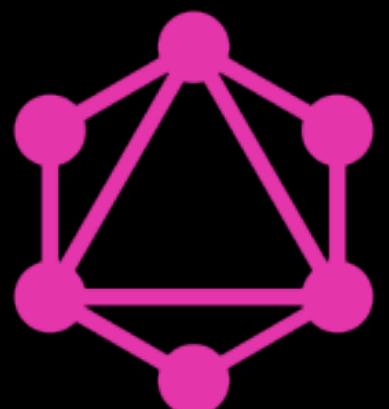
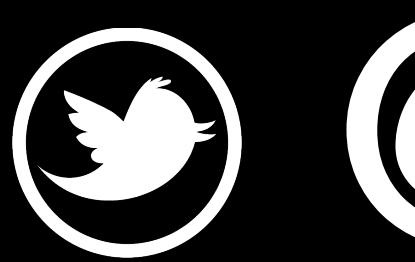


GraphQL

the holy contract between
client and server





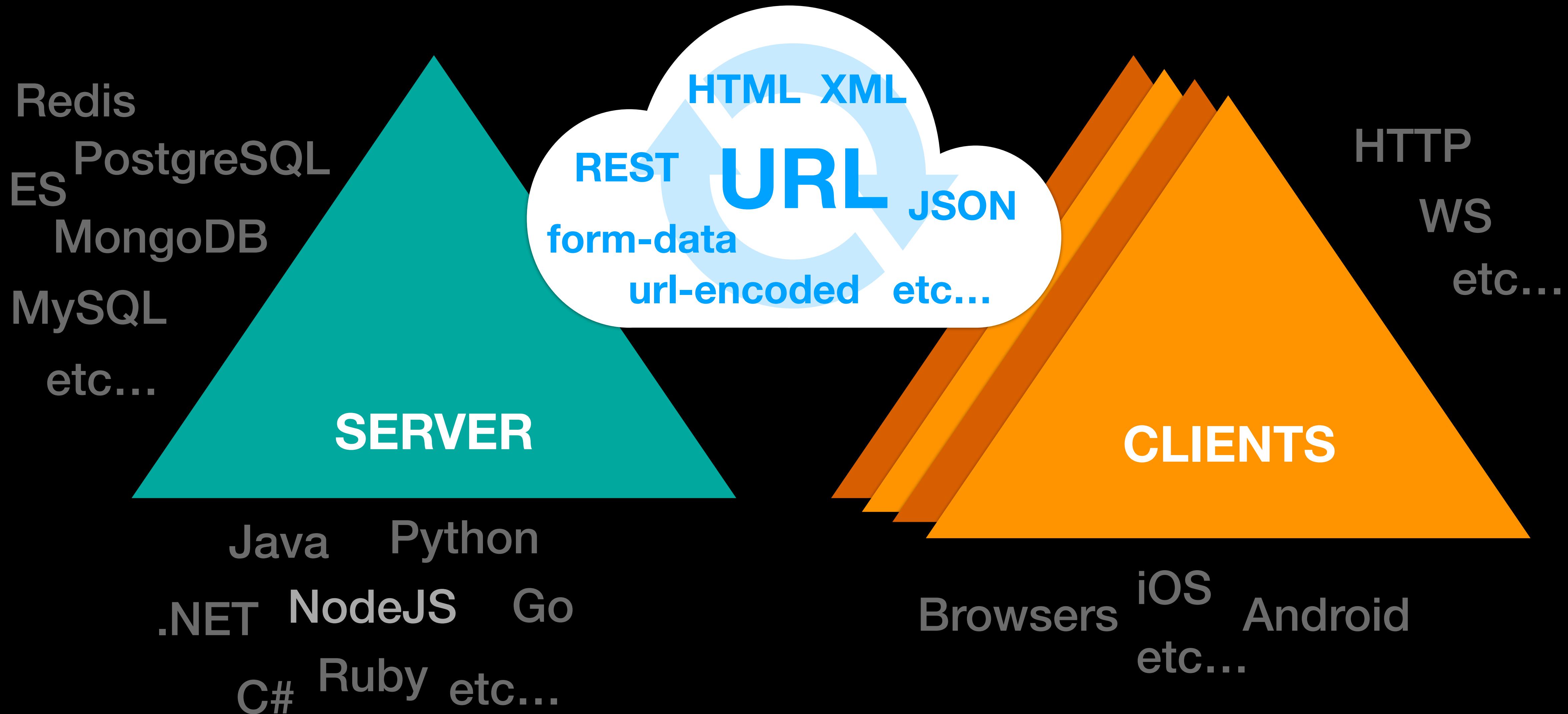
@nodkz



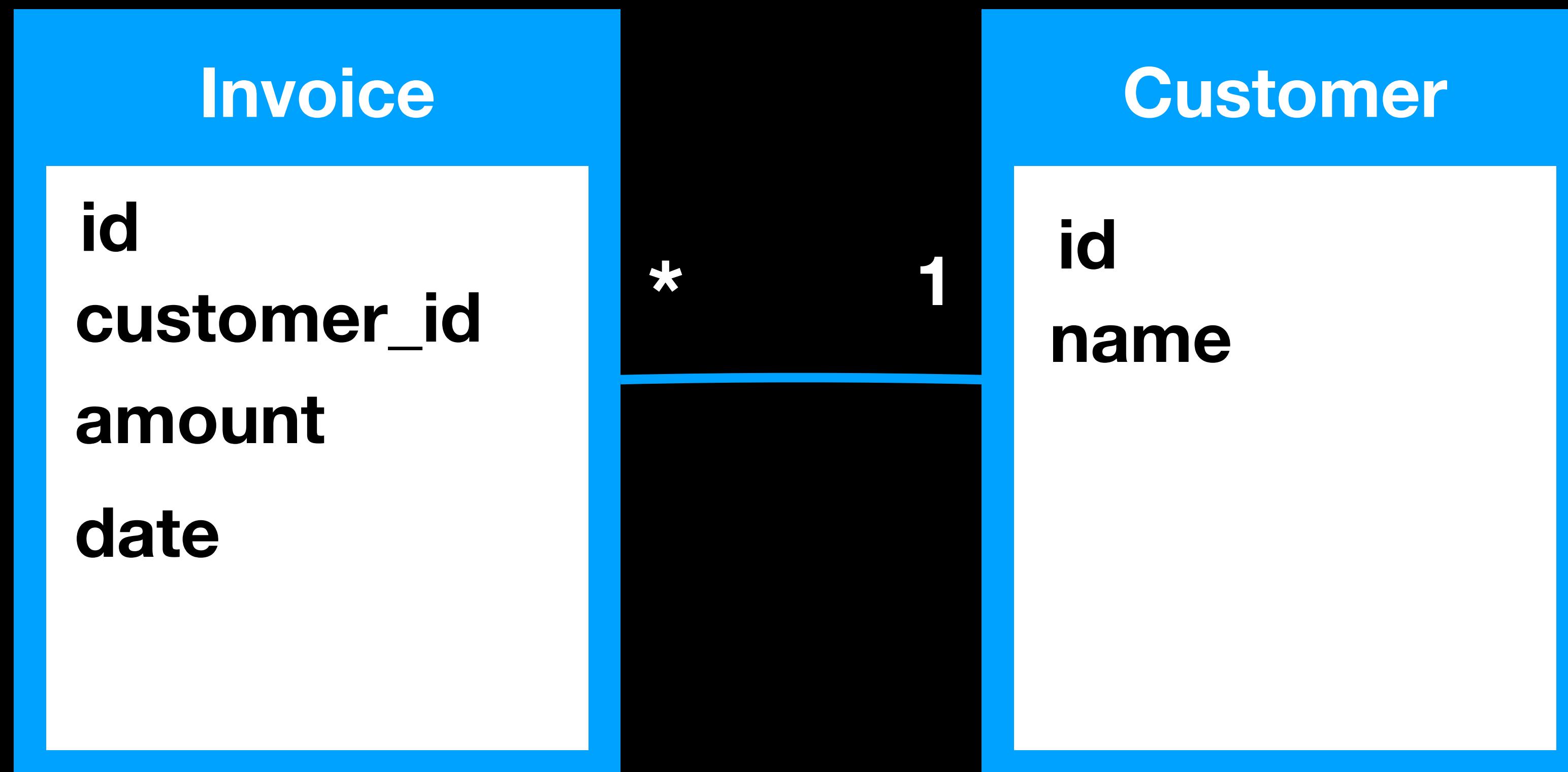
Pavel Chertorogov
with GraphQL since 2015

Client-server intro

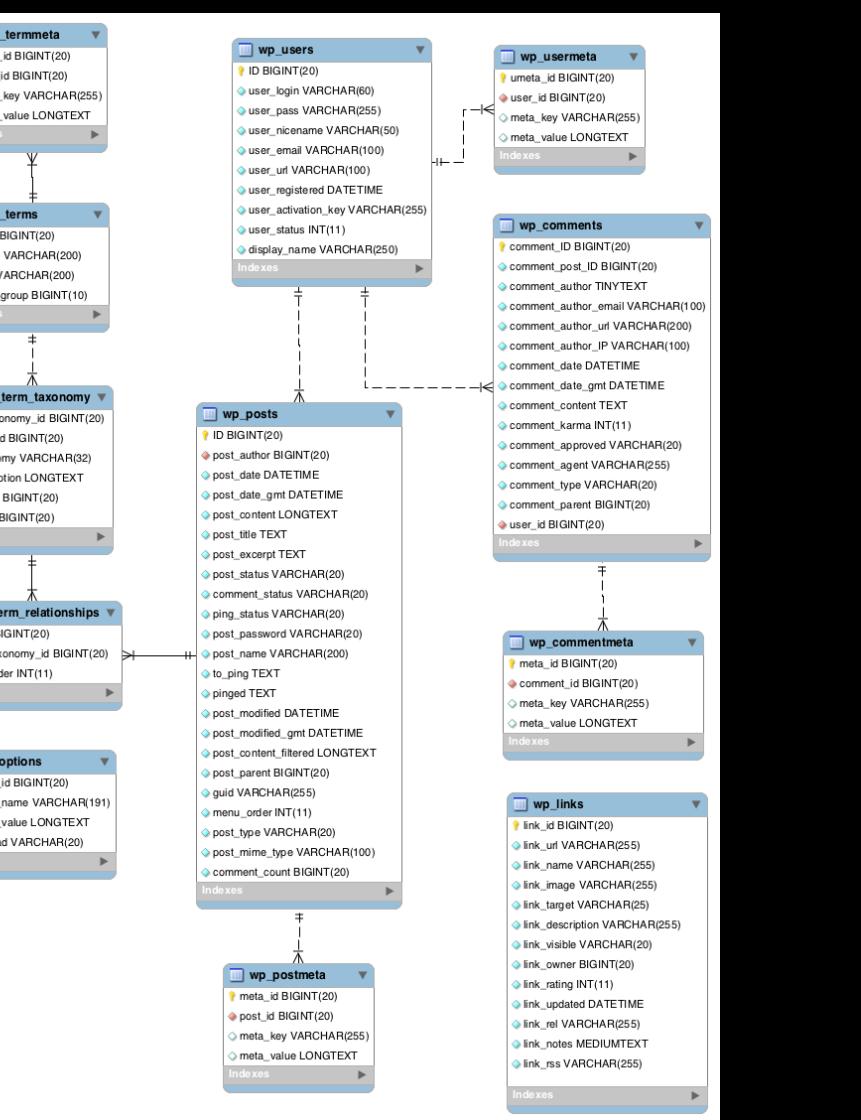
Client-server apps



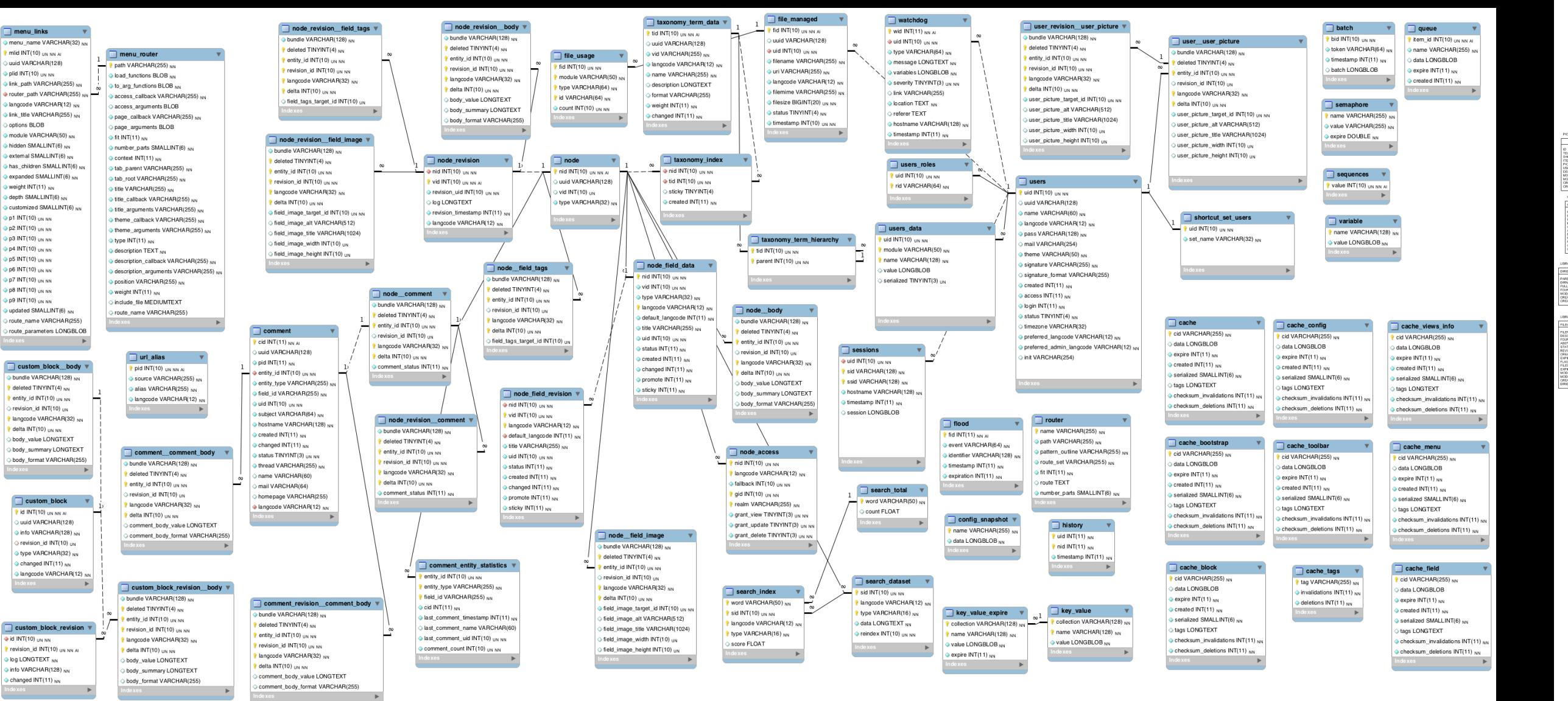
Backend capabilities



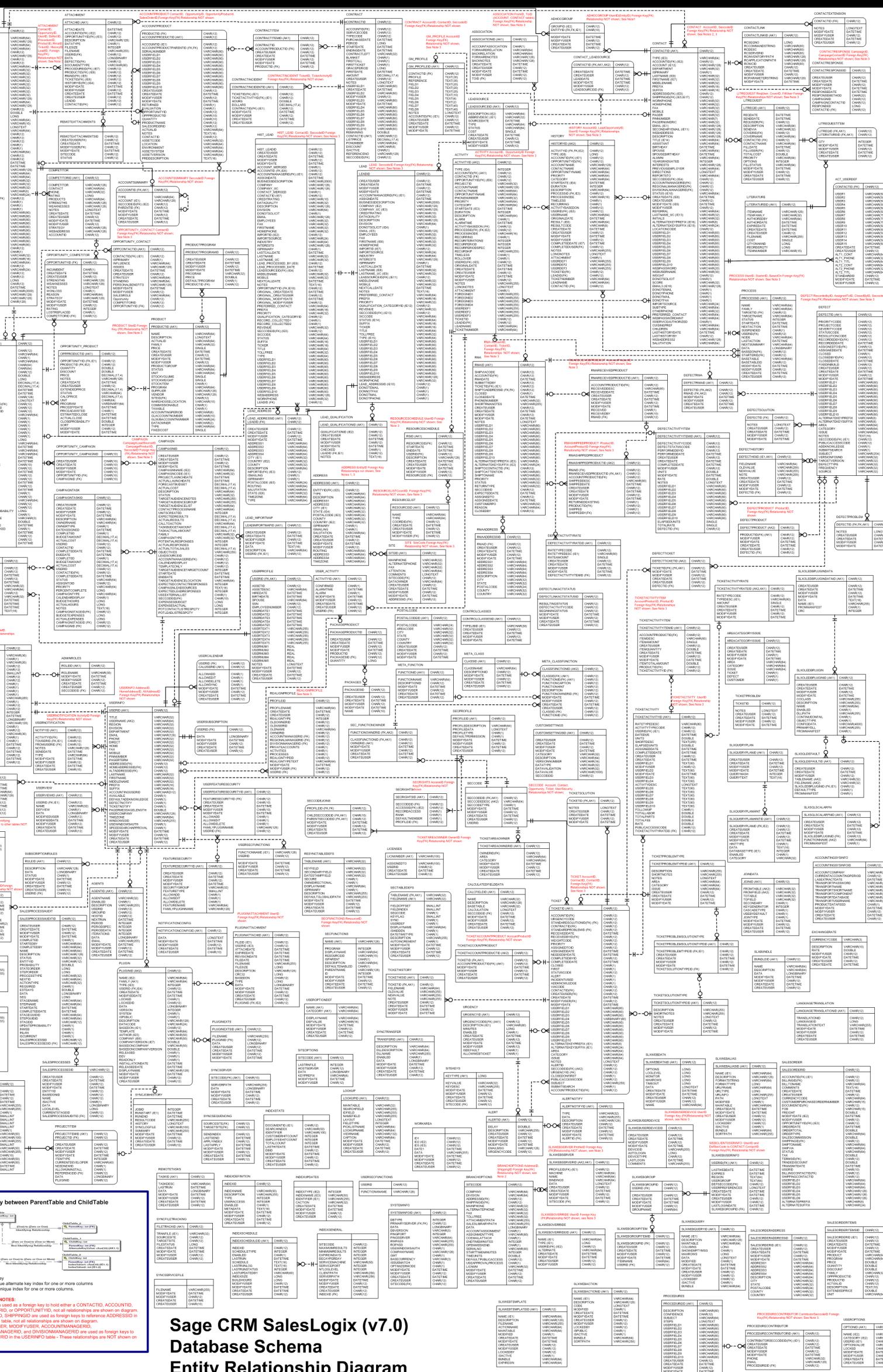
Simple DB Schema Diagram



Wordpress 4.4.2 Schema (12 tables)



