GraphQL - when REST API is not enough - lessons learned

Marcin Stachniuk
collibra

Marcin Stachniuk
mstachniuk.github.io

/mstachniuk/graphql-java-example
@MarcinStachniuk

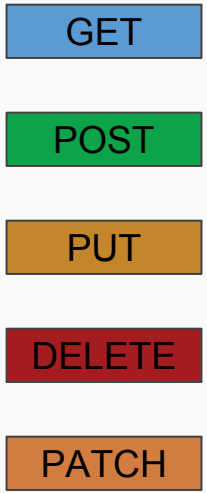
wroclaw.jug.pl

collibra.com

collibra®

WROCŁAW
JAVA USER GROUP

REST - REpresentational State Transfer



https://api.example.com/customers/123

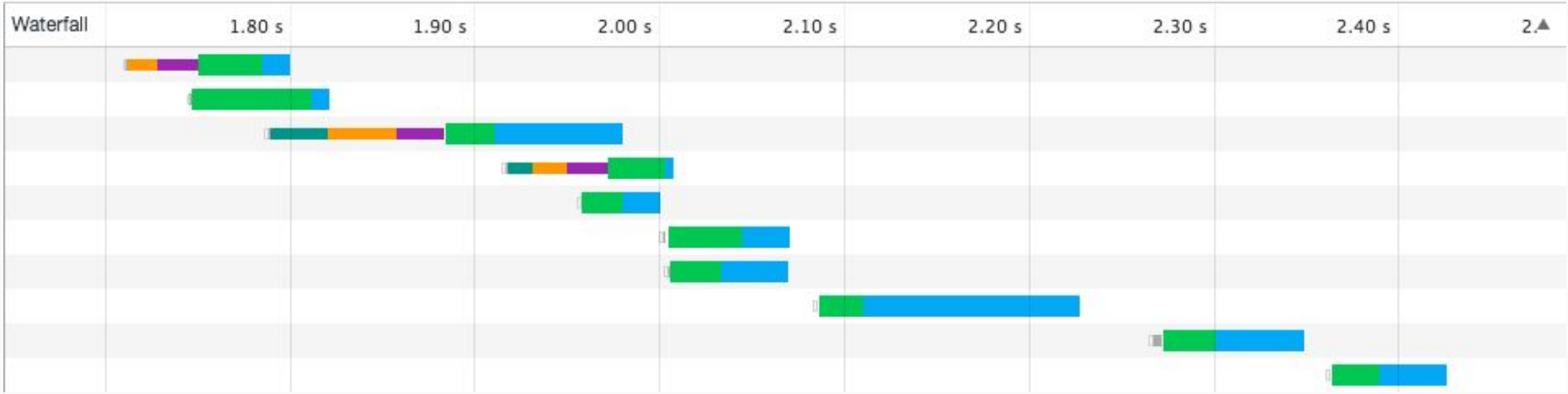
REST fixed response

GET /customers/111

```
{
 "customer": {
  "id": "111",
  "name": "John Doe",
  "email": "john@doe.com",
  "company": {
   "id": "222"
  },
  "orders": [
   {
    "id": "333"
   },
   {
    "id": "444"
   }
  ]
 }
}
```

```
{
 "customer": {
  "id": "111",
  "name": "John Doe",
  "email": "john@doe.com",
  "company": {
   "href": "https://api.example.com/companies/222"
  },
  "orders": [
   {
    "href": "https://api.example.com/orders/333"
   },
   {
    "href": "https://api.example.com/orders/444"
   }
  ]
 }
}
```

REST consequences: several roundtrips



REST response with nested data

GET /customers/111

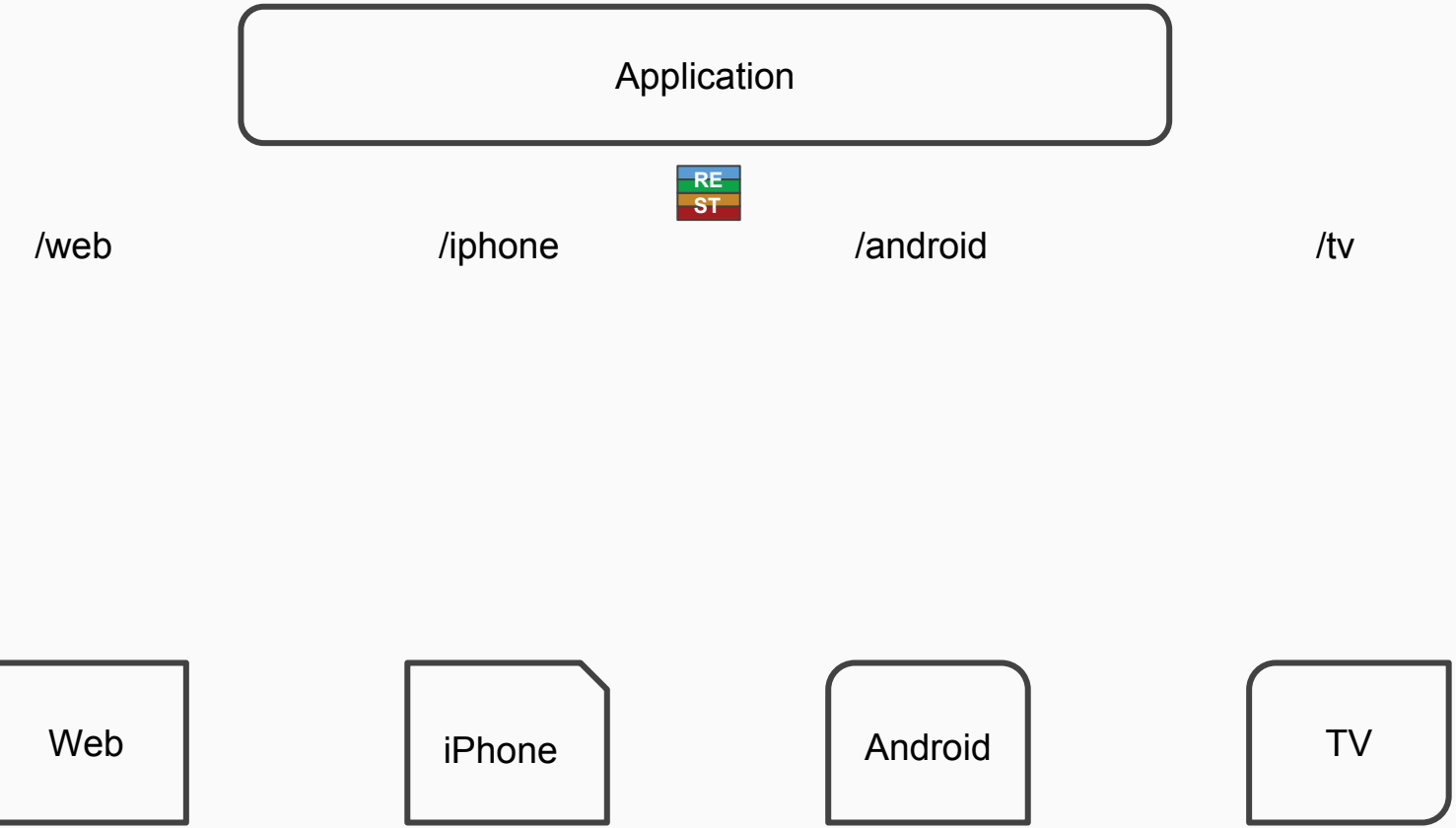

```
{
 "customer": {
  "id": "111",
  "name": "John Doe",
  "email": "john@doe.com",
  "company": {
   "id": "222",
   "name": "My Awesome Corporation",
   "website": "MyAwesomeCorporation.com"
  },
  "orders": [
   {
    "id": "333",
    "status": "delivered",
    "items": [
     {
      "id": "555",
      "name": "Silver Bullet",
      "amount": "42",
      "price": "10000000",
      "currency": "USD",
      "producer": {
       "id": "777",
       "name": "Lorem Ipsum",
       "website": "LoremIpsum.com"
      }
     }
    ]
   },
   {
    "id": "444",
    "name": "Golden Hammer",
    "amount": "5",
    "price": "10000",
    "currency": "USD",
    "producer": {
     ...
    }
   }
  ]}}
```

