

---

A REST alternative



By day

---



Software developer  
@ Ontario Institute for Cancer Research

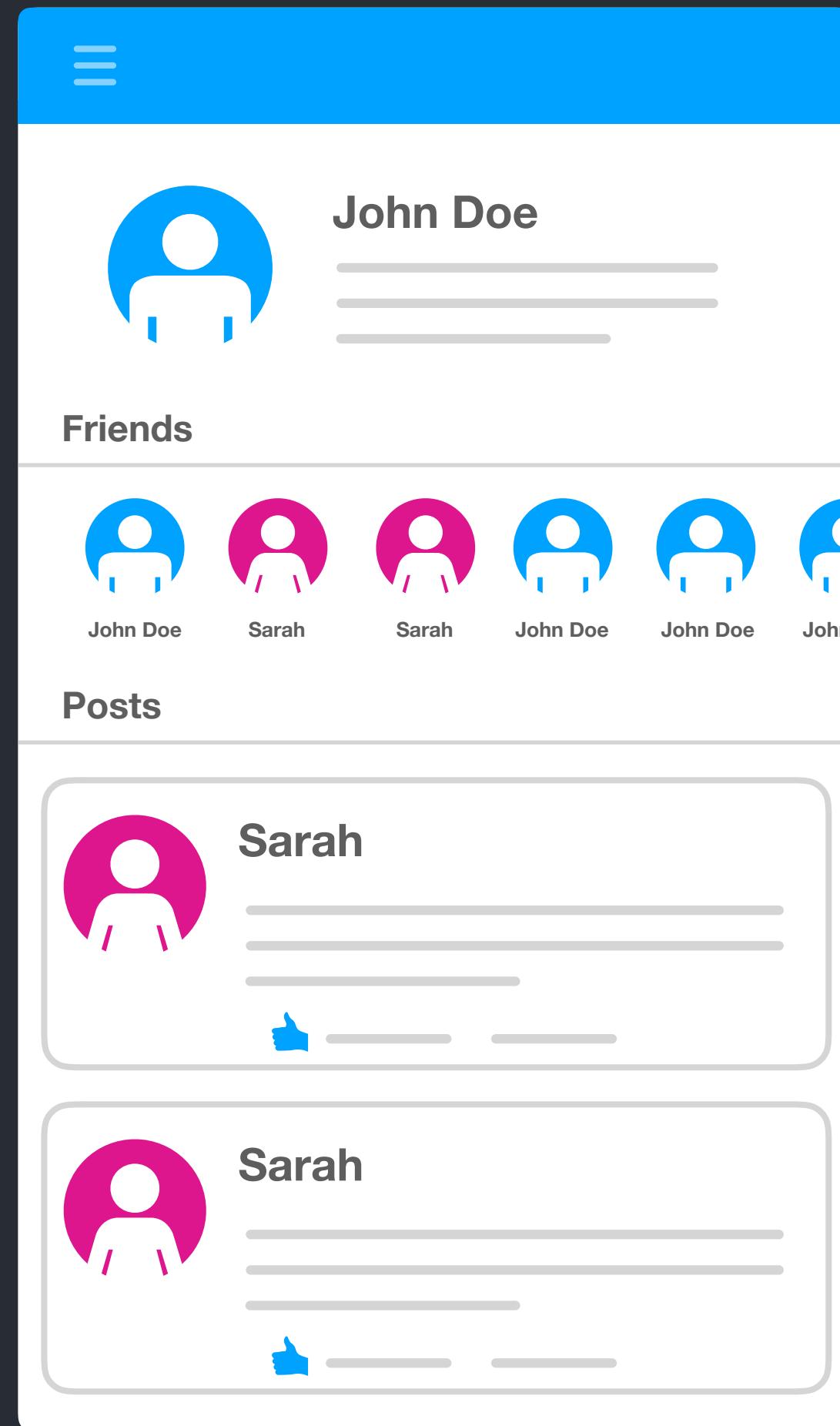
By night

---



Organizer & mentor  
@ freeCodeCamp Toronto study group

# Traditional REST API model



/user/1234

```
{  
  "id": 1234,  
  "name": "John Doe",  
  "email": "johndoe@email.com",  
  "age": 23,  
  "postIds": [23, 54, 12],  
  "friendsIds": [2341, 3453]}
```

/user/2341

```
{  
  "id": 2341,  
  "name": "Justin Timberlake",  
  "email": "justin@email.com",  
  "age": 23,  
  "postIds": [45, 76, 32],  
  "friendsIds": [4565, 6756]}
```

/user/2345

```
{  
  "id": 2345,  
  "name": "Justin Richardson",  
  "email": "justinrichardson@email.com",  
  "age": 9000,  
  "postIds": [56, 233, 756],  
  "friendsIds": [2234, 678]}
```

/post/23

```
{  
  "text": "I'm feeling happy",  
  "likes": [ {"user": "Steve"}, {"user": "Someone"} ],  
  "comments": [  
    { "author": "Steve", "text": "Me too" },  
    { "author": "John", "text": "What's up" }  
  ]},  
  ...  
  ...
```

- takes at least 3 network requests

- under & over fetching

- how do you know what endpoint to hit?

