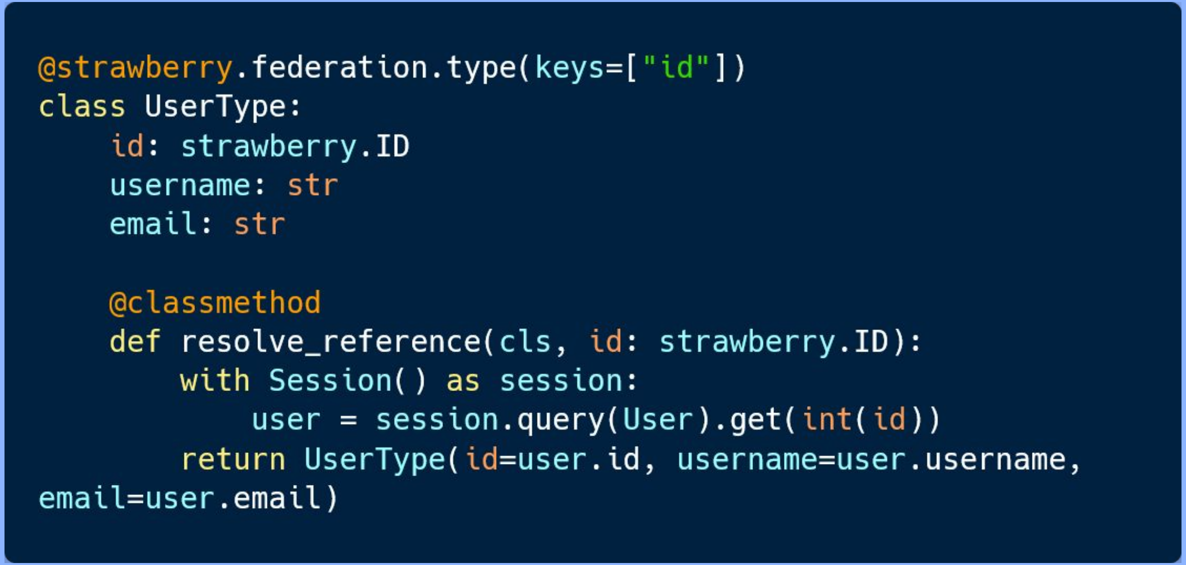
```
@strawberry.federation.type(keys=["id"])
class UserType:
    id: strawberry.ID
    username: str
    email: str
```

Federation Gateway/Router

- Combines multiple microservice schemas and exposes a single combined endpoint
- Intelligent - It holds the logic to resolve entities and shared types

Reference resolver function

The reference resolver function enables the Federation gateway's query planner to resolve a particular entity by its @key fields.



```
@strawberry.federation.type(keys=["id"])
class UserType:
    id: strawberry.ID
    username: str
    email: str

    @classmethod
    def resolve_reference(cls, id: strawberry.ID):
        with Session() as session:
            user = session.query(User).get(int(id))
        return UserType(id=user.id, username=user.username,
email=user.email)
```

Demo

References

- Graphql Specification: https://spec.graphql.org/
- Graphql.org Docs: https://graphql.org/learn/
- Apollo Federation Docs: https://www.apollographql.com/docs/federation/
- Strawberry GraphQL Docs: https://strawberry.rocks/docs
- 8 Years of GraphQL: Unraveling the Trade-Offs: Marc-Andre Giroux

in adarsh-d

adarshd905 𝕏

Thank You

Slides and Demo code: go.adarsh.pizza/flaskcon