Building GraphQL server on *2018*

timqian

Me

- Nodejs backend at work
- Full stack at home
- Javascript is the best language ~~in the universe~~ to me

Target audiance

- Some experience on building/using REST API
- Little experience on building/using GraphQL

Table of contents

What is an APP from the perspective of data?

TJ Holowaychuk 🙃
@tjholowaychuk

正在关注

my js state management library: {}

翻译推文

下午9:51 - 2018年1月28日

349 转推 **1,634** 喜欢

48 349 1.6千

Frontend's Job:

1. Get the `{}` from backend
2. Render the page based on the `{}`
3. Update the `{}` based on user's input
4. Maybe store the update back to backend

Example

In a blog system, we want to render the user with his blogs.

How does frontend get the { } through REST API

```
GET /users/:id # get user info
GET /users/:id/blogs # get all blogs of user
```

```
// The object frontend used to render the page
{
  user: {
    id: 1,
    username: 'timqian',
    blogs: [{
      id: 1,
      title: 'hi world'
      content: 'hello world'
    }]
  }
}
```

How does frontend get the { } through GraphQL

```
# GraphQL query
query {
  user(id: 1) {
    username
    blogs {
      id
      title
      content
    }
  }
}
```

How does frontend get the { } through GraphQL

```
// returned object from GraphQL backend
{
  user: {
    username: 'timqian',
    blogs: [{
      id: 1,
      title: 'hi world'
      content: 'hello world'
    }]
  }
}
```

Better experience for receiving more complicated object

Want more info about the user and less about the blog?

```
query {
  user(id: 1) {
+   id
    username
    blogs {
-     id
      title
-     content
    }
  }
}
```

Want more info about the user and less about the blog?

```
// returned object from GraphQL backend
{
  user: {
    id: 1,
    username: 'timqian',
    blogs: [{
      title: 'hi world'
    }]
  }
}
```

So *What* is GraphQL

GraphQL is a query language for your API. Basically it is about selecting fields on objects

How to implement

How to implement (1): Define Schema

- GraphQL query language is about selecting fields on objects
- We will need a **Schema** to describe/define the data we can ask for
- **Schema**: a set of types which completely describe the set of possible data you can query on that service

How to implement (1): Define Schema

```
# 1. Define schema
type Query {
  user(id: ID!): User!
  blogs: [Blog]
}
type User {
  id: ID!
  username: String!
  blogs: [Blog]
}
type Blog {
  id: ID!
  title: String!
  content: String!
  createdBy: User!
}
schema {
  query: Query
```

How to implement (2): Write resolvers

- **Schema** describes all of the fields, arguments, and result types
- Now we need a collection of functions that are called to actually execute these fields, and this collection of functions are called **Resolvers**

How to implement (2): Write resolvers

```
// 2. Define resolvers as a nested object that
// maps type and field names to resolver functions
const resolver = {
  Query: {
    user: (obj, args) => daos.User.get(args.id),
    blogs: (obj, args) => daos.Blog.getAll(),
  },
  User: {
    blogs: (obj, args) => daos.Blog.getByUser(obj.id),
  },
  Blog: {
    createdBy: (obj, args) => daos.User.get(obj.createdBy)
  },
}
```

How to implement (3): Bind schema and resolver together

```
// 3. Bind schema and resolver together using
// graphql-yoga which is based on `graphql-tool`
import { GraphQLServer } from 'graphql-yoga';

const server = new GraphQLServer({ typeDefs, resolvers });

server.start(() =>
  console.log('Server is running on localhost:4000'));
```

Summary of implementing a GraphQL server

1. Write a **Schema** to define the data graph
2. Write **Resolvers** to resolve fileds of the defined data graph

Summary of implementing a GraphQL server

1. Write a **Schema** to define the data graph
2. Write **Resolvers** to resolve fileds of the defined data graph

So Easy!