- how do you know what data will be received?

GET /posts/?user\_id=1234

GET /user/1234/friends

POST /postsByUser

```
{  
  userId: 1234  
}
```

## Is this the best we can do?

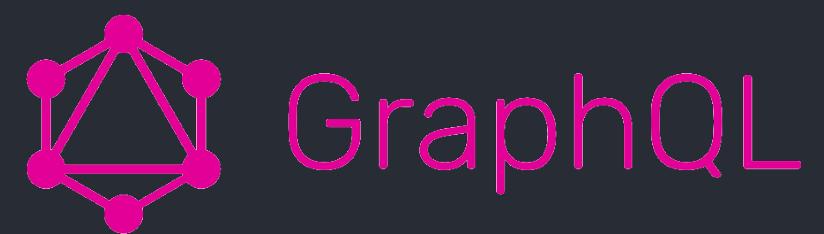
The problem over time:

- Too many ad-hoc endpoints
- Some end points grows too big
- WAIT, THAT'S NOT EVEN REST!

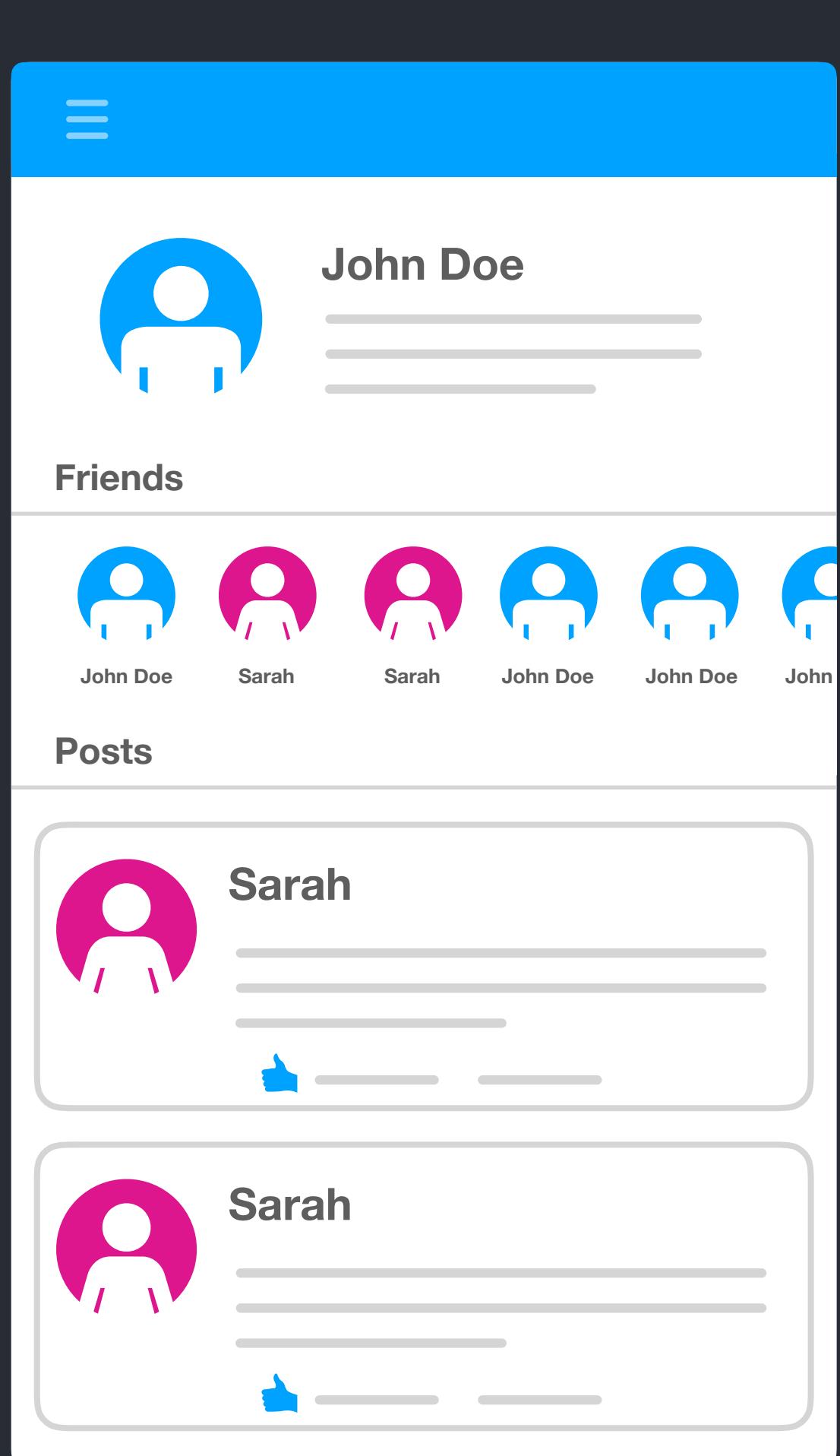
Then for writing data...