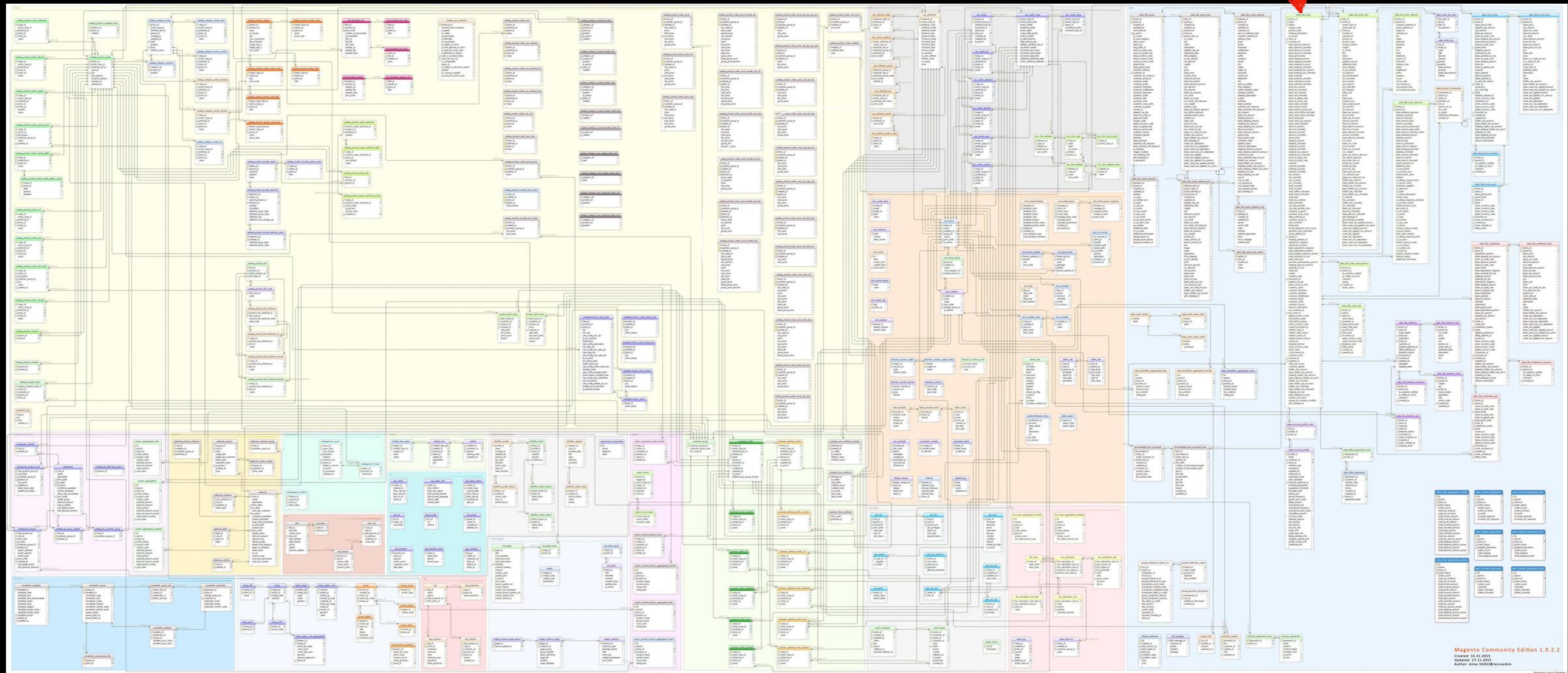
ipal8 Schema (60 tables)



alesLogix Schema (168 tables)

... and even more Monster with 333 tables

Order table (138 fields)

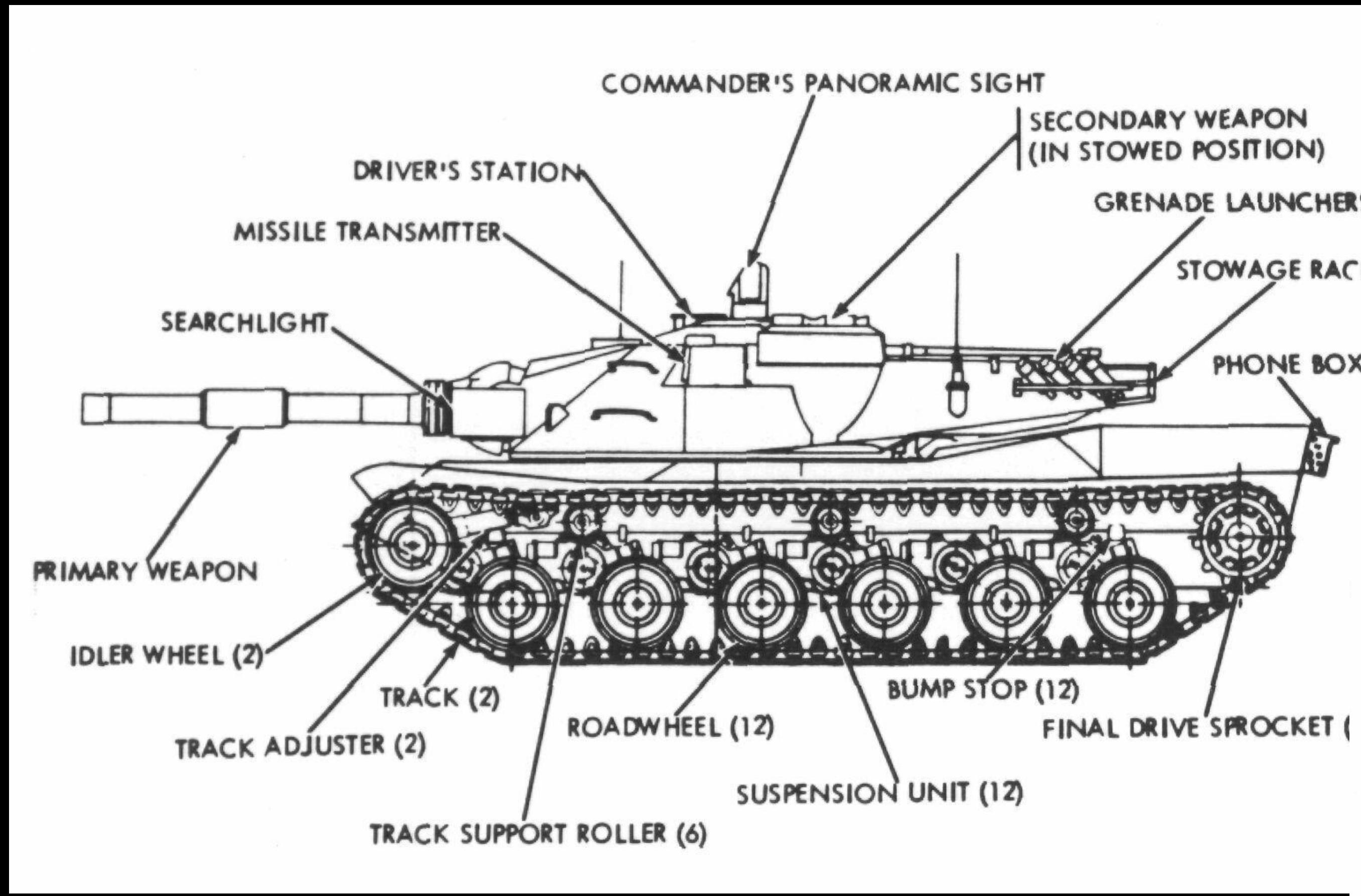


**How to create API
for this db-schema
HELL?**

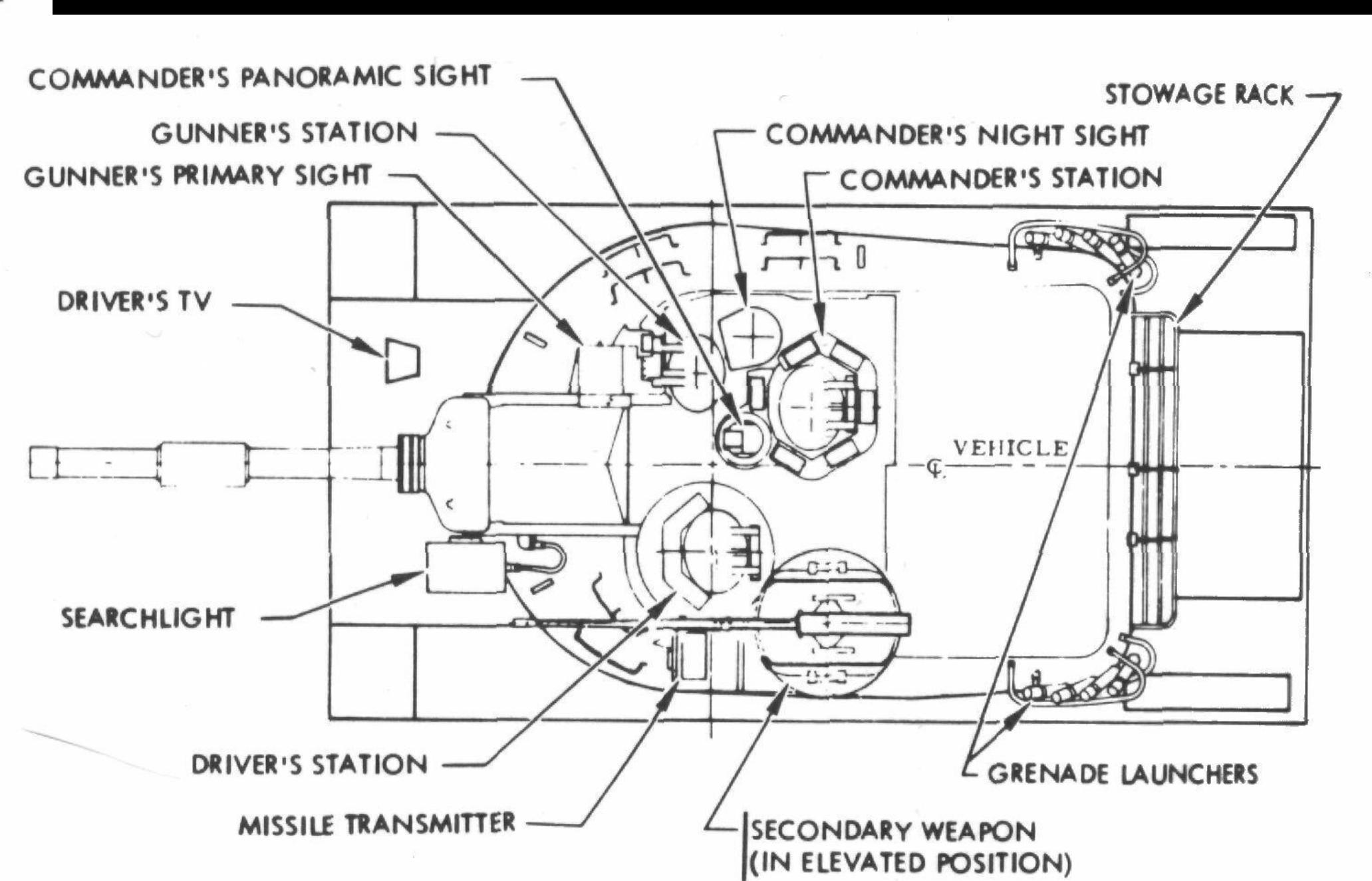


Wait! I have a better plan!

This is GraphQL Schema



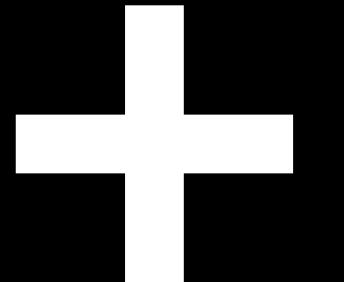
Wikipedia MBT-70 schema



GraphQL basics

GraphQL - is a ...

query
language
for APIs



query
executor
on Schema

for Frontenders

for Backenders

C#
.NET
Python
NodeJS
Ruby
Go
etc...

GraphQL Query Language

```
1
2  query {
3    viewer {
4      product {
5        name
6        unitPrice
7        category {
8          name
9          description
10         }
11       }
12     }
13   }
```

```
  {
    "data": {
      "viewer": {
        "product": {
          "name": "Uncle Bob's Organic Dried Pears",
          "unitPrice": 30,
          "category": {
            "name": "Produce",
            "description": "Dried fruit and bean curd"
          }
        }
      }
    }
  }
```



GraphQL query



Response in JSON

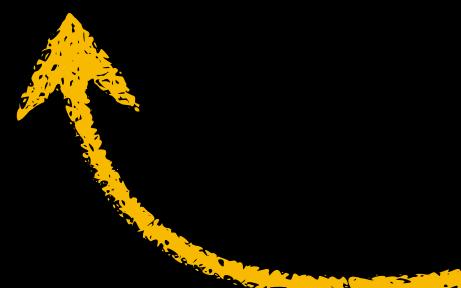
GraphQL Query Language

```
1
2 query {
3   viewer {
4     product {
5       name
6       unitPrice
7       category {
8         name
9         description
10      }
11    }
12  }
13}
14
```



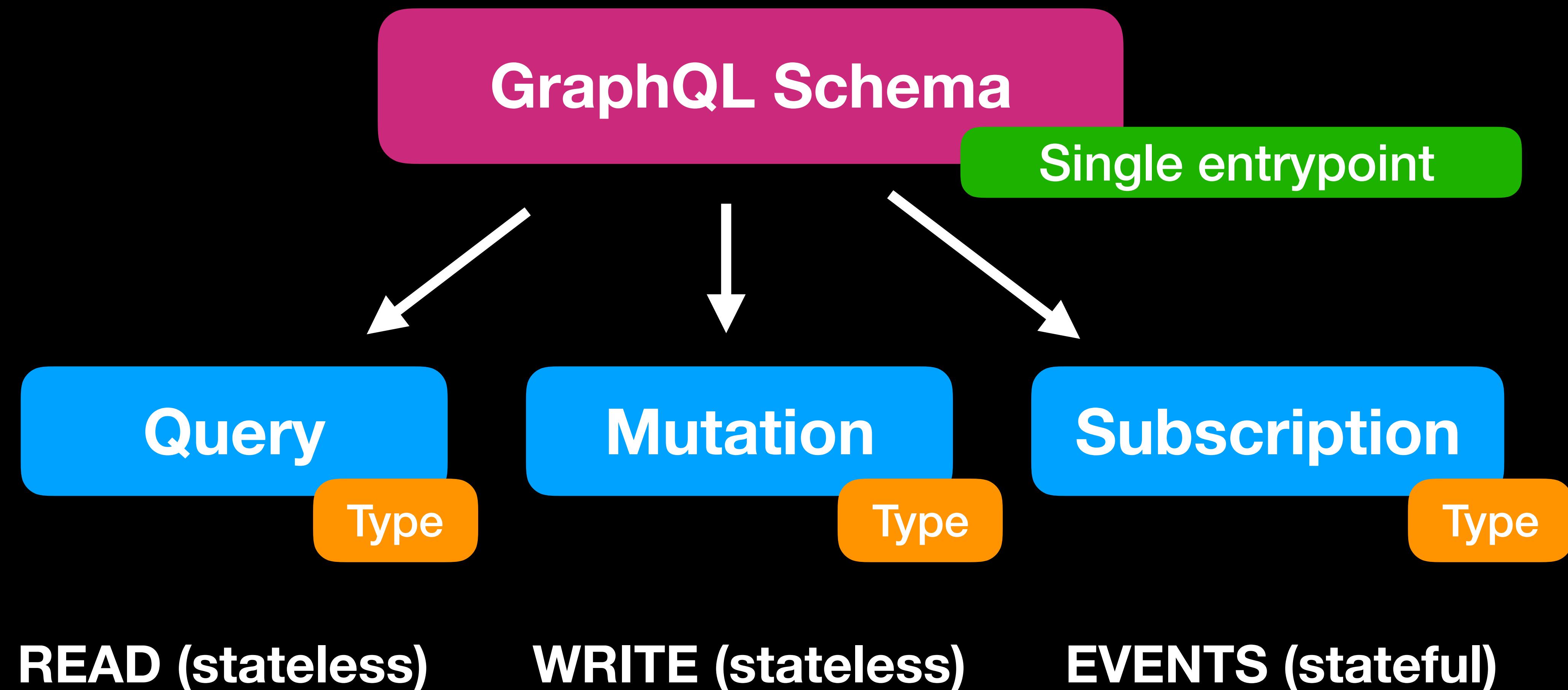
GraphQL query

```
{
  "data": {
    "viewer": {
      "product": {
        "name": "Uncle Bob's Organic Dried Pears",
        "unitPrice": 30,
        "category": {
          "name": "Produce",
          "description": "Dried fruit and bean curd"
        }
      }
    }
  }
}
```

