

# GraphQL

the **holy** contract between  
client and server



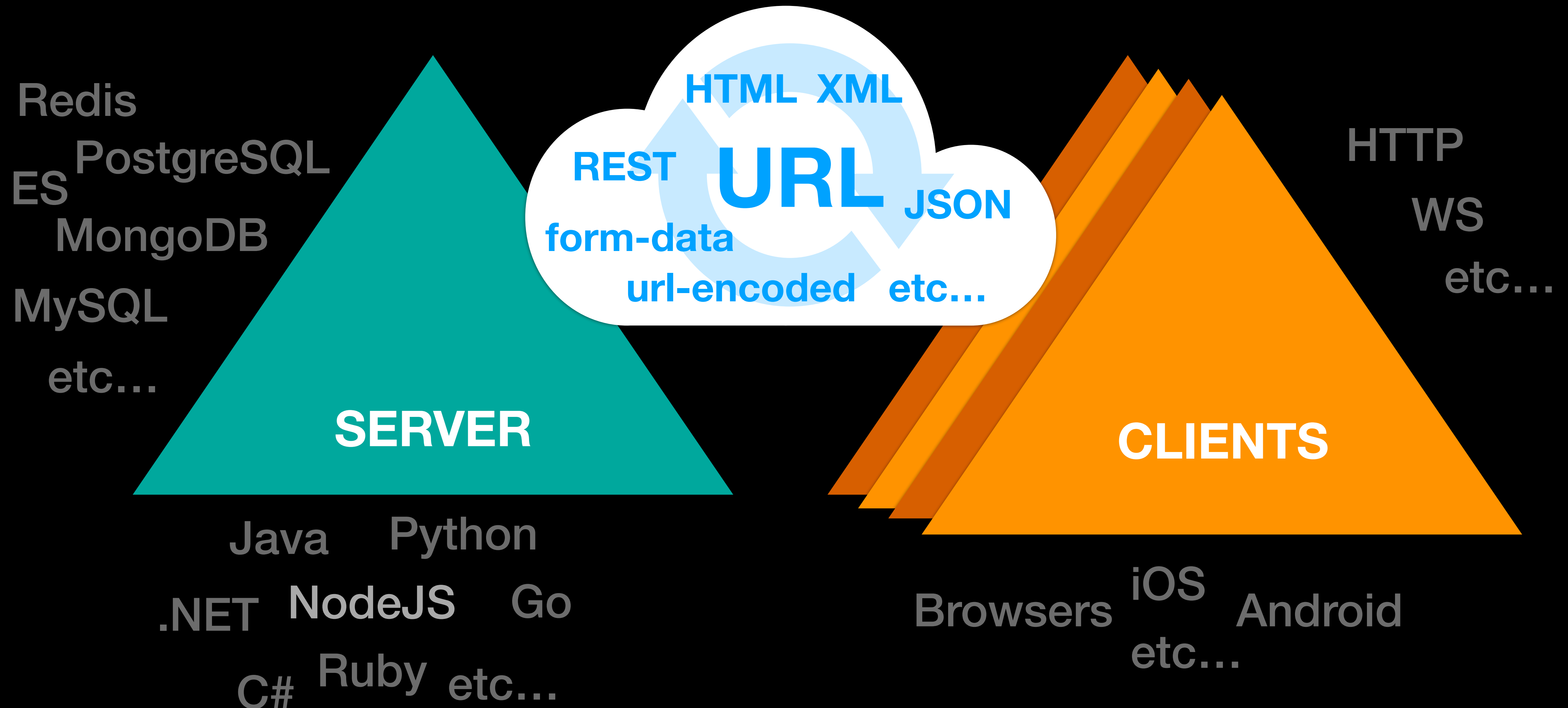


 @nodkz 

**Pavel Chertorogov**  
**with GraphQL since 2015**

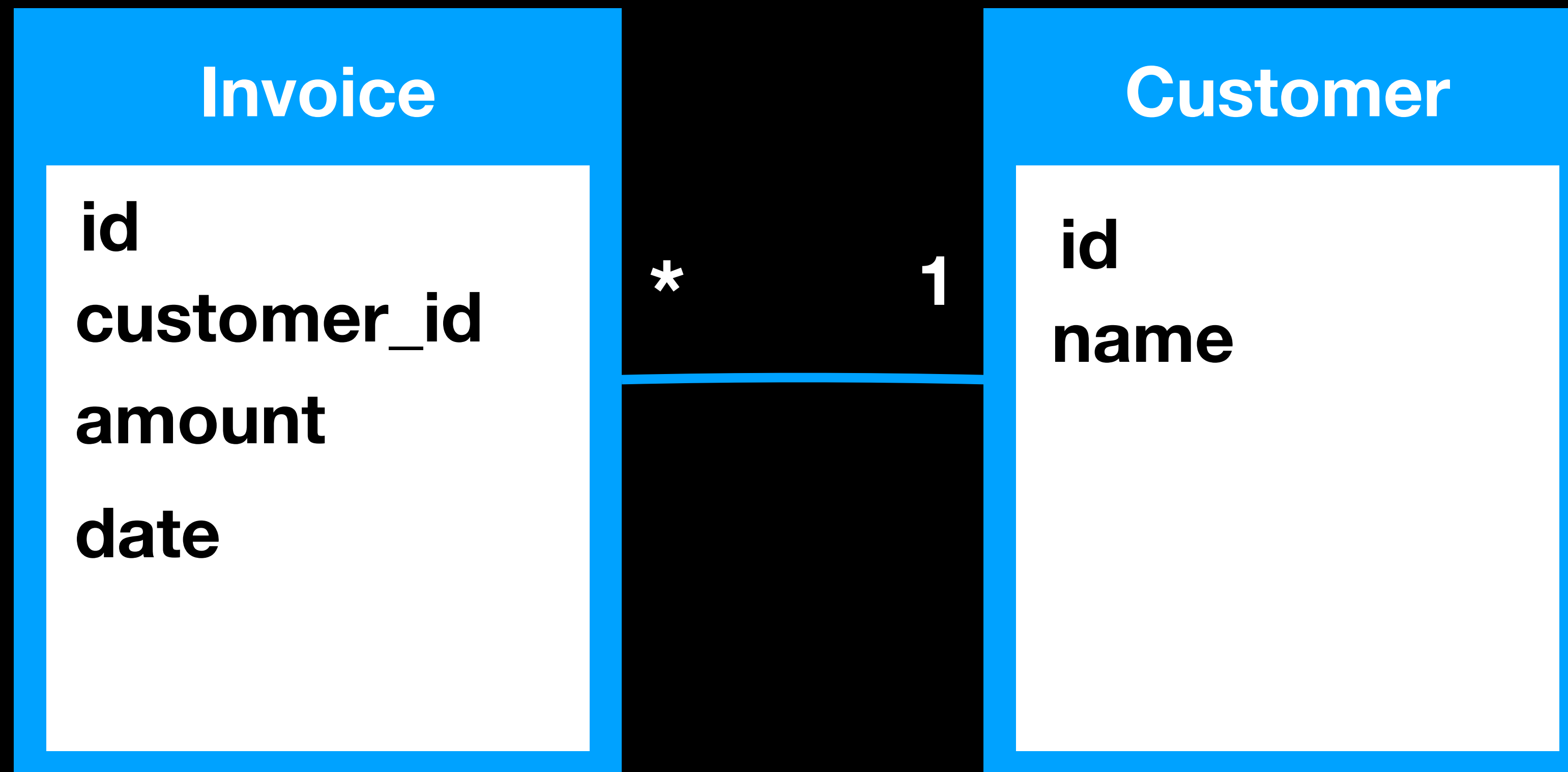
# Client-server intro

# Client-server apps



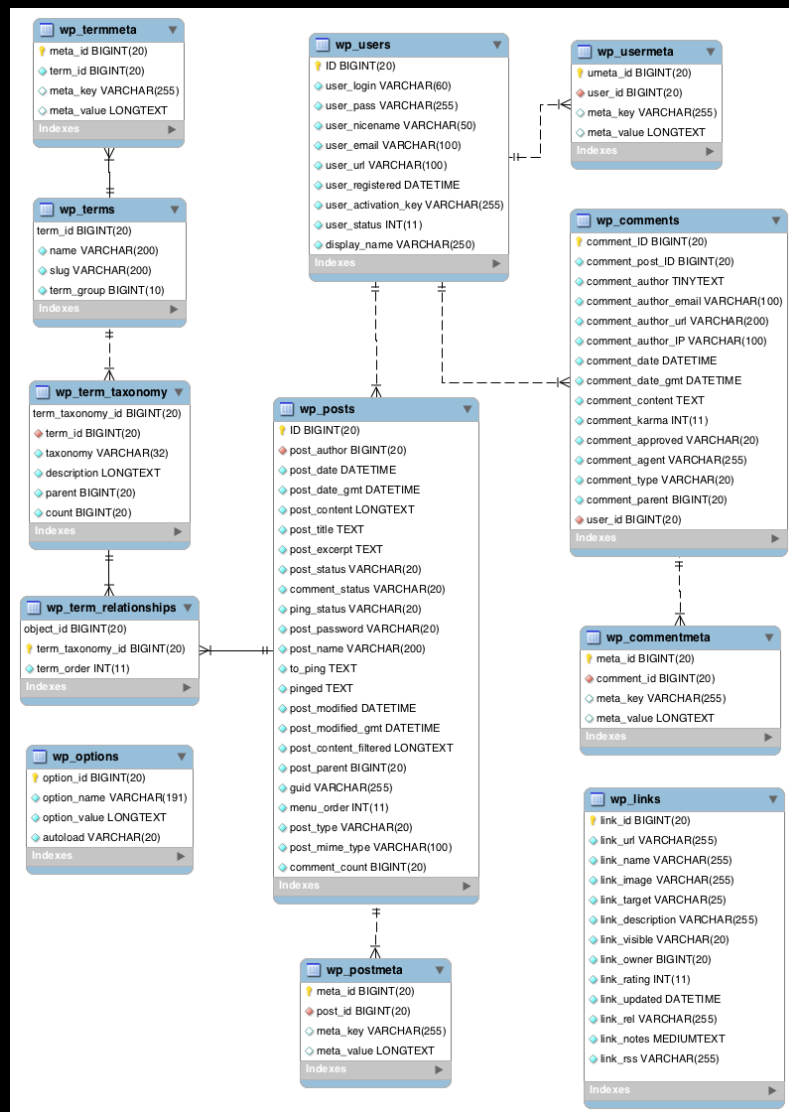


# Backend capabilities

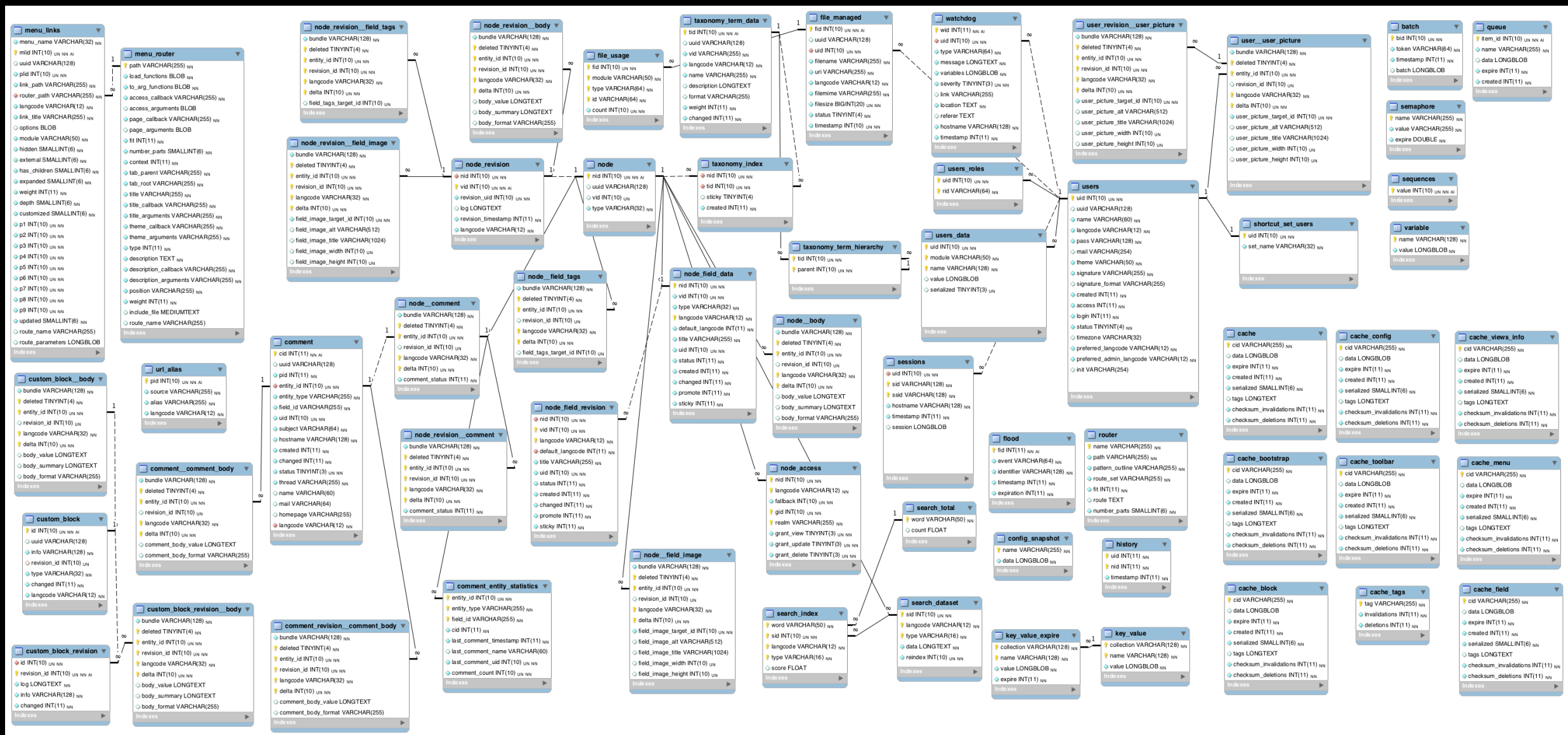


## Simple DB Schema Diagram

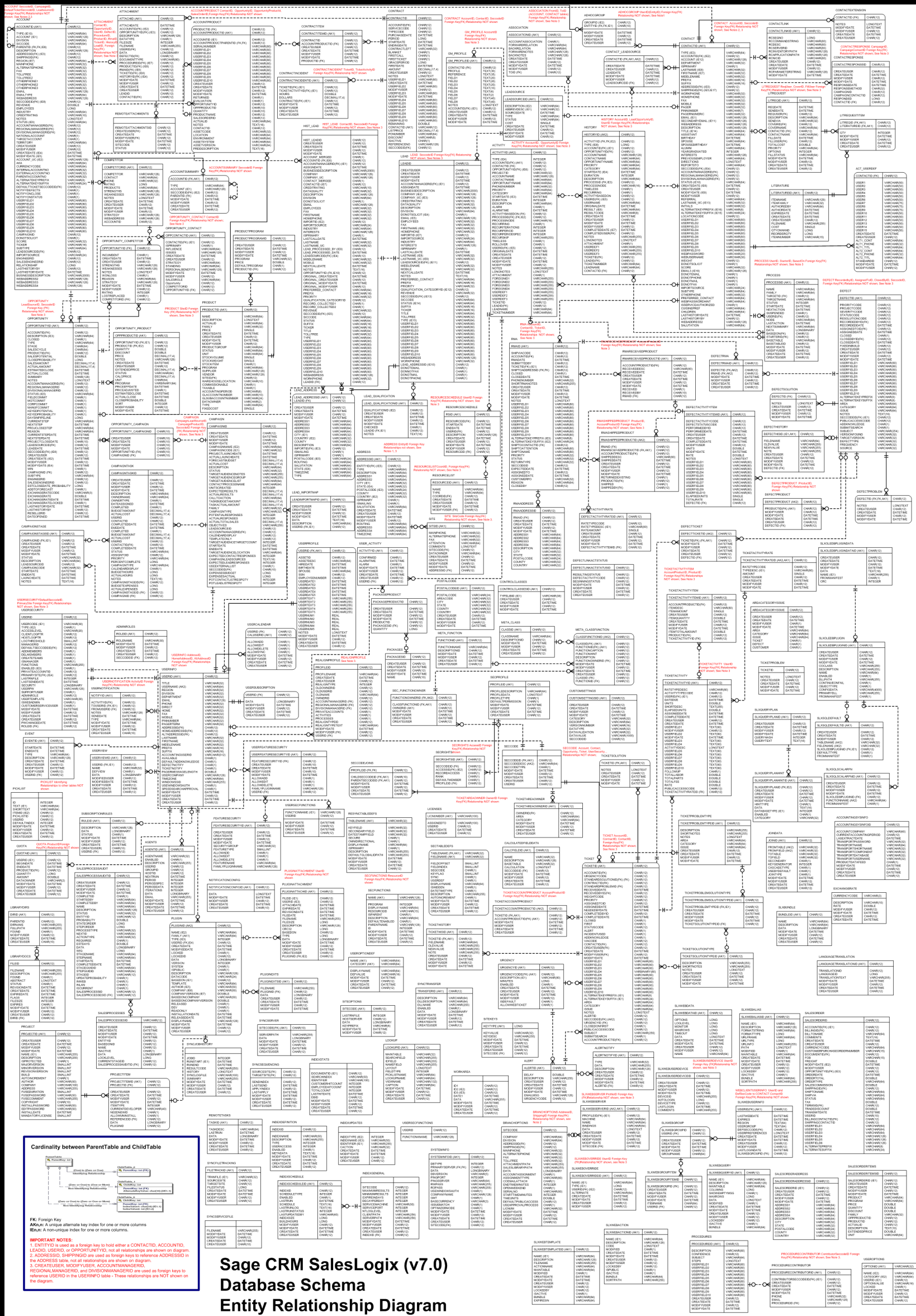




## Wordpress 4.4.2 Schema (12 tables)



# Drupal8 Schema (60 tables)

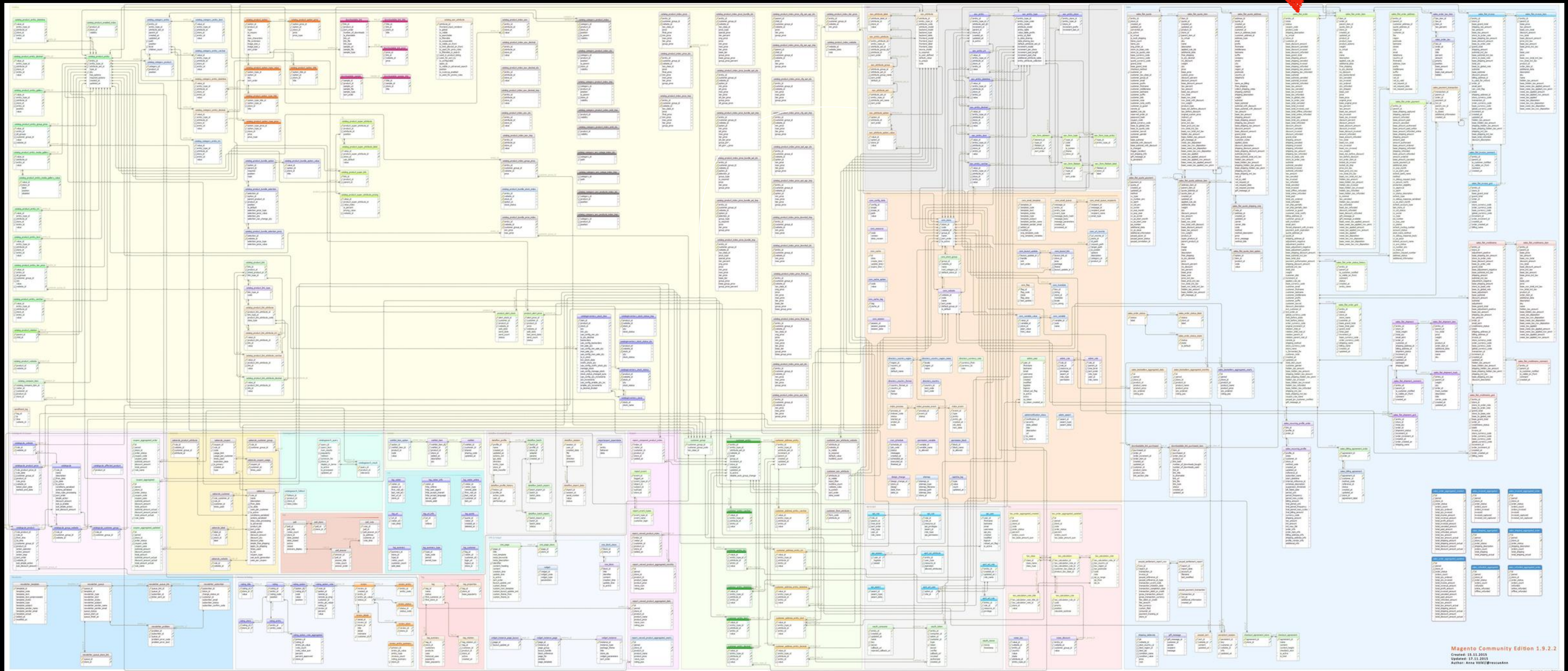


## Sage CRM SalesLogix (v7.0) Database Schema Entity Relationship Diagram



... and even more Monster with 333 tables

Order table  
(138 fields)