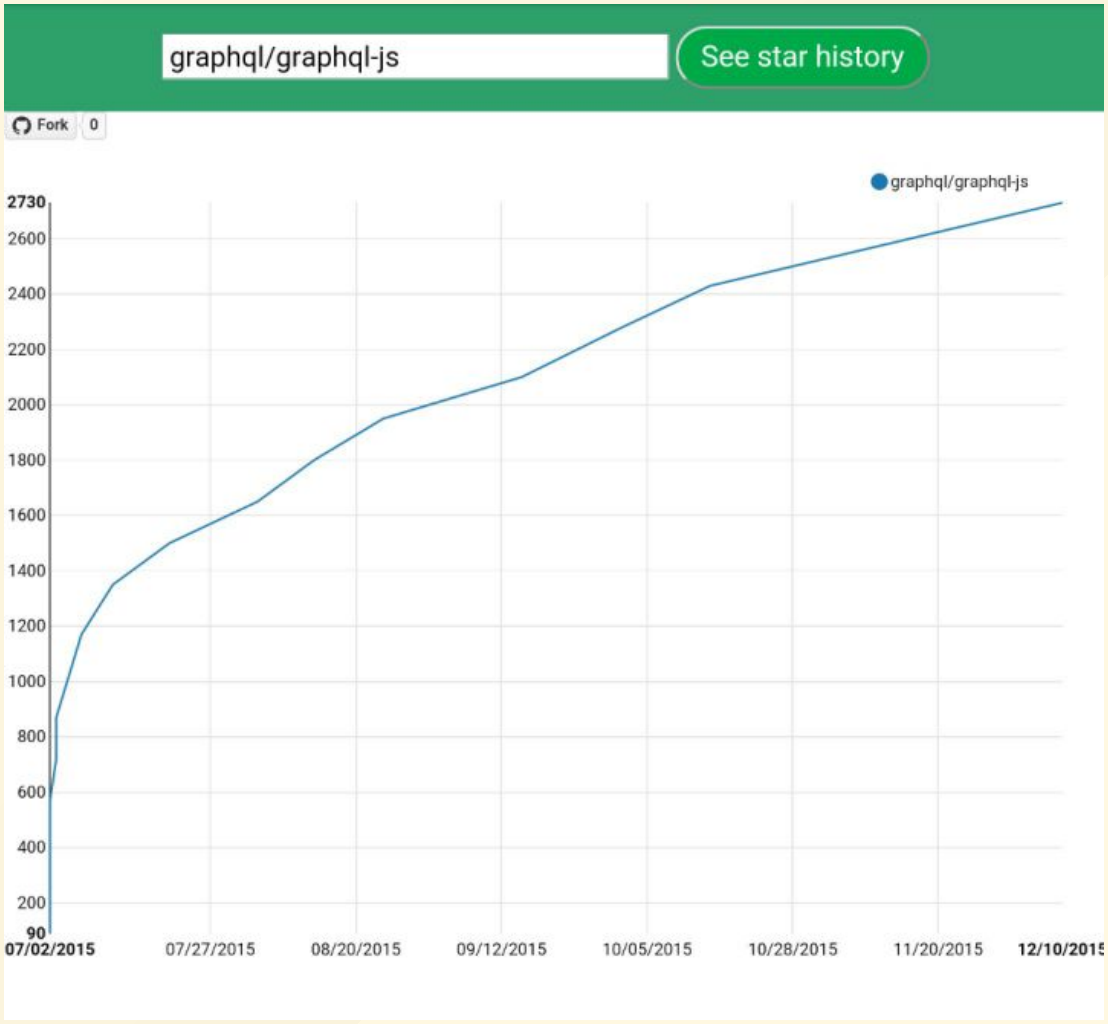
but it wasn't that easy before graphql-tool was invented