**ONE DOES NOT SIMPLY**

**KNOW WHETHER TO PUT OR POST**



# GraphQL



/graphql

```
Query {  
  user(id: 1234) {  
    id  
    name  
    description  
    friends {  
      name  
    }  
    posts {  
      text  
      likes  
    }  
  }  
}
```

```
{  
  "id": 1234,  
  "name": "John Doe",  
  "description": "some guy who likes to code",  
  "friends": [  
    { "name": "Justin Timberlake"},  
    { "name": "Justin Richardson"},  
    ...  
  ],  
  "posts": [  
    {  
      "text": "I'm feeling happy",  
      "likes": [{"user": "Steve"}, {"user": "Someone"}]  
    },  
    ...  
  ],  
}
```

- Single end point to serve all data needs
- Provides a typed schema for your API
- HTTP is only the transport mechanism

# Some facts

---

Developer



Facebook  
Open Source

Initial release

July 2015

Latest release

June 2018 (first “non Draft RFC”)

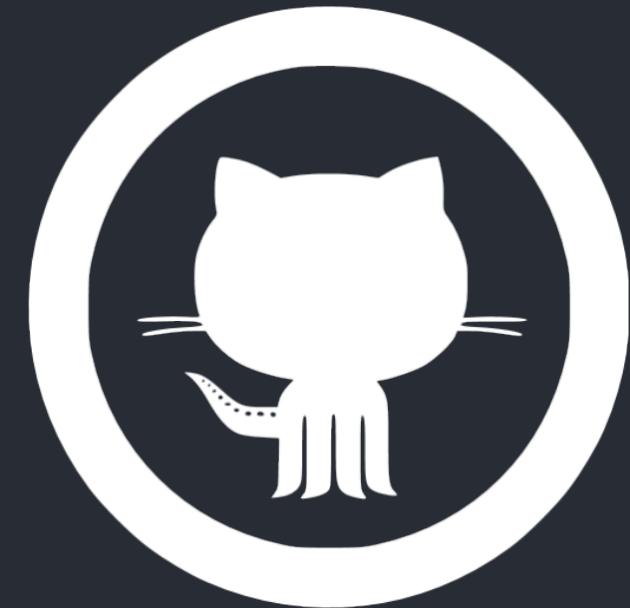
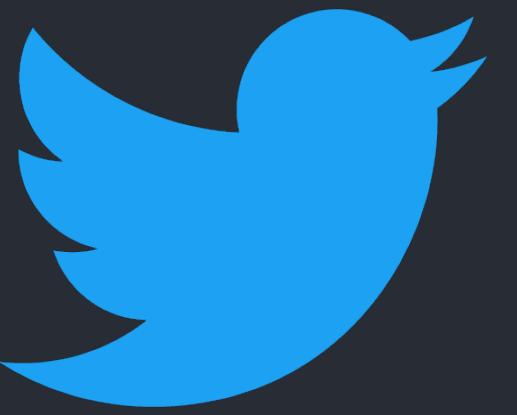
Languages with  
implementation  
libraries



and more...

# Who uses GraphQL?

---



intuit®



dailymotion

yelp



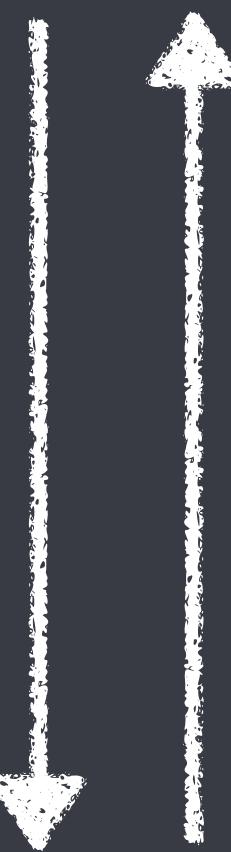
# A UI client for GraphQL APIs



## MongoDB Compass

A screenshot of the MongoDB Compass interface. It shows a list of documents in the 'cluster0-shard-00-00-00000.contacts' collection. The interface includes tabs for 'SCHEMA', 'DOCUMENTS', 'EXPLAIN PLAN', and 'INDEXES'. A search bar at the top has the filter 'name: "example"' applied. Below the search bar, there's a list of 1000 documents displayed in a grid.