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

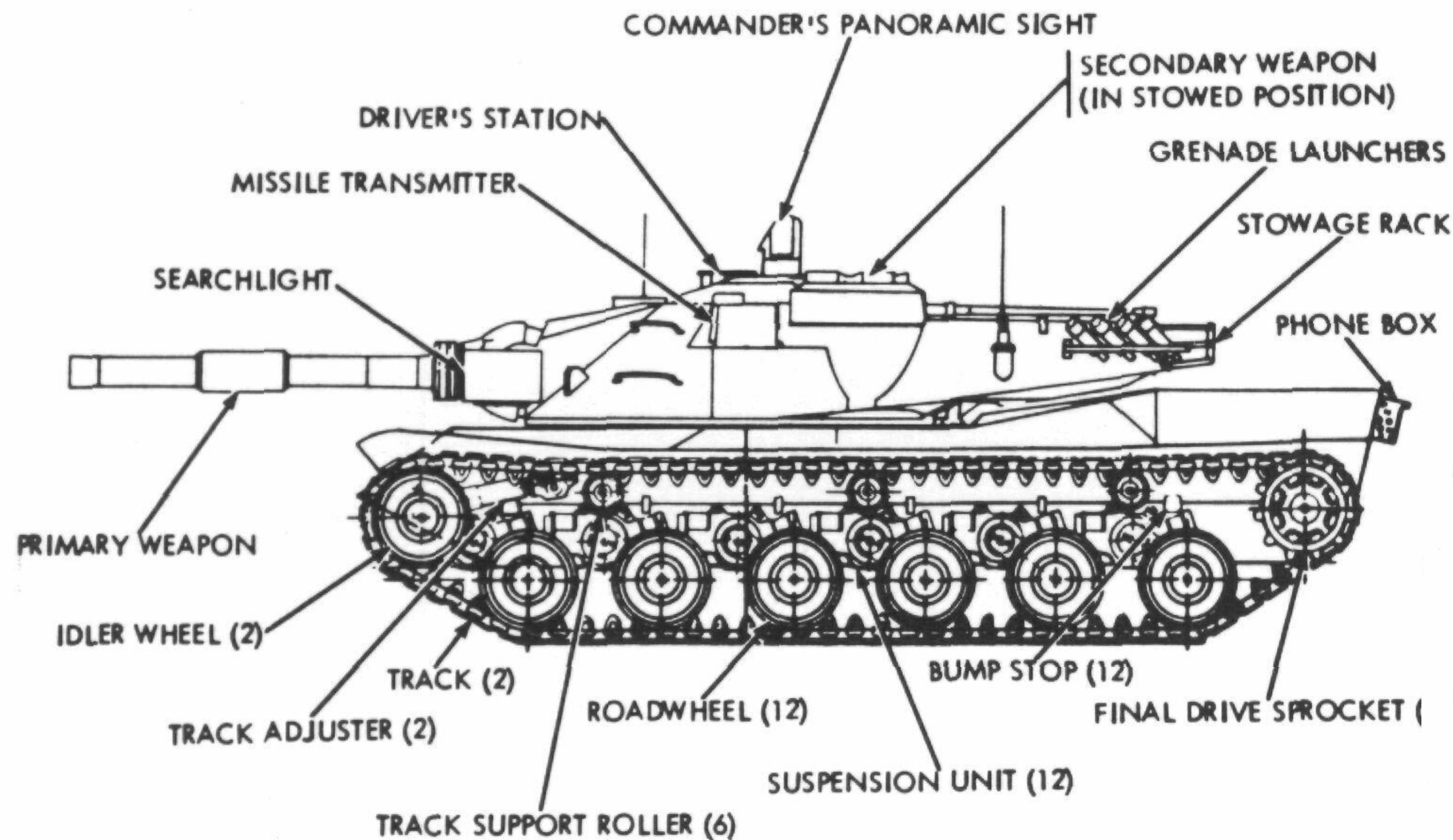




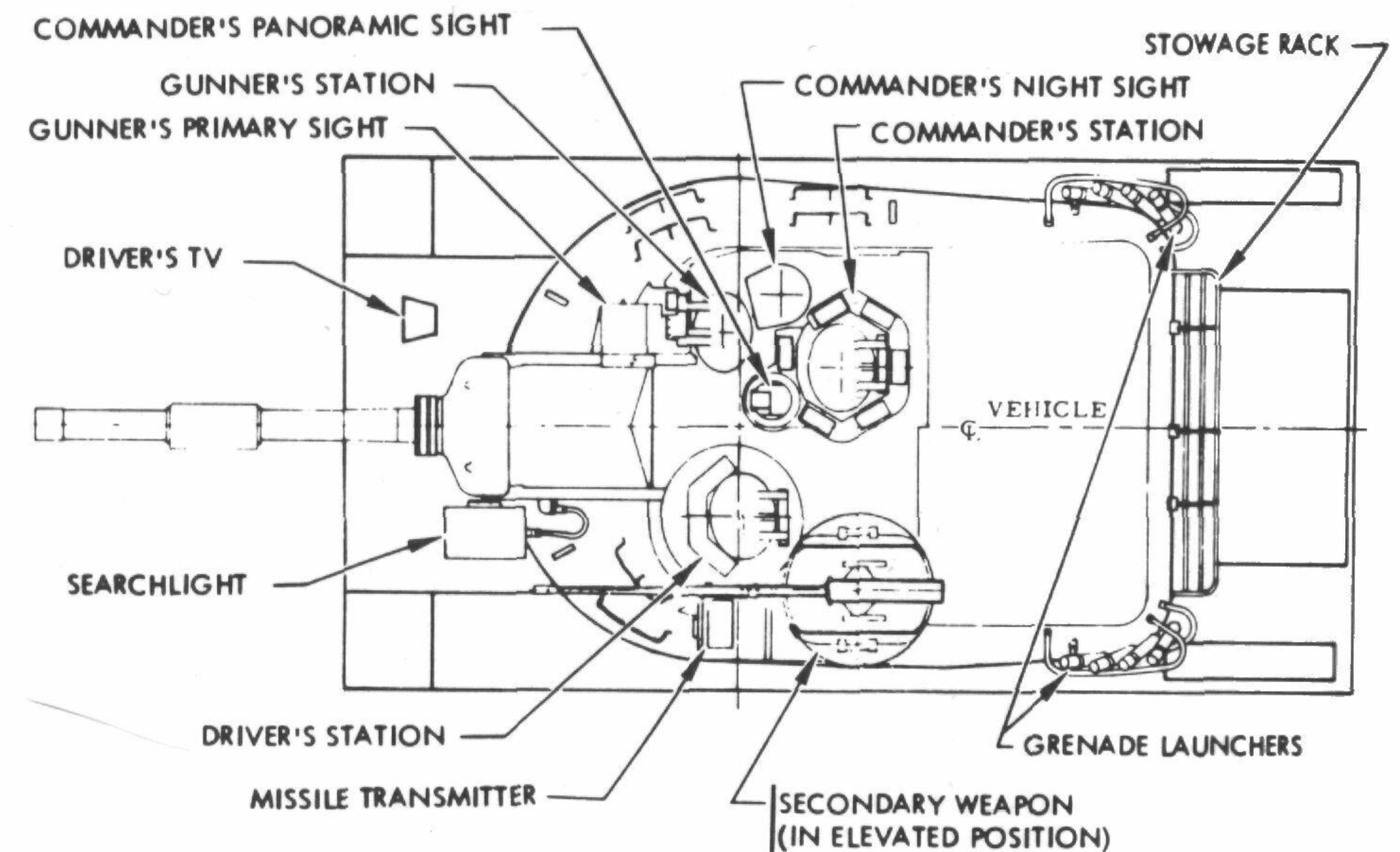
**Wait! I have a better plan!**



# This is GraphQL Schema



Wikipedia MBT-70 schema





# GraphQL basics

# GraphQL – is a ...

query  
language  
for APIs

for Frontenders

+

query  
executor  
on Schema

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 }
14
```

```
{
  "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

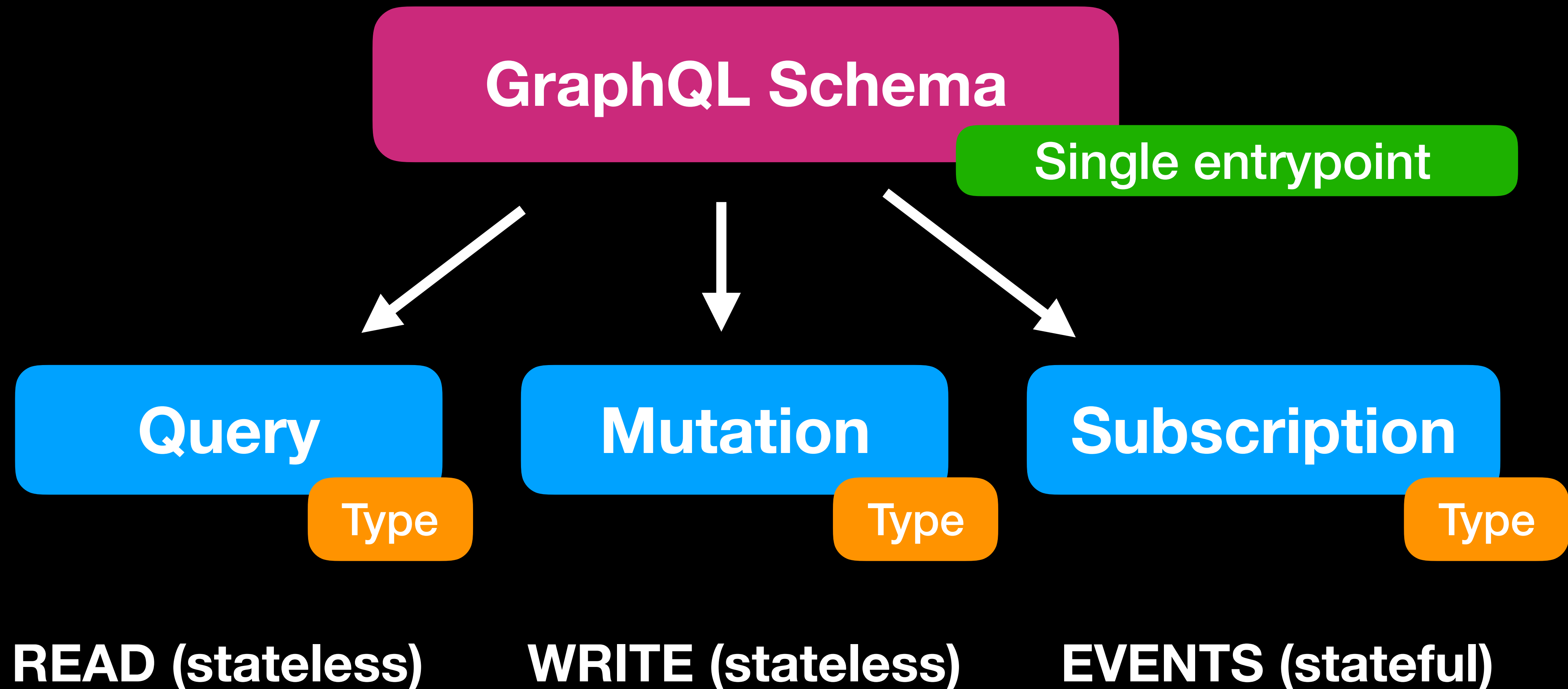
```
1
2 query {
3   viewer {
4     product {
5       name
6       unitPrice
7       category {
8         name
9         description
10      }
11    }
12  }
13 }
14
```

```
{
  "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 Schema



# ObjectType

Type

Name

Field 1

Field N

Type

Invoice

Name

id Field

customerId Field

amount Field

date Field



# Field Config

Type

Field 1

Field N

Type

**Scalar or Object Type which returns resolve**

Resolve

**Function with fetch logic from any data source**

Args

**Set of input args for resolve function**

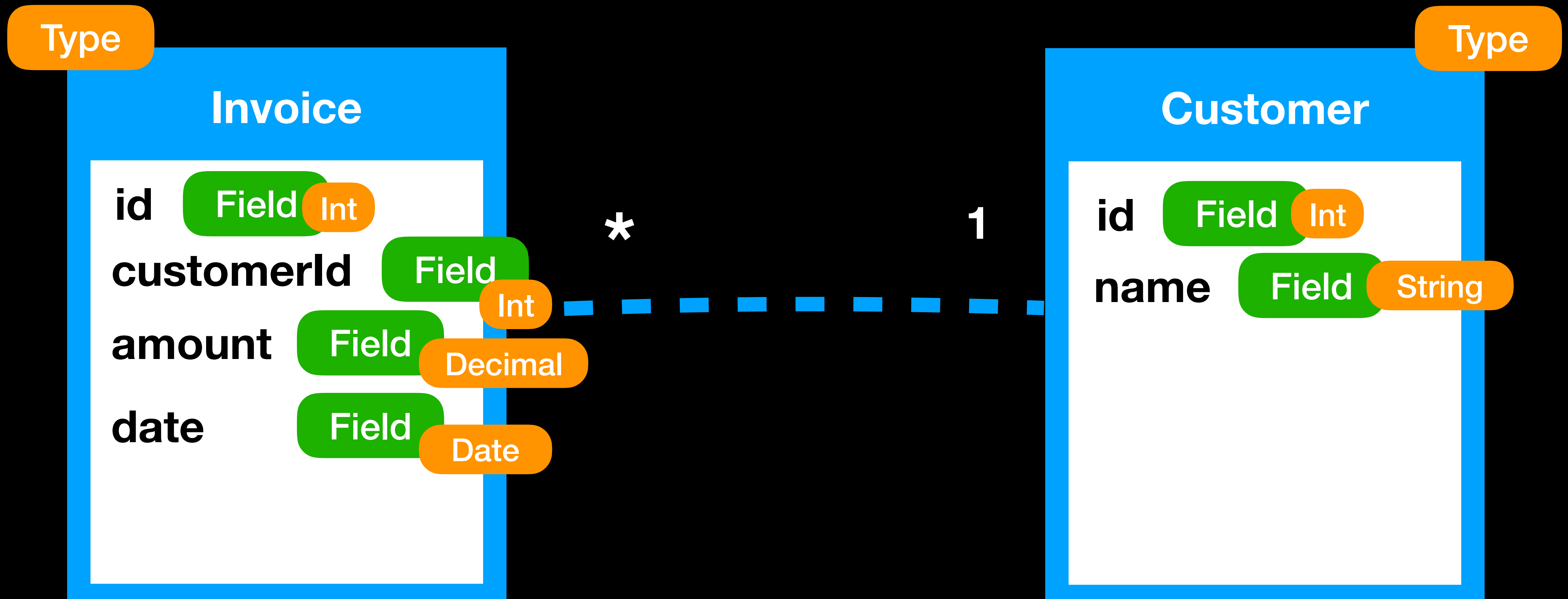
Description

**Documentation**

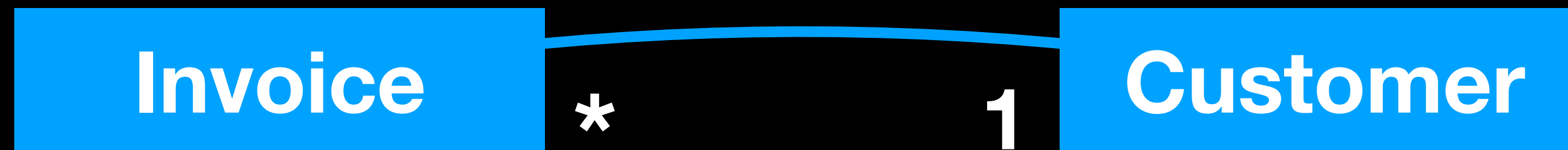
DeprecationReason

**Field hiding**

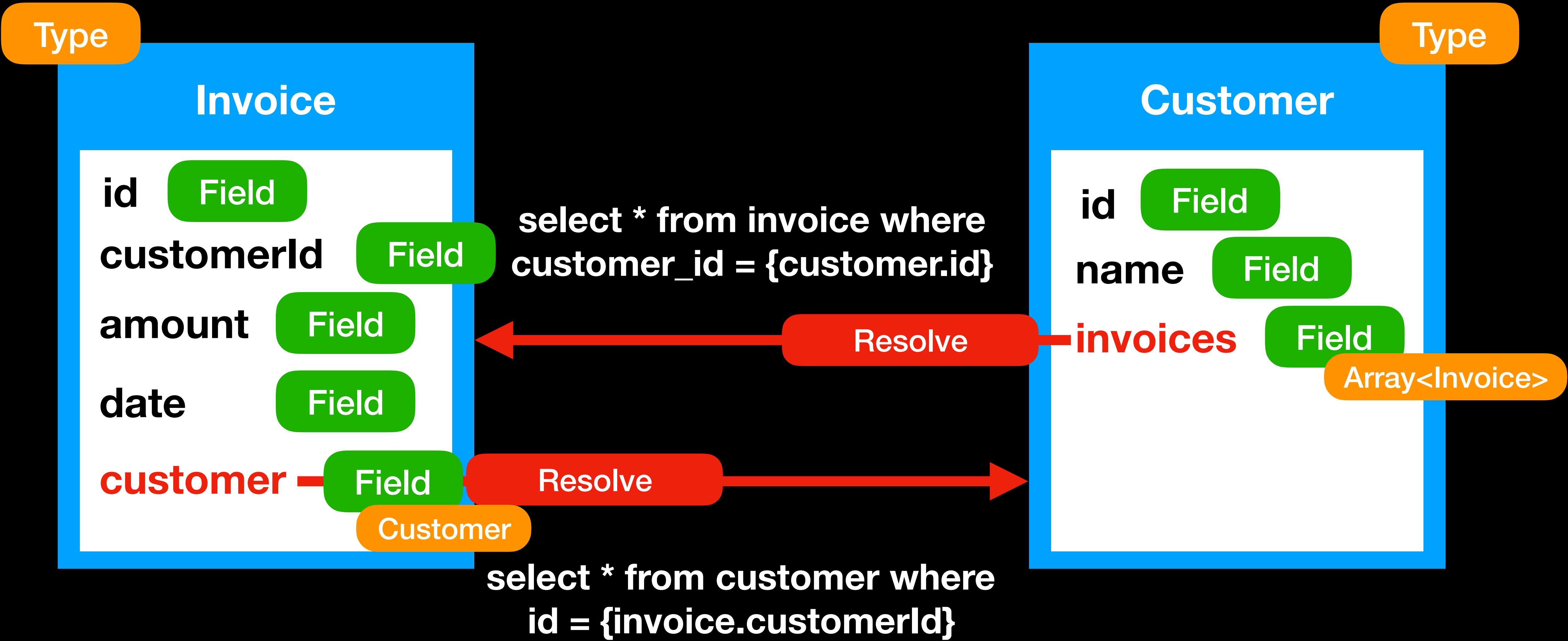
# Relations between types



# 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!

Type

## Invoice

id Field  
customerId Field  
amount Field  
date Field

select \* from invoice where  
customer\_id = {customer.id}  
limit {args.limit}

Resolve

Args

Limit: 3

Type

## Customer

id Field  
name Field  
invoices Field  
Array<Invoice>

## **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
- ~~resolve~~
- directives
- input types
- enums
- interfaces
- unions

**SDL format  
(txt)**