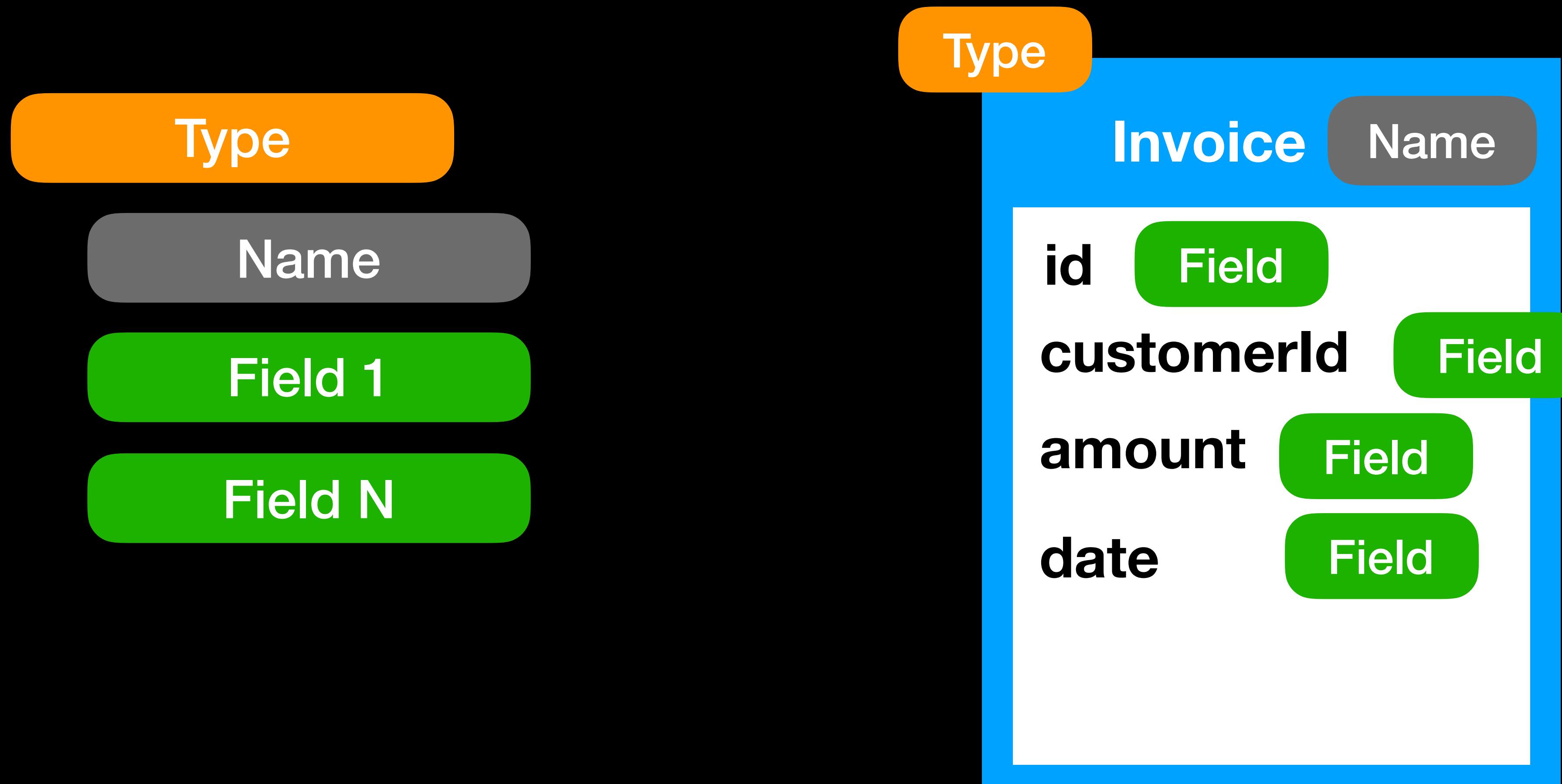


Response in JSON

GraphQL Schema



ObjectType



Field Config

Type

Field 1

Field N

Type

Resolve

Args

Description

DeprecationReason

Scalar or Object Type which returns resolve

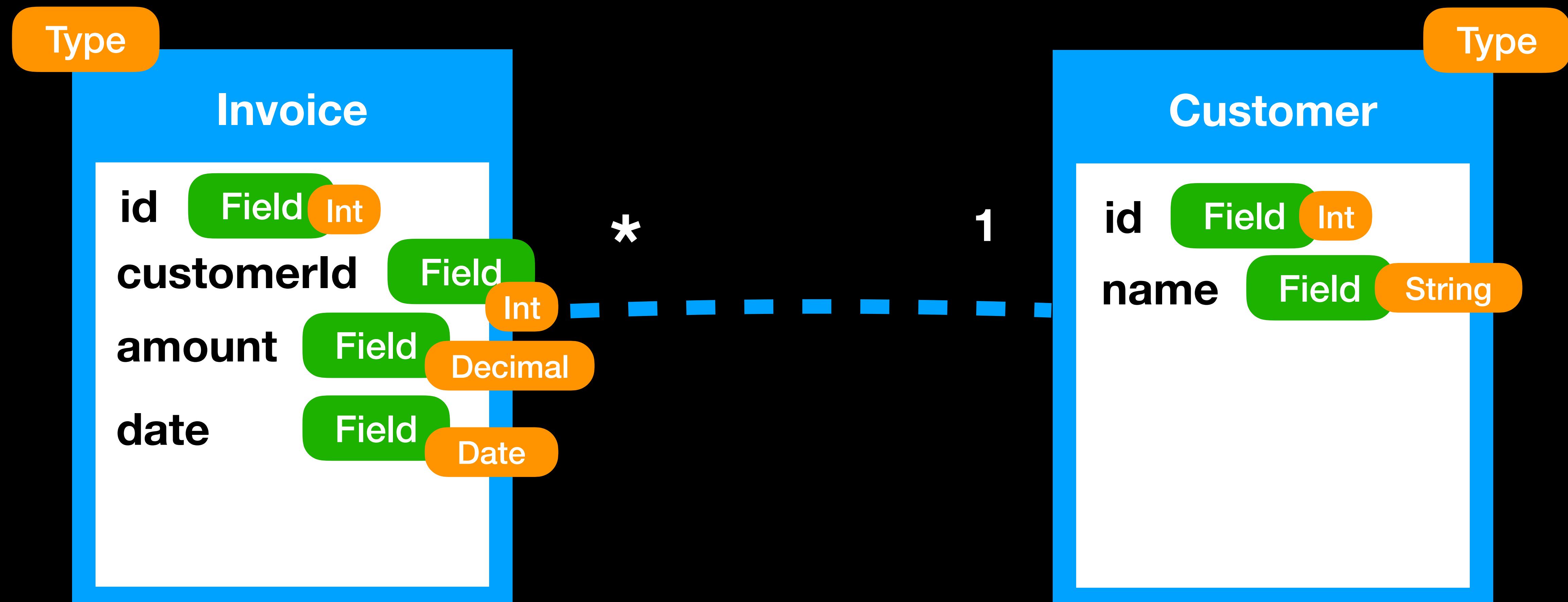
Function with fetch logic from any data source

Set of input args for resolve function

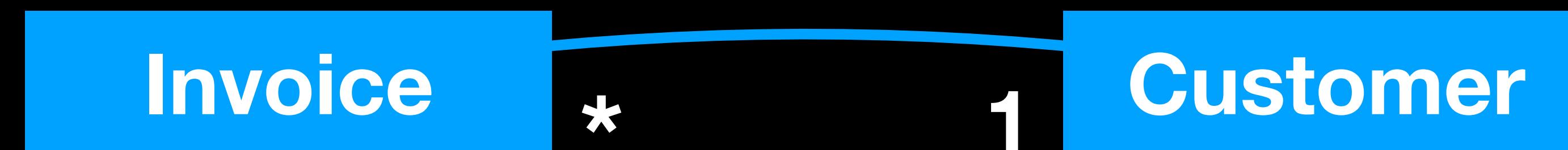
Documentation

Field hiding

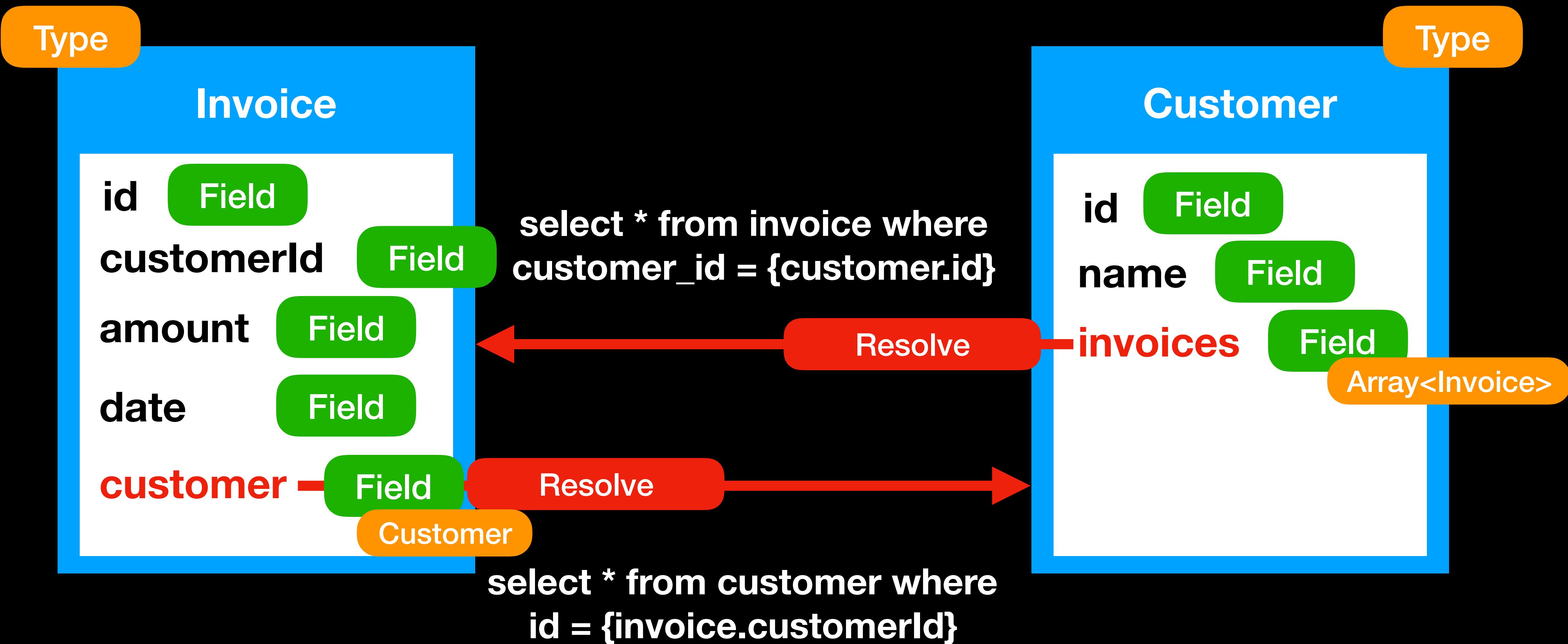
Relations between types



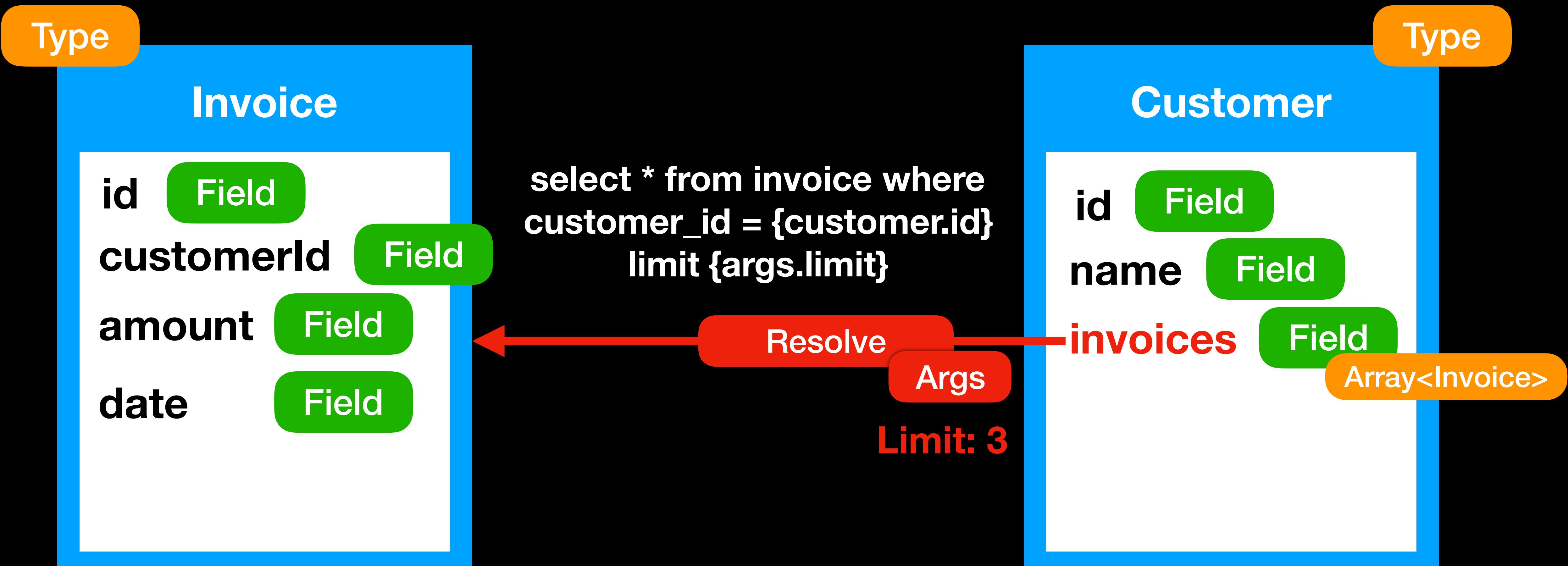
One does not simply draw a relation line in GraphQL!



Relation is a new field in your Type



- Limit? Offset? Sort? Additional filtering?
- No problem, add them to args and process in resolve function!



Resolve function

```
function (source, args, context, info) {  
  // access logic (check permissions)  
  // fetch data logic (from any mix of DBs)  
  // processing logic (operations, calcs)  
  return data;  
}
```

ANY PRIVATE BUSINESS LOGIC...

Schema Introspection

```
printSchema(schema); // txt output (SDL format)
```

```
graphql(schema, introspectionQuery); // json output (AST)
```

Just remove **Resolve** functions
(private business logic)
and you get PUBLIC schema

Schema Introspection example

- types
- fields
- args
- docs
- ~~resolvers~~
- directives
- input types
- enums
- interfaces
- unions

```
type Customer {  
    id: Int  
    name: String  
    # List of Invoices for current Customer  
    invoices(limit: Int): [Invoice]  
}
```

```
# Show me the money  
type Invoice {  
    id: Int  
    customerId: Int  
    amount: Decimal  
    # Customer data for current Invoice  
    customer: Customer  
    oldField: Int @deprecated(reason: "will be removed")  
}
```

**SDL format
(txt)**

Schema Introspection provides an ability for awesome tooling:

- Autocompletion
- Query validation
- Documentation
- Visualization
- TypeDefs generation for static analysis (Flow, TypeScript)

GraphiQL – graphical interactive in-browser GraphQL IDE

Eslint-plugin-graphql – check queries in your editor, CI

Relay-compiler – generates type definitions from queries

Type definition example

```
const QueryType = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    films: {
      type: new GraphQLList(FilmType),
      args: {
        limit: { type: GraphQLInt, defaultValue: 5 },
      },
      resolve: async (source, args) => {
        const data = await loadData(`https://swapi.co/api/films/`);
        return data.slice(0, args.limit);
      },
    },
    ...otherFields,
  }),
});
```

Type definition example

```
const Query = new GraphQLObjectType({  
  name: 'Query',  
  fields: () => ({  
    field: {  
      type: new GraphQLList(FilmType),  
      args: {  
        limit: { type: GraphQLInt, defaultValue: 5 },  
      },  
      resolve: async (source, args) => {  
        const data = await loadData(`...`);  
        return data.slice(0, args.limit);  
      },  
    },  
    ...otherFields,  
  }),  
});
```

Type

Field 1

Field N

Type

Args

Resolve

Description

DeprecationReason

FieldConfig

Don't forget to read about

- input types
- directives
- enums
- interfaces
- unions
- fragments

<http://graphql.org/learn/>

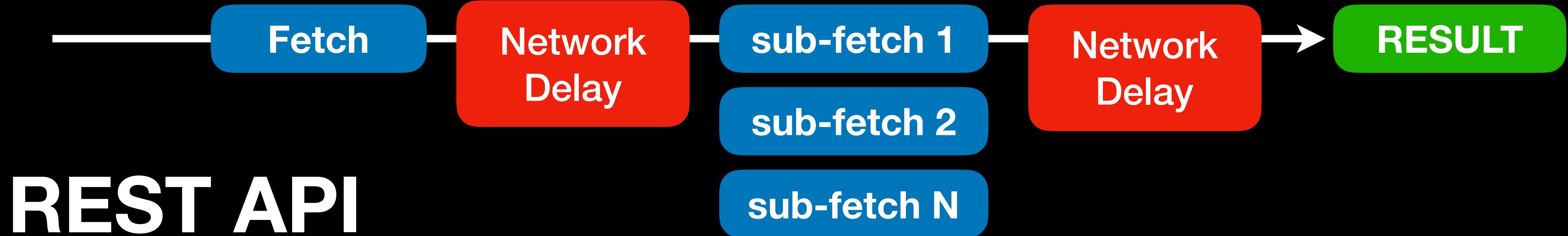


Backend capabilities

Client requirements

<https://graphql-compose.herokuapp.com/>

GraphQL Demo



REST API

- Sub-fetch logic on client side (increase bundle size)
- Over-fetching (redundant data transfer/parsing)