REST response with nested data and limit fields

GET /customers/111?fields=name,company/*,orders.status,orders.items(name,producer/name)


```
{
 "customer": {
  "id": "111",
  "name": "John Doe",
  "email": "john@doe.com",
  "company": {
   "id": "222",
   "name": "My Awesome Corporation",
   "website": "MyAwesomeCorporation.com"
  },
  "orders": [
   {
    "id": "333",
    "status": "delivered",
    "items": [
     {
      "id": "555",
      "name": "Silver Bullet",
      "amount": "42",
      "price": "10000000",
      "currency": "USD",
      "producer": {
       "id": "777",
       "name": "Lorem Ipsum",
       "website": "LoremIpsum.com"
      }
     }
    ]
   },
   {
    "id": "444",
    "name": "Golden Hammer",
    "amount": "5",
    "price": "10000",
    "currency": "USD",
    "producer": {
     ...
    }
   }
  ]
 }
}
```

Different clients - different needs



Different clients - different needs

Application

/web /iphone /android /tv

Content-Type: application/vnd.myawesomecorporation.com+v1+**web**+json
iphone
android
tv

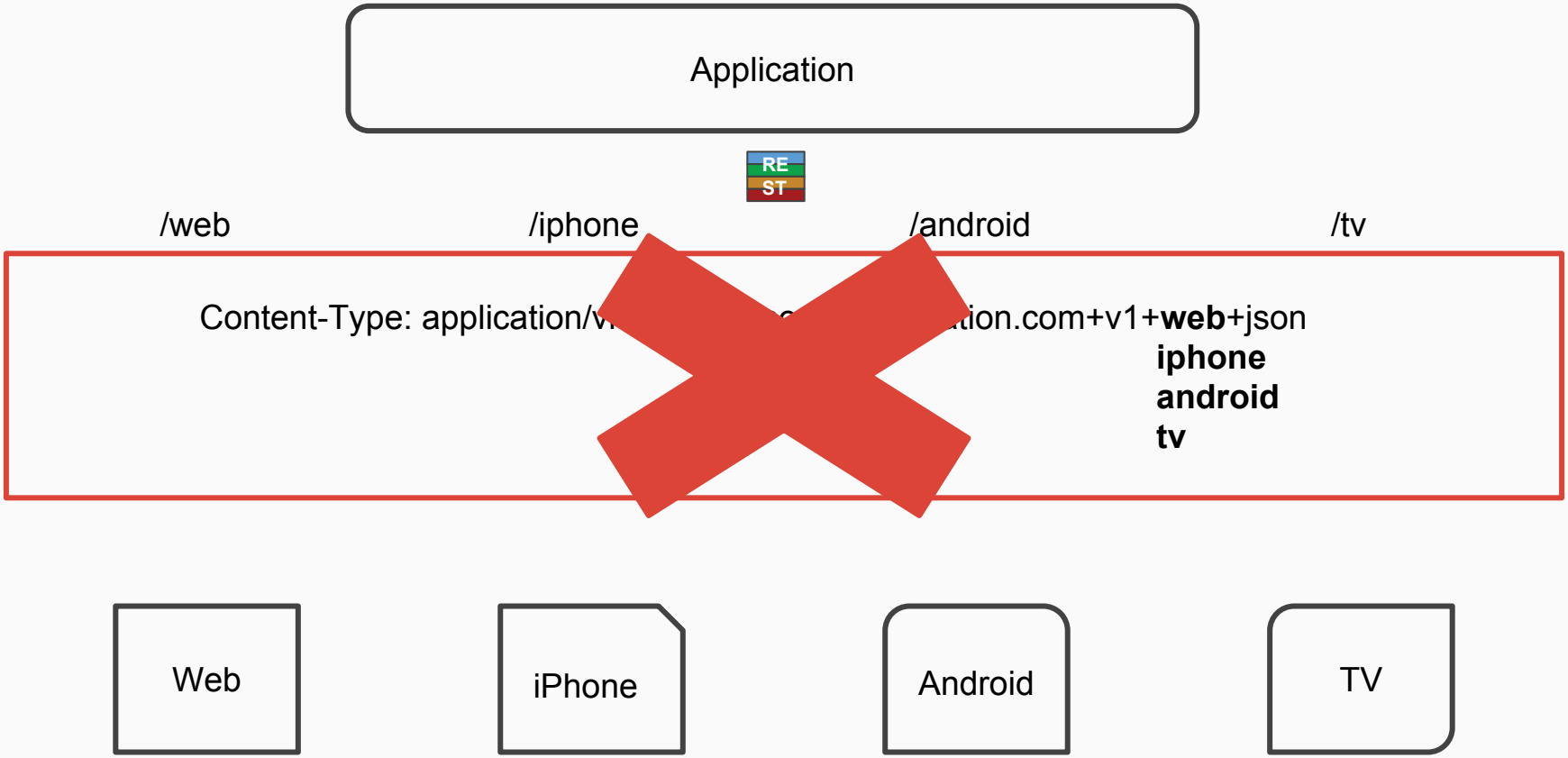
Web

iPhone

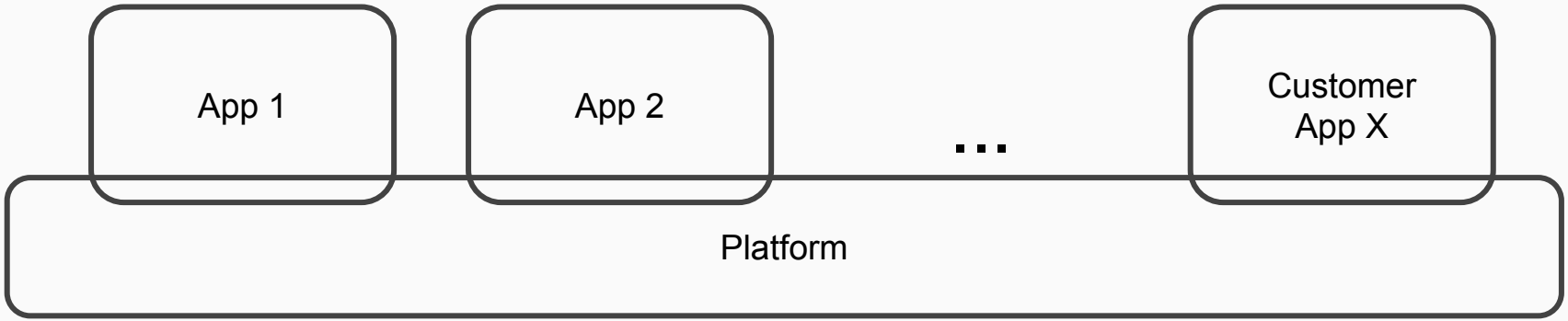
Android

TV

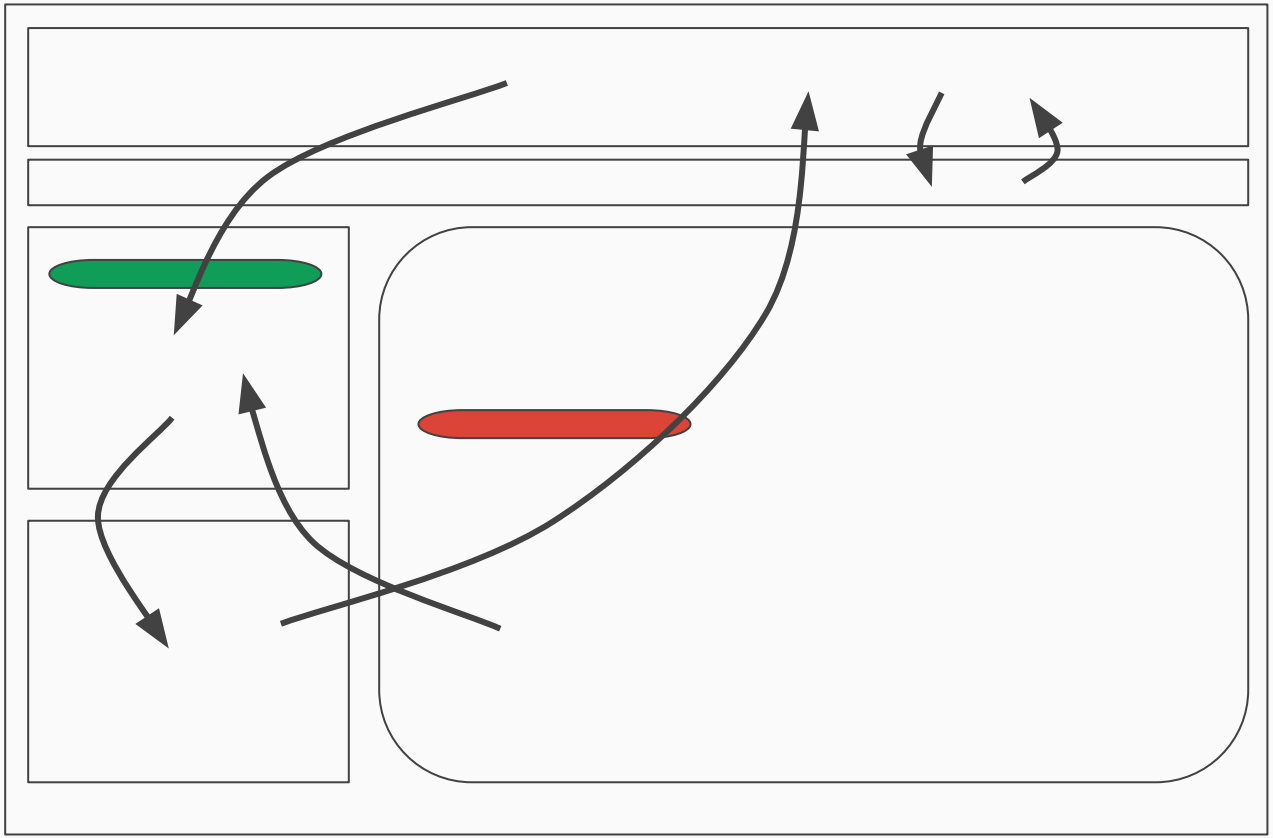
Different clients - different needs



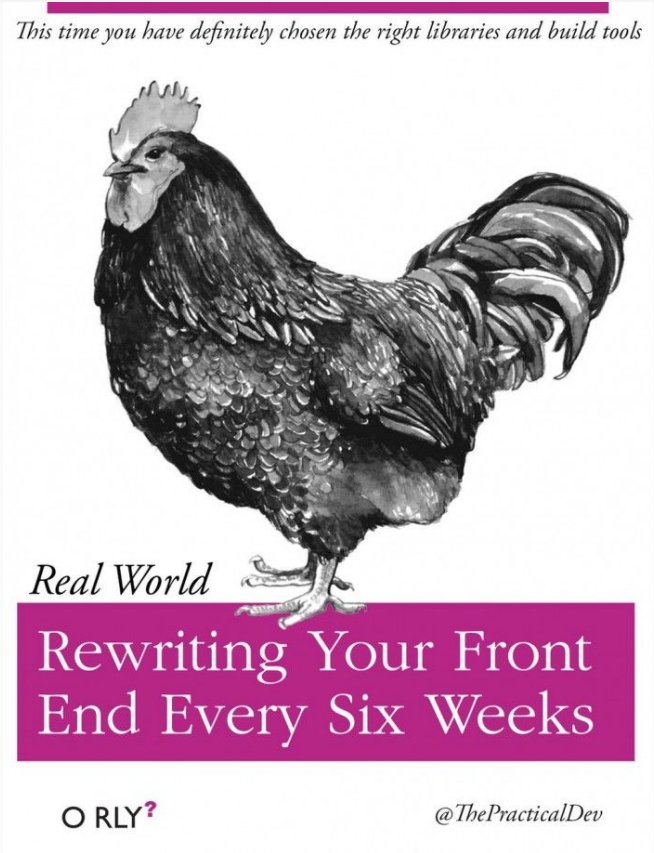
Platform Architecture



Modularisation at UI



New Frontend Framework



ReactJS

TypeScript

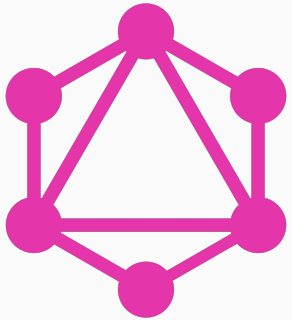
Relay

GraphQL

GraphQL

- Graph Query Language
- Published by Facebook in 2015
- Growth from Facebook Graph API
- Reference implementation in JavaScript
- First version of Java Library: 18 Jul 2015

  https://github.com/graphql-java/graphql-java
- First usage: 21 Sep 2015

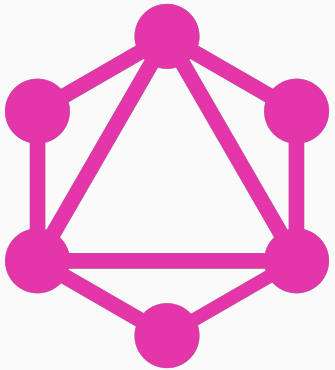


Never add a library to your project few days after init release

- No community
- A lot of bugs
- Bad documentation
- Strict following reference implementation and specification