```
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")  
}
```

# 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

FieldConfig

Type

Args

Resolve

Description

DeprecationReason

# Don't forget to read about

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

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

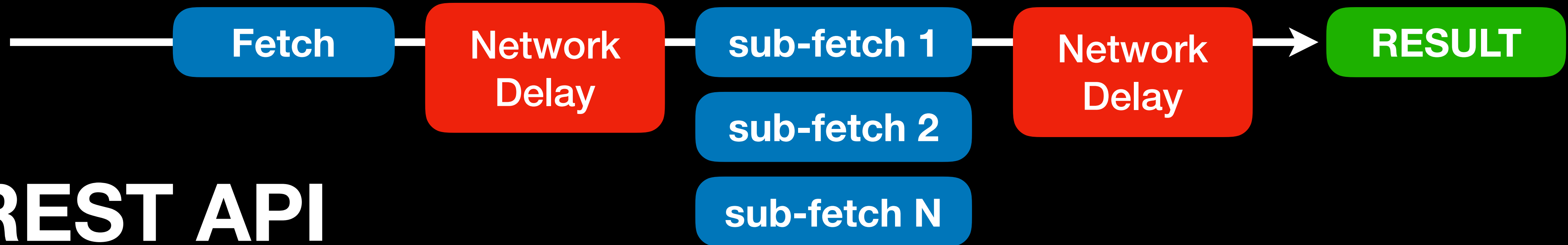




**Backend capabilities**  
**Client requirements**

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

# GraphQL Demo



## REST API

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



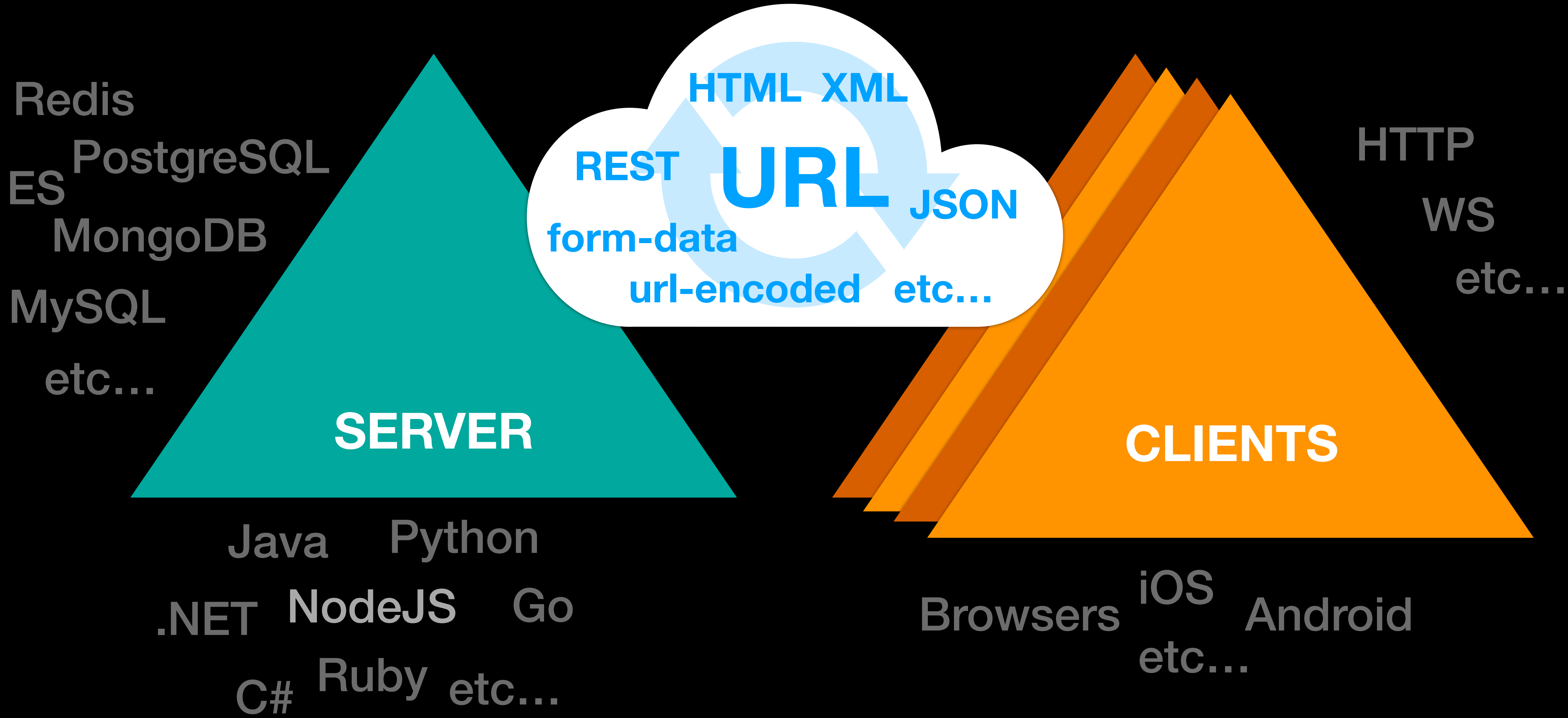
## GraphQL

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

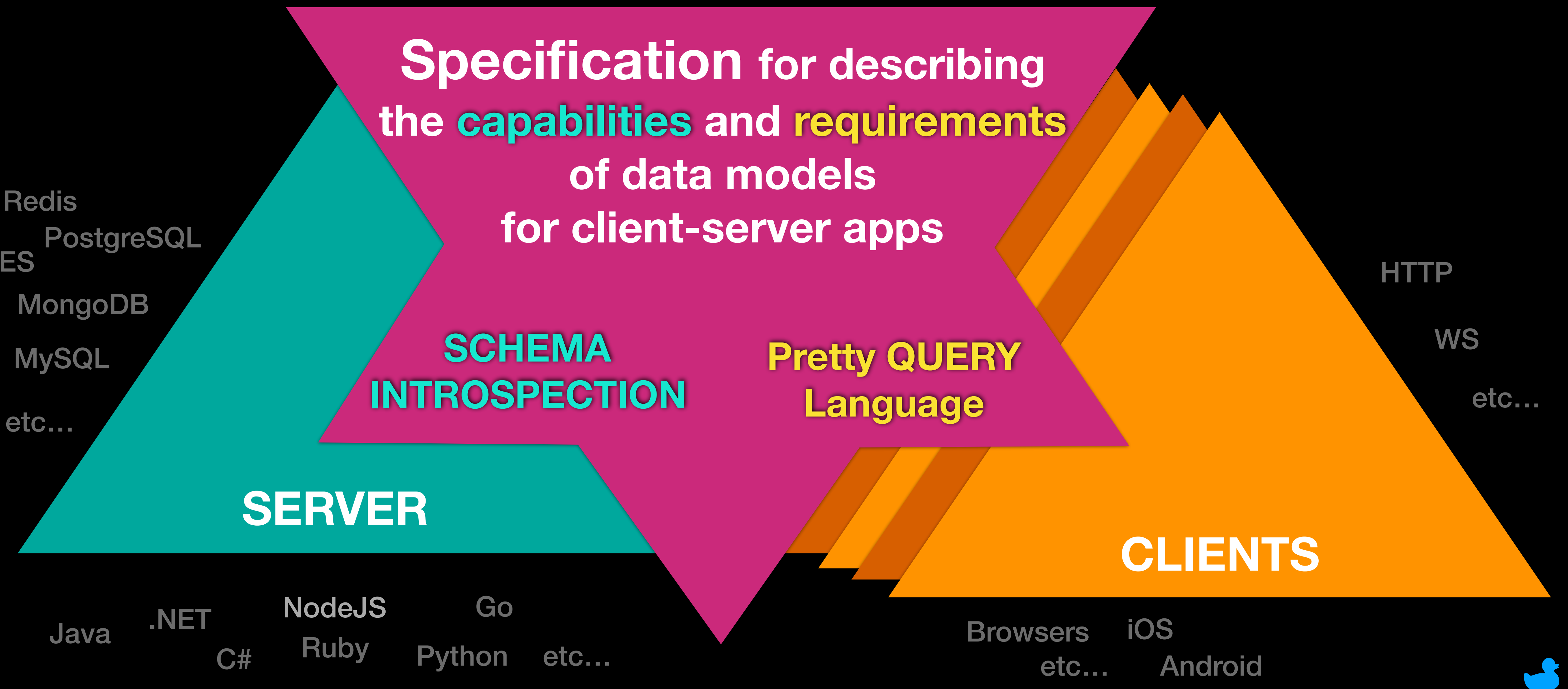


A copy from one of the previous slides...

# Client-server apps



# GraphQL – is a query language for APIs



**For frontend  
developers**