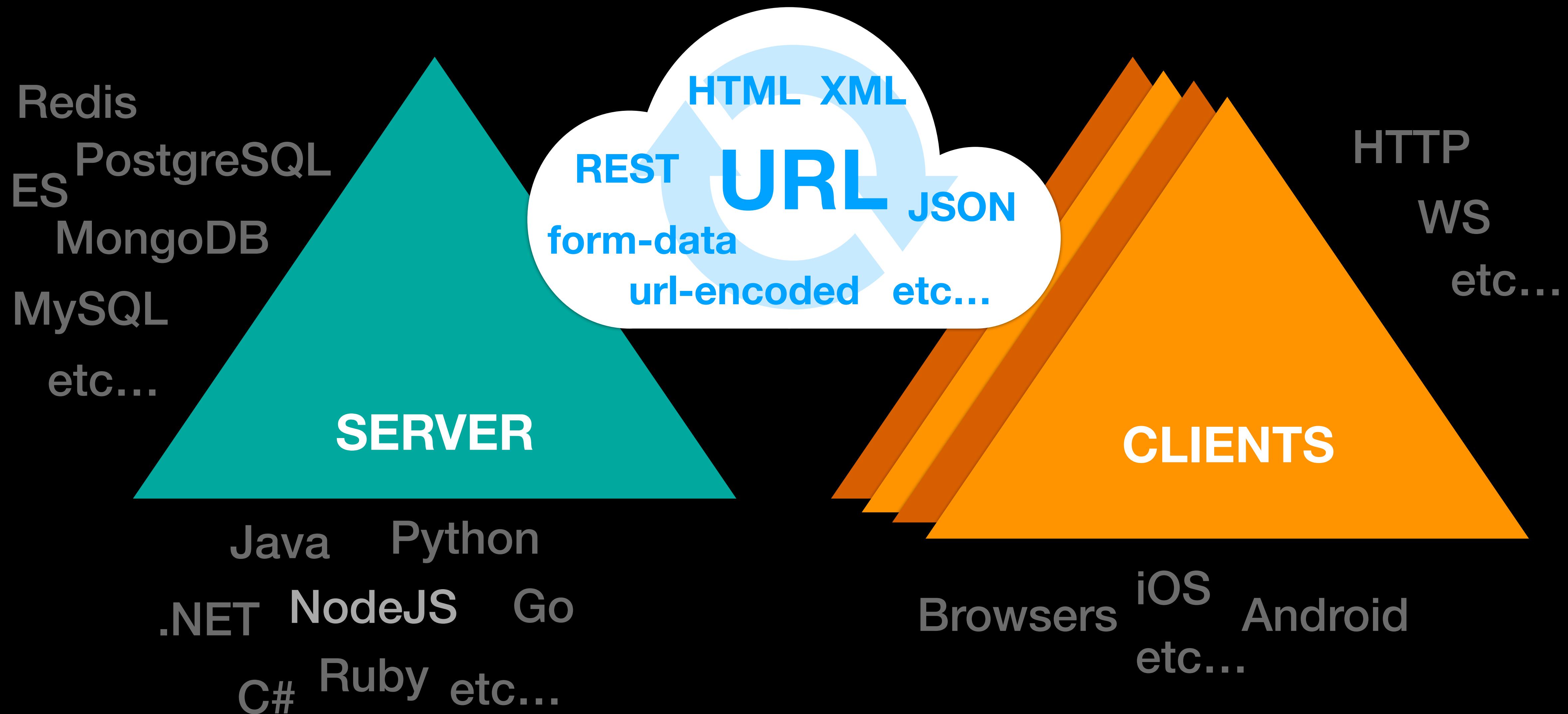
GraphQL

- + No additional network round-trip (speed)
- + Exactly requested fields (speed)
- + Sub-fetch logic implemented on server side

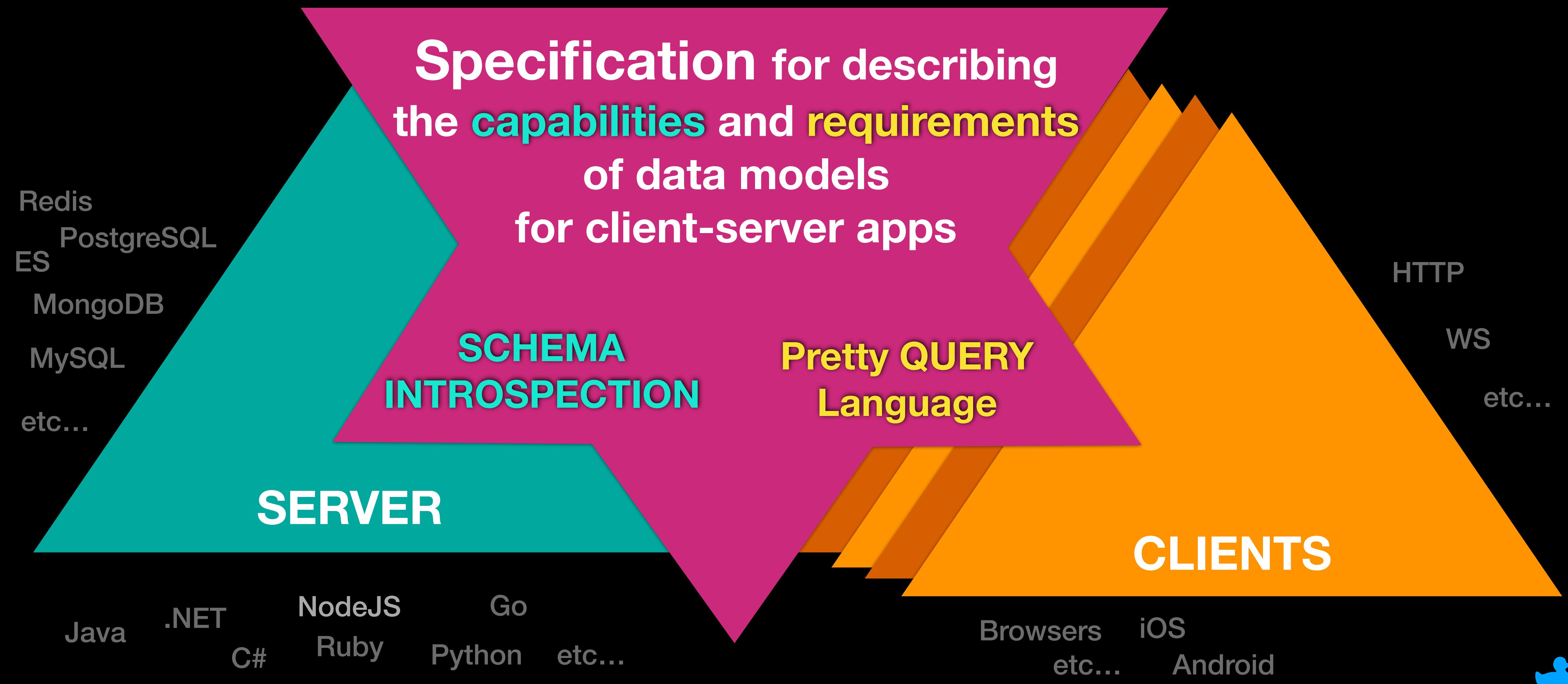


A copy from one of the previous slides...

Client-server apps



GraphQL – is a query language for APIs



Static Analysis

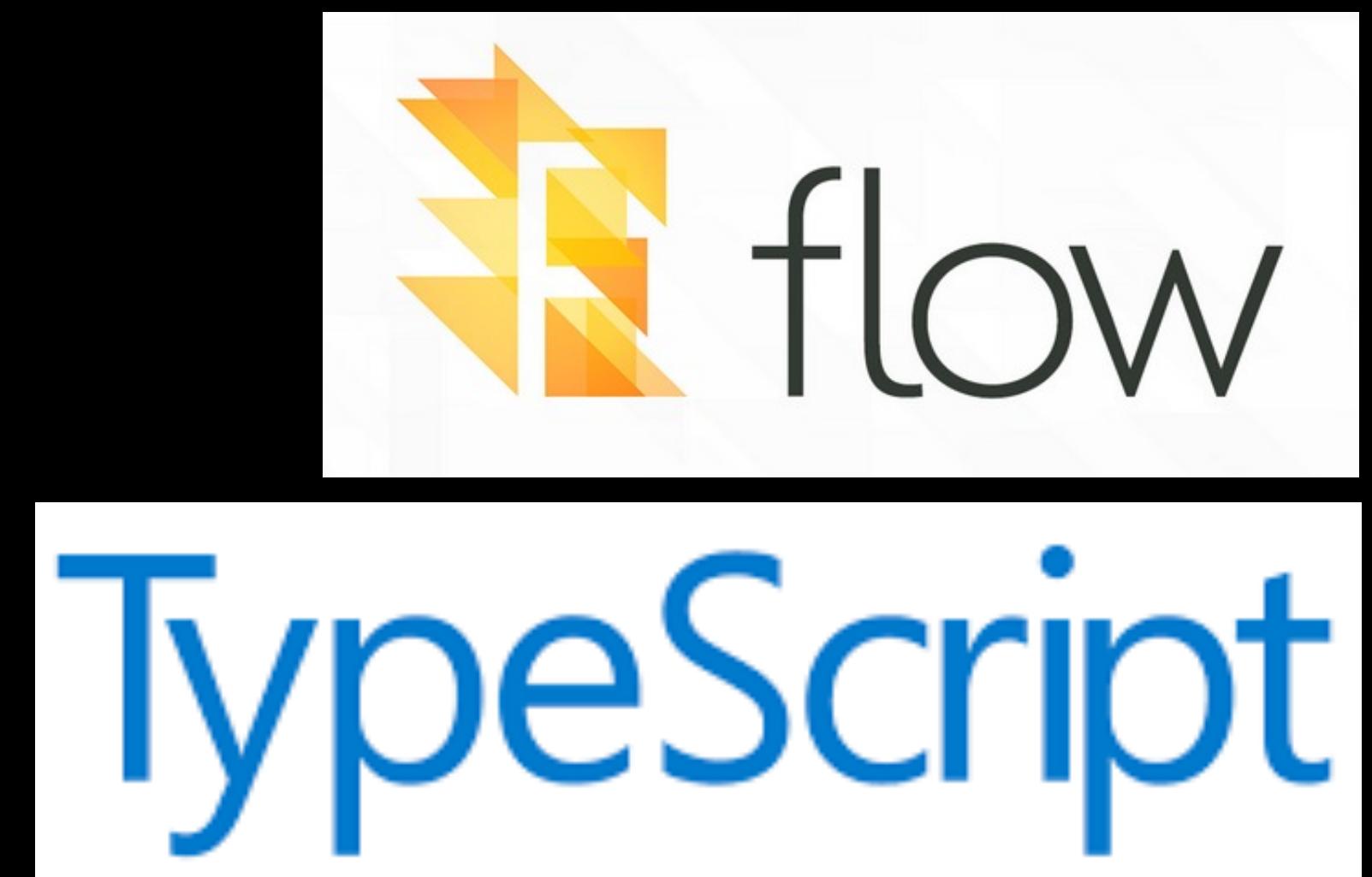
For frontend
developers

Static type checks



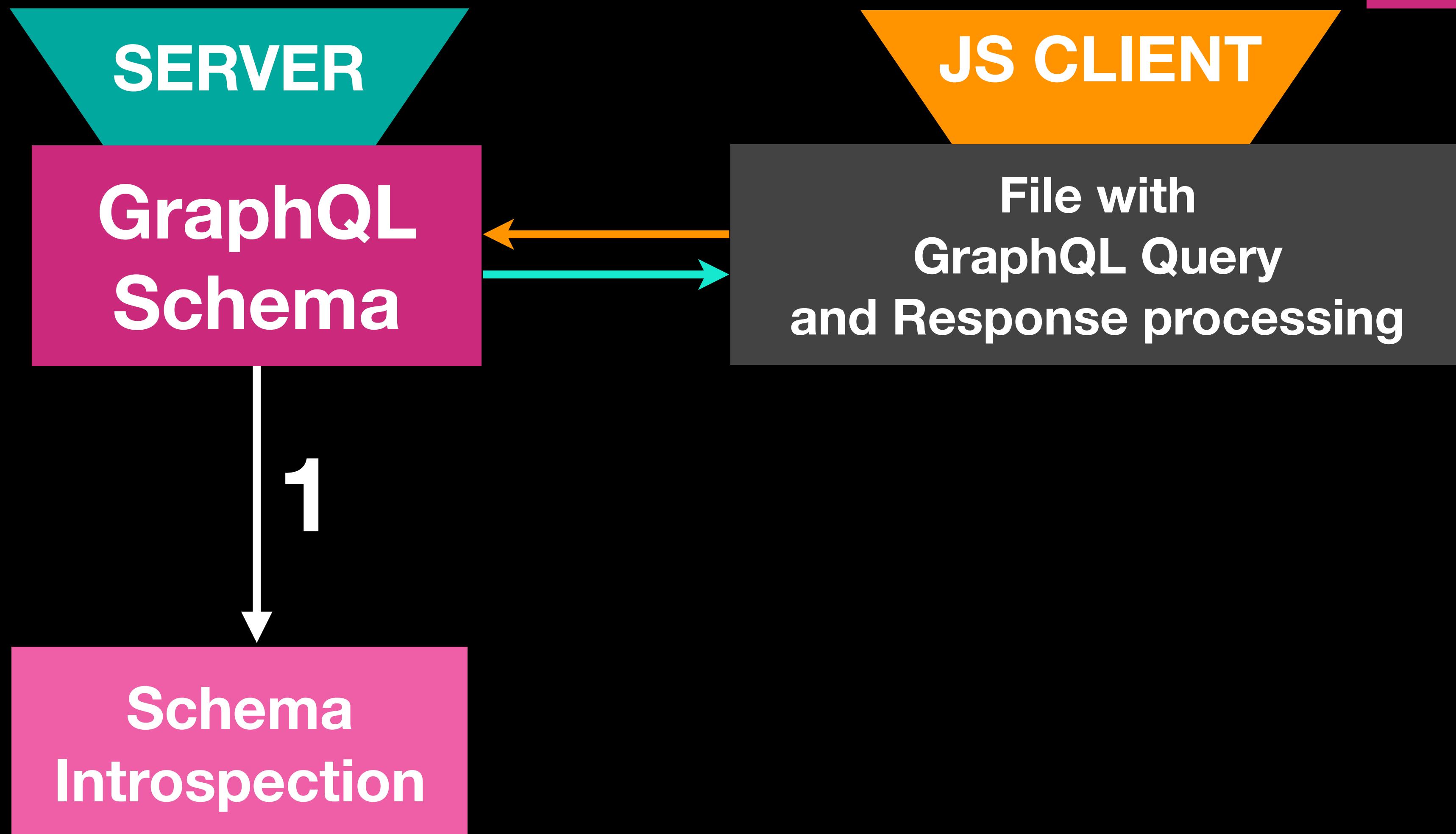
PRODUCtIVITY

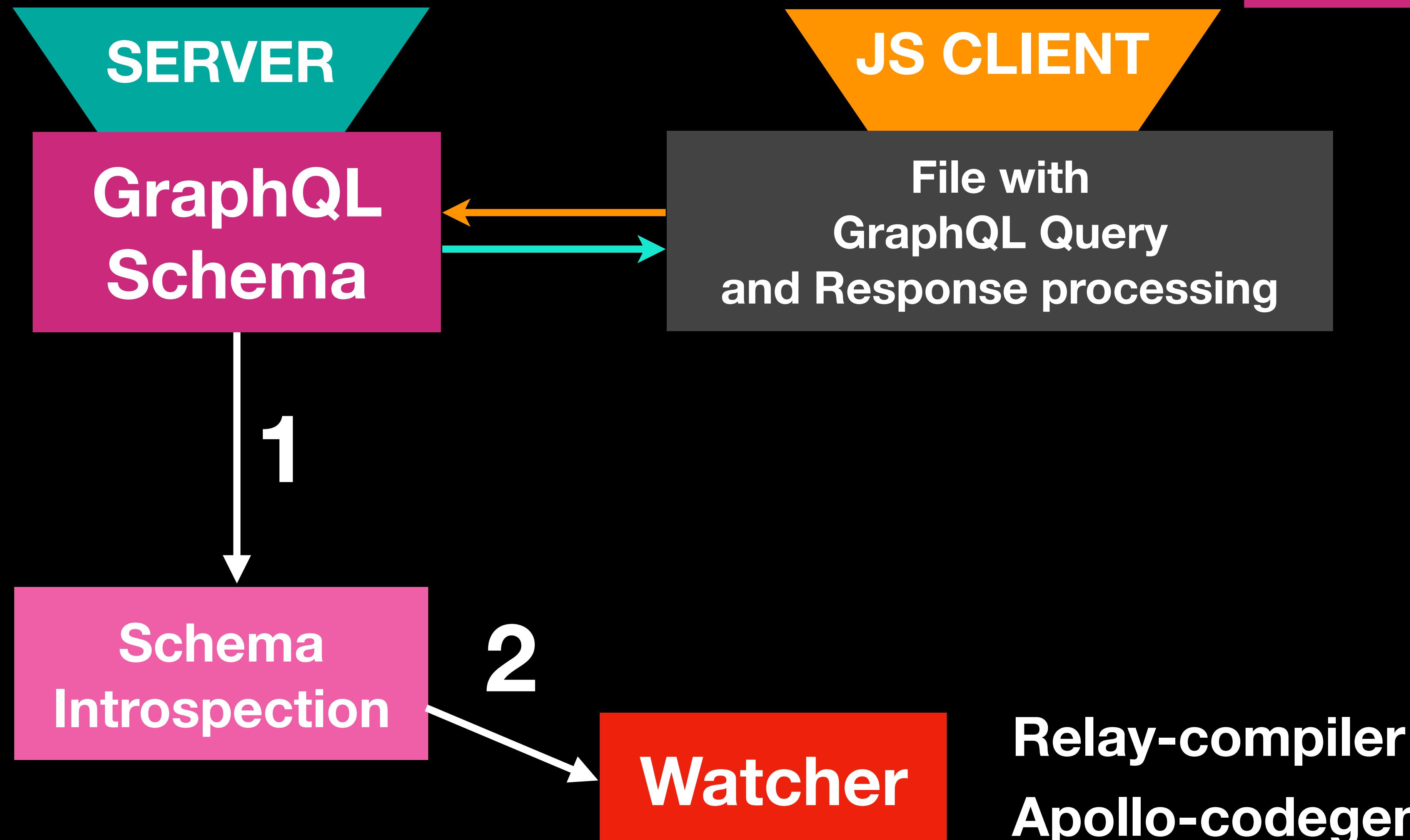
- Types checks
- Functions call checks
- Auto-suggestion
- Holy refactoring