GraphQL main concepts

- One endpoint for all operations
- Always define in request what you need
- Queries, Mutations and Subscriptions
- Defined by schema



Graphs, graphs everywhere...

GraphQL Simple API

GET /customers/2?fields=id,name,email

```
{
  customer(id: "2") {
    id
    name
    email
  }
}
```

```
{
  "data": {
    "customer": {
      "id": "2",
      "name": "name",
      "email": "a@b.com"
    }
  }
}
```

```
type Customer {
  #fields with ! are required
  id: ID!
  name: String!
  email: String!
}

type Query {
  customer(id: String!): Customer!
}
```

GraphQL Bad Request

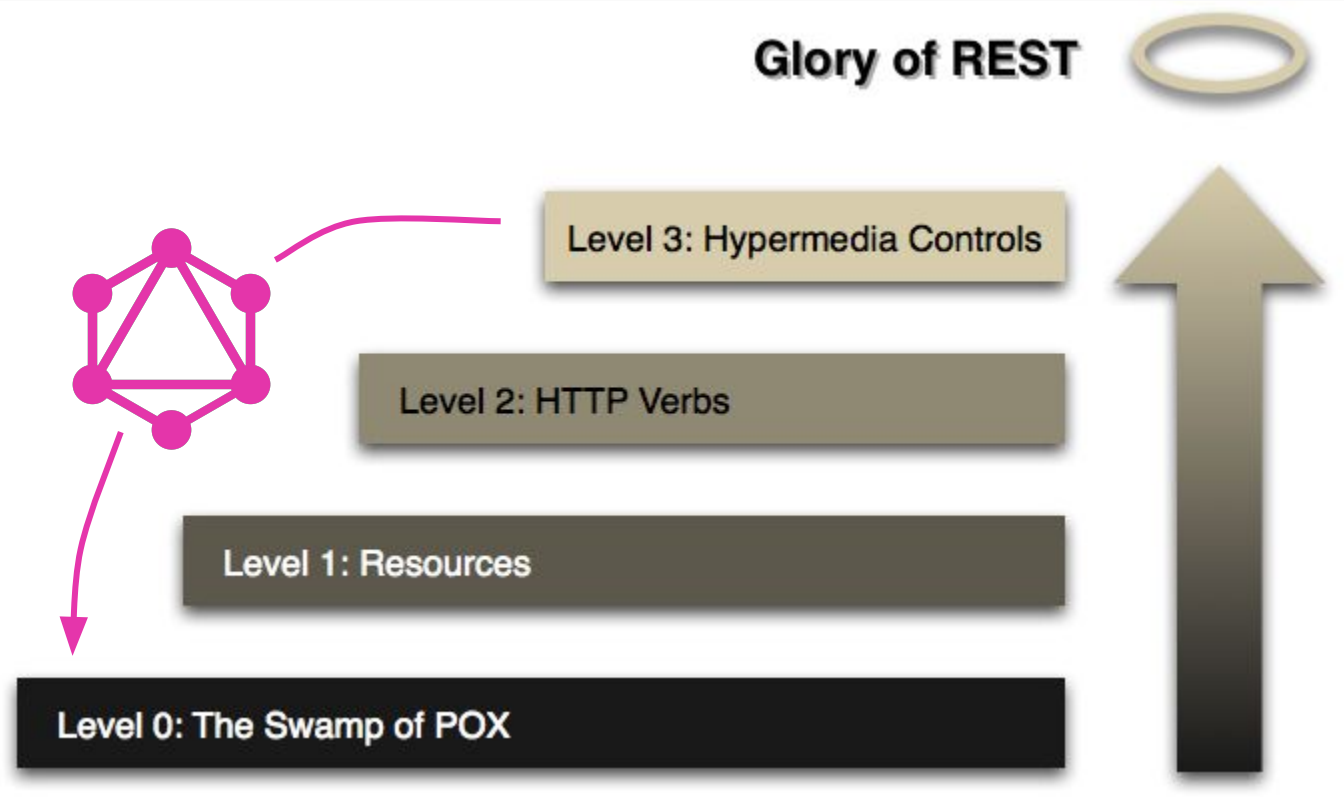
GET /custo!@#$

```
{
  custo!@#$
}
```

```
{
  "data": null,
  "errors": [
    {
      "message": "Invalid Syntax",
      "locations": [
        {
          "line": 2,
          "column": 8
        }
      ],
      "errorType": "InvalidSyntax",
      "path": null,
      "extensions": null
    } ] }
```

http.cat/200

Go back to the roots



GraphQL Simple API

GET /customers/2?fields=id,name,email,company(id,name)

```
{
  customer(id: "2") {
    id
    name
    email
    company {
      id
      name
    }
  }
}
```

```
{
  "data": {
    "customer": {
      "id": "2",
      "name": "name",
      "email": "a@b.com",
      "company": {
        "id": "211",
        "name": "Company Corp."
      }
    }
  }
}
```

```
type Customer {
  id: ID!
  name: String!
  email: String!
  company: Company
}

type Company {
  id: ID!
  name: String!
  website: String!
}

type Query {
  customer(id: String!): Customer!
}
```

GraphQL Simple API

GET /customers/2?fields=id,name,email,orders(id,status)

```
{
  customer(id: "2") {
    id
    name
    orders {
      id
      status
    }
  }
}
```

```
{
  "data": {
    "customer": {
      "id": "2",
      "name": "name",
      "orders": [
        {
          "id": "55",
          "status": "NEW"
        },
        {
          "id": "66",
          "status": "DONE"
        }
      ]
    }
  }
}
```

```
type Customer {
  id: ID!
  name: String!
  email: String!
  company: Company
  orders: [Order]
}

type Order {
  id: ID!
  status: Status
}

type Status {
  NEW, CANCELED, DONE
}
```