# Static Analysis

**Static type checks**

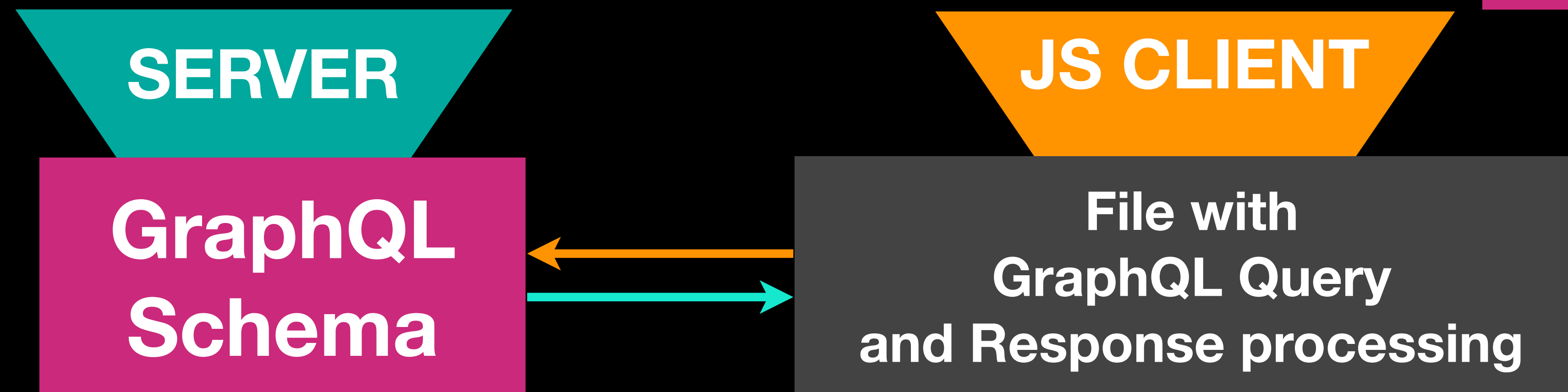


# PRODUCTIVITY

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

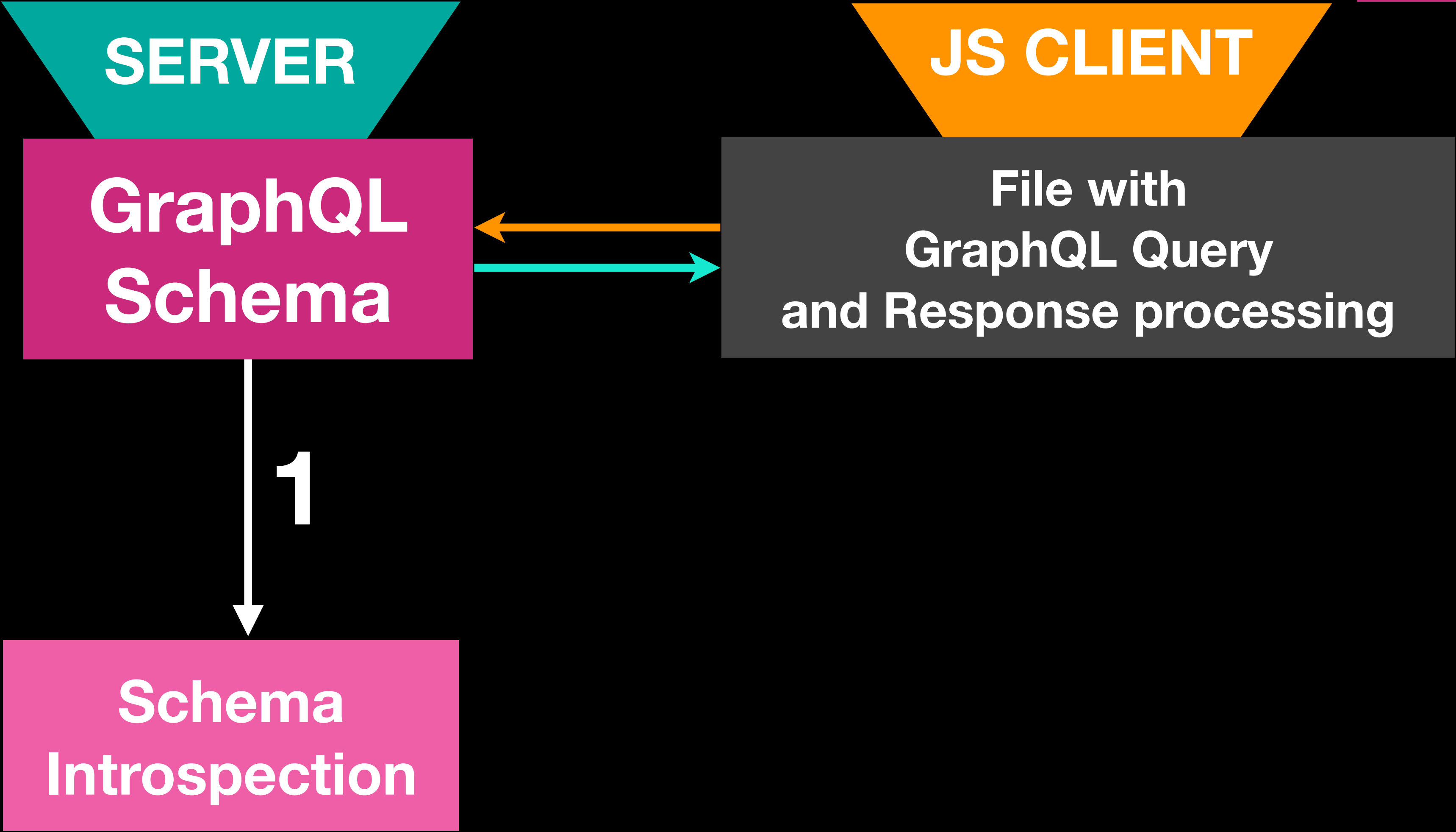


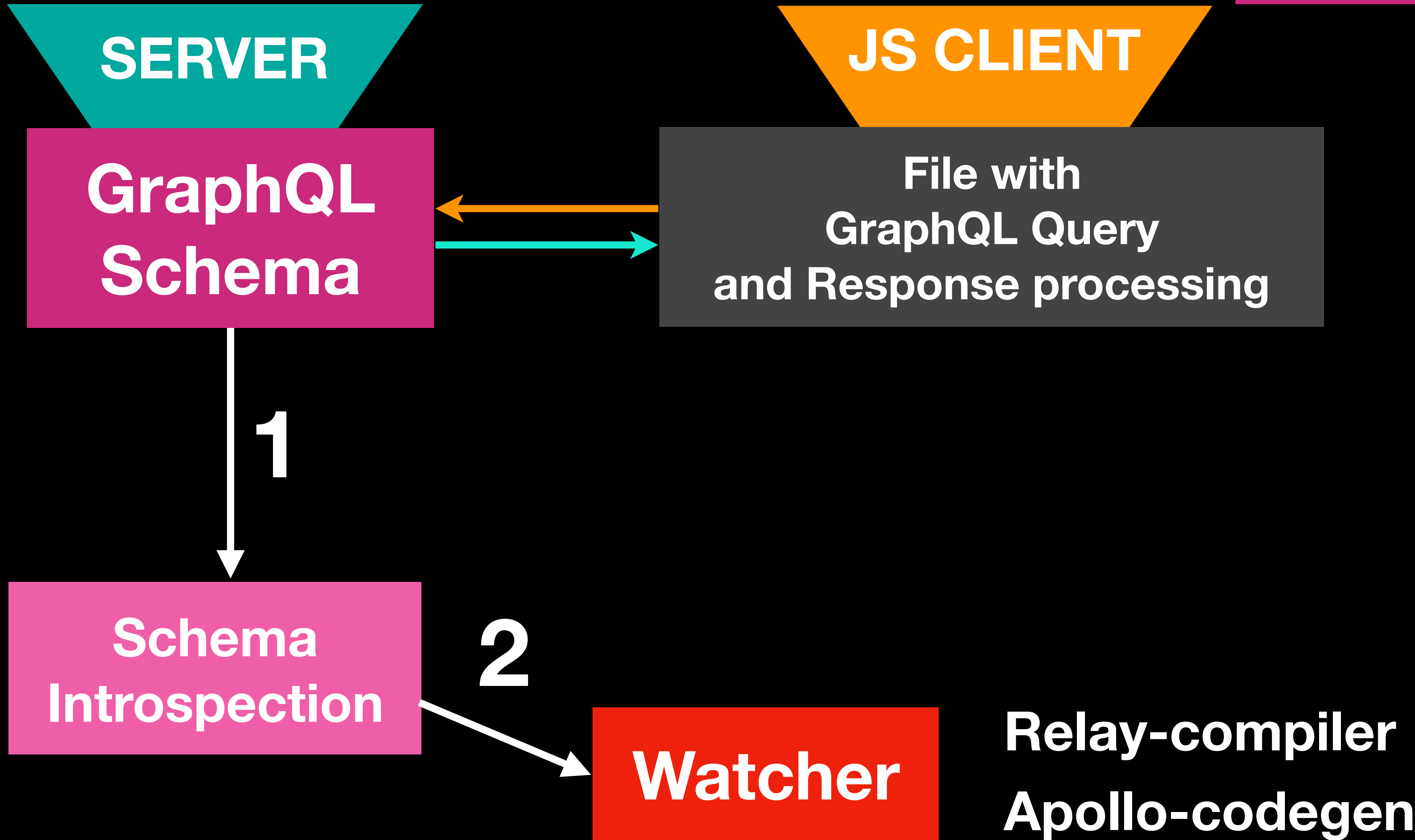
Static program analysis is the analysis of computer software that is performed without actually executing programs

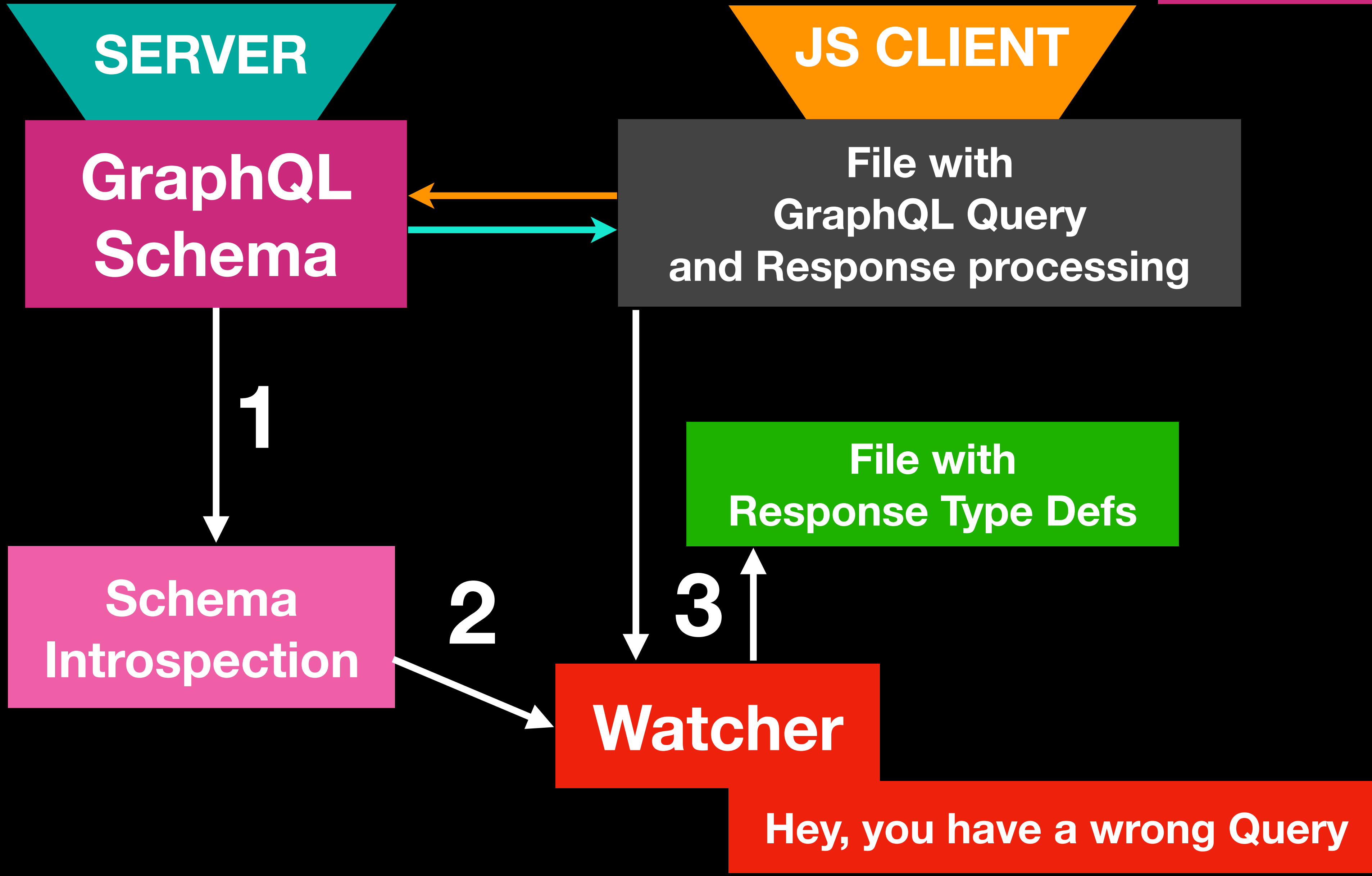


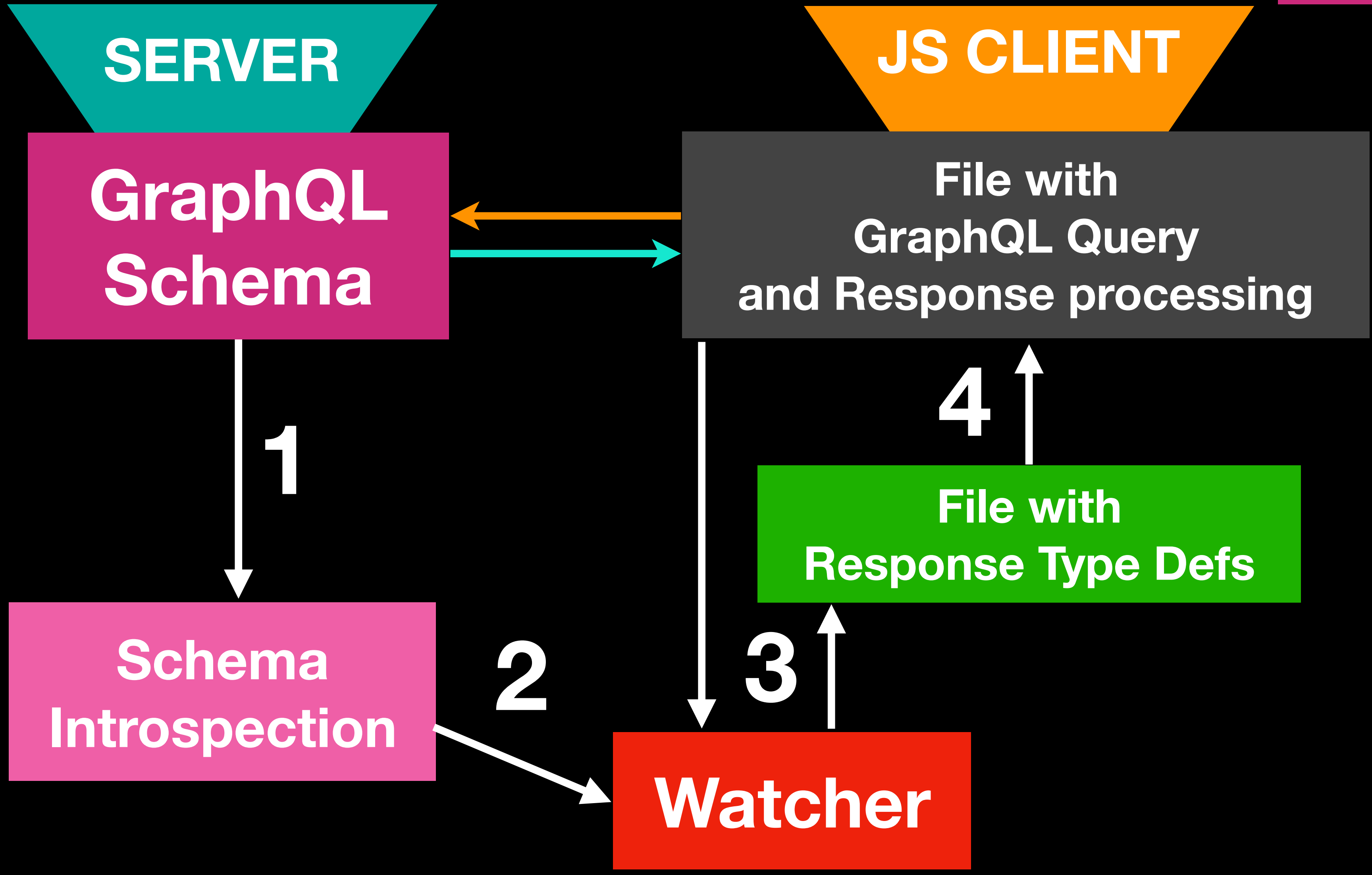
**Let's turbo-charge  
our client apps static analysis  
with GraphQL queries**

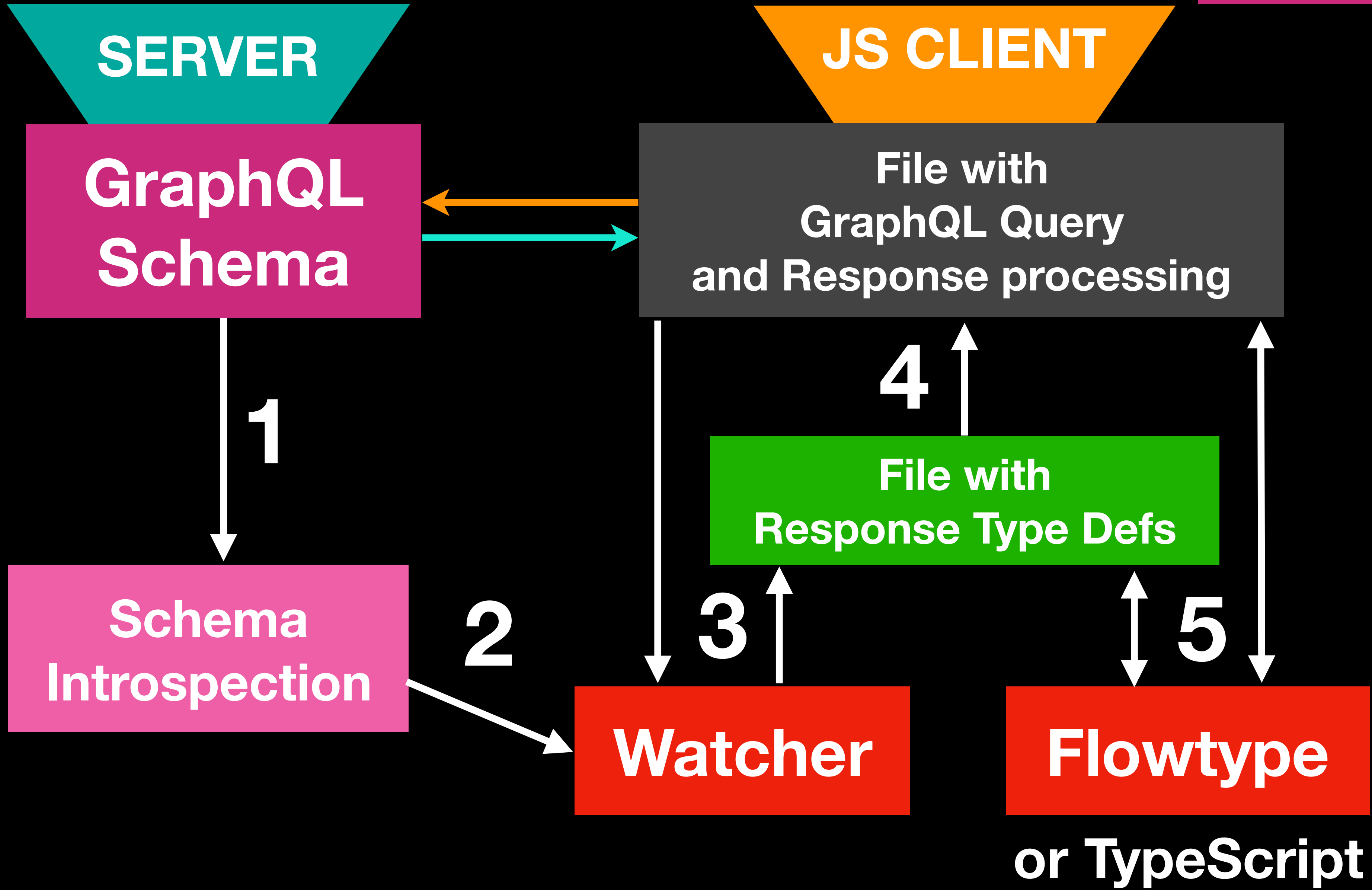


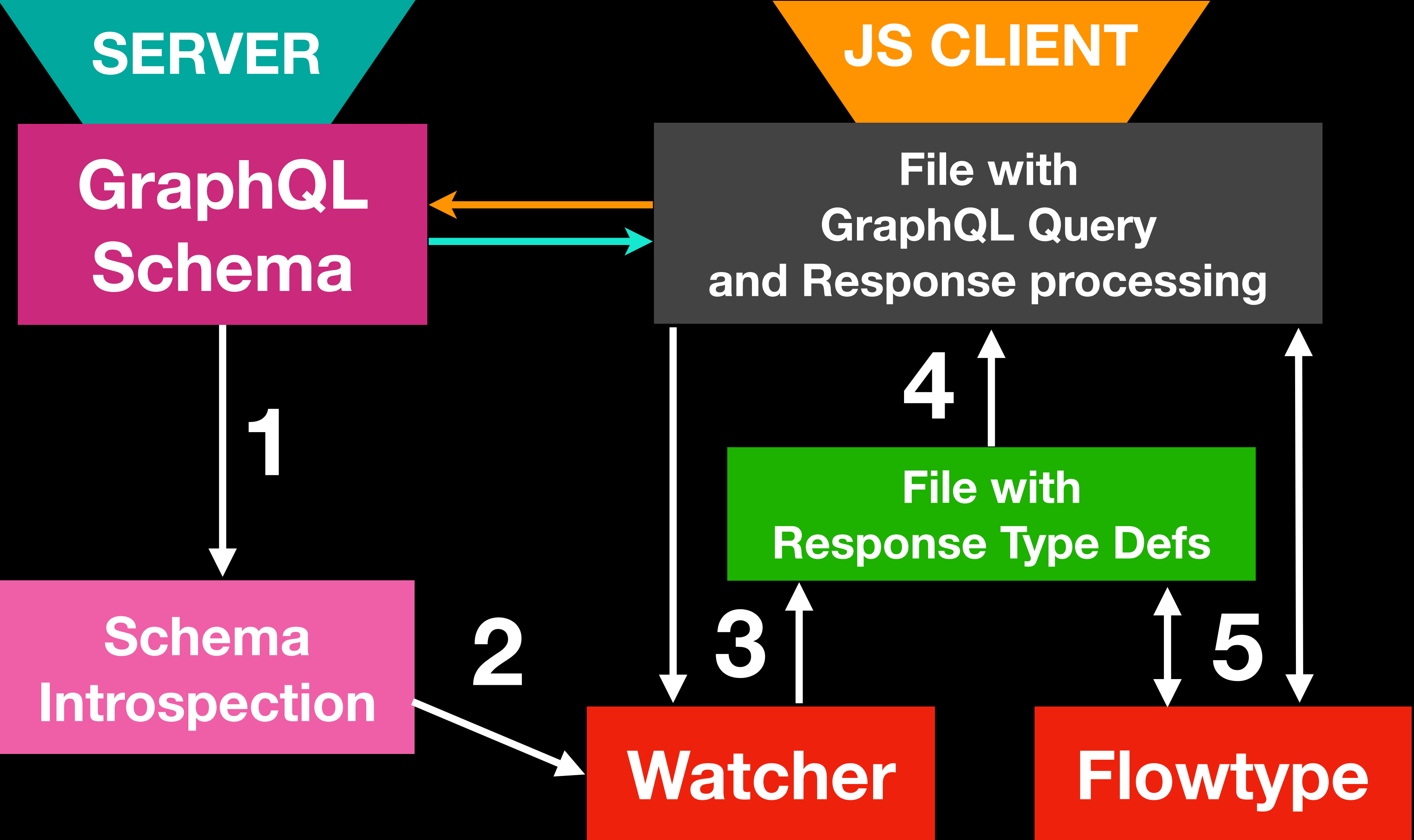












*Houston, we have a Type Check problem at line 19287 col 5: possible undefined value*

# DEMO

GraphQL Query

Generated Response Type Def

Crappy Code

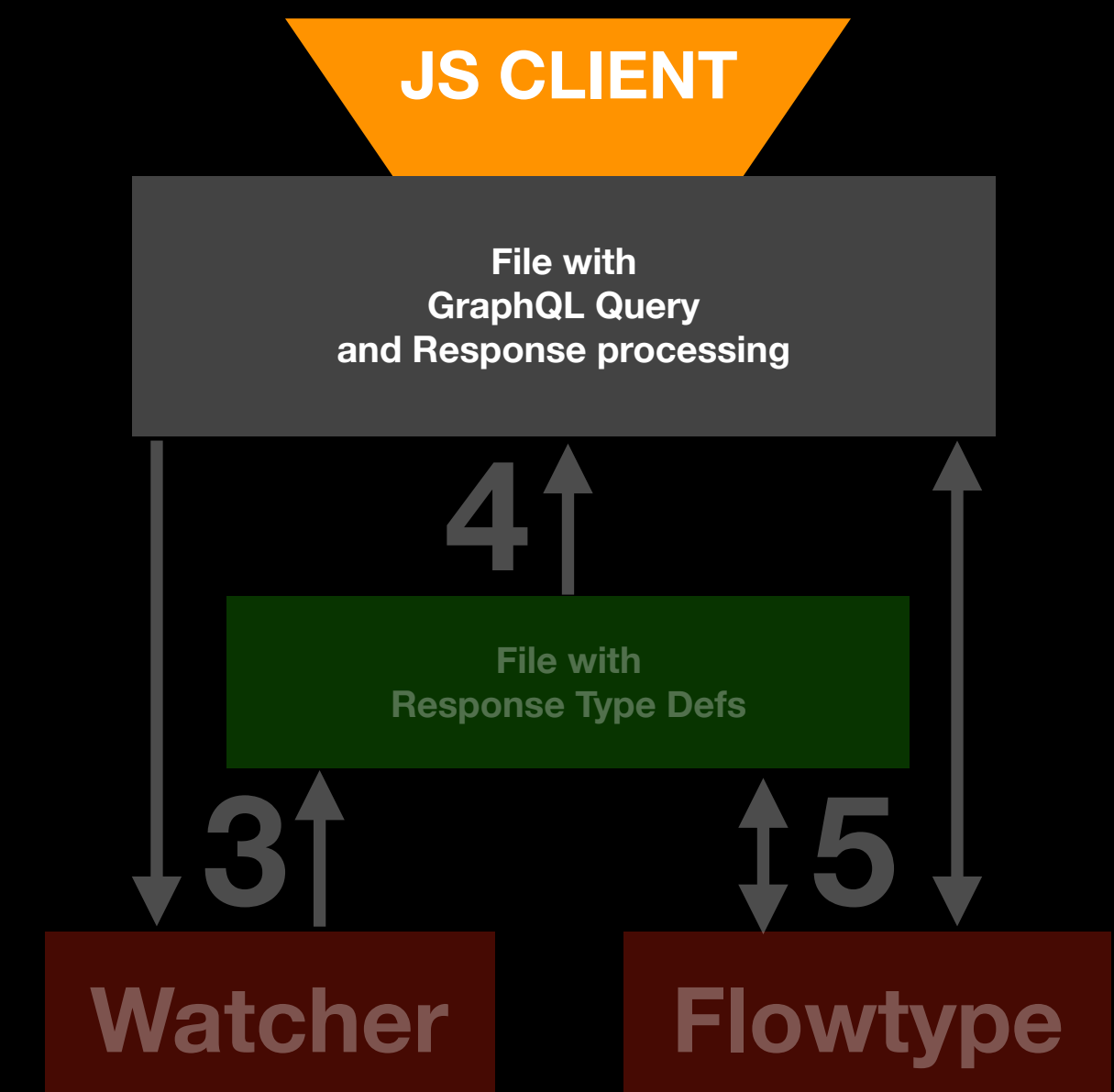
Flow typed Code

Flow error



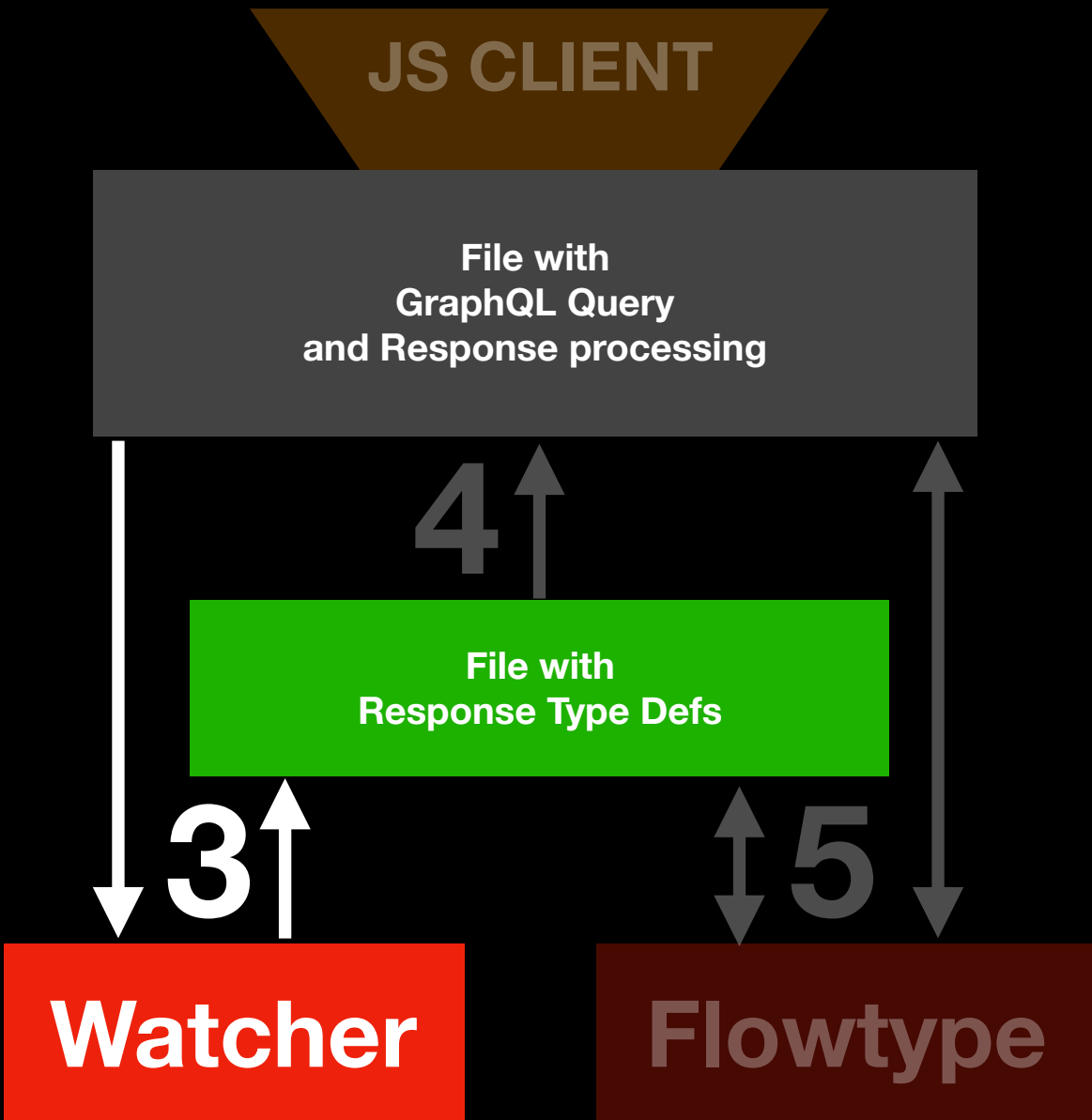
## GraphQL Query

