Designing for GraphQL

Why this talk?

Why this talk?

→ GraphQL invalidates existing assumptions about client–server APIs

Why this talk?

→ GraphQL invalidates existing assumptions about client–server APIs

→ Shift from a *server-driven* model to a *client-driven* model

Why this talk?

→ GraphQL invalidates existing assumptions about client–server APIs

→ Shift from a *server-driven* model to a *client-driven* model

→ Powerful and flexible, but introduces new challenges

Why GraphQL?

The Before Times

The Before Times

1. CORBA

The Before Times

1. CORBA
2. SOAP

The Before Times

1. CORBA
2. SOAP
3. REST

The Before Times

1. CORBA
2. SOAP
3. REST
4. HATEOAS *(hypermedia)*

Problem: over-fetching

Problem: over-fetching

→ Server can't know which fields the client needs

Problem: over-fetching

→ Server can't know which fields the client needs

→ Different clients need different data (e.g., mobile vs. desktop)

Problem: over-fetching

→ Server can't know which fields the client needs

→ Different clients need different data (e.g., mobile vs. desktop)

→ Wasteful to include unnecessary data in responses

Problem: under-fetching

Problem: under-fetching

→ New app features means additional data required

Problem: under-fetching

→ New app features means additional data required

→ Either the server code has to change or the client has to make more queries

Problem: under-fetching

→ New app features means additional data required

→ Either the server code has to change or the client has to make more queries

→ End up with N+1 patterns if data is too limited

GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them.

— *GraphQL specification*

Client-driven

Client-driven

→ Client gets exactly what it wants

Client-driven

→ Client gets exactly what it wants

→ Other architectures are friendly to server implementors but not clients

Client-driven

→ Client gets exactly what it wants

→ Other architectures are friendly to server implementors but not clients

→ Requires the server to be more defensive!

Strongly-typed & introspective

Strongly-typed & introspective

→ Queries are type-checked before execution

Strongly-typed & introspective

→ Queries are type-checked before execution

→ The server can guarantee the shape of the response

Strongly-typed & introspective

→ Queries are type-checked before execution

→ The server can guarantee the shape of the response

→ Tools can check query validity, inspect the schema

GraphQL primitives

Queries

→ Equivalent to REST `GET`, `HEAD`

```
query {
  hero {
    name
    friends {
      name
    }
  }
}
```

```
{
  "hero": {
    "name": "R2-D2",
    "friends": [
      { "name": "Luke Skywalker" },
      { "name": "Han Solo" },
      { "name": "Leia Organa" }
    ]
  }
}
```

Mutations

→ Optional operation type

→ Equivalent to REST POST, PUT, DELETE

```
mutation {
  likeStory(storyID: "12345") {
    story {
      likeCount
    }
  }
}
```

```
{
  "likeStory": {
    "story": {
      "likeCount": 21
    }
  }
}
```

Subscriptions

→ Optional operation type

→ Streaming, pub/sub as a first-class concept

```
subscription {
  newMessage(roomID: "123") {
    sender
    text
  }
}
```

Subscriptions

[]

Subscriptions

```
[{
  "newMessage": {
    "sender": "Alice",
    "text": "Hello world!"
  }
}]
```

Subscriptions


```
[{
  "newMessage": {
    "sender": "Alice",
    "text": "Hello world!"
  }
},
{
  "newMessage": {
    "sender": "Bob",
    "text": "yo"
  }
}]
```

Subscriptions


```
[{
  "newMessage": {
    "sender": "Alice",
    "text": "Hello world!"
  }
},
{
  "newMessage": {
    "sender": "Bob",
    "text": "yo"
  }
},
{
  "newMessage": {
    "sender": "Taylor",
    "text": "👋"
  }
}]
```

Fragments

```
{
  user(id: 4) {
    friends(first: 10) {
      ...friendFields
    }
    mutualFriends(first: 10) {
      ...friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  profilePic(size: 50)
}
```

Fragments

→ Fetch the same fields in multiple places

```
{
  user(id: 4) {
    friends(first: 10) {
      ...friendFields
    }
    mutualFriends(first: 10) {
      ...friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  profilePic(size: 50)
}
```

Fragments

→ Fetch the same fields in multiple places

→ Compose many different fetches into one

```
{
  user(id: 4) {
    friends(first: 10) {
      ...friendFields
    }
    mutualFriends(first: 10) {
      ...friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  profilePic(size: 50)
}
```