How to implement DataFetcher for queries

GET /customers/2?fields=id,name,email,orders(id,status)

```
{
  customer(id: "2") {
    id
    name
    orders {
      id
      status
    }
  }
}
```

```
@Component
public class CustomerFetcher extends PropertyDataFetcher<Customer> {

    @Autowired
    private CustomerService customerService;

    @Override
    public Customer get(DataFetchingEnvironment environment) {
        String id = environment.getArgument("id");
        return customerService.getCustomerById(id);
    }
}
```

How to implement DataFetcher for queries

GET /customers/2?fields=id,name,email,orders(id,status)

```
{
  customer(id: "2") {
    id
    name
    orders {
      id
      status
    }
  }
}
```

```
public class Customer {
   private String id;
   private String name;
   private String email;      // getters are not required
}

public class OrderDataFetcher extends PropertyDataFetcher<List<Order>> {

  @Override
  public List<Order> get(DataFetchingEnvironment environment) {
     Customer source = environment.getSource();
     String customerId = source.getId();
     return orderService.getOrdersByCustomerId(customerId);
  }
}
```

GraphQL mutations

POST /customers PUT /customers/123 DELETE /customers/123 PATCH /customers/123

```
mutation {
  createCustomer(input: {
      name: "MyName"
      email: "me@me.com"
      clientMutationId: "123"
    }) {
    customer {
      id
    }
    clientMutationId
  }
}
```

```
{
  "data": {
    "createCustomer": {
      "customer": {
        "id": "40",
      },
      "clientMutationId":
          "123"
    }
  }
}
```

```
input CreateCustomerInput {
  name: String
  email: String
  clientMutationId: String!
}

type CreateCustomerPayload {
  customer: Customer
  clientMutationId: String!
}

type Mutation {
  createCustomer(input: CreateCustomerInput):
      CreateCustomerPayload!
}
```

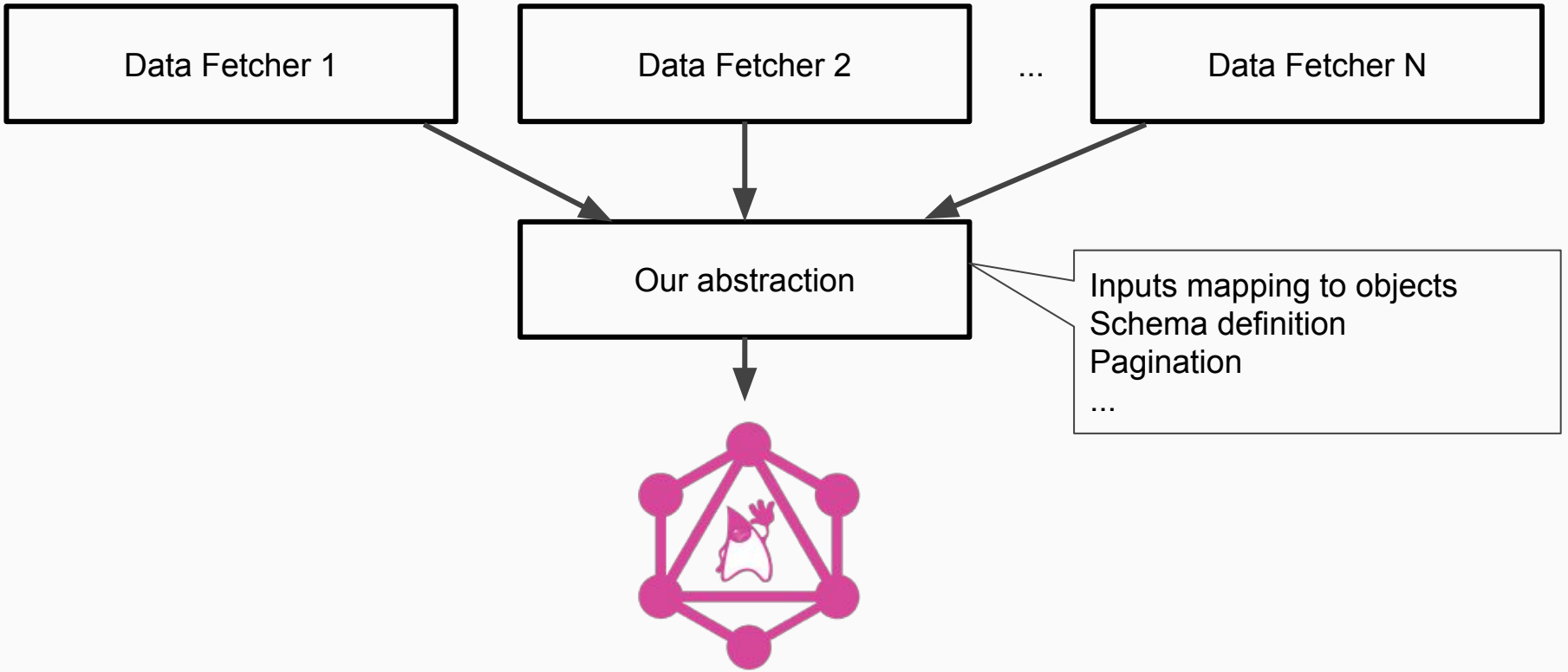
How to implement DataFetcher for mutations

POST /customers PUT /customers/123 DELETE /customers/123 PATCH /customers/123

```
mutation {
  createCustomer(input: {
      name: "MyName"
      email: "me@me.com"
      clientMutationId: "123"
    }) {
     customer {
       id
     }
     clientMutationId
  }
}
```

```
@Override
public CreateCustomerPayload get(DataFetchingEnvironment environment) {
  Map<String, Object> input = environment.getArgument("input");
  String name = (String) input.get("name");
  String email = (String) input.get("email");
  String clientMutationId = (String) input.get("clientMutationId");
  Customer customer = customerService.create(name, email);
  return new CreateCustomerPayload(customer, clientMutationId);
}
```

Abstraction over GraphQL Java



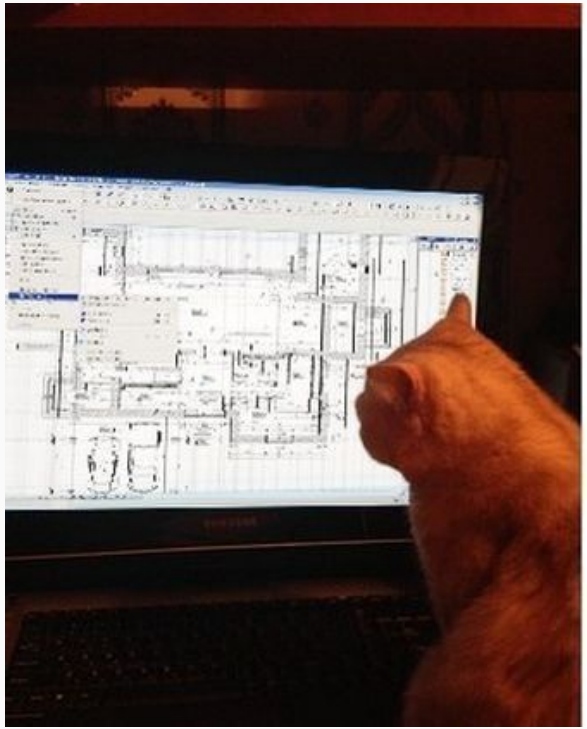
Abstraction is not good if you don’t understand how it works under the hood

- Copy paste errors
- Wrong usage
- Hard to update to new version

GraphQL can do more!