```
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;
    };
  };
};
*/
```



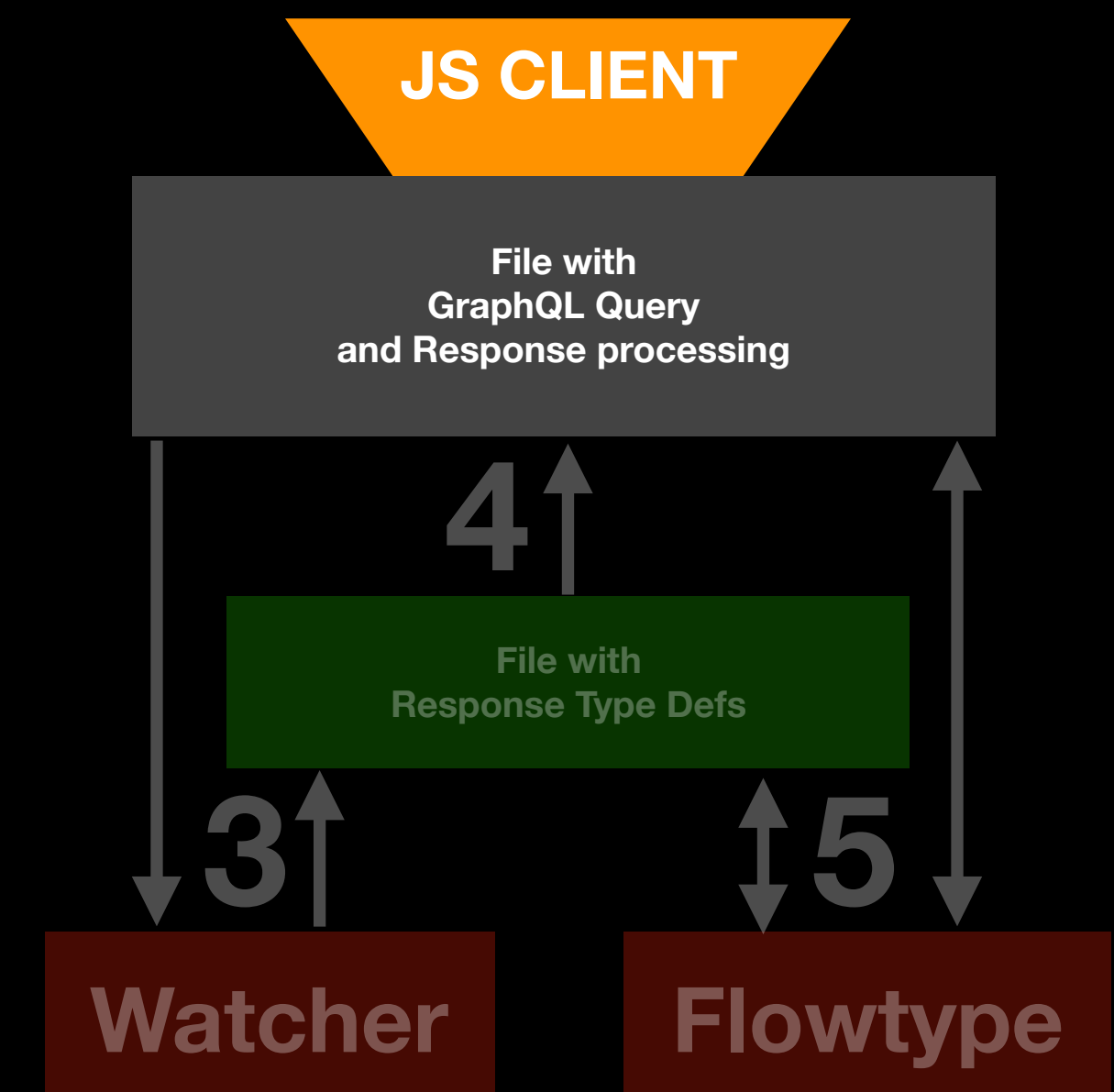
Writer time: 0.53s [0.37s compiling, ...]  
Created:  
- BalanceQuery.graphql.js  
Unchanged: 291 files  
Written default in 0.61s

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

## Crappy Code

