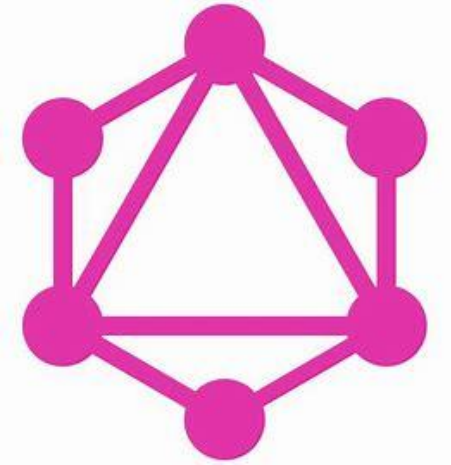
Fun with GraphQL Schema Stitching

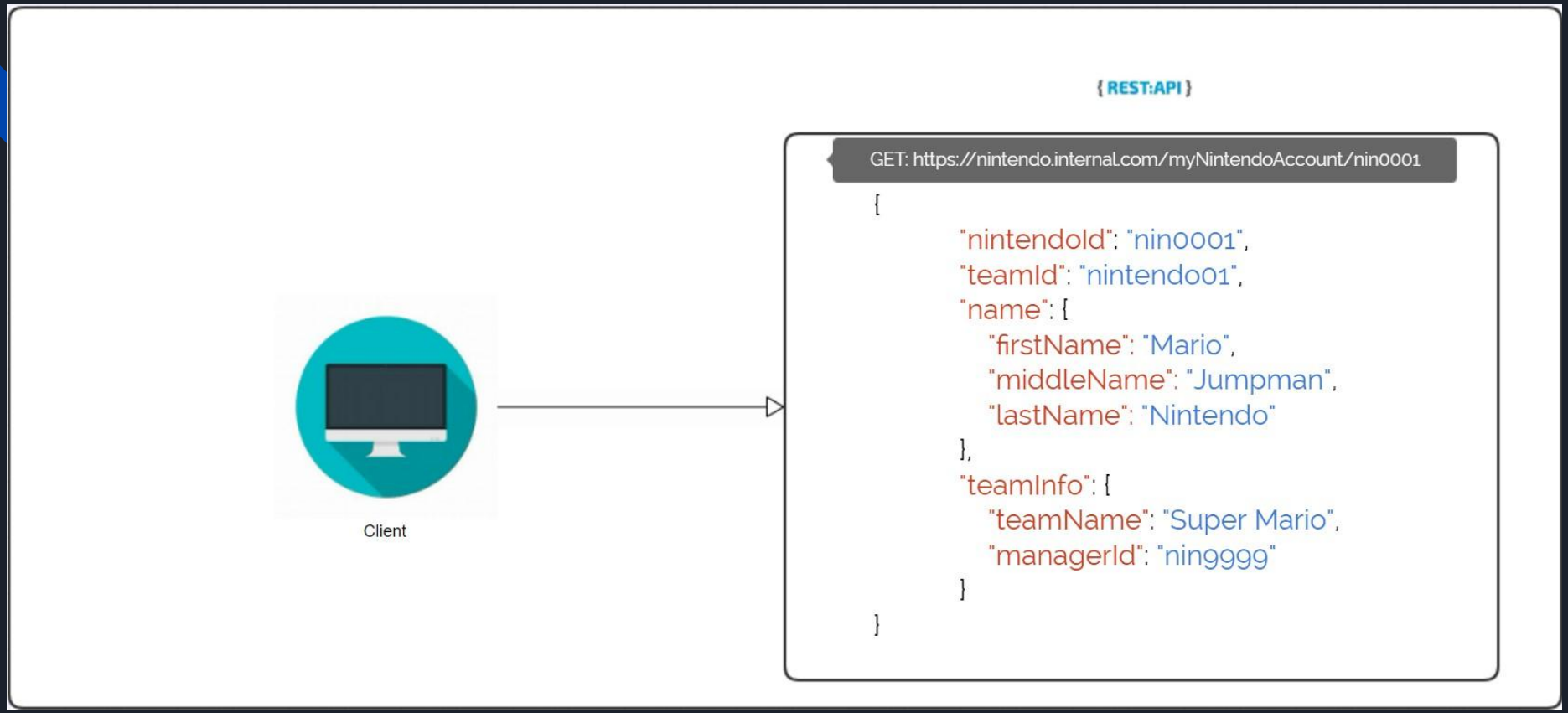
A natural extension of GraphQL

Agenda

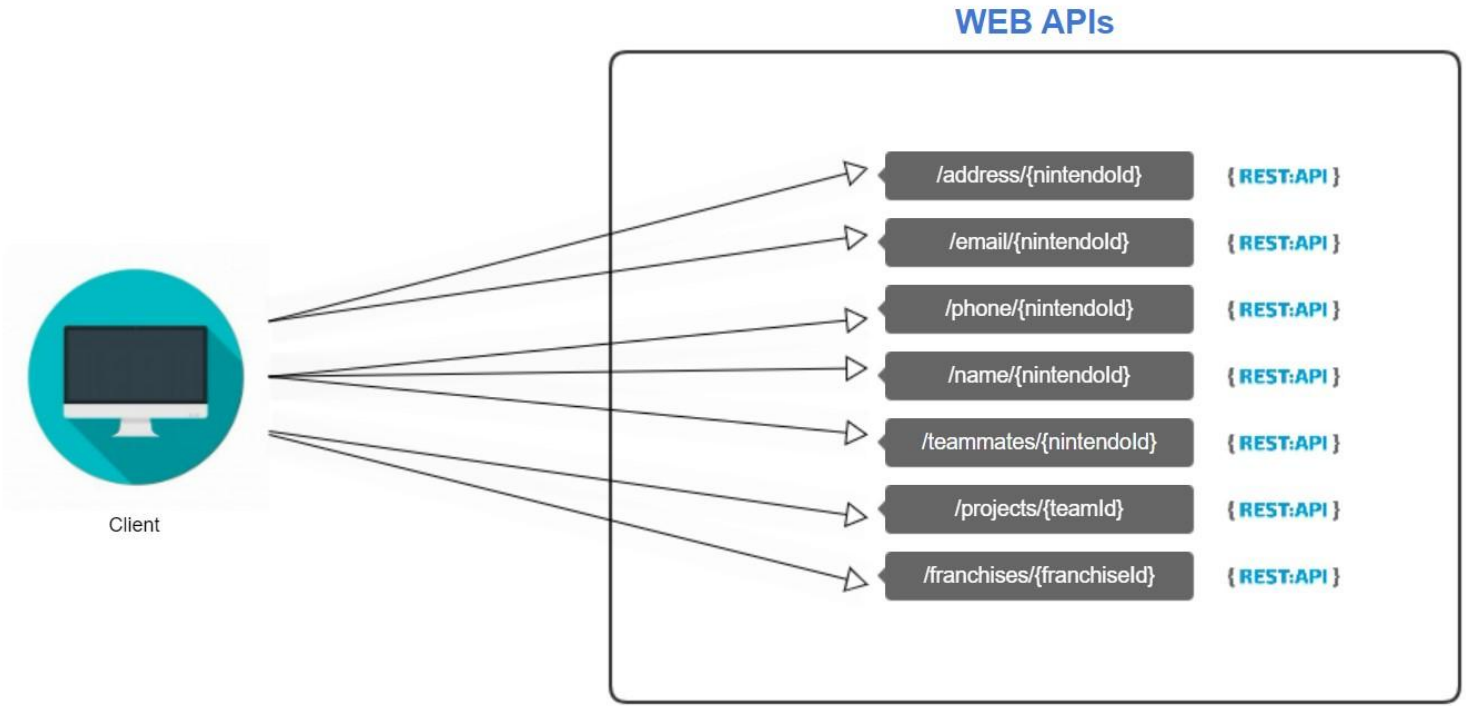
- Overview of GraphQL
- Schema Stitching
- Demo - Fun with GraphQL Schema Stitching
- Recap
- Q&A

What is GraphQL?





REST is pretty manageable when making 1 or 2 calls



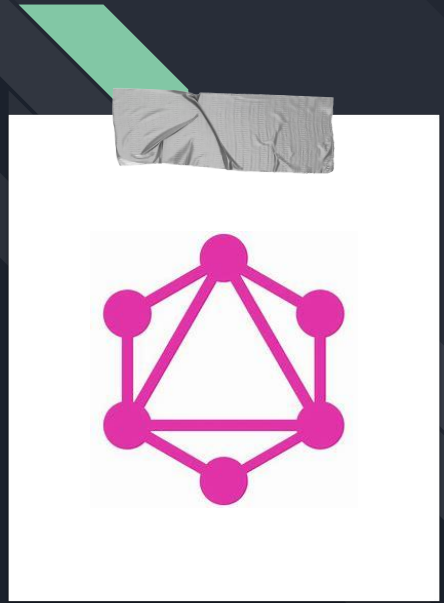
A client making calls to multiple REST APIs can get a little messy.

So how do we solve that?

So how do we solve that?

A query language for your API

“GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask exactly what they need and nothing more, makes it easier to evolve APIs over time, and enable powerful developer tools.”



Source: [GraphQL | A query language for your API]

GraphQL Query Example

```
query contactInformation($nintendoId: NintendoId!){
    addresses(id: $nintendoId) {
        stateProvince
        cityName
        streetAddress
        postalCode
    }
    phones(id: $nintendoId) {
        type
        purpose
        number
    }
    emails(id: $nintendoId) {
        purpose
        emailAddress
    }
}
```


```
{
  "data": {
    "addresses": [
      {
        "stateProvince": "Texas",
        "cityName": "San Antonio",
        "streetAddress": "1985 Nintendo Avenue Apt 1964",
        "postalCode": "78240"
      }
    ],
    "phones": [
      {
        "type": "MOBILE",
        "purpose": "BOTH",
        "number": "2104892777"
      }
    ],
    "emails": [
      {
        "purpose": "WORK",
        "emailAddress": "mario@nintendo.com"
      },
      {
        "purpose": "PERSONAL",
        "emailAddress": "mario@switch.com"
      }
    ]
  }
}
```


A query is a GraphQL Operation that allows you to retrieve specific data from the server

GraphQL Query Example

