GraphQL in Cisco's B2B

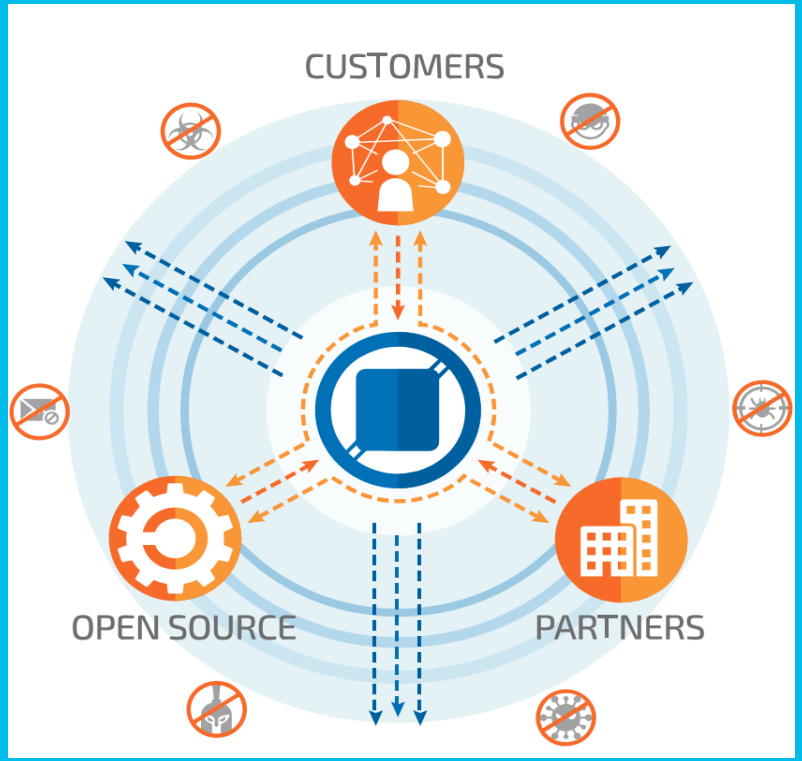
Jay Chow

Software Engineer, Cisco eCommerce, IT

Cisco, San Jose

20th March 2020

Agenda



1. What is GraphQL?
2. A more efficient alternative to REST
3. GraphQL is not only for React developers
4. GraphQL is better than REST
5. GraphQL provides insightful analytics
6. Schema Definition Language
7. Core concepts/ use cases
8. Benefits of GraphQL on the client side
9. References

What is GraphQL?

- New API standard that was invented and open-sourced by Facebook
- Enables declarative data fetching
- GraphQL server exposes single endpoint and responds to precise queries

A more efficient Alternative to REST

- REST has been a popular way to expose data from server
- But we need to consider 3 factors on how APIs are designed in the future:

1) Increased mobile usage creates need for efficient data loading
2) Variety of different frontend frameworks and platforms on the client, with GraphQL each client can access precisely the data it needs
3) Fast development speed and expectation for rapid feature development

Important: GraphQL is great for frontend developers as **data fetching complexity is pushed to the server-side**, opportunities for new abstractions on the frontend

GraphQL is NOT only for React developers

- Facebook uses GraphQL since 2012 in their native mobile apps
- GraphQL can be used with any programming language and many different backend and frontend frameworks
- Already used in production by companies such as Github, Yelp, Shopify, Twitter, Coursera
- Conferences: GraphQL-Europe, GraphQL Summit in San Francisco