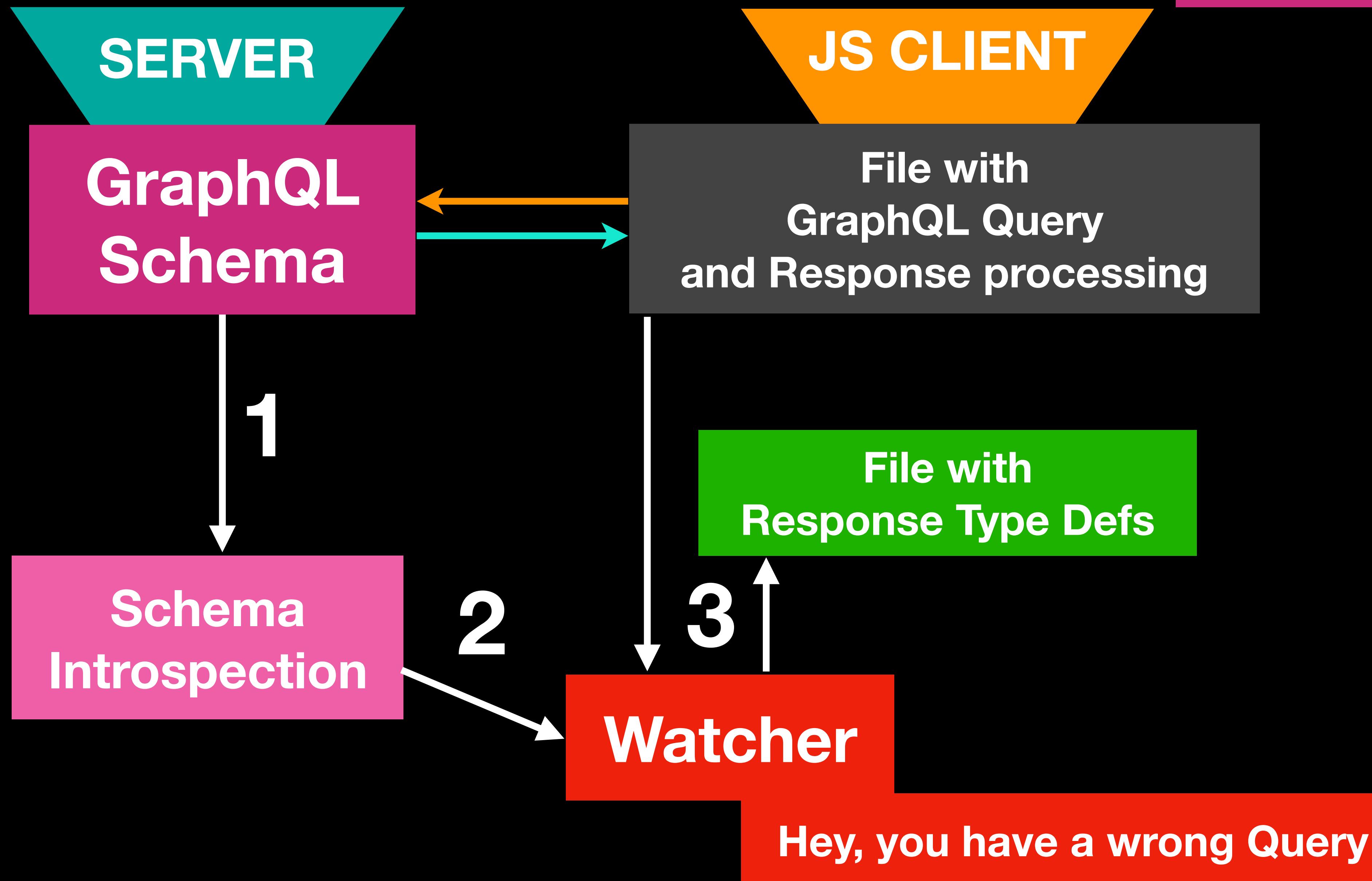


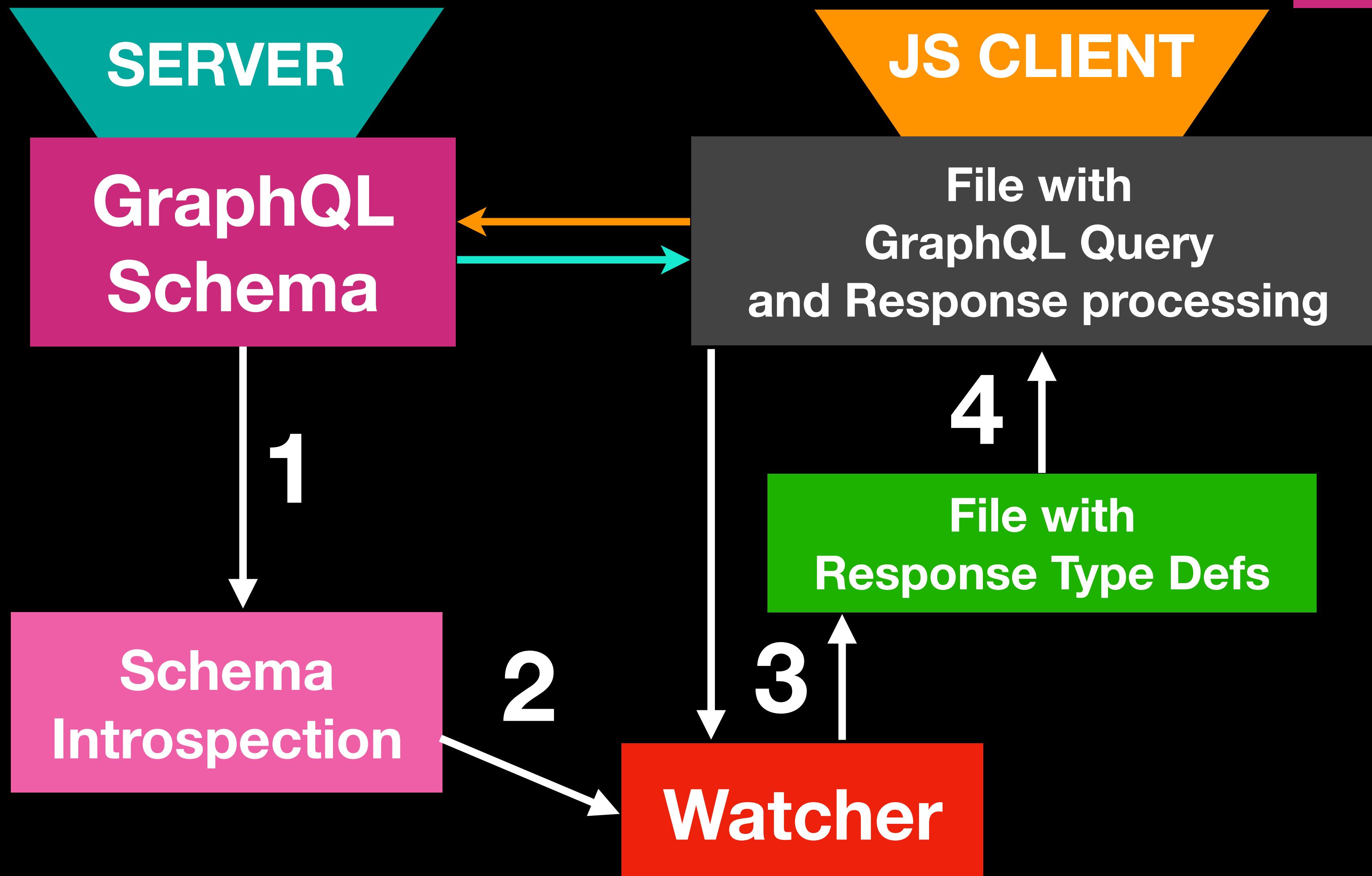


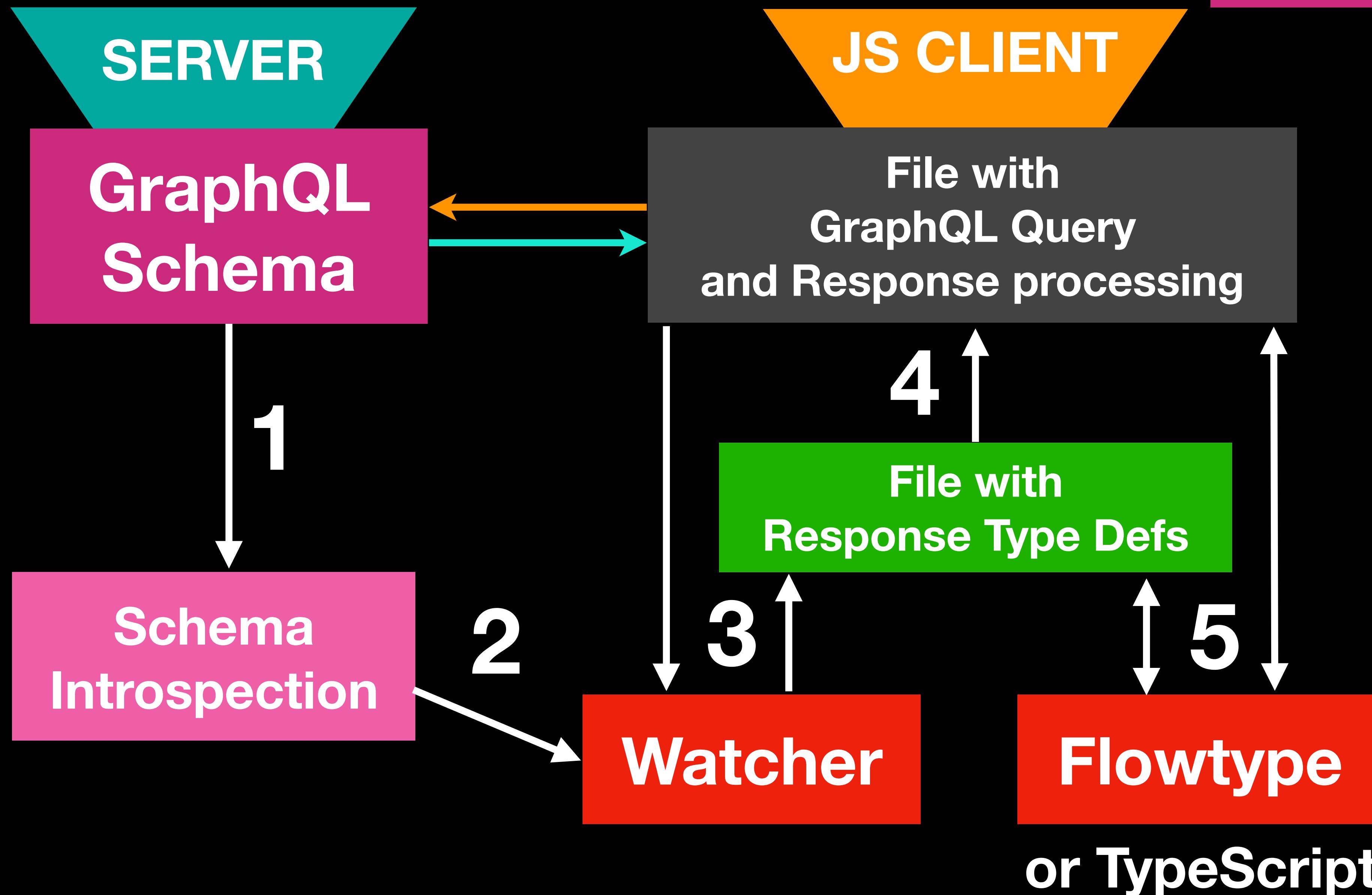
Let's turbo-charge
our client apps static analysis
with GraphQL queries

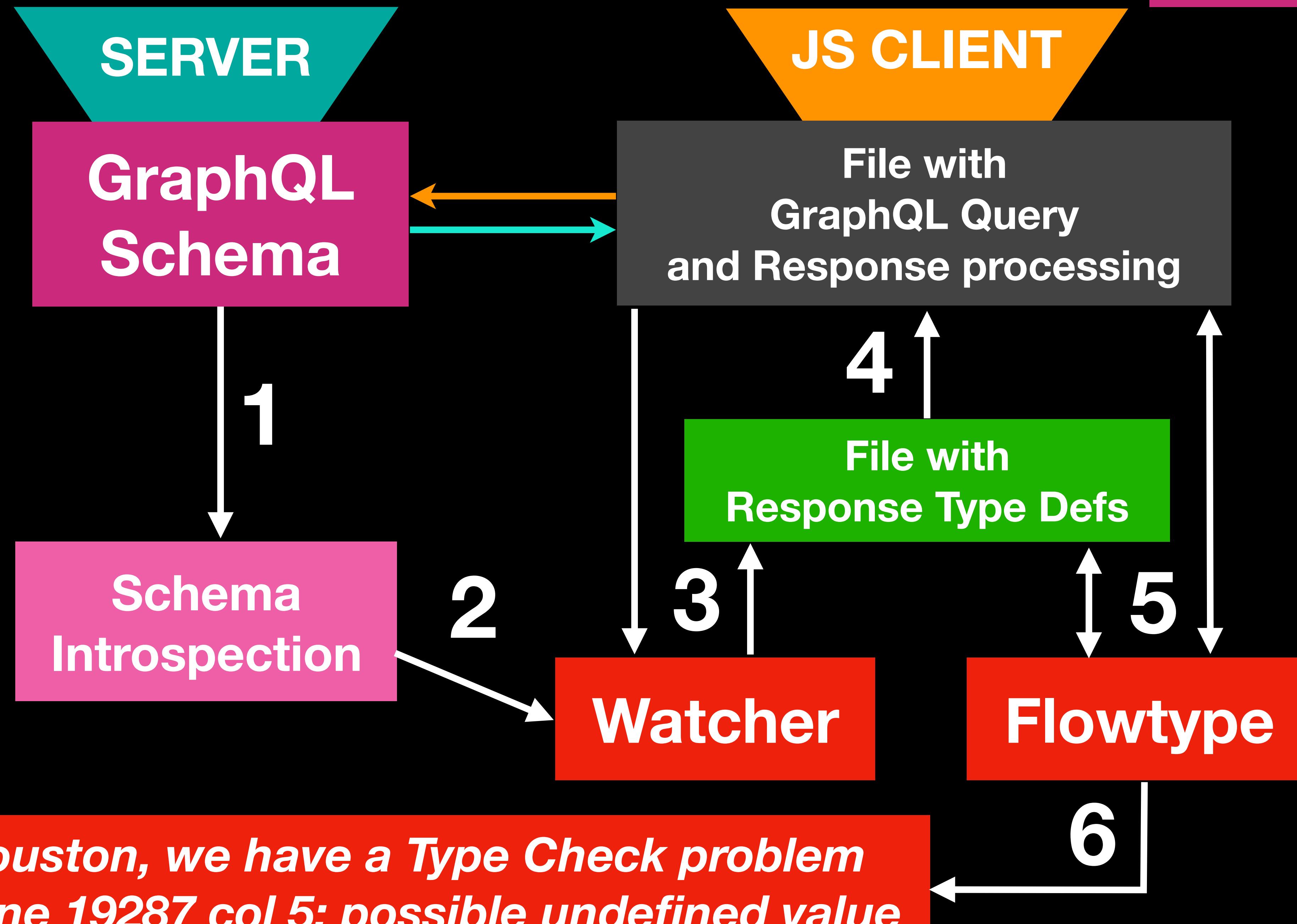












DEMO

GraphQL Query

Generated Response Type Def

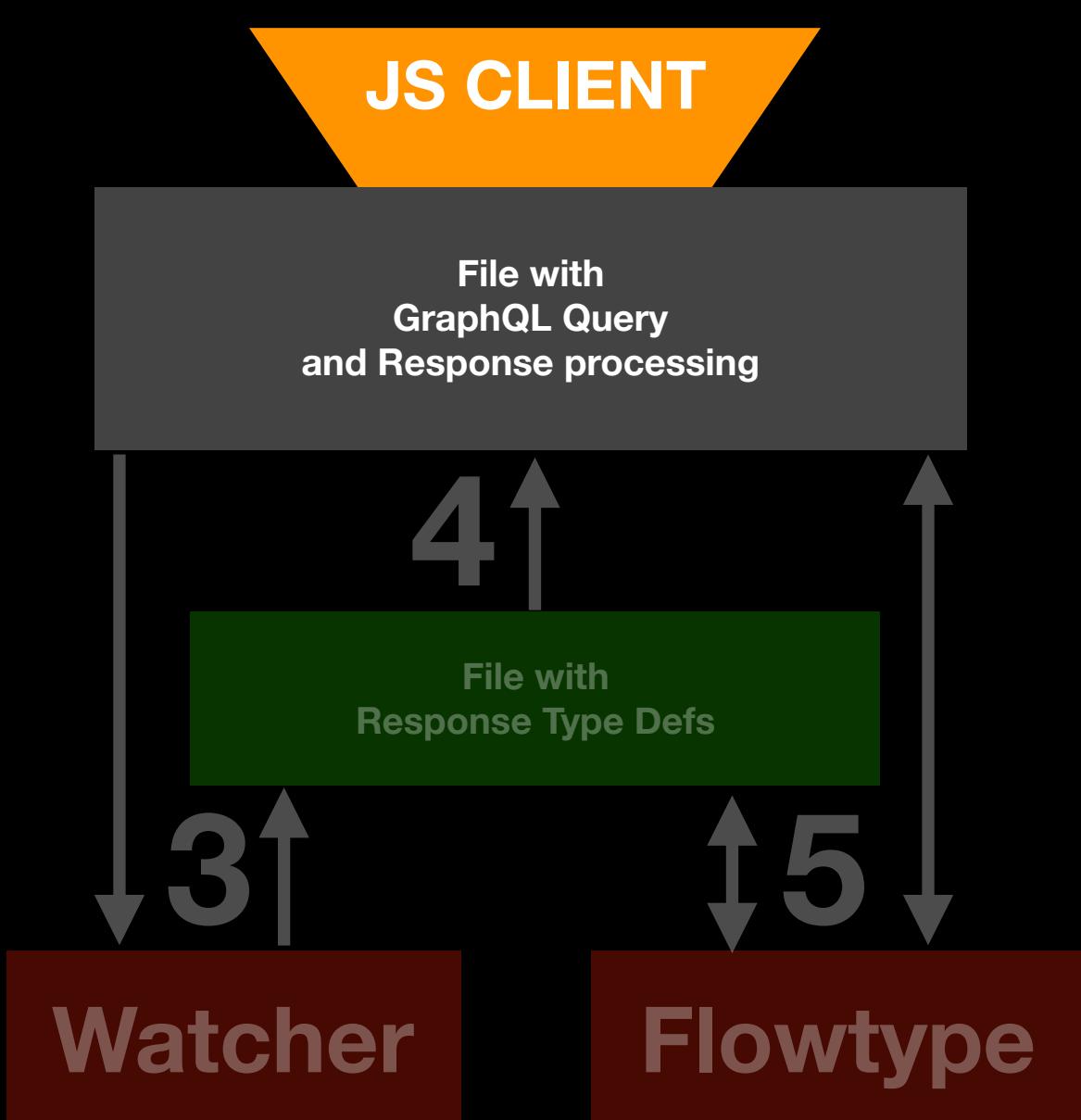
Crappy Code

Flow typed Code

Flow error

GraphQL Query

```
import { graphql } from 'react-relay';
const query = graphql`  
query BalanceQuery {  
  viewer {  
    cabinet {  
      accountBalance  
    }  
  }  
}`;
```

