# 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

# The Before Times

# The Before Times

1. CORBA

# The Before Times

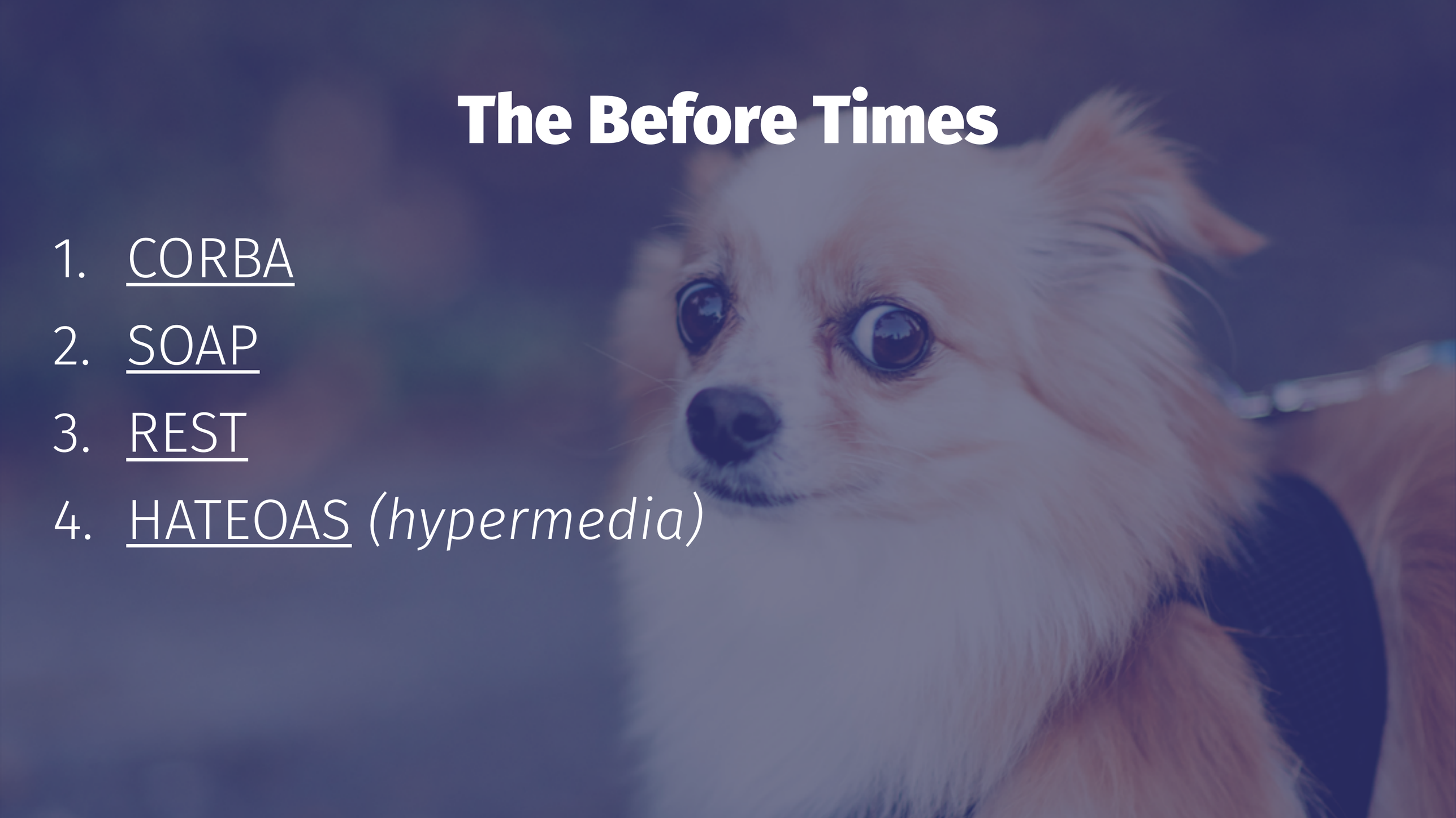
1. CORBA
2. SOAP

# 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

# **Problem:** over-fetching

→ Server can't know which fields the client needs

→ Different clients need different data

→ **Bad solution:** include *all data* in responses

# Problem: under-fetching

# Problem: under-fetching

→ More app features = more data required

# Problem: under-fetching

→ More app features = more data required

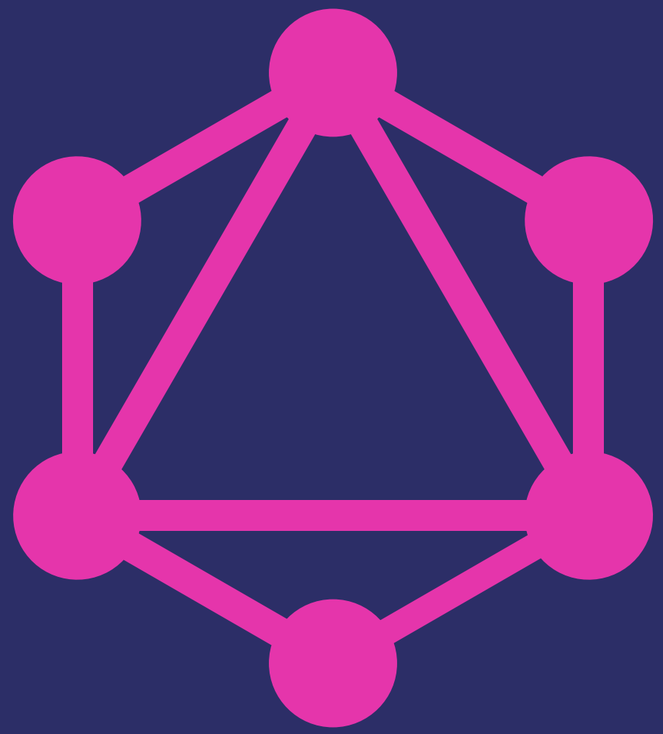
→ Server implementation needs to be updated

# **Problem:** under-fetching

→ More app features = more data required

→ Server implementation needs to be updated

→ **Bad solution:** N+1 query patterns



# Introducing GraphQL

***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

→ Fetch only the data needed for presentation

# Client-driven

→ Client gets **exactly** what it wants

→ Fetch only the data needed for presentation

→ Requires the server to be more flexible!

# Strongly-typed & introspective

# Strongly-typed & introspective

→ Queries are **type checked** before execution

# Strongly-typed & introspective

→ Queries are **type checked** before execution

→ Server can guarantee the shape of the response

# Strongly-typed & introspective

→ Queries are **type checked** before execution

→ 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

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

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

→ Fetch similar fields in **multiple places**

# Fragments

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

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

→ Fetch similar fields in **multiple places**

→ **Compose** many different fetches into one

# Fragments

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

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

→ Fetch similar fields in **multiple places**

→ **Compose** many different fetches into one

→ **Pattern match** on types within a query


(some)
Best practices

# Server should design types

# Server should design types

→ Schema should be clean, **graph-based**

# Server should design types

→ Schema should be clean, **graph-based**

→ Don't assume anything about client fetch patterns

# Server should design types

→ Schema should be clean, **graph-based**

→ Don't assume anything about client fetch patterns

→ Abstract away implementation details

# Client should design fragments

# Client should design fragments

→ GraphQL queries are **hierarchical**

# Client should design fragments

→ GraphQL queries are **hierarchical**

→ **UI views** are also hierarchical!

# Client should design fragments

→ GraphQL queries are **hierarchical**

→ **UI views** are also hierarchical!

→ Define **one fragment per view**

# Client should design fragments

→ GraphQL queries are **hierarchical**