GraphQL is better than REST

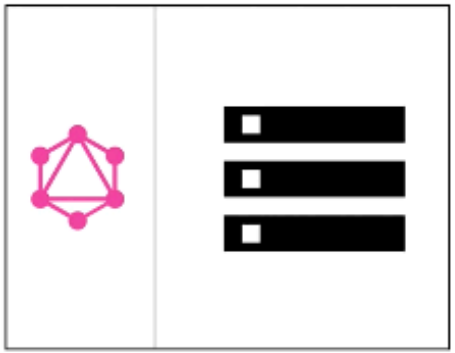
- REST: stateless servers and structured access to resources
- REST is a strict specification
- BUT changing requirements on client-side don't go well with static nature of REST
- GraphQL was developed to cope with need for more flexibility and efficiency in client-server communication
- 1 single API endpoint, only send 1 single POST request and server will respond all information that is needed
- Minimizes over-fetching and under-fetching of data from server back to client

Over-fetching: Downloading unnecessary data

Under-fetching: An endpoint doesn't return enough of the right information, need to send multiple requests

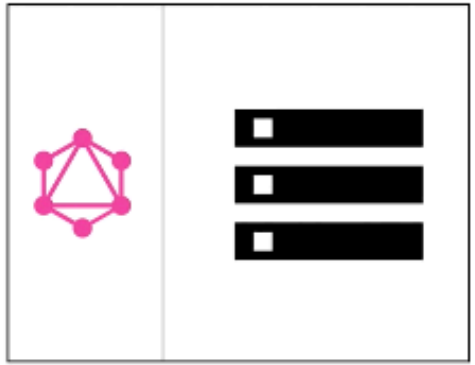
① Fetch everyting with a single request


```
query {
  User(id: “er3tg439frjw”) {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```



① Fetch everyting with a single request

HTTP POST



```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" },
        { title: "React & GraphQL - A declarative love story" }
        { title: "Why GraphQL is better than REST" }
        { title: "Relay vs Apollo - GraphQL Clients" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```

GraphQL provides insightful analytics

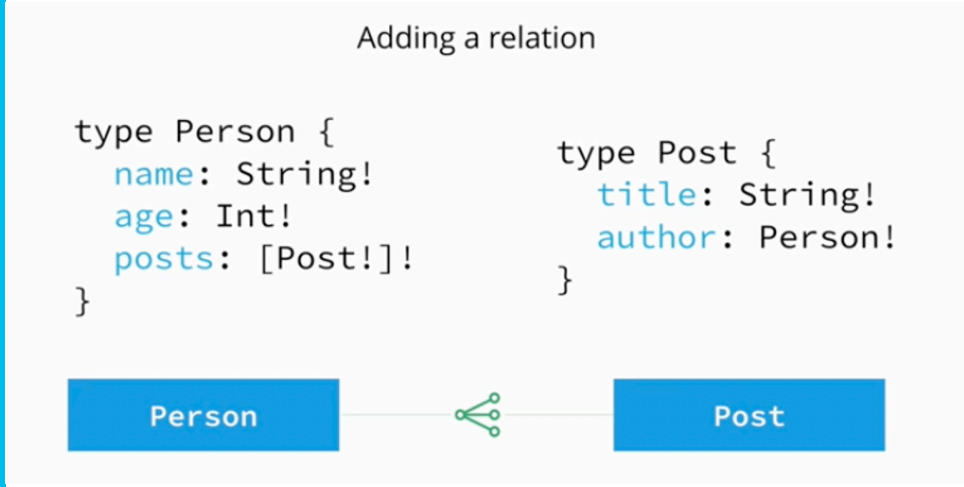
Fine-grained info about what data is read by clients

Enables evolving API and deprecating unneeded data

Client specifies exactly what information it is interested in

Strong type system to define capabilities of an API using a **schema**

Schema Definition Language



GraphQL Core Concepts

1) Fetching data with Queries

2) Writing Data with Mutations

3 kinds: creating new data, updating existing data and deleting existing data

3) Realtime updates with Subscriptions

When a client subscribes to an event(i.e when a new user gets created), it will initiate and hold a steady connection to the server, when that even happens, server will push corresponding data to the client

Note: Can think of subscription as a stream of data sent over to the client

The Query Type

```
{
  allPersons {
    name
  }
}
```

```
type Query {
  allPersons(last: Int): [Person!]!
}
```