BSON  
queries



`mongodb://user@pass:host:27017/db`



mongoDB®



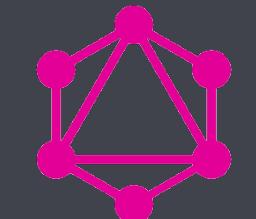
## MySQL Workbench

A screenshot of the MySQL Workbench interface. It shows a query editor with several lines of SQL code. The queries are related to a 'flights' table and involve joins with other tables like 'airports' and 'airlines'. The results of the queries are shown in a table below the code.

SQL  
queries



`jdbc:mysql://hostname:3306/db`



## GraphiQL

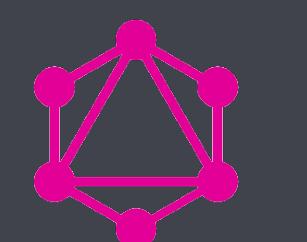
A screenshot of the GraphiQL interface. It features a 'Schema' browser on the left with definitions for 'product', 'brand', 'Product', 'Manufacturer', 'Customer', and 'Customer'. On the right, there's a 'Query' editor with a JSON-based query and its results.

GraphQL  
Queries



`http://myApi.com/graphql`

API



GraphQL

# Front-end: Performing Query

```
Query {  
  user(id: 1234) {  
    id  
    name  
    friends {  
      id  
      name  
      favoriteColor  
    }  
  }  
}
```



```
{  
  "data": {  
    "user": {  
      "id": 1234,  
      "name": "John Doe",  
      "friends": [  
        {  
          "id": 2341,  
          "name": "Justin Timberlake",  
          "age": 23  
        },  
        {  
          "id": 2345,  
          "name": "Justin Richardson",  
          "age": 9001  
        }  
      ]  
    }  
  }  
}
```

# Front-end: Performing Mutation

```
Mutation {  
  newUser(userData: {  
    name: "Andrew",  
    age: 10  
  }) {  
    id  
    name  
    age  
    friends {  
      name  
    }  
  }  
}
```



```
{  
  "data": {  
    "newUser": {  
      "id": 2342,  
      "name": "Andrew",  
      "age": 10,  
      "friends": []  
    }  
  }  
}
```

# Front-end: Using variables

query

```
Mutation($myNewUser: UserInput) {  
  newUser(userData: $myNewUser) {  
    id  
    name  
    age  
    friends {  
      name  
    }  
  }  
}
```

variables

```
{  
  "myNewUser": {  
    "name": "Andrew",  
    "age": 10  
  }  
}
```



Can be factored out of GraphQL  
and passed in from your  
Javascript

# Front-end: The resulting http request

```
Mutation($myNewUser: UserInput) {  
  newUser(userData: $myNewUser) {  
    id  
    name  
    age  
    friends {  
      name  
    }  
  }  
}  
  
{  
  "myNewUser": {  
    "name": "Andrew",  
    "age": 10  
  }  
}
```

```
fetch("http://myApp.com/graphql", {  
  method: "POST",  
  headers: { "Content-Type": "application/json" },  
  body: JSON.stringify({  
    query: `  
      Mutation($myNewUser: UserInput) {  
        newUser(userData: $myNewUser) {  
          id  
          name  
          age  
          friends {  
            name  
          }  
        }  
      }  
    `,  
    variables: {  
      "myNewUser": {  
        "name": "Andrew",  
        "age": 10  
      }  
    }  
  })  
})
```

# In practice... example usage with a GraphQL client library

```
export default gql`  
  user(id: 1234) {  
    id  
    name  
    age  
    friends {  
      id  
      name  
      favoriteColor  
    }  
  }  
`))(({data: {user}}) => (  
  <div>  
    <h1>{user.name}</h1>  
    <h2>{user.age}</h2>  
    <li>  
      {user.friends.map(friend => (  
        <div key={friend.id}>  
          <h3>{friend.name}</h3>  
        </div>  
      ))}  
    </li>  
  </div>  
) );
```

```
export default graphql(gql`  
  Mutation($myNewUser: UserInput) {  
    newUser(userData: $myNewUser) {  
      id  
      name  
      age  
      friends {  
        name  
      }  
    }  
  }  
`)(({ mutate }) => (  
  <button  
    onClick={() => mutate({  
      variables: {  
        "myNewUser": {  
          "name": "Andrew",  
          "age": 10  
        }  
      }  
    })}  
    > New User  
  </button>  
) );
```

- Directly bind UI data and actions to server model
- Reduces need for client-side state management and “data plumbing”