Fragments

→ Fetch the same fields in multiple places

→ Compose many different fetches into one

→ Pattern-match on type within a query

```
{
  user(id: 4) {
    friends(first: 10) {
      ...friendFields
    }
    mutualFriends(first: 10) {
      ...friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  profilePic(size: 50)
}
```

(some)

Best practices

Server design should focus on types

Server design should focus on types

→ GraphQL schema should be designed around types that make sense

Server design should focus on types

→ GraphQL schema should be designed around types that make sense

→ Don't make assumptions about client fetch patterns

Server design should focus on types

→ GraphQL schema should be designed around types that make sense

→ Don't make assumptions about client fetch patterns

→ Abstract away ugly details where possible

Client design should focus on fragments

Client design should focus on fragments

→ GraphQL queries are hierarchical

Client design should focus on fragments

→ GraphQL queries are hierarchical

→ UI views are also hierarchical!

Client design should focus on fragments

→ GraphQL queries are hierarchical

→ UI views are also hierarchical!

→ Define one fragment per view

Client design should focus on fragments

→ GraphQL queries are hierarchical

→ UI views are also hierarchical!

→ Define one fragment per view

→ Compose all the fragments on screen together into one single query

Fetch-and-subscribe

Fetch-and-subscribe

→ Client fetches initial state in a query, observes changes with a subscription

Fetch-and-subscribe

→ Client fetches initial state in a query, observes changes with a subscription

→ No polling, repeat queries necessary

Fetch-and-subscribe

→ Client fetches initial state in a query, observes changes with a subscription

→ No polling, repeat queries necessary

→ Local data store can be kept fully consistent

Fetch-and-subscribe

→ Client fetches initial state in a query, observes changes with a subscription

→ No polling, repeat queries necessary

→ Local data store can be kept fully consistent

→ Be careful about race conditions!

Pagination

```
{
  friends(first: 10, after: "opaqueCursor") {
    edges {
      cursor
      node {
        id
        name
      }
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

Pagination

→ Use opaque cursors, not page numbers or offsets

```
{
  friends(first: 10, after: "opaqueCursor") {
    edges {
      cursor
      node {
        id
        name
      }
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

Pagination

→ Use opaque cursors, not page numbers or offsets

→ Each page should yield a fully hierarchical object type

```
{
  friends(first: 10, after: "opaqueCursor") {
    edges {
      cursor
      node {
        id
        name
      }
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

Backwards compatibility

Backwards compatibility

→ GraphQL is unversioned

Backwards compatibility

→ GraphQL is unversioned

→ Fields, types should not be removed

Backwards compatibility

→ GraphQL is unversioned

→ Fields, types should not be removed

→ Fields should not *change* type

Backwards compatibility

→ GraphQL is unversioned

→ Fields, types should not be removed

→ Fields should not *change* type

→ *Can* replace list results with empty lists

Backwards compatibility

→ GraphQL is unversioned

→ Fields, types should not be removed

→ Fields should not *change* type

→ *Can* replace list results with empty lists

→ Use deprecation warnings!

Nullability

Nullability

→ Changing a field to nullable is a breaking change

Nullability

→ Changing a field to nullable is a breaking change

→ Errors within a non-null field will fail the parent

Nullability

→ Changing a field to nullable is a breaking change

→ Errors within a non-null field will fail the parent

→ Better to make fields nullable by default

Demand control

Demand control

→ GraphQL requires the server to assume more complexity

Demand control

→ GraphQL requires the server to assume more complexity

→ The possible space of unreasonable or malicious queries is *huge*

Demand control

→ GraphQL requires the server to assume more complexity

→ The possible space of unreasonable or malicious queries is *huge*

Demand control

→ GraphQL requires the server to assume more complexity

→ The possible space of unreasonable or malicious queries is *huge*

→ Estimate query complexity

Demand control

→ GraphQL requires the server to assume more complexity

→ The possible space of unreasonable or malicious queries is *huge*

→ Estimate query complexity

→ Preregister supported queries

Demand control

→ GraphQL requires the server to assume more complexity

→ The possible space of unreasonable or malicious queries is *huge*

→ Estimate query complexity

→ Preregister supported queries

→ Apply backend rate limits

Remember...

GraphQL is client-driven

Design accordingly!

Resources

Resources

→ GraphQL website

Resources

→ GraphQL website

→ GraphQL specification

Resources

→ GraphQL website

→ GraphQL specification

→ Principled GraphQL