A bit history of GraphQL

How GraphQL server looks before graphql-tool existed

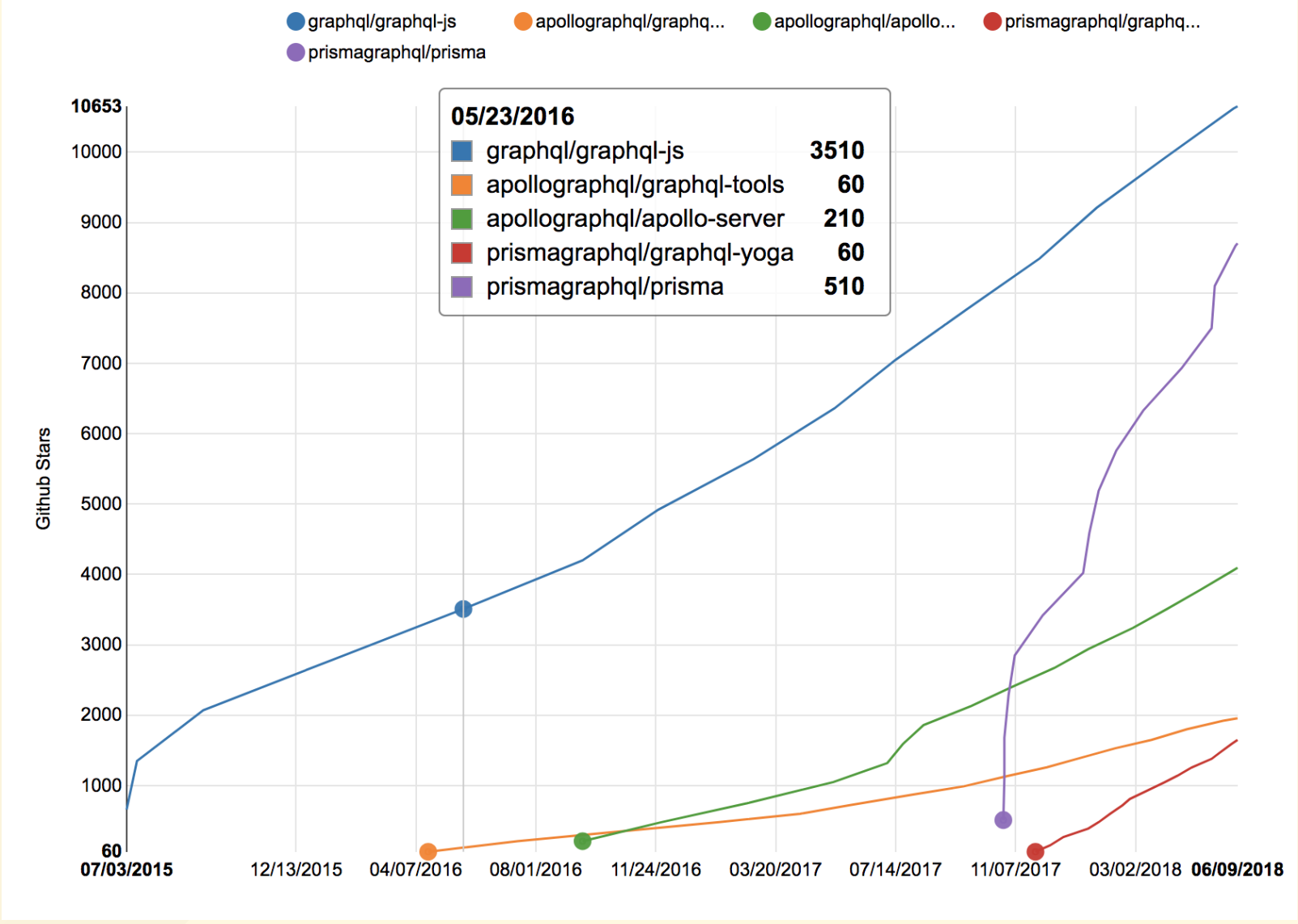
```
import {
  graphql,
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString
} from 'graphql';
var schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'RootQueryType',
    fields: {
      hello: {
        type: GraphQLString,
        resolve() {
          return 'world';
        }
      }
    }
  })
```

History of popularity: 2015 (bottleneck)



- timqian.com/star-history

History of popularity: 2016 -2018 (boom)



- timqian.com/star-history

Why choose GraphQL over REST

- Performance
  - Less roundtrips
- Development experance
  - Self documented (no outdated apidoc anymore!)
  - Less endpoints
  - Ask for what you want
  - Real-time data push (subscription)

Issues and Solutions

- N+1 problem
- Writing test
- Similar code for normal usage

Issue (1): N+1 problem

```
# Will do N + 1 database query if there is N blogs
query {
  blogs {
    id
    title
    createdBy {
      id
      name
    }
  }
}
```

Situation can be worse when the query becomes more complex.

Can we do the N user query together?

Solution: `Dataloader` (1)

```
const DataLoader = require('dataloader');

// Provid a batch loading function
const myBatchGetUsers = ids =>
  daos.User.whereIn('id', ids);

// Create your data loader
const userLoader =
  new DataLoader(myBatchGetUsers);
```

Solution: `Dataloader` (2)