→ **UI views** are also hierarchical!

→ Define **one fragment per view**

→ Compose all fragments on screen into one query

# Unidirectional data flow

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

→ Fetch initial state in a **query**

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

→ Fetch initial state in a **query**

→ Observe changes over time with a **subscription**

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

→ Fetch initial state in a **query**

→ Observe changes over time with a **subscription**

→ Make changes with a **mutation**

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

→ Fetch initial state in a **query**

→ Observe changes over time with a **subscription**

→ Make changes with a **mutation**

→ Refresh the UI whenever the data store changes

# **Unidirectional** data flow

→ **Client data store** should be the source of truth

→ Fetch initial state in a **query**

→ Observe changes over time with a **subscription**

→ Make changes with a **mutation**

→ Refresh the UI whenever the data store changes

→ Be careful about race conditions!

# Pagination

→ Use **opaque** cursors—
avoid page numbers
or offsets

→ Each page should have
an **object graph**

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

# Backwards compatibility

# Backwards compatibility

→ GraphQL is **unversioned**

# Backwards compatibility

→ GraphQL is **unversioned**

→ **Removing** fields, types will break clients

# Backwards compatibility

→ GraphQL is **unversioned**

→ **Removing** fields, types will break clients

→ **Changing** field types will break clients

# Backwards compatibility

→ GraphQL is **unversioned**

→ **Removing** fields, types will break clients

→ **Changing** field types will break clients

→ 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

→ **Flexibility** makes possible many **unreasonable or malicious queries**

# Demand control

→ **Flexibility** makes possible many **unreasonable or malicious queries**

→ The **GraphQL server** must manage this complexity

# Demand control

→ **Flexibility** makes possible many **unreasonable or malicious queries**

→ The **GraphQL server** must manage this complexity

# Demand control

→ **Flexibility** makes possible many **unreasonable or malicious queries**

→ The **GraphQL server** must manage this complexity

1. Estimate query complexity

# Demand control

→ **Flexibility** makes possible many **unreasonable or malicious queries**

→ The **GraphQL server** must manage this complexity

1. Estimate query complexity
2. Preregister supported queries

# Demand control

→ **Flexibility** makes possible many **unreasonable or malicious queries**

→ The **GraphQL server** must manage this complexity

1. Estimate query complexity
2. Preregister supported queries
3. Apply backend rate limits


Remember...

# GraphQL is client-driven

*Design accordingly!*

# Resources

→ GraphQL website

→ GraphQL specification

→ Principled GraphQL