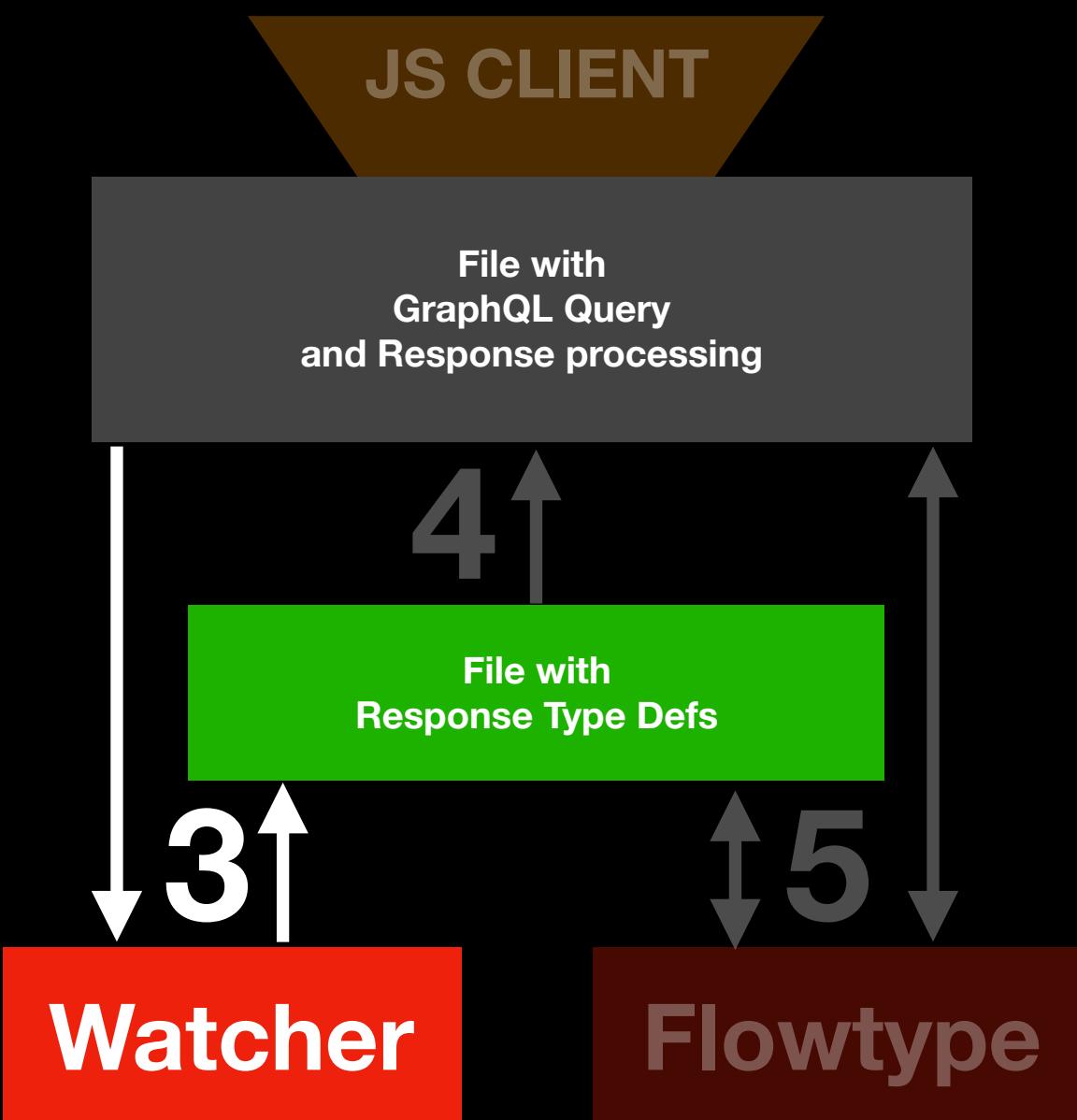


Generated Response Type Def

```
/* @flow */
/*::
export type BalanceQueryResponse = {
  +viewer: ?{
    +cabinet: ?{
      +accountBalance: ?number;
    };
  };
  +cabinet: ?{
    +accountBalance: ?number;
  };
};*/

```

Writer time: 0.53s [0.37s compiling, ...]
 Created:
 - BalanceQuery.graphql.js
 Unchanged: 291 files
 Written default in 0.61s

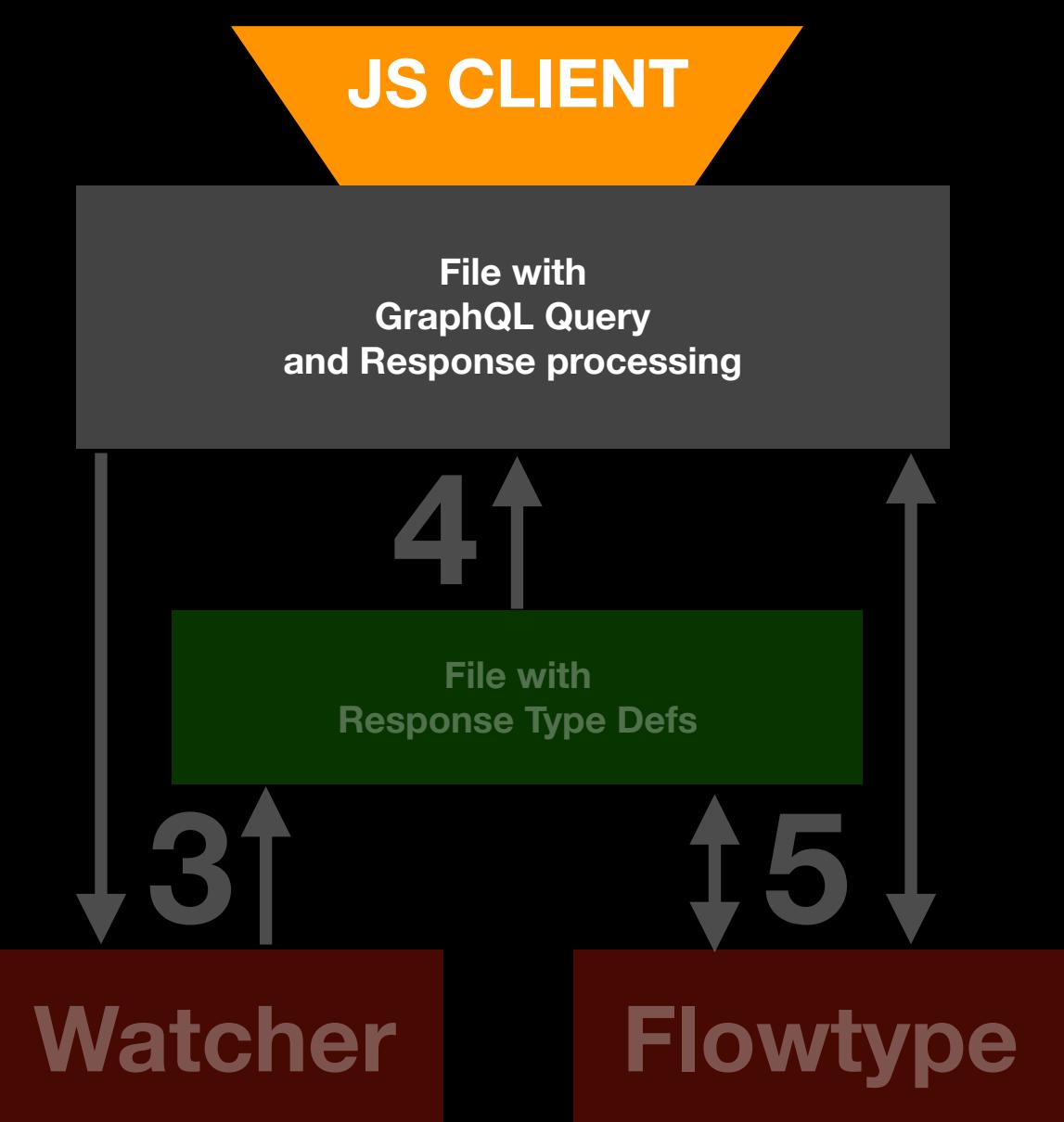


```
import { graphql } from 'react-relay';
import * as React from 'react';
```

```
export default class Balance extends React.Component {
  render() {
    const { viewer } = this.props;
    return <div>
      Balance {viewer.cabinet.accountBalance}
    </div>;
  }
}

const query = graphql`query BalanceQuery {
  viewer { cabinet { accountBalance } }
}`;
```

Crappy Code



Flow typed Code

```

import { graphql } from 'react-relay';
import * as React from 'react';
import type { BalanceQueryResponse } from './__generated__/BalanceQuery.graphql';

type Props = BalanceQueryResponse;

export default class Balance extends React.Component<Props> {
  render() {
    const { viewer } = this.props;
    return <div>Balance {viewer.cabinet.accountBalance}</div>;
  }
}
const query = graphql`query BalanceQuery {
  viewer { cabinet { accountBalance } }
}`;

```

The diagram illustrates the static analysis process for the code above. It features three main components arranged vertically:

- JS CLIENT**: A grey box labeled "File with GraphQL Query and Response processing".
- Watcher**: A brown box at the bottom.
- Flowtype**: A brown box at the bottom right.

Arrows indicate the flow of data and analysis results:

- A downward arrow from the JS CLIENT box to the Watcher box is labeled "3↑".
- An upward arrow from the Watcher box to the JS CLIENT box is labeled "4↑".
- A double-headed horizontal arrow between the Watcher and Flowtype boxes is labeled "5↓↑".

```

/* @flow */
import { graphql } from 'react-relay';
import * as React from 'react';
import type { BalanceQueryResponse } from './__generated__/BalanceQuery.graphql';

type Props = BalanceQueryResponse;

export default class Balance extends React.Component<Props> {
  render() {
    const { viewer } = this.props;
    return <div>Balance {viewer.cabinet.accountBalance}</div>;
  }
}
const query = graphql`query BalanceQuery {
  viewer { cabinet { accountBalance } }
}`;

```

Flow typed Code

The diagram illustrates the static analysis process:

- JS CLIENT**: A grey box labeled "File with GraphQL Query and Response processing".
- Flowtype**: A red box labeled "File with Response Type Defs".
- Watcher**: A brown box.
- Arrows and Labels:**
 - A grey arrow labeled "3↑" points from the Watcher box down to the JS CLIENT box.
 - A green arrow labeled "4↑" points from the Flowtype box up to the JS CLIENT box.
 - A double-headed red arrow labeled "5" connects the Flowtype and Watcher boxes horizontally.

Flow errors

Error: src/_demo/Balance.js:11

```
11:     return <div>Your balance: {viewer.cabinet.accountBalance}</div>;  
                                         ^^^^^^ property `cabinet`.  
  
Property cannot be accessed on possibly null value  
11:     return <div>Your balance: {viewer.cabinet.accountBalance}</div>;  
                                         ^^^^^^ null
```

Error: src/ demo/Balance.js:11

Flow errors for missing field

```
type Props = BalanceQueryResponse;
class Balance extends React.Component<Props> {
  render() {
    const { viewer } = this.props;
    return <div>{viewer.invoices}</div>;
  }
}
```

Flow errors for missing field

Error: src/_demo/Balance.js:11

```
11:     return <div>{viewer.invoices}</div>;  
                         ^^^^^^^^^^ property `invoices`.
```

Property not found in

```
      v-  
13: +viewer: ?{ |  
14:   +cabinet: ?{ |  
15:     +accountBalance: ?number;  
16:   | };  
17: | };  
-^ object type. See: src/_demo/__generated__/  
BalanceQuery.graphql.js:13
```



For backend
developers

GraphQL Query Problems

Denial of Service attacks

aka Resource exhaustion attacks

```
query HugeResponse {  
  user {  
    friends(limit: 1000) {  
      friends(limit: 1000) {  
        friends(limit: 1000) {  
          ...  
        }  
      }  
    }  
  }  
}
```

Solutions:

- avoid nesting relations
- cost analysis on the query
- pre-approve queries that the server can execute (persisted queries by unique ID) using by Facebook

N+1 query problem

```
query NestedQueryN1 {  
  productList {  
    id  
    categoryId  
    category {  
      id  
      name  
    }  
  }  
}
```

1 query for ProductList

N queries for fetching every Category by id

Solution: DataLoader

```
const CatLoader = new DataLoader(  
  ids => Category.findByIds(ids)  
);  
  
CatLoader.load(1);  
CatLoader.load(2);  
CatLoader.load(1);  
CatLoader.load(4);
```

will do just one BATCH request on next tick to the Database



For backend
developers

Schema construction problems

Query Type example

Problem #1: too much copy/paste

```
const QueryType = new GraphQLObjectType({  
  name: 'Query',  
  fields: () => ({  
    films: ....,  
    persons: ....,  
    planets: ....,  
    species: ....,  
    starships: ....,  
    vehicles: ....,  
  }),  
});
```

**6 fields and
every FieldConfig
consists from
almost identical
12 ctrl+c/ctrl+v lines**

The Star Wars API
<https://swapi.co>

FieldConfig example for films field

```
films: {  
  type: new GraphQLList(FilmType),  
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
  resolve: async (source, args) => {  
    const data = await loadData(`https://swapi.co/api/films/`);  
    if (args && args.limit > 0) {  
      return data.slice(0, args.limit);  
    }  
    return data;  
  },
```

Comparison of two FieldConfigs

Problem #1: too much copy/paste

```
{
  films: {
    type: new GraphQLList(FilmType),
    args: { limit: { type: GraphQLInt, defaultValue: 5 } },
    resolve: async (source, args) => {
      const data = await loadData(`https://swapi.co/api/films/`);
      if (args && args.limit > 0) {
        return data.slice(0, args.limit);
      }
      return data;
    },
  },
  planets: {
    type: new GraphQLList(PlanetType),
    args: { limit: { type: GraphQLInt, defaultValue: 5 } },
    resolve: async (source, args) => {
      const data = await loadData(`https://swapi.co/api/planets/`);
      if (args && args.limit > 0) {
        return data.slice(0, args.limit);
      }
      return data;
    },
  },
}
```



**differs
only by url**

Solution 1: you may generate your resolve functions

```
function createListResolve(url) {  
  return async (source, args) => {  
    const data = await loadData(url);  
    if (args && args.limit > 0) {  
      return data.slice(0, args.limit);  
    }  
    return data;  
  };  
}
```

create a function
which returns a resolve function

Solution 1: you may generate your resolve functions

```
{
  films: {
    type: new GraphQLList(FilmType),
    args: { limit: { type: GraphQLInt, defaultValue: 5 } },
    resolve: async (source, args) => {
      const data = await loadData(`https://swapi.co/api/films/`);
      if (args && args.limit > 0) {
        return data.slice(0, args.limit);
      }
      return data;
    },
  },
},
```

```
films: {
  type: new GraphQLList(FilmType),
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },
  resolve: createListResolve(`https://swapi.co/api/films/`),
},
```

reduce N times 7 LoC to 1 LoC

```
planets: {
  type: new GraphQLList(PlanetType),
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },
  resolve: async (source, args) => {
    const data = await loadData(`https://swapi.co/api/planets/`);
    if (args && args.limit > 0) {
      return data.slice(0, args.limit);
    }
    return data;
  },
},
```

```
planets: {
  type: new GraphQLList(PlanetType),
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },
  resolve: createListResolve(`https://swapi.co/api/planets/`),
},
```

Solution 2: you may generate your FieldConfigs

```
films: {  
  type: new GraphQLList(FilmType),  
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
  resolve: createListResolve(`https://swapi.co/api/films/`),  
},
```

differs only by `Type` and `url`

```
planets: {  
  type: new GraphQLList(PlanetType),  
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
  resolve: createListResolve(`https://swapi.co/api/planets/`),  
},
```

Solution 2: you may generate your FieldConfigs

```
function createFieldConfigForList(type, url) {  
  return {  
    type: new GraphQLList(type),  
    args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
    resolve: createListResolve(url),  
  };  
}
```

create a function
which returns a FieldConfig

Solution 2: you may generate your FieldConfigs

```
films: {  
  type: new GraphQLList(PlanetType),  
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
  resolve: createListResolve(`https://swapi.co/api/films/`),  
},  
planets: {  
  type: new GraphQLList(FilmType),  
  args: { limit: { type: GraphQLInt, defaultValue: 5 } },  
  resolve: createListResolve(`https://swapi.co/api/planets/`),  
},
```