Update resolver

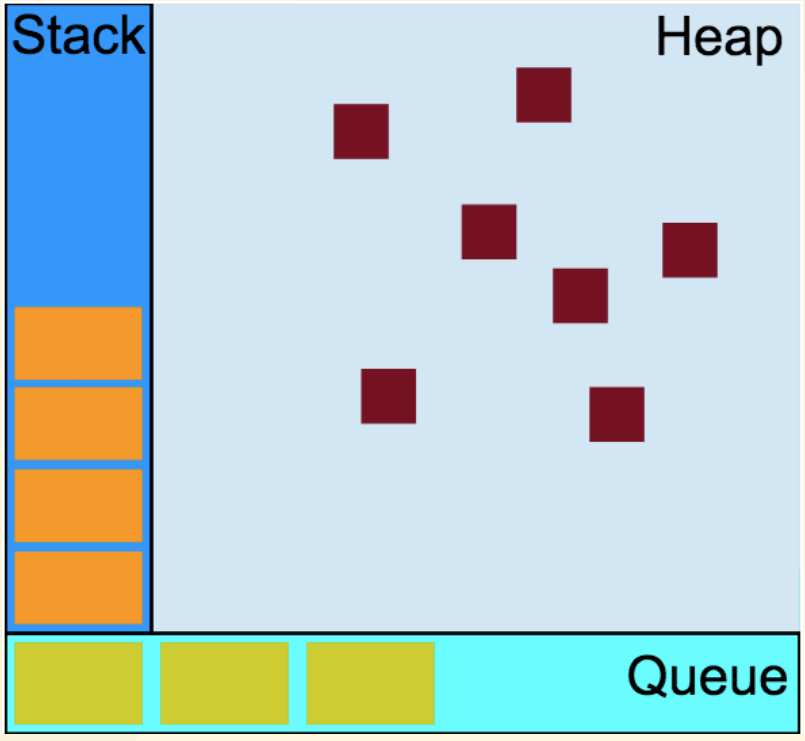
```
const resolver = {
  Query: {
    user: (obj, args) => daos.User.get(args.id),
  },
  User: {
    blogs: (obj, args) => daos.Blog.getByUser(obj.id),
  },
  Blog: {
-   createdBy: (obj, args) => daos.User.get(obj.createdBy),
+   createdBy: (obj, args) =>
+     userLoader.load(obj.createdBy),
  },
}
```

`Dataloader` Caching

```
load(key)

clear(key)

loadMany(keys)

clearAll()
```

How does Dataloader work

DataLoader will coalesce all individual loads which occur within a single frame of execution (a single tick of the event loop) and then call your batch function with all requested keys.



Application level dataloader?

The official Readme encourage user to create a new DataLoader per request. Because:

1. Many different users with different access permissions. It may be dangerous to use one cache across many users
2. In memary cache can not scale among servers

But they are actually solvable

Application level dataloader?

1. Use dataloader in the dao layer of your app and do ACL on resolver
2. Use redis/memcached as the cache of dataloader

Refs

- Disscussions on an issue of dataloader repo
- Use redis instead of memary as the cache

Issue (2): Writing test query(String) is error prone

Issue (2): Writing tests

```
# Sample schema
type Query {
    user(id: Int!): User!
}

type User {
    id: Int!
    username: String!
    email: String!
    createdAt: String!
}
```

Issue (2): Writing tests

```
# Sample query
query {
    user(id: 1) {
        id
        username
        email
        createdAt
    }
}
```

Issue (2): Writing tests

```
# Sample query
query user($id: Int!) {
    user(id: $id) {
        id
        username
        email
        createdAt
    }
}
```

gql-generator: generate sample queries for you based on the schema

Issue (3): 70% of the resolvers are doing similar things: query db by ID; query db by foreign key.

Issue (3): 70% of the resolvers are doing similar things: query db by ID; query db by foreign key.

Prisma: Automatically mapping your API to database:

Define your types and it will do the resolves for you.

```graphql
type User {
  id: ID! @unique
  createdAt: DateTime!
  name: String!
  admin: Boolean! @default(value: "true")
}
```

DATAMODEL IN GRAPHQL SDL

```
CREATE TABLE User(
    `id` CHAR(25),
    `createdAt` DATETIME NOT NULL DEFAULT CURRENT
    `updatedAt` DATETIME NOT NULL DEFAULT CURRENT
    `name` MEDIUMTEXT NOT NULL,
    `admin` BOOLEAN NOT NULL DEFAULT TRUE,
    PRIMARY KEY (`id`),
    UNIQUE INDEX `id_UNIQUE` (`id` ASC)
)
```

GENERATED SQL

RESOLVERS

With Prisma | Without Prisma

```
const Query = {
  userList: (_, args, context, info) => {
    return mysql.query(
      `SELECT
          "user"."id",
          "user"."name",
          "user"."isAdmin"
        FROM tblUsers as "user"`
    )
  }
}
```

SEND A QUERY

SCHEMA

```
type Query {
  userList: [User!]!
}

type User {
  id: ID!
  name: String!
  isAdmin: Boolean
}
```

RESOLVERS

With Prisma | Without Prisma

```
const Query = {
  userList: (_, args, context, info) => {
    return context.prisma.query.users({}, info)
  }
}
```

SEND A QUERY

SCHEMA

```
type Query {
  userList: [User!]!
}

type User {
  id: ID!
  name: String!
  isAdmin: Boolean
}
```

Summary

- Building GraphQL is easy and the tool chain is still evolving
- GraphQL benifits both frontend and backend
- Give it a try in your next awsome project

Questions?