```
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 } }
}`;
```



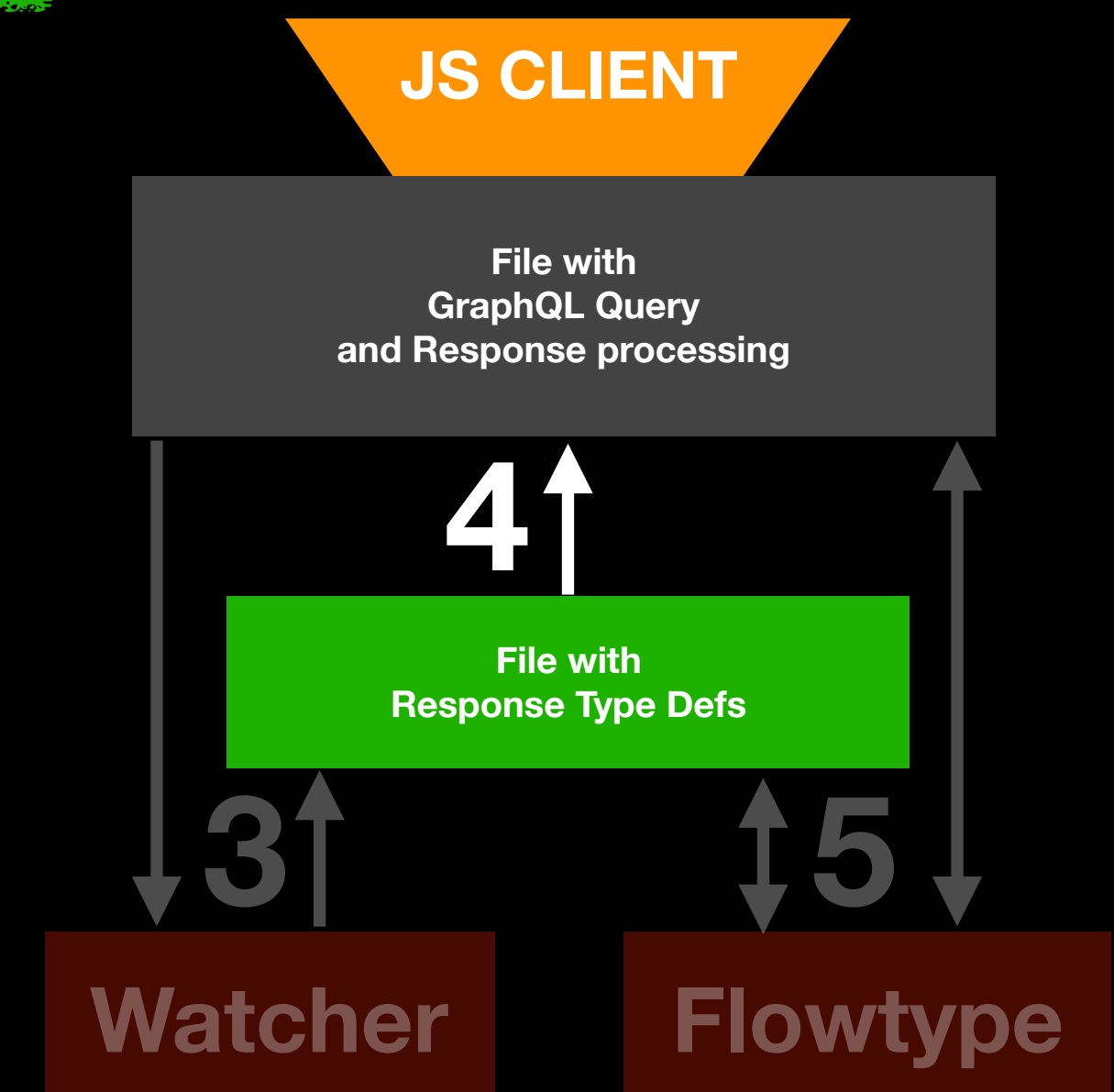
## Flow typed Code