# Back-end: GraphQL schema

```
interface Entity {  
    id: ID!  
}  
  
type User implements Entity{  
    id: ID!  
    name: String!  
    favoriteColor: String!  
    friends: [User]  
}  
  
type FriendShip {  
    from: User  
    to: User  
    createdTime: String!  
}  
  
input UserInput {  
    name: String!  
    favoriteColor: String!  
}
```

```
type Query {  
    user(id: ID!): User  
}  
  
type Mutation {  
    newUser(userData: UserInput): User  
    newFriendship(from: ID!, to: ID!): FriendShip  
}
```

Built-in scalar types:

**Int**: A signed 32-bit integer

**Float**: A signed floating-point value

**String**: A UTF-8 character sequence

**Boolean**: true or false

**ID**: signifies a unique identifier,  
meant to be non-human readable

**interface**: abstract types that include a list of fields for other types to implement

**type**: represents a type of object you can **READ** from the API

**input**: represents an object type you can **SEND** to the API

Note: notice the difference between the User type and the UserInput type

**Query**: a special type every GraphQL schema has that defines all the entry points

**Mutation**: a special type that defines all the write operations that can be performed  
(in the REST world, that's all of POST, PUT, PATCH and DELETE)

Other features: **enum**, **union** types, custom **scalar** types, **directives**

# Back-end: Resolvers

## Schema

```
type User implements Entity {  
  id: ID!  
  name: String!  
  age: Number!  
  friends: [User]  
}  
  
type Query {  
  user(id: ID!): User  
}
```

## Resolver

```
const resolveUserId = (obj, args, context, info) => {  
  return getUserId(args.id).then(user => {  
    id: user.id,  
    name: user.name,  
    age: user.age,  
    friends: (obj, _, context, info) =>  
      user.friendIds.map(id => resolveUserId(obj, { id }, context, info))  
  }));  
};  
  
const resolvers = {  
  Query: {  
    user: resolveUserId  
  }  
};
```

# Back-end: Resolver function signature

```
const resolveUserId = (obj, args, context, info) => {
  return getUserId(args.id).then(user => ({
    id: user.id,
    name: user.name,
    age: user.age,
    friends: (obj, _, context, info) =>
      user.friendIds.map(id => resolveUserId(obj, { id }, context, info))
  }));
};

const resolvers = {
  Query: {
    user: resolveUserId
  }
};
```

## Arguments:

**obj**: object containing the result returned from the resolver on the parent field

**args**: object containing the arguments passed in from the query

**context**: object shared by all resolvers in a particular query, used for holding per-request states

**info**: contains information about the execution state of the query (field name, path from root, etc...)  
This is only used in very advanced cases

## Return types:

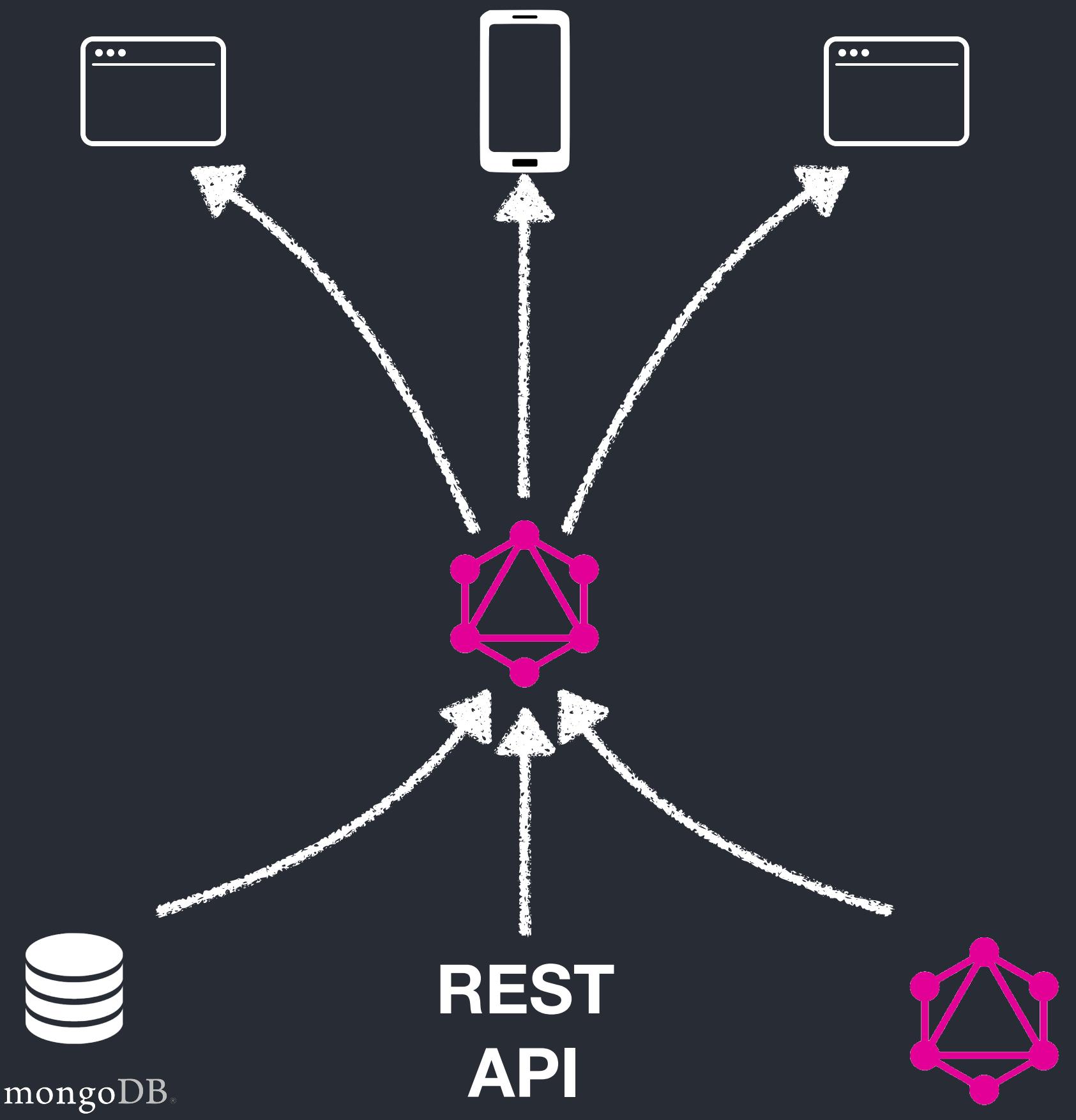
One of the following:

- **null** or **undefined**, indicating object could not be found
- An **array**, if the schema indicates a list type
- A **promise**. Can be combined with arrays (array of promises or promise that resolves to an array)
- A **scalar** or **object** value

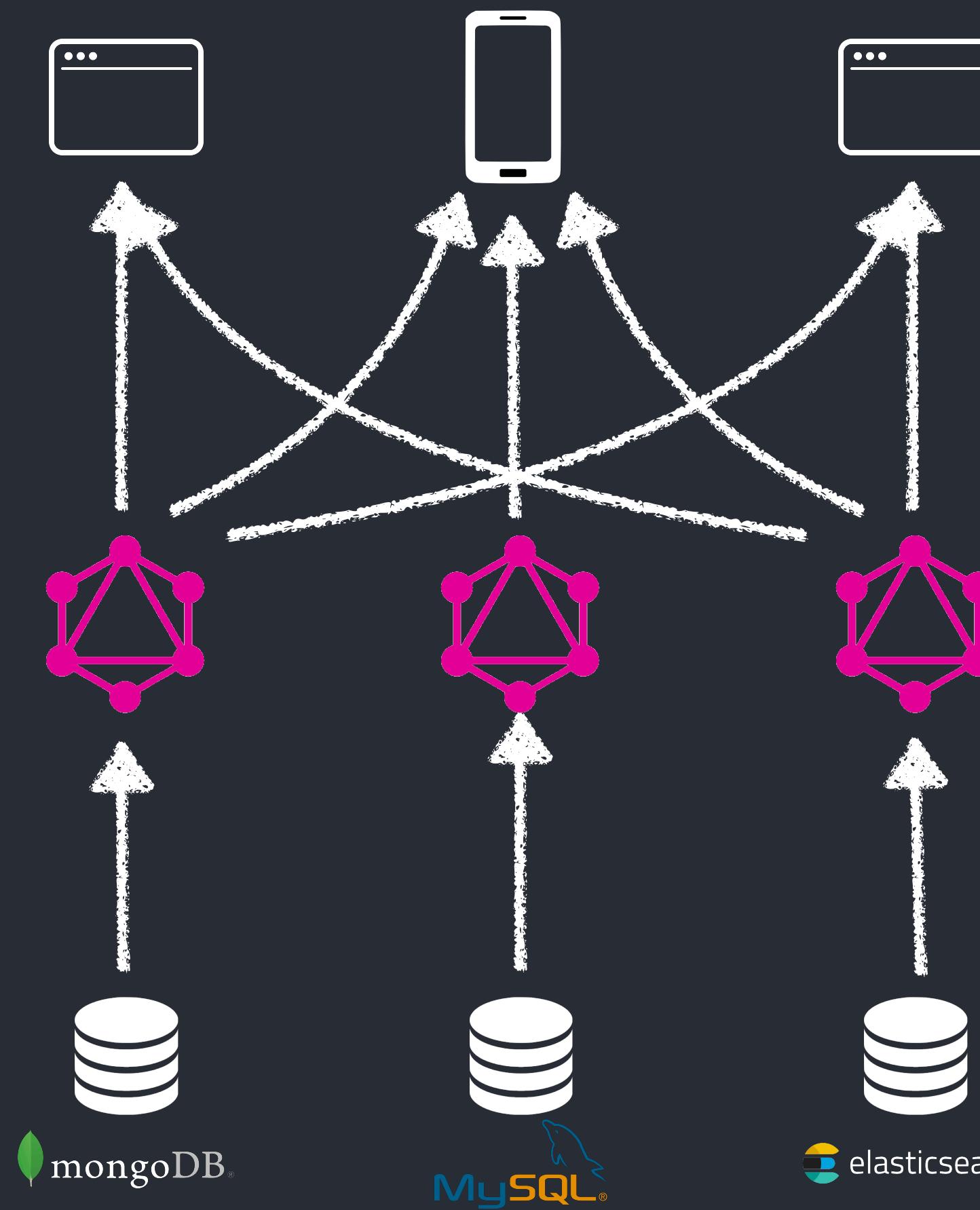
# DOs and DON'Ts



One endpoint to resolve from multiple resources



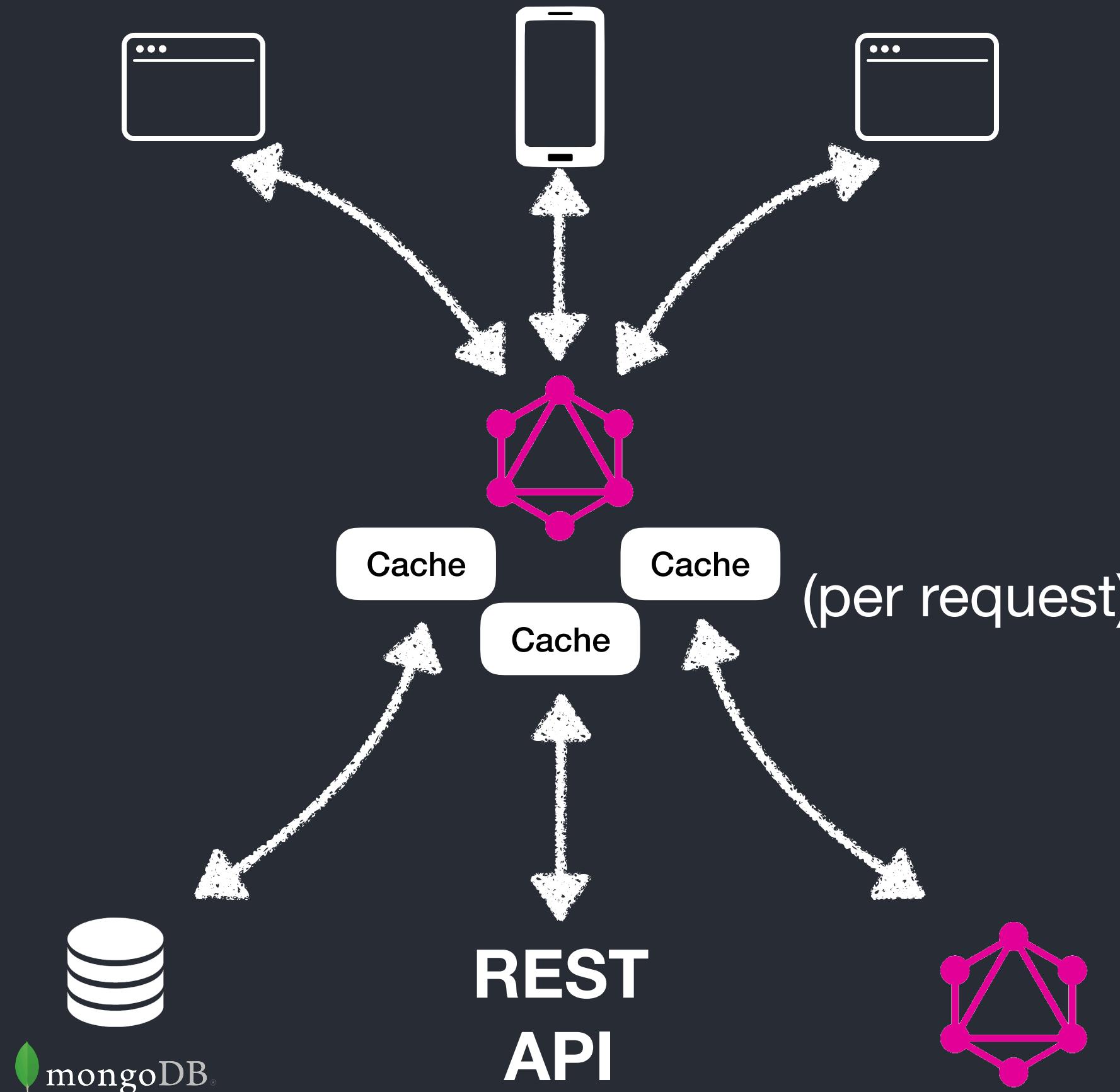
Multiple apps (or end points) to wrap each resource