10 LoC reduced to 2 LoC

```
{  
  films: createFieldConfigForList(FilmType, `https://swapi.co/api/films/`),  
  planets: createFieldConfigForList(PlanetType, `https://swapi.co/api/planets/`),  
}
```

Solution 1: you may generate your resolve functions

Solution 2: you may generate your FieldConfigs

```

1  /* eslint-disable */
2  const QueryType = new GraphQLObjectType({
3    name: 'Query',
4    fields: () => ({
5      people: {
6        type: new GraphQLList(PersonType),
7        args: {
8          limit: { type: GraphQLInt, defaultValue: 3 },
9          offset: { type: GraphQLInt, defaultValue: 0 },
10        },
11        resolve: async (source, args) => {
12          const data = await loadData('https://swapi.co/api/people/');
13          if (args && args.limit > 0 && args.offset >= 0) {
14            return data.slice(args.offset, args.limit + args.offset);
15          }
16          return data;
17        },
18      },
19      planets: {
20        type: new GraphQLList(PlanetType),
21        args: {
22          limit: { type: GraphQLInt, defaultValue: 3 },
23          offset: { type: GraphQLInt, defaultValue: 0 },
24        },
25        resolve: async (source, args) => {
26          const data = await loadData('https://swapi.co/api/planets/');
27          if (args && args.limit > 0 && args.offset >= 0) {
28            return data.slice(args.offset, args.limit + args.offset);
29          }
30          return data;
31        },
32      },
33      films: {
34        type: new GraphQLList(FilmType),
35        args: {
36          limit: { type: GraphQLInt, defaultValue: 3 },
37          offset: { type: GraphQLInt, defaultValue: 0 },
38        },
39        resolve: async (source, args) => {
40          const data = await loadData('https://swapi.co/api/films/');
41          if (args && args.limit > 0 && args.offset >= 0) {
42            return data.slice(args.offset, args.limit + args.offset);
43          }
44          return data;
45        },
46      },
47      species: {
48        type: new GraphQLList(SpeciesType),
49        args: {
50          limit: { type: GraphQLInt, defaultValue: 3 },
51          offset: { type: GraphQLInt, defaultValue: 0 },
52        },
53        resolve: async (source, args) => {
54          const data = await loadData('https://swapi.co/api/species/');
55          if (args && args.limit > 0 && args.offset >= 0) {
56            return data.slice(args.offset, args.limit + args.offset);
57          }
58          return data;
59        },
60      },
61      starships: {
62        type: new GraphQLList(StarshipType),
63        args: {
64          limit: { type: GraphQLInt, defaultValue: 3 },
65          offset: { type: GraphQLInt, defaultValue: 0 },
66        },
67        resolve: async (source, args) => {
68          const data = await loadData('https://swapi.co/api/starships/');
69          if (args && args.limit > 0 && args.offset >= 0) {
70            return data.slice(args.offset, args.limit + args.offset);
71          }
72          return data;
73        },
74      },
75      vehicles: {
76        type: new GraphQLList(VehicleType),
77        args: {
78          limit: { type: GraphQLInt, defaultValue: 3 },
79          offset: { type: GraphQLInt, defaultValue: 0 },
80        },
81        resolve: async (source, args) => {
82          const data = await loadData('https://swapi.co/api/vehicles/');
83          if (args && args.limit > 0 && args.offset >= 0) {
84            return data.slice(args.offset, args.limit + args.offset);
85          }
86          return data;
87        },
88      },
89    });
90  });
91
92  export default QueryType;
93

```

90 LoC

was
reduced
in 3 times

```

1  /* eslint-disable */
2  function createListResolve(url) {
3    return source({ args }) => {
4      const data = await loadData(url);
5      if (args && args.limit > 0) {
6        return data.slice(0, args.limit);
7      }
8      return data;
9    };
10  }
11
12  function createFieldConfigForList(type, url) {
13    return {
14      type: new GraphQLList(type),
15      args: { limit: { type: GraphQLInt, defaultValue: 3 } },
16      resolve: createListResolve(url),
17    };
18  }
19
20  const QueryType = new GraphQLObjectType({
21    name: 'Query',
22    fields: () => ({
23      people: createFieldConfigForList(PersonType, 'https://swapi.co/api/people/'),
24      planets: createFieldConfigForList(PlanetType, 'https://swapi.co/api/planets/'),
25      films: createFieldConfigForList(FilmType, 'https://swapi.co/api/films/'),
26      species: createFieldConfigForList(SpeciesType, 'https://swapi.co/api/species/'),
27      starships: createFieldConfigForList(StarshipType, 'https://swapi.co/api/starships/'),
28      vehicles: createFieldConfigForList(VehicleType, 'https://swapi.co/api/vehicles/'),
29    });
30  });
31
32  export default QueryType;
33

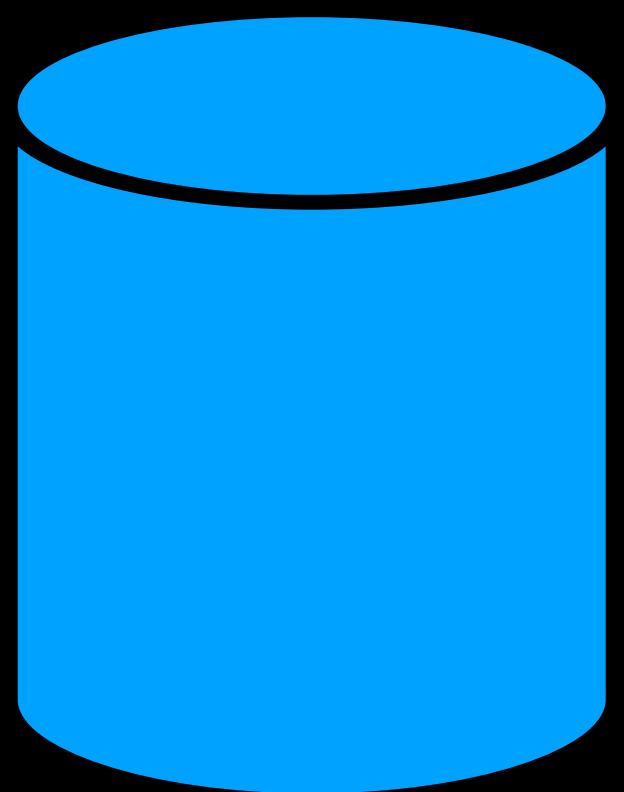
```

30 LoC

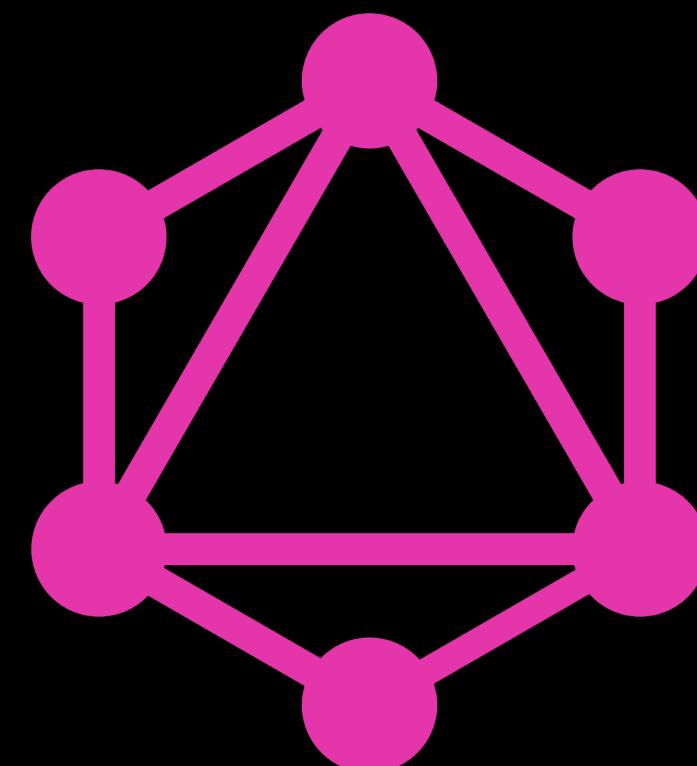
DRY principle
(don't repeat yourself)

How to keep to Schemas in SYNC?

Model: User



Type: User

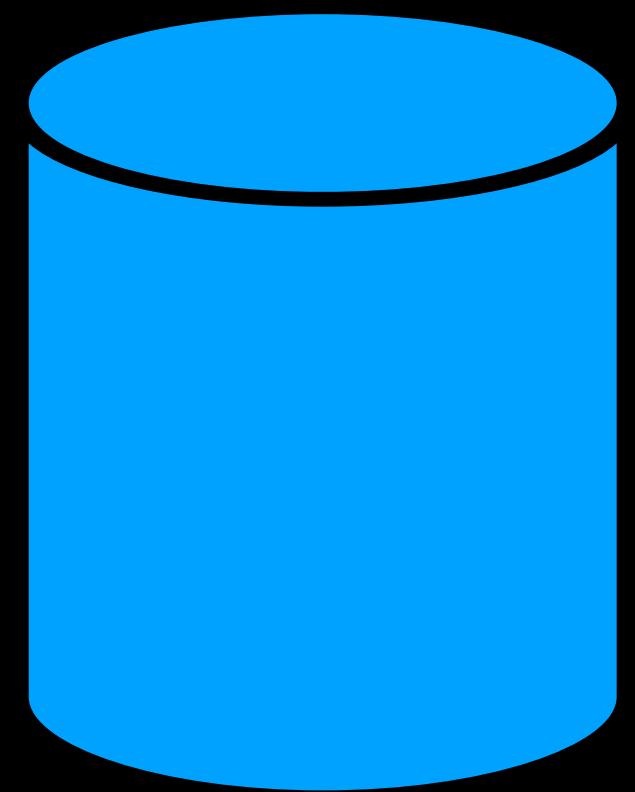


With time you may:

- add new fields
- change field types
- remove fields
- rename fields

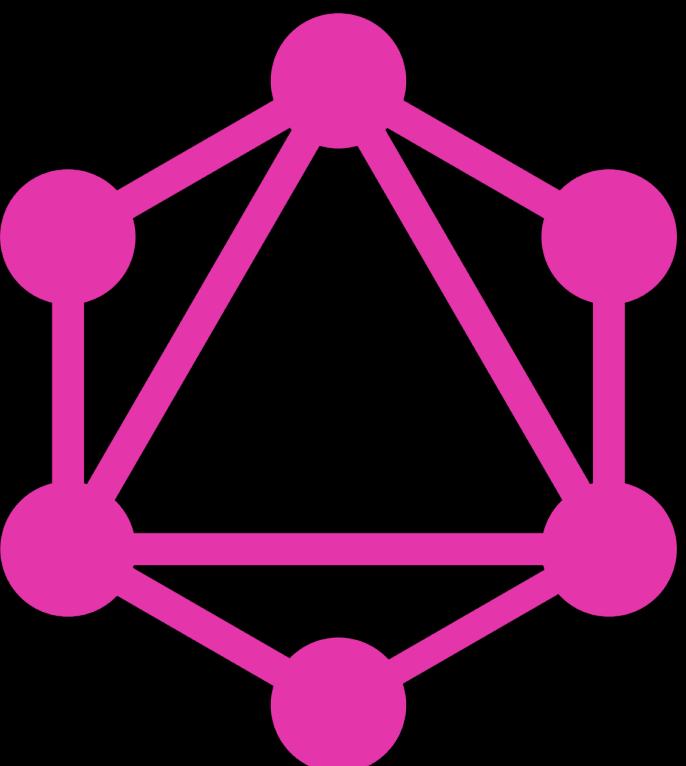
Solution: generate GraphQL types from ORM models

Model: User



GENERATE
→

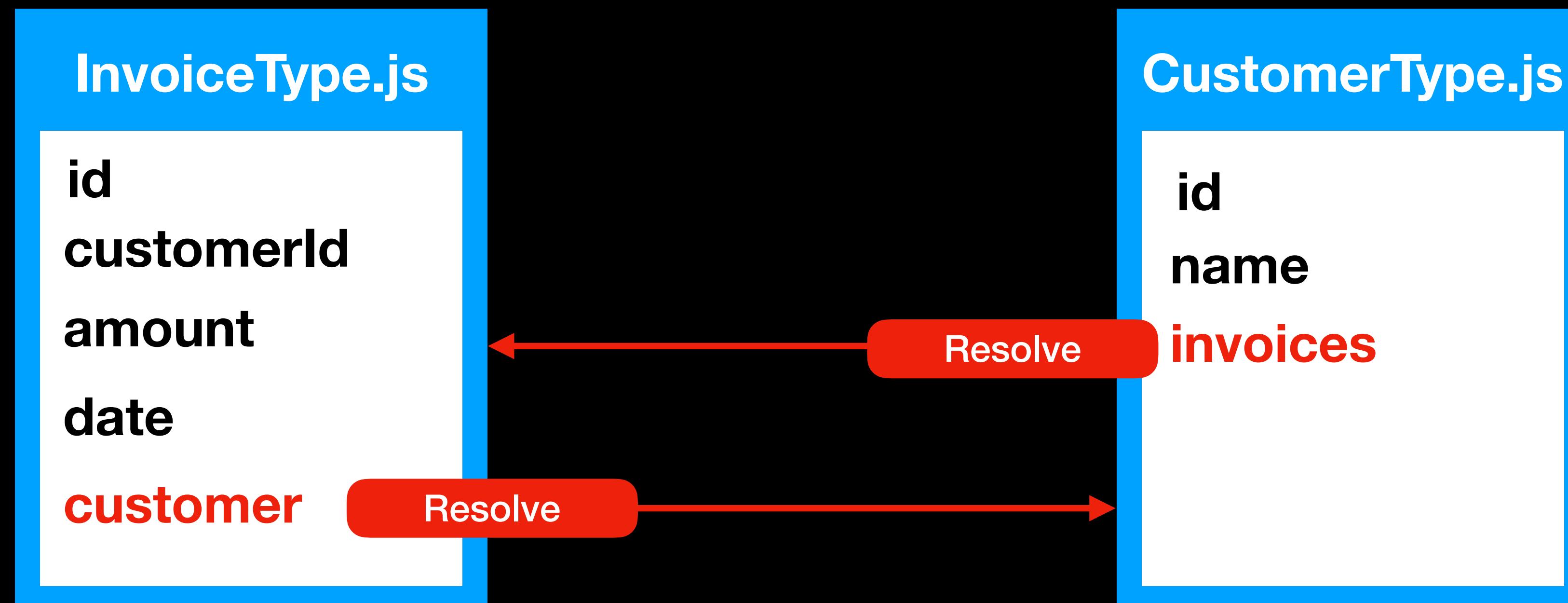
Type: User



- via some cli/script
- on server boot load (better)

ssOT principle
(single source of truth)

Problem #3: mess in types and resolve functions



`select * from CUSTOMER
where ...`

`select * from INVOICE
where ...`

InvoiceType.js contains CUSTOMER query
CustomerType.js contains INVOICE query

Problem #3: mess in types and resolve functions

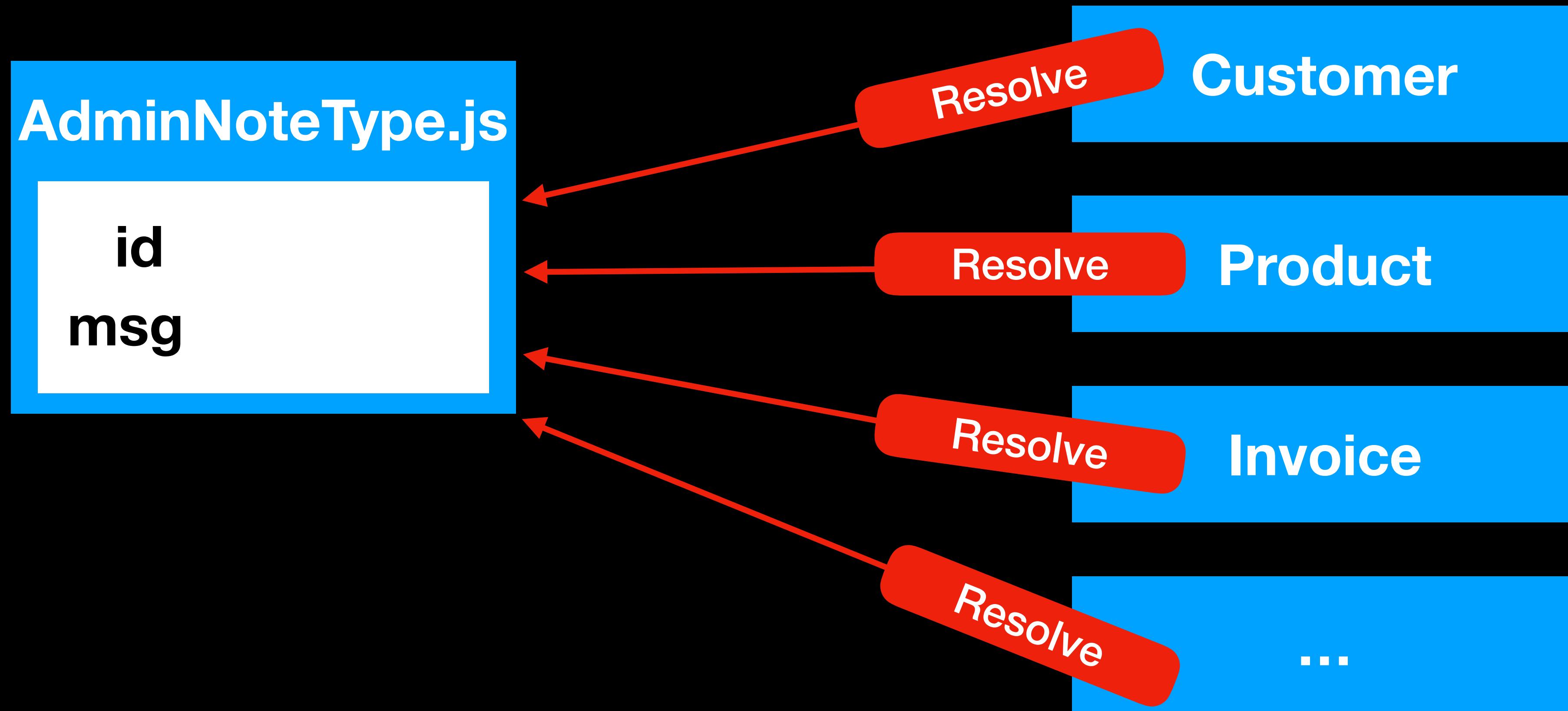
CustomerType.js

```
id  
name  
invoices  
transactions  
tickets  
events  
likes  
messages  
...
```

```
select * from INVOICES where ...  
select * from TRANSACTIONS where ...  
select * from TICKETS where ...  
select * from EVENTS where ...  
select * from LIKES where ...  
select * from MESSAGES where ...  
select * from ...
```

**CustomerType.js knows too much
about queries of others types**

Problem #3: mess in types and resolve functions



What if you need to restrict access for some group of users?
Modify resolvers in all places?