```
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 } }
}`;
```



## Flow typed Code

```
/* @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>;
```

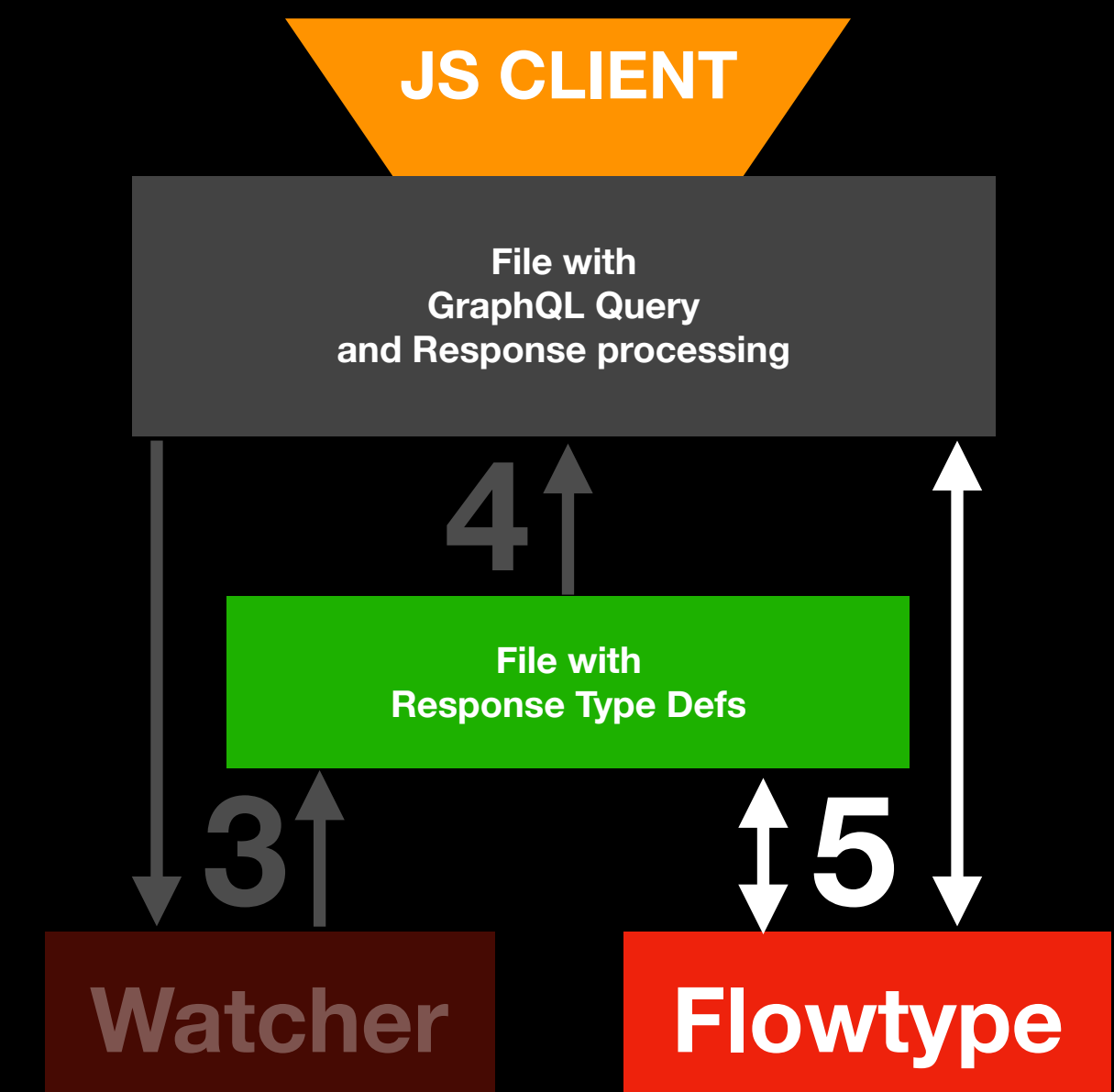
```
  }
```

```
}
```

```
const query = graphql`query BalanceQuery {
```

```
  viewer { cabinet { accountBalance } }
```

```
`};
```



# Flow 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

```
11:      return <div>Your balance: {viewer.cabinet.accountBalance}</div>;
                                     ^^^^^^^ property `cabinet`.
```

Property cannot be accessed on possibly undefined value

```
11:      return <div>Your balance: {viewer.cabinet.accountBalance}</div>;
                                     ^^^^^^^ undefined
```



# 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      persons: {
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 export default QueryType;
```

90 LoC

was reduced in 3 times

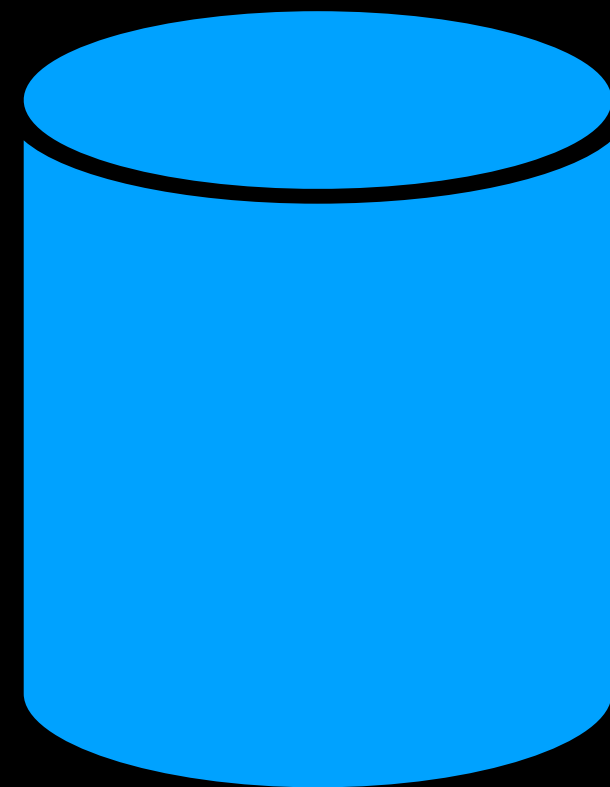
```
1  /* eslint-disable */
2
3  function createListResolve(url) {
4    return async (source, args) => {
5      const data = await loadData(url);
6      if (args && args.limit > 0) {
7        return data.slice(0, args.limit);
8      }
9      return data;
10   };
11 }
12
13 function createFieldConfigForList(type,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

30 LoC

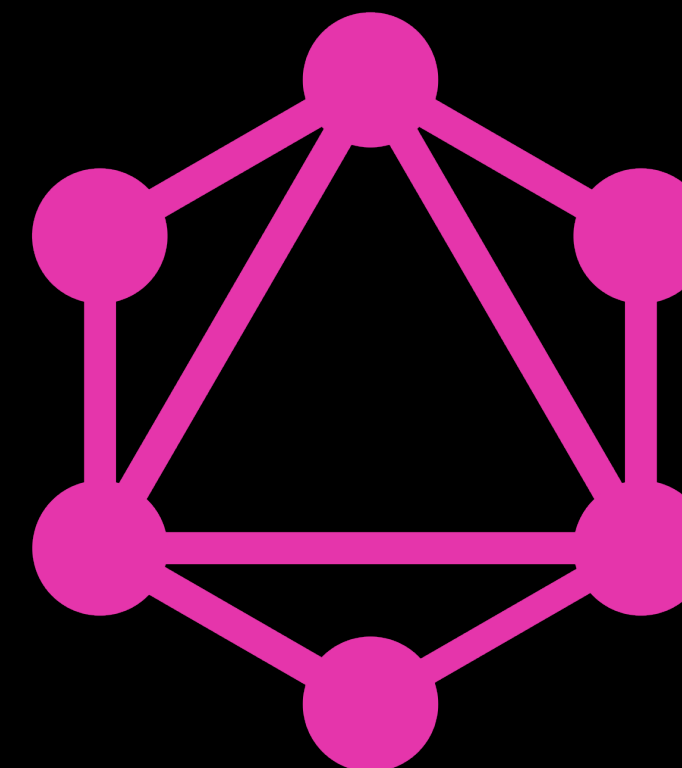
DRY principle (don't repeat yourself)

# How to keep to Schemas in SYNC?

Model: User



Type: User

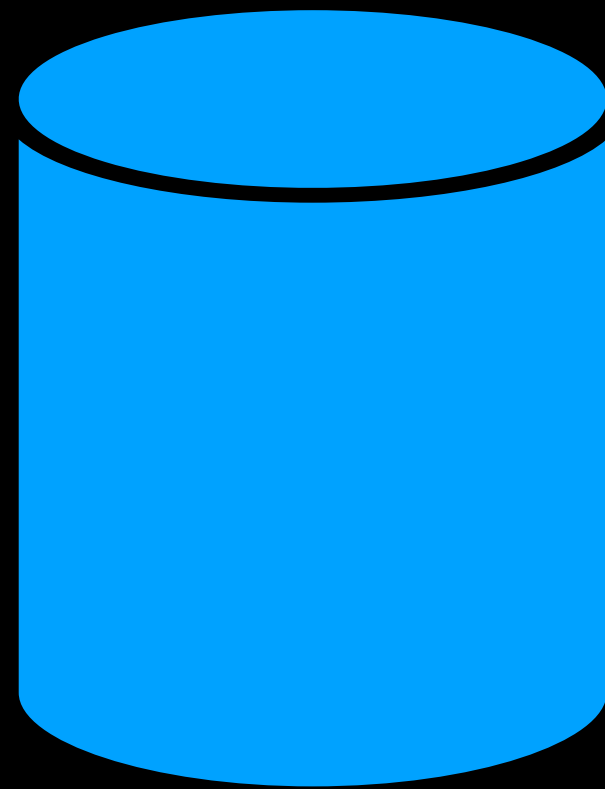


With time you may:

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

# Solution: generate GraphQL types from ORM models

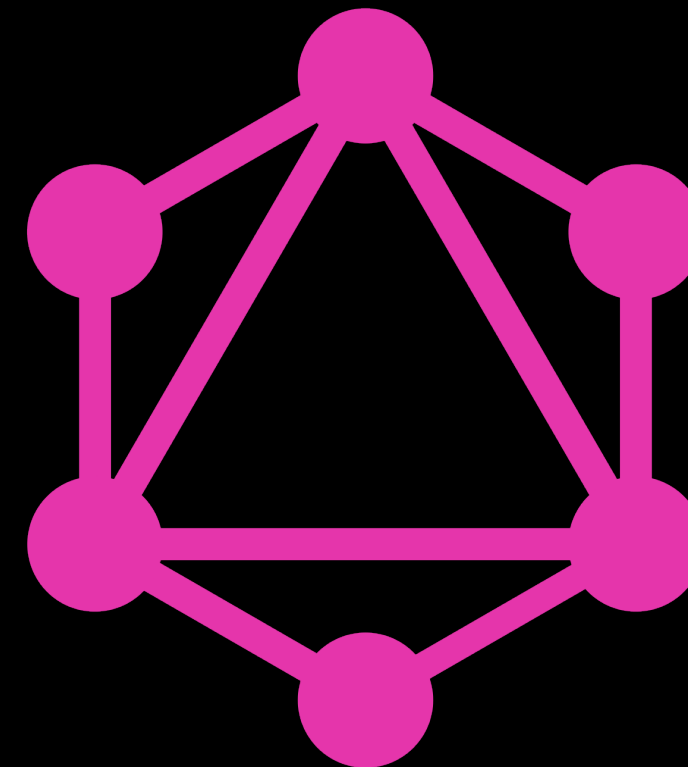
Model: User



GENERATE



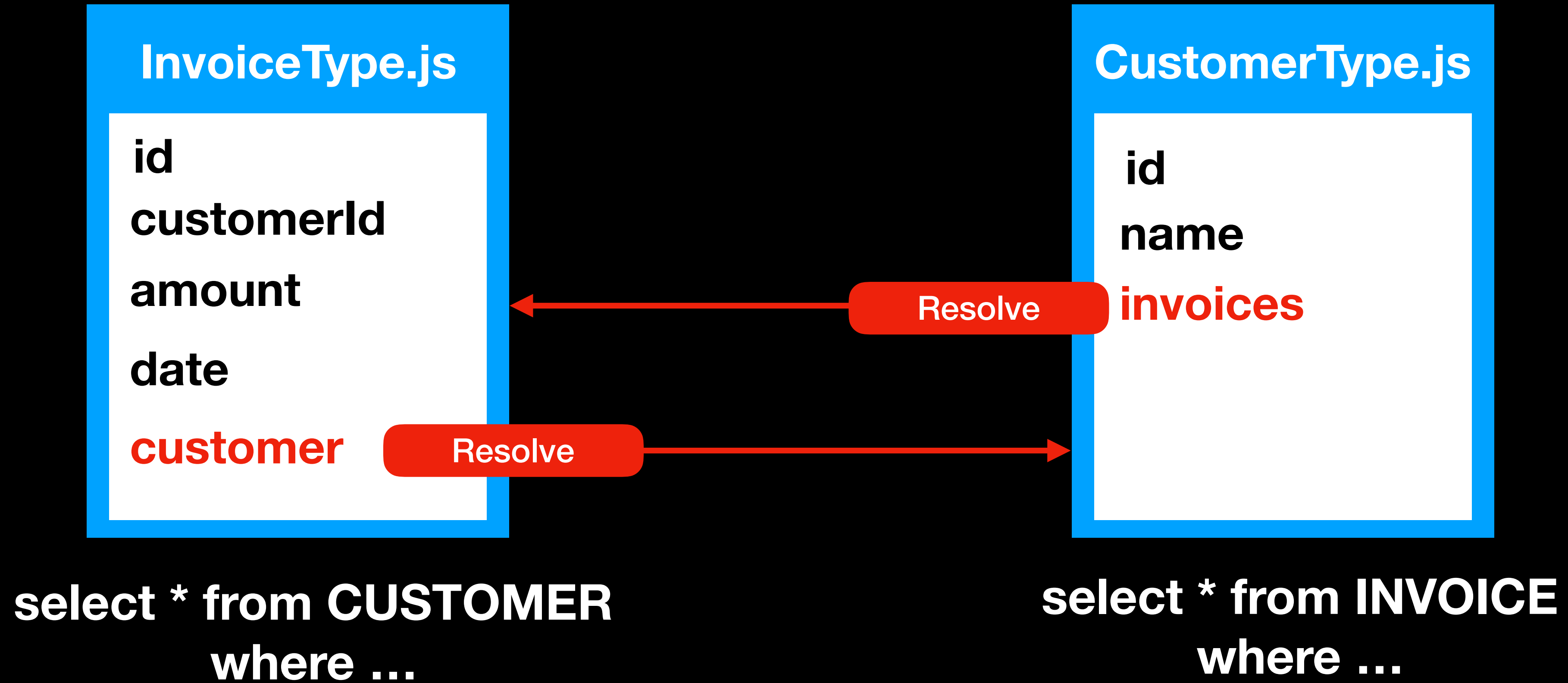
Type: User



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

**SSOT** principle  
(single source of truth)

Problem #3: mess in types and resolve functions



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

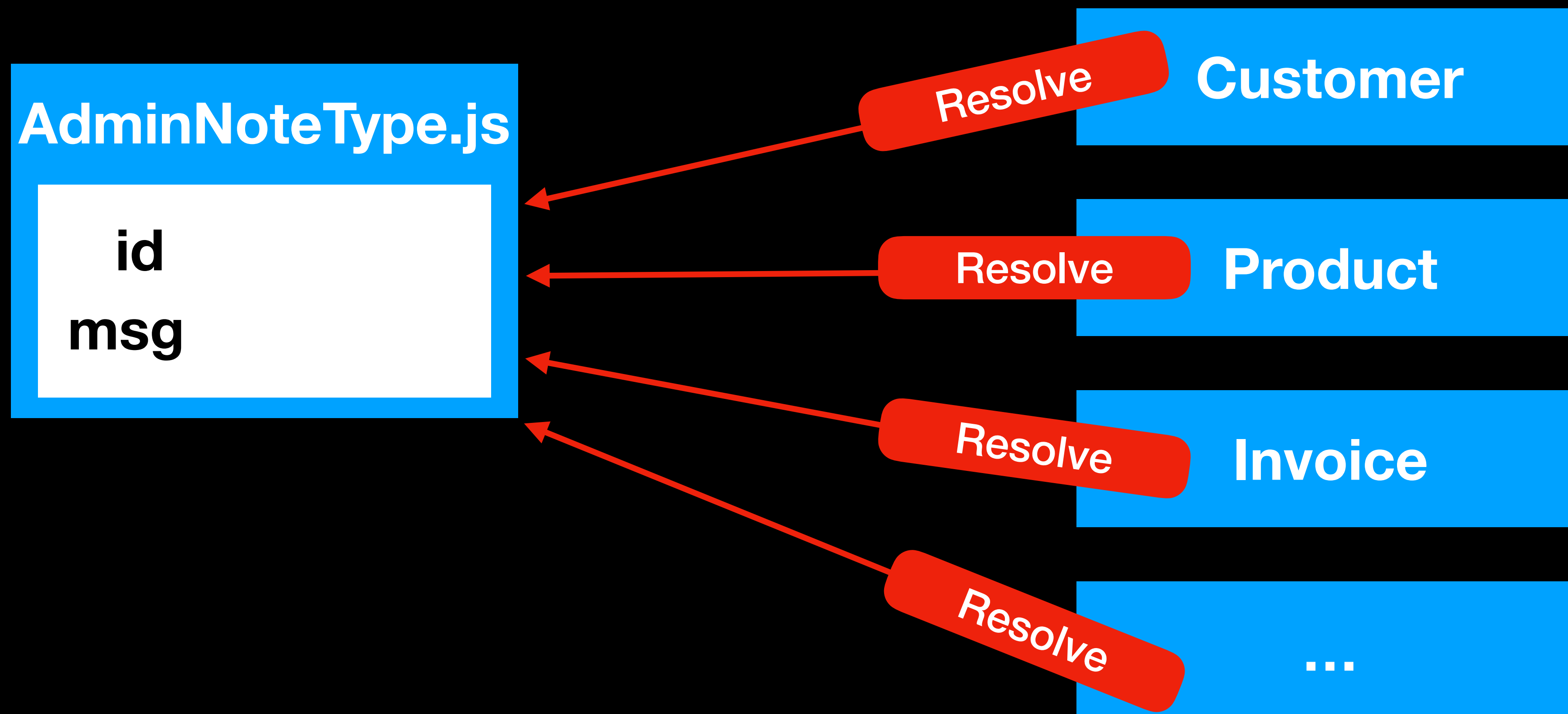


## CustomerType.js