- Variables
- Aliases
- Fragments
- Operation name
- Directives
- Interfaces
- Unions

How to define your schema?



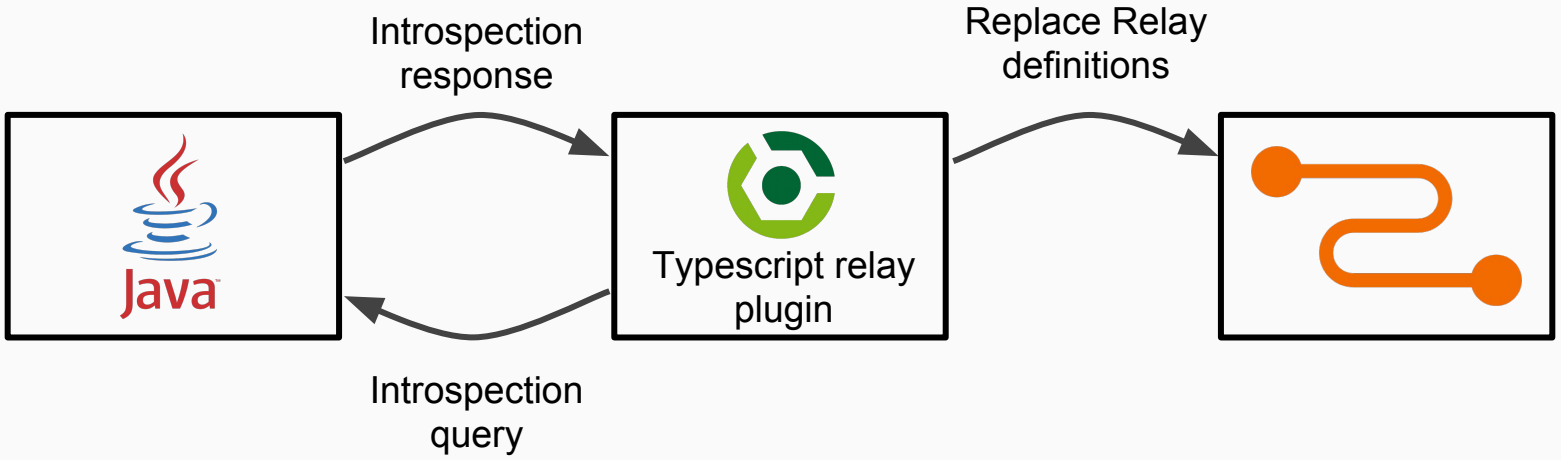
Code First approach

```
private GraphQLFieldDefinition customerDefinition() {
  return GraphQLFieldDefinition.newFieldDefinition()
        .name("customer")
        .argument(GraphQLArgument.newArgument()
              .name("id")
              .type(new GraphQLNonNull(GraphQLString)))
        .type(new GraphQLNonNull(GraphQLObjectType.newObject()
              .name("Customer")
              .field(GraphQLFieldDefinition.newFieldDefinition()
                    .name("id")
                    .description("fields with ! are requred")
                    .type(new GraphQLNonNull(GraphQLID))
                    .build())
              ….
              .build()))
        .dataFetcher(customerFetcher)
        .build();
}
```

Schema First approach

```
type Query {
  customer(id: String!): Customer!
}


type Customer {
  #fields with ! are required
  id: ID!
  name: String!
  email: String!
  company: Company
  orders: [Order]
}
```

Code First approach - How to build



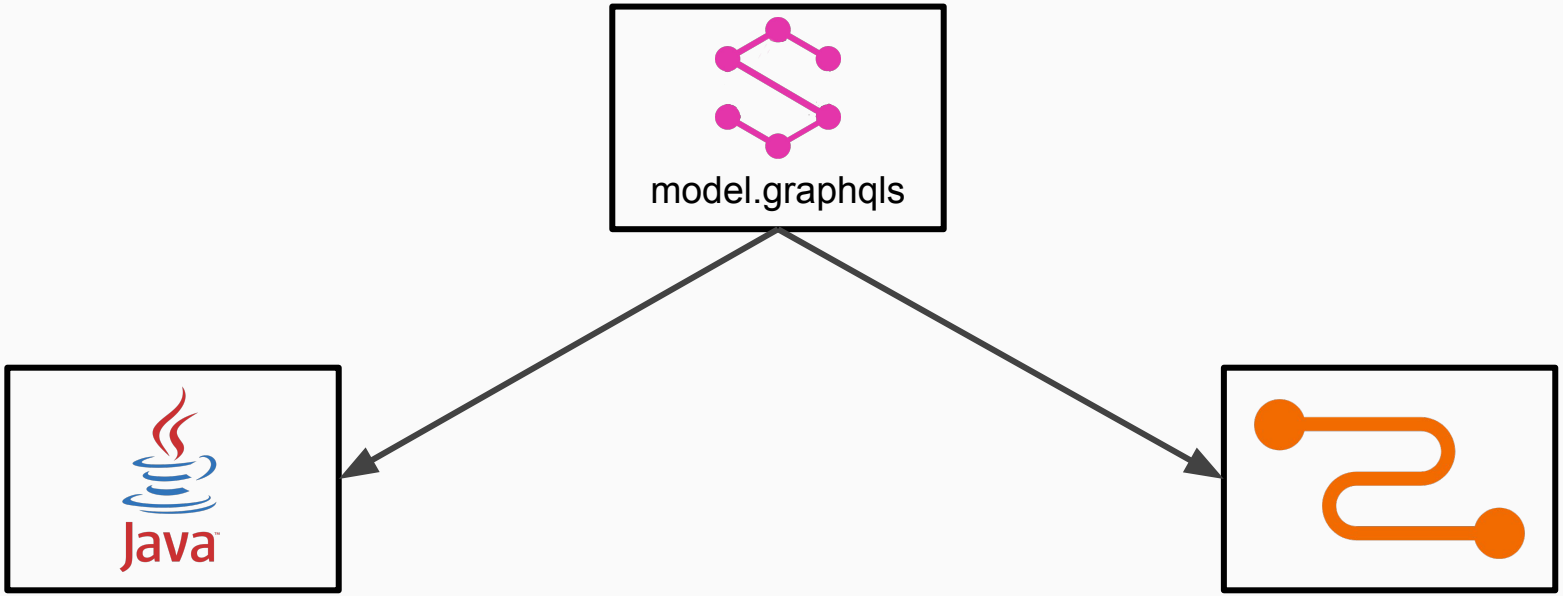
Schema First approach

```
type Customer {
  # fields with ! are required
  id: ID!
  name: String!
  email: String!
  company: Company
  orders: [Order]
}
```