Solution: GraphQL Models*

* “GraphQL Model” is not a part of GraphQL specification.

It's suggested additional layer of abstraction for more comfortable way to construct and maintain your schema and relations into it.

Solution: GraphQL Models

```
class CustomerGQLModel {  
    type: CustomerGraphQLType;  
    resolvers: {  
        findById: {  
            type: CustomerGraphQLType,  
            args: { id: 'Int!' },  
            resolve: (_, args) =>  
                load(`select * from customer where id = ${args.id}`),  
        },  
        findMany: { ... },  
        createOne: { ... },  
        updateOne: { ... },  
        removeOne: { ... },  
        ...  
    };  
    inputType: CustomerGraphQLInputType;  
}
```

1 Type

2 MAP<FieldConfig>

3 InputType

Contains:

1. type definition
2. all possible ways to CRUD data
3. may have other helper methods and data

Writing Types via SDL and providing resolvers separately.

```
const typeDefs = ` type args
type Query {
  customer(id: Int!): Customer
  invoices(limit: Int): [Invoice]
}
type Customer {
  id: Int!
  firstName: String
  invoices: [Invoice]
}`;
```

```
const schema = makeExecutableSchema({ typeDefs, resolvers });
```

```
const resolvers = { resolve
Query: {
  customer: (_, { id }) =>
    Customer.find({ id }),
  invoices: (_, { limit }) =>
    Invoice.findMany({ limit }),
},
Customer: {
  invoices: (source) =>
    Invoice.find({ customerId: source.id }),
},
};
```

It's nice developer experience for small to medium sized schema

BUT...

Hard to work with complex input args

```
type Query {
  invoices(filter: FilterInput): [Invoice]
}
```

```
input FilterInput {
  num: Int
  dateRange: DateRangelnput
  status: InvoiceStatusEnum
}
```

```
input DateRangelnput {
  min: Date
  max: Date
}
```

```
enum InvoiceStatusEnum {
  unpaid paid declined
}
```

All highlighted parts with red lines should be in sync

```
invoices: (_, { filter }) => {
  const { num, dateRange, status } = filter;
  const q = {};
  if (num) q.num = num;
  if (dateRange)
    q['date.$inRange'] = dateRange;
  if (status) q.status = status;
  return Post.findMany(q);
},
```

- If one InputType used in several resolvers, then the complexity of refactoring increases dramatically.
- If one InputType per resolver, then too much copy/paste almost similar types.

* This example contains an error in the code, try to find it ;)

Solution: build the schema programmatically

Generate FieldConfigs via your custom functions (Resolvers) ...

```
class InvoiceGQLModel {  
  findManyResolver(configOptions) {  
    return {  
      type: InvoiceType,  
      args: {  
        filter: { type: new GraphQLInputObjectType({ ... })},  
        resolve: (_ , args) => Invoice.findMany(args),  
      }  
    }  
  }  
  findByIdResolver() { ... }  
  ...
```

... and then ...

Solution: build the schema programmatically

...and then build your Schema from fields and your Resolvers

```
import { GraphQLSchema, GraphQLObjectType } from 'graphql';
import InvoiceResolvers from './InvoiceResolvers';
```

```
const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      invoices: InvoiceResolvers.findManyResolver(),
      ...
    },
  }),
});
```

<http://graphql.org/graphql-js/constructing-types/>

Problem #4: import { makeExecutableSchema } from 'graphql-tools';

```
type Query {  
  invoices(filter: FilterInput): [Invoice]  
}  
  
resolve  
invoices: (_, { filter }) => { ... }
```

combine code from different places

back to FieldConfig

```
{  
  type: new GraphQLList(Invoice),  
  args: {  
    filter: { type: new GraphQLInputObjectType({ ... })},  
  },  
  resolve: (_, { filter }) => { ... },  
}
```

Problem #4: import { makeExecutableSchema } from 'graphql-tools';

```
type Query {  
  invoices(filter: FilterInput): [Invoice]  
}
```

```
invoices: (_, { filter }) => { ... }
```

combine code from different places

back to FieldConfig

```
{  
  type: [Invoice],  
  args: {  
    filter: `input FilterInput { ... }`,  
  },  
  resolve: (_, { filter }) => { ... },  
}
```

my code with
graphql-compose



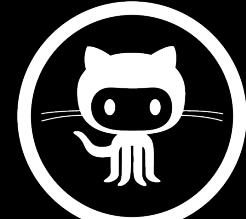
For backend
developers

Graphql-compose packages

Graphql-compose-*

- OSS packages family
for generating GraphQL Types

The main idea is to generate GraphQL Schema from your ORM/Mappings at the server startup with a small lines of code as possible.



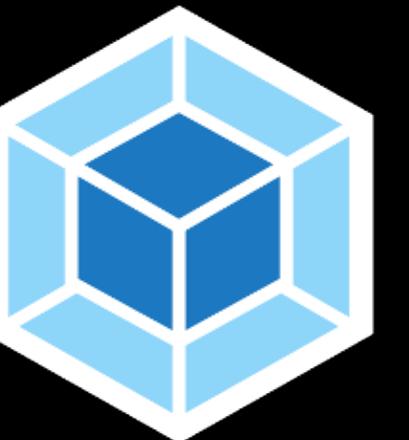
Help wanted



MIT License

Expose Flowtype/TypeScript declarations

With awful docs all packages have more than 460 starts on GitHub

**Graphql-compose
works almost like a  webpack**

**It bundles your Schema
from different type sources**

Schema creation workflow

1 ORM Schema,
Mapping

↓
Generate editable GraphQL Types
with a set of CRUD Resolvers (FieldConfigs w/ args, type, resolve)

2 TypeComposer
with Resolvers

Manually created TypeComposers
or Vanilla GraphQL types

Remove/add fields
Wrap default Resolvers with custom business logic
Create own Resolvers (FieldConfigs)
Build relations between types

3 GraphQL Schema

Graphql-compose provides handy syntax for manual type creation

```
const InvoiceItemTC = TypeComposer.create(`  
  type InvoiceItem {  
    description: String  
    qty: Int  
    price: Float  
  }  
`);
```

SDL syntax for simple types
(schema definition language)

Graphql-compose provides handy syntax for manual type creation

```
const InvoiceTC = TypeComposer.create({  
  name: 'Invoice',  
  fields: {  
    id: 'Int!',  
    now: {  
      type: 'Date',  
      resolve: () => Date.now()  
    },  
    items: () => [InvoiceItemTC],  
  },  
});
```

SDL syntax inside

Type as function, [] as List

Config Object Syntax
for complex types

Graphql-compose provides methods for modifying Types

TC.addFields({ field1: ..., field2: ... });

TC.removeField(['field2', 'field3']);

TC.extendField('lat', { description: 'Latitude', resolve: () => {} });

```
TC.hasField('lon'); // boolean
TC.getFieldNames(); // ['lon', 'lat']
TC.getField('lon'); // FieldConfig
TC.getField('lon'); // return FieldConfig
TC.getFields(); // { lon: FieldConfig, lat: FieldConfig }
TC.setFields({ ... }); // completely replace all fields
TC.setField('lon', { ... }); // replace `lon` field with new FieldConfig
TC.removeField('lon');
TC.removeOtherFields(['lon', 'lat']); // will remove all other fields
TC.reorderFields(['lat', 'lon']); // reorder fields, lat becomes first
TC.deprecateFields({ 'lat': 'deprecation reason' }); // mark field as deprecated
TC.getFieldType('lat'); // GraphQLFloat
TC.getFieldTC('complexField'); // TypeComposer
TC.getFieldArgs('lat'); // returns map of args config or empty {} if no args
TC.hasFieldArg('lat', 'arg1'); // false
TC.getFieldArg('lat', 'arg1'); // returns arg config
```



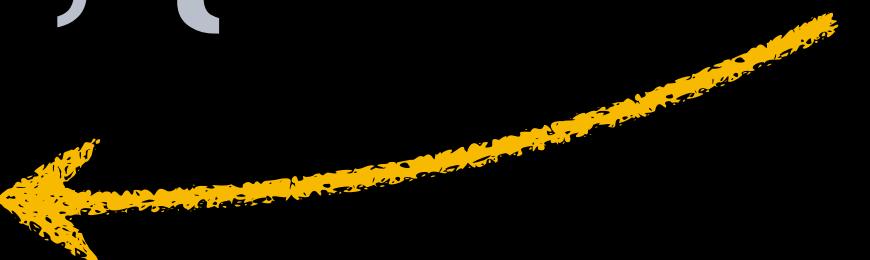
TOP 3 commonly used methods

Bunch of other useful methods

Graphql-compose create relations between Types via FieldConfig

```
InvoiceTC.addField('items', {  
  type: () => ItemsTC,  
  resolve: (source) => {  
    return Items.find({ invoiceId: source.id })  
  },  
});
```

Type as function
solves hoisting problems



Graphql-compose create relations between Types via Resolvers

```
InvoiceTC.addRelation('items', {  
  resolver: () => ItemsTC.getResolver('findMany'),  
  prepareArgs: {  
    filter: source => ({ invoiceId: source.id }),  
  },  
});
```



Prepare args for Resolver

Graphql-compose is a great tool for writing your own type generators/plugins

graphql-compose-json

type generator

graphql-compose-mongoose

type generator

resolver generator

graphql-compose-pagination

resolver generator

graphql-compose-connection

resolver generator

graphql-compose-relay

type/resolver modifier

graphql-compose-elasticsearch

type generator

resolver generator

http API wrapper



graphql-compose-aws

SDK API wrapper

Huge GraphQL Schema example

graphql-compose-aws

~700 lines of code, 2 days of work

generates more than 10 000 GraphQL Types

schema size ~2 Mb in SDL, ~9 Mb in json

JUST IN 2 DAYS



nodkz @nodkz · Dec 3

★ Star

76

Just published whole [@awscloud](#) API via [@GraphQL](#) 🎉

Discover AWS API via GraphQL 🤝

125 AWS services described by more than 10_000 GraphQL types. 😊

Live demo inside github.com/graphql-compose/graphql-compose-aws 😊



3



116



245



AWS Cloud API in GraphQL

125 Services

```
redshift(...): AwsRedshift
rekognition(...): AwsRekognition
resourcegroups(...): AwsResourceGroups
resourcegroupstaggingapi(...): AwsResourceGroupsTaggingAPI
route53(...): AwsRoute53
route53domains(...): AwsRoute53Domains
s3(...): AwsS3
ses(...): AwsSES
sms(...): AwsSMS
sns(...): AwsSNS
sqrs(...): AwsSQS
ssm(...): AwsSSM
sts(...): AwsSTS
swf(...): AwsSWF
```

3857 Operations

```
Amazon Simple Storage Service (2006-03-01)
type AwsS3 {
    abortMultipartUpload(...): AwsS3AbortMultipartUploadOutput
    completeMultipartUpload(...): AwsS3CompleteMultipartUploadOutput
    copyObject(...): AwsS3CopyObjectOutput
    createBucket(...): AwsS3CreateBucketOutput
    createMultipartUpload(...): AwsS3CreateMultipartUploadOutput
    deleteBucket(...): JSON
    deleteBucketAnalyticsConfiguration(...): JSON
    deleteBucketCors(...): JSON
    deleteBucketEncryption(...): JSON
    deleteBucketInventoryConfiguration(...): JSON
    getBucket(...): AwsS3GetBucketOutput
    getBucketAcl(...): AwsS3GetBucketAclOutput
    getBucketAnalyticsConfiguration(...): AwsS3GetBucketAnalyticsConfigurationOutput
    getBucketCors(...): AwsS3GetBucketCorsOutput
    getBucketEncryption(...): AwsS3GetBucketEncryptionOutput
    getBucketInventoryConfiguration(...): AwsS3GetBucketInventoryConfigurationOutput
    getBucketLifecycle(...): AwsS3GetBucketLifecycleOutput
    getBucketLocation(...): AwsS3GetBucketLocationOutput
    getBucketMetricsConfiguration(...): AwsS3GetBucketMetricsConfigurationOutput
    getBucketPolicy(...): AwsS3GetBucketPolicyOutput
    getBucketReplication(...): AwsS3GetBucketReplicationOutput
    getBucketTagging(...): AwsS3GetBucketTaggingOutput
    getBucketVersioning(...): AwsS3GetBucketVersioningOutput
    getBucketWebsite(...): AwsS3GetBucketWebsiteOutput
    headBucket(...): AwsS3HeadBucketOutput
    headObject(...): AwsS3HeadObjectOutput
    listBucket(...): AwsS3ListBucketOutput
    listBucketPrefixes(...): AwsS3ListBucketPrefixesOutput
    listBucketVersions(...): AwsS3ListBucketVersionsOutput
    listMultipartUploads(...): AwsS3ListMultipartUploadsOutput
    listObjects(...): AwsS3ListObjectsOutput
    listObjectsV2(...): AwsS3ListObjectsV2Output
    putBucket(...): AwsS3PutBucketInput!
    putBucketAcl(...): AwsS3PutBucketAclInput!
    putBucketAnalyticsConfiguration(...): AwsS3PutBucketAnalyticsConfigurationInput!
    putBucketCors(...): AwsS3PutBucketCorsInput!
    putBucketEncryption(...): AwsS3PutBucketEncryptionInput!
    putBucketInventoryConfiguration(...): AwsS3PutBucketInventoryConfigurationInput!
    putBucketLifecycle(...): AwsS3PutBucketLifecycleInput!
    putBucketLocation(...): AwsS3PutBucketLocationInput!
    putBucketMetricsConfiguration(...): AwsS3PutBucketMetricsConfigurationInput!
    putBucketPolicy(...): AwsS3PutBucketPolicyInput!
    putBucketReplication(...): AwsS3PutBucketReplicationInput!
    putBucketTagging(...): AwsS3PutBucketTaggingInput!
    putBucketVersioning(...): AwsS3PutBucketVersioningInput!
    putObject(...): AwsS3PutObjectInput!
```

6711 Input/Output params

```
createBucket(
    input: AwsS3CreateBucketInput!,
    config: AwsConfig
): AwsS3CreateBucketOutput
TYPE DETAILS
type AwsS3CreateBucketOutput {
    Location: String
}
ARGUMENTS
input: AwsS3CreateBucketInput!
config: AwsConfig
}
TYPE DETAILS
type AwsS3CreateBucketInput {
    ACL: String
    Bucket: String!
    CreateBucketConfiguration: AwsS3CreateBucketCreateBucketConfig
    GrantFullControl: String
    GrantRead: String
    GrantReadACP: String
    GrantWrite: String
    GrantWriteACP: String
}
```

<https://graphqlbin.com/plqhO>

Graphql-compose schema demos

Mongoose, Elastic, Northwind

<https://github.com/nodkz/graphql-compose-examples>

<https://graphql-compose.herokuapp.com>

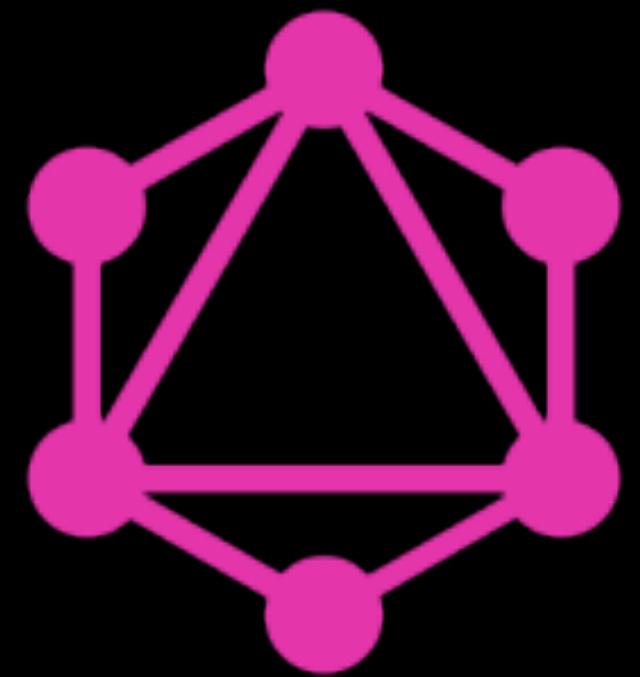
Wrapping REST API

<https://github.com/lyskos97/graphql-compose-swapi>

<https://graphql-compose-swapi.herokuapp.com>



Last words...



GraphQL is awesome!

less stress more success

less time on coding

less network traffic

less errors



Read

medium graphql

Watch

youtube graphql

Glue

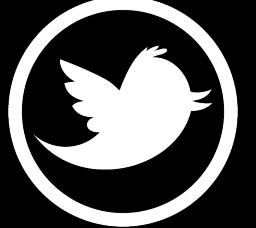
howtographql.com

Take away!

**GraphQL is powerful query language
with great tools**

**GraphQL is typed so it helps with
static analysis on clients**

Generate GraphQL Schemas on server

THANKS!
Pavel Chertorogov
 nodkz 

GraphQL is a



for your
server and client apps