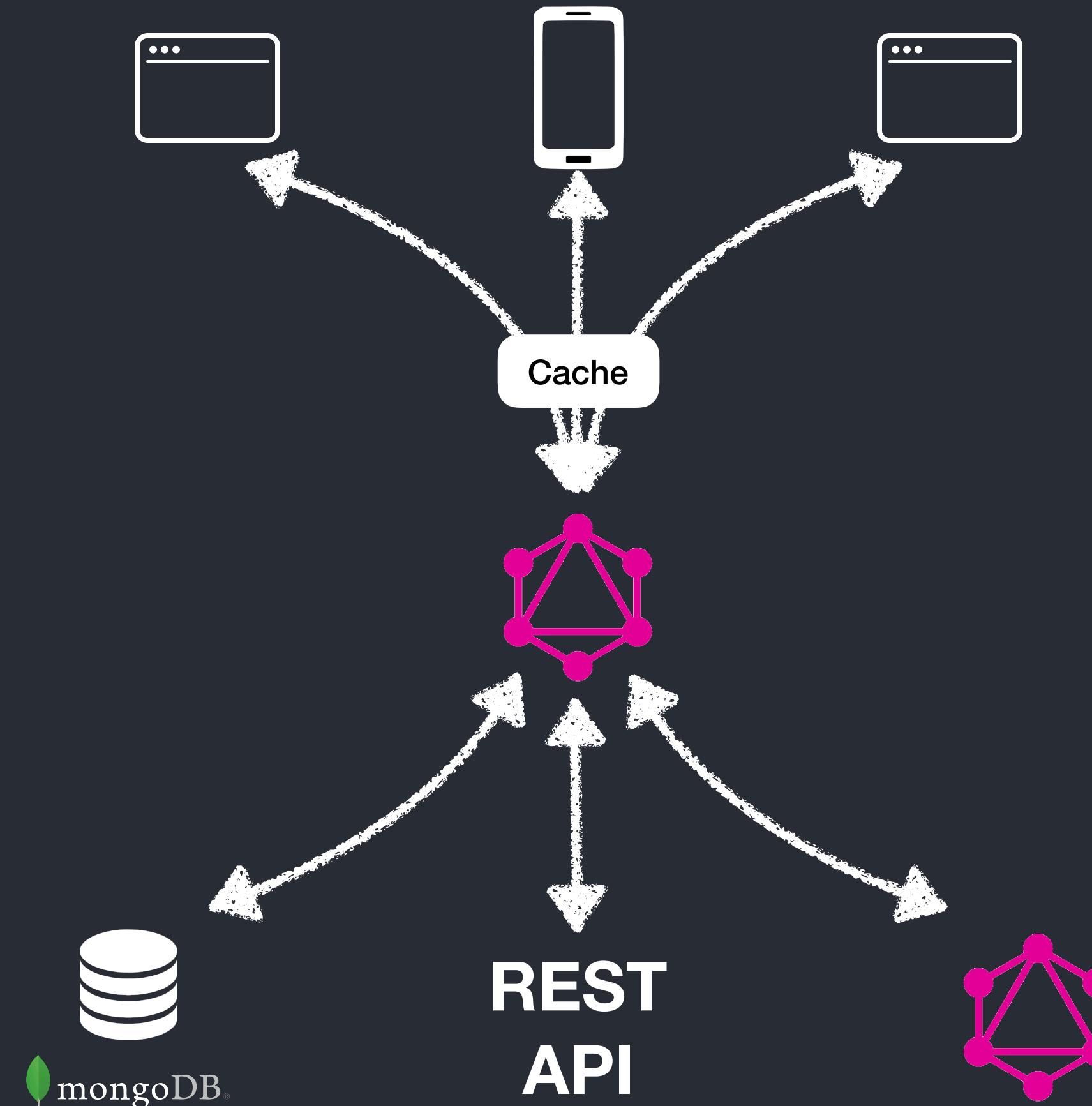
# DOs and DON'Ts



Per-request caching and batching in resolver to reduce unnecessary requests



Catch-all http cache will cause bad things to happen



# We just scratched the surface...



# Cool GraphQL resources

---

## Some cool GraphQL APIs

<https://github.com/APIs-guru/graphql-apis/blob/master/README.md>

## Apollo Launchpad - a “codepen” for prototyping Graphql APIs

<https://launchpad.graphql.com/new>

## How to GraphQL - The Full Stack tutorial for GraphQL

<https://www.howtographql.com/>