```
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**



**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

Contains:

```
class CustomerGraphQLModel {  
  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

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 Query {
    customer(id: Int!): Customer
    invoices(limit: Int): [Invoice]
  }

  type Customer {
    id: Int!
    firstName: String
    invoices: [Invoice]
  }
`;
```

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

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

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

BUT...



## Hard to work with complex input args

All highlighted parts with red lines should be in sync

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

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

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

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

```
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/>

```
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 }) => { ... },  
}
```

```
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**

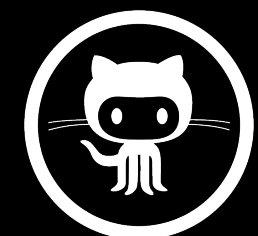
**GraphQL-compose  
packages**

# GraphQL-compose-\*

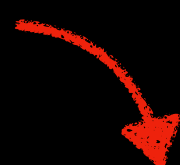
- OSS packages family  
for generating GraphQL Types

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

MIT License



Help wanted



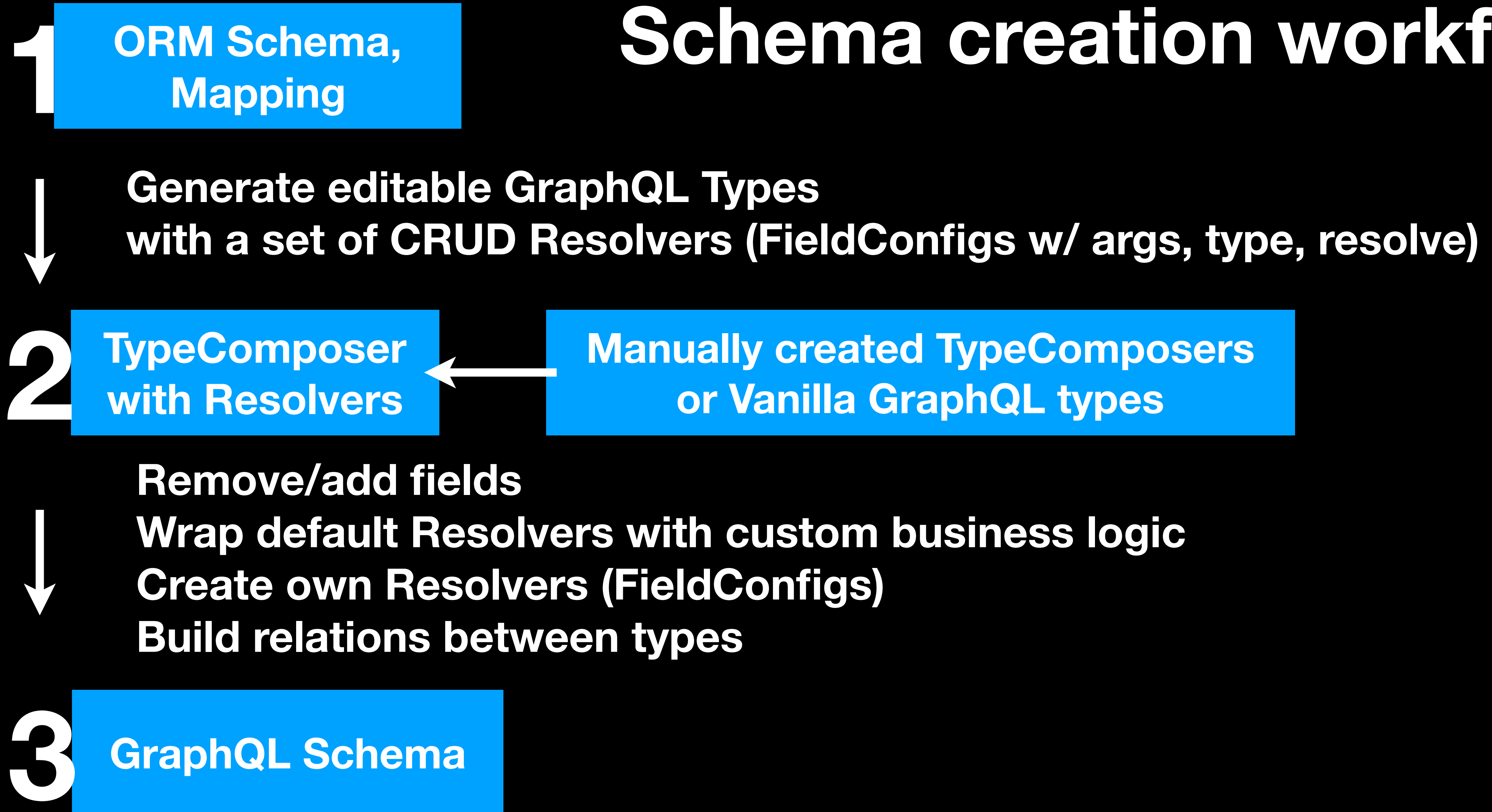
Exposes Flowtype/Typescript declarations

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



# 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

Type as function  
solves hoisting problems

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



# GraphQL-compose create relations between Types via Resolvers

```
InvoiceTC.addRelation('items', {  
  resolver: () => ItemsTC.getResolver('findMany'),  
  prepareArgs: {  
    filter: source => ({ invoicelId: 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](https://github.com/graphql-compose/graphql-compose-aws) 😊



3



116



245



# AWS Cloud API in GraphQL

Schema generation takes about ~1-2 seconds

125 Services

3857 Operations

6711 Input/Output params

rekognition(...): <b>AwsRekognition</b>	▶			
resourcegroups(...): <b>AwsResourceGroups</b>	▶			
resourcegroupstaggingapi(...): <b>AwsResourceGroupsTaggingAPI</b>	▶			
route53(...): <b>AwsRoute53</b>	▶			
route53domains(...): <b>AwsRoute53Domains</b>	▶			
s3(...): <b>AwsS3</b>	▶	createBucket(...): <b>AwsS3CreateBucketOutput</b>	<div>createBucket(   input: <b>AwsS3CreateBucketInput!</b>,   config: <b>AwsConfig</b> ): <b>AwsS3CreateBucketOutput</b></div> <div>TYPE DETAILS</div> <div>type <b>AwsS3CreateBucketOutput</b> {   Location: <b>String</b> }</div> <div>ARGUMENTS</div> <div>input: <b>AwsS3CreateBucketInput!</b> ▶ config: <b>AwsConfig</b> ▶</div>	<div>input: <b>AwsS3CreateBucketInput!</b></div> <div>TYPE DETAILS</div> <div>type <b>AwsS3CreateBucketInput</b> {   ACL: <b>String</b> ▶   Bucket: <b>String!</b> ▶   CreateBucketConfiguration: <b>AwsS3CreateBucketCreateBucketConfig</b> ▶   GrantFullControl: <b>String</b> ▶   GrantRead: <b>String</b> ▶   GrantReadACP: <b>String</b> ▶   GrantWrite: <b>String</b> ▶   GrantWriteACP: <b>String</b> ▶ }</div>
ses(...): <b>AwsSES</b>	▶			
sms(...): <b>AwsSMS</b>	▶			
sns(...): <b>AwsSNS</b>	▶			
sqs(...): <b>AwsSQS</b>	▶			
ssm(...): <b>AwsSSM</b>	▶			
sts(...): <b>AwsSTS</b>	▶			
swf(...): <b>AwsSWF</b>	▶			

<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**

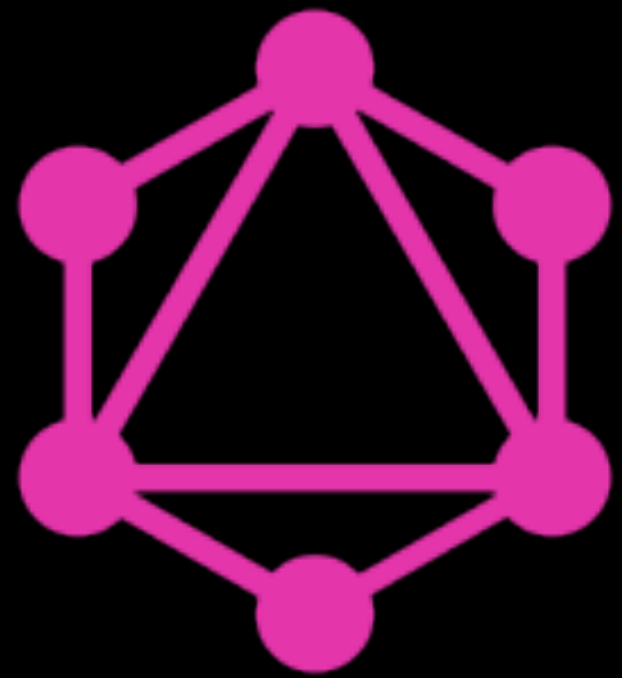
<https://github.com/lyskos97/graphql-compose-swapi>

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



# Last words...





# GraphQL is awesome!

less stress more success

less time on coding

less network traffic

less errors



PS. SOMETIMES A LOT LESS



**Read**

[medium graphql](#)

**Watch**

[youtube graphql](#)

**Glue**

[howtographql.com](#)

**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