*.graphqls

```
SchemaParser schemaParser = new SchemaParser();
File file = // ...
TypeDefinitionRegistry registry = schemaParser.parse(file);
SchemaGenerator schemaGenerator = new SchemaGenerator();
RuntimeWiring runtimeWiring = RuntimeWiring.newRuntimeWiring()
    .type("Query", builder ->
        builder.dataFetcher("customer", customerFetcher))
    // ...
    .build();
return schemaGenerator.makeExecutableSchema(registry, runtimeWiring);
```

Schema First approach - project building diagram



Schema First Approach is better

Code First approach:

- Hard to maintain
- It was the only way at the beginning to define a schema
- No possibility to mix both
- No easy way to migrate to Schema First

Schema First Approach:

- Easy to maintain and understand
- Helps organise work
- Demo schema is 2x smaller

GraphQL - How to define pagination, filtering, sorting?

Pagination:

- before, after
- offset, limit

Filtering:

- filter: {name: “Bob” email: “%@gmail.com”}
- filter: {
  OR: [{
  AND: [{
  releaseDate_gte: "2009"
  }, {
  title_starts_with: "The Dark Knight"
  }]
  }, name: “Bob”
  }

Sorting:

- orderBy: ASC, DESC
- sort: NEWEST, IMPORTANCE

GraphQL is not full query language

- Flexibility
- Less common conventions
- Dgraph.io created GraphQL+-

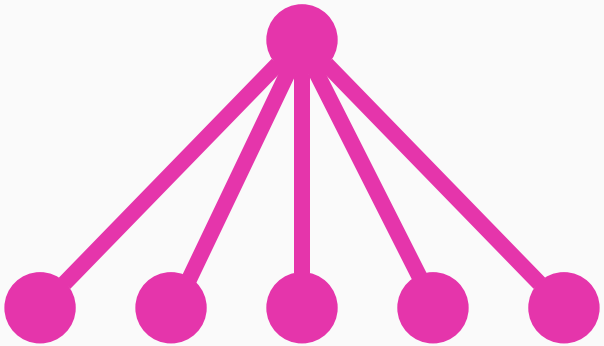
N+1 problem

```
{
  customers {      <-----  1 call
    id
    name
    orders {       <-----  n calls
      id
      status
    }
  }
}
```

java-dataloader

- Add async BatchLoader
- Add caching

If you have N + 1 problem
use java-dataloader



Bad GraphQL API definition - examples

```
{
  customer(id: "2") { … }
  customerFull(id: "2") { … }
  customerFull2(id: "2") { … }
  customerWithDetails(id: "2") { … }
  ...
}
```

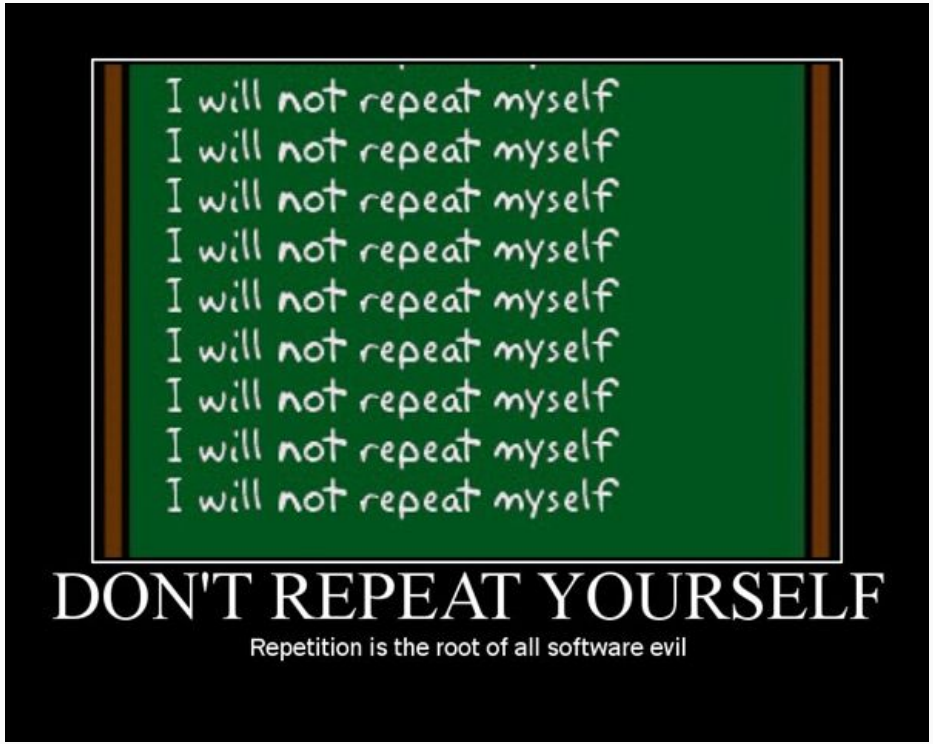
Bad GraphQL API definition - examples

```
{
  usersOrGroups(ids: ["User:123", "UserGroup:123"]) {
    ... on User {
      id
      userName
    }
    ... on UserGroup {
      id
      name
    }
  }
}
```

```
{
  user(id: "123") {
    id
    userName
  }
  userGroup(id: "123") {
    id
    userName
  }
}
```

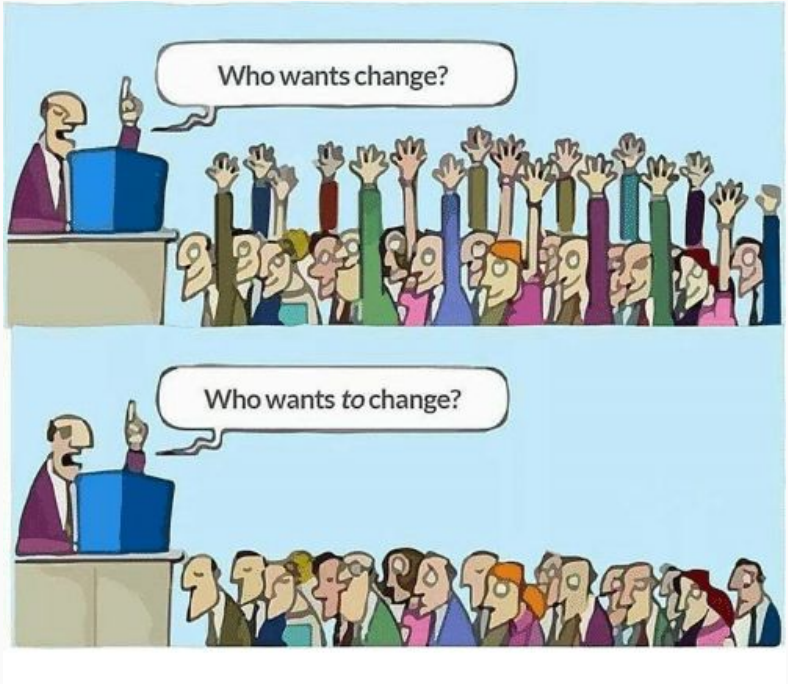
Bad GraphQL API definition - examples

```
{
    orders (input: {
        status: "NEW"
        first: "2"
        offset: "3"
    }, first: "1", offset: "3") {
        items { … }
}
```