The Mutation Type

```
mutation {
  createPerson(name: "Bob", age: 36) {
    id
  }
}
```

```
type Mutation {
  createPerson(name: String!, age: String!): Person!
}
```

The Subscription Type

```
subscription {
  newPerson {
    name
    age
  }
}
```

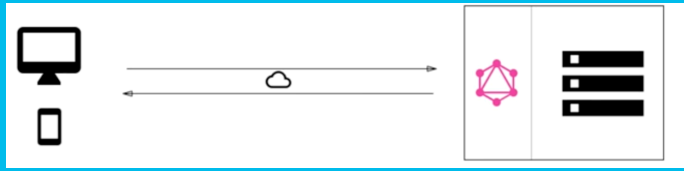
```
type Subscription {
  newPerson: Person!
}
```

```
type Query {
  allPersons(last: Int): [Person!]!
  allPosts(last: Int): [Post!]!
}

type Mutation {
  createPerson(name: String!, age: String!): Person!
  updatePerson(id: ID!, name: String!, age: String!): Person!
  deletePerson(id: ID!): Person!
  createPost(title: String!): Post!
  updatePost(id: ID!, title: String!): Post!
  deletePost(id: ID!): Post!
}

type Subscription {
  newPerson: Person!
  updatedPerson: Person!
  deletedPerson: Person!
  newPost: Post!
  updatedPost: Post!
  deletedPost: Post!
}
```

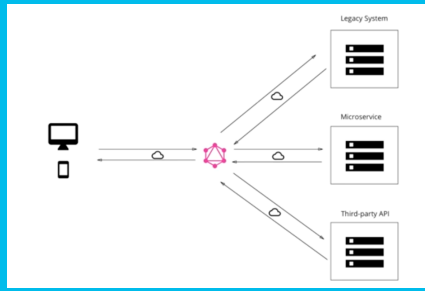
```
type Person {
  id: ID!
  name: String!
  age: Int!
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  author: Person!
}
```

3 Architectural Use Cases for GraphQL

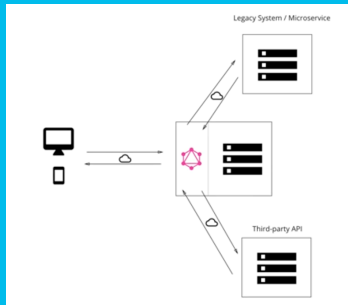
1) GraphQL server with a connected database
- Often used for greenfield projects
- Uses single web server that implements GraphQL
- Server resolves queries and constructs response with data that it gets from the database



2) GraphQL Server to integrate existing system without connected database
- For companies with legacy infrastructure and many different APIs
- Used to unify existing systems and hide complexity of data fetching on the backend
- GraphQL web server doesn't need to care about data sources



3) Hybrid approach with a connected database and integration of existing system



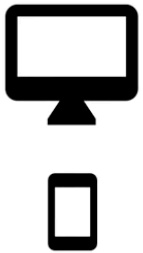
Important Concept: Resolver functions

Recall that queries/mutations consists of set of fields

GraphQL has 1 resolver function per field

Purpose of each resolver function is to fetch the data for its corresponding field

Important Concept: Resolver functions (example)



```
query {
  User(id: “er3txsa9frju”) {
    name
    friends(first: 5) {
      name
      age
    }
  }
}
```

Resolvers

```
User(id: String!): User

name(user: User!): String!

age(user: User!): Int!

friends(first: Int, user: User!): [User!]!
```

With GraphQL, there is a positive shift from imperative to declarative data fetching on the client side

Imperative data fetching:

1) Construct and send HTTP request (i.e fetch in JS)
2) Receive and parse server response
3) Store data locally
4) Display information in UI

Declarative data fetching:

1) Describe data requirements
2) Display information in UI

What’s next?

- Will apply GraphQL in Cisco’s B2B for checking order status

References

https://www.howtographql.com