Thinking shift is a key

- Let’s think in graphs and NOT in endpoints / resources / entities / DTOs
- Bad design of our API



GraphQL Testing

Testing GraphQL

```
@SpringBootTest
@ContextConfiguration(classes = Main)
class CustomerFetcherSpec extends Specification {

  @Autowired
  GraphQLSchema graphQLSchema

  GraphQL graphQL

  def setup() {
    graphQL = GraphQL.newGraphQL(graphQLSchema).build()
  }
```

Testing GraphQL

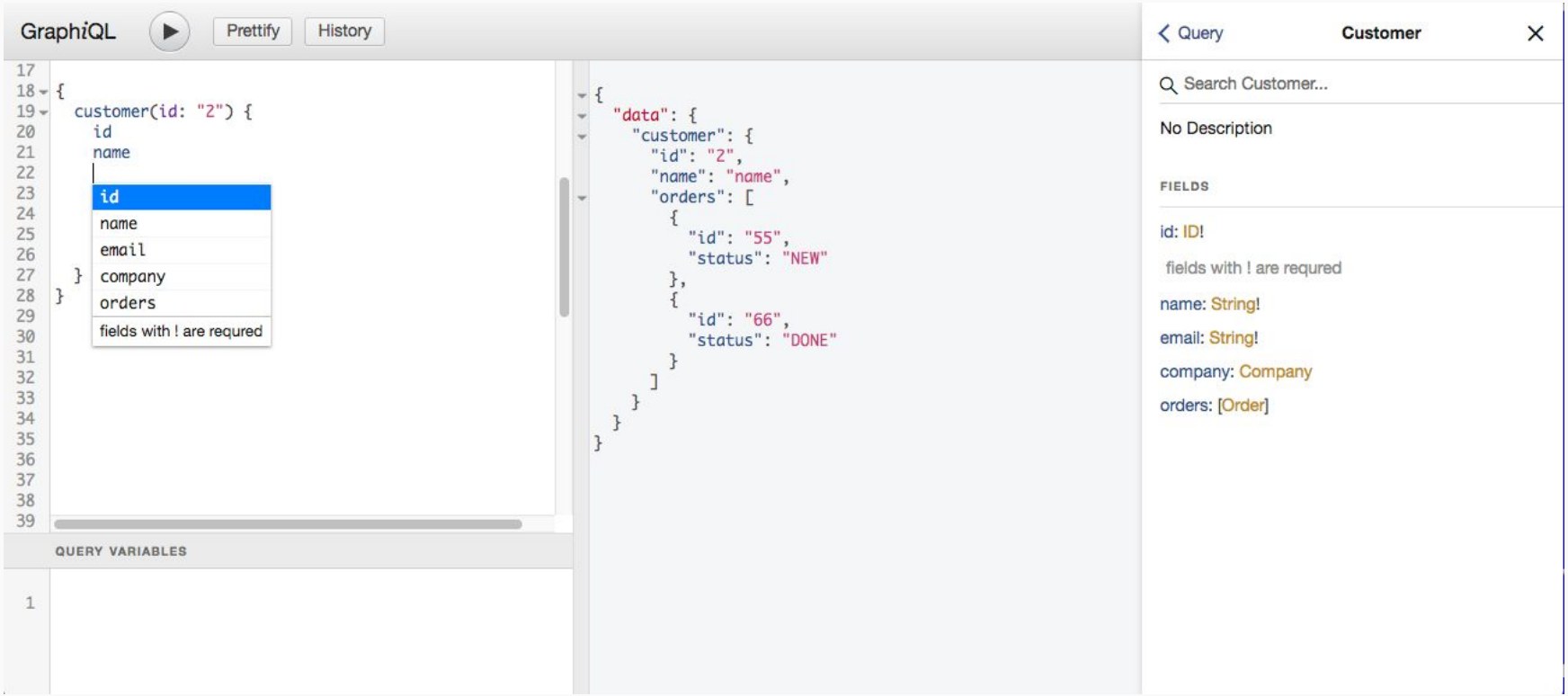
```
def "should get customer by id"() {
   given:
   def query = """{ customer(id: "2") { … } }"""

   def expected = [ "customer": [ … ] ]

   when:
   def result = graphQL.execute(query)

   then:
   result.data == expected
}
```

Testing is easy

Trap Adventure 2 - "The Hardest Retro Game"

Tools

GraphiQL: github.com/graphql/graphiql



More libraries and projects related to graphql-java

https://github.com/graphql-java/awesome-graphql-java

Tooling is nice now

Summary

GraphQL Pros:

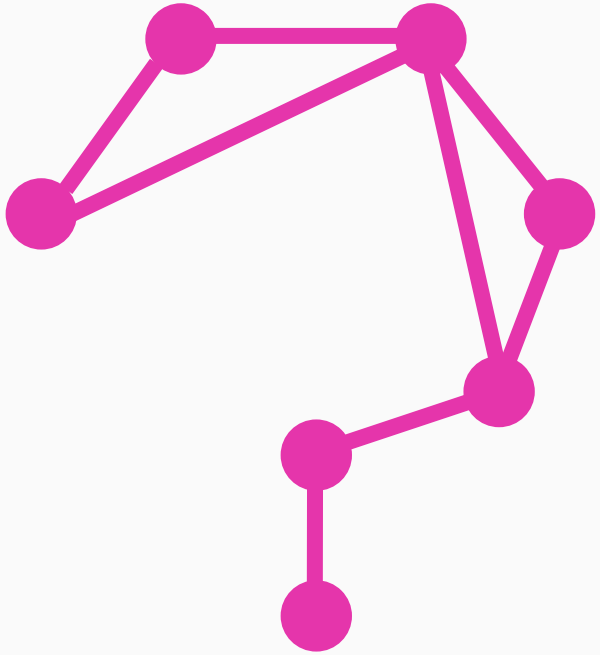
- Nice alternative to REST
- It can be used together with REST
- Good integration with Relay / ReactJS
- You get exactly what you want to get
- Good for API with different clients
- Good to use on top of existing API
- Self documented
- Easy testing
- Nice tooling

GraphQL Cons:

- High entry barrier
- Hard to return simple Map
- Not well know (yet)
- Performance overhead
- A lot of similar code to write

Nothing is a silver bullet

Q&A



GraphQL - when REST API is not enough - lessons learned

Marcin Stachniuk

collibra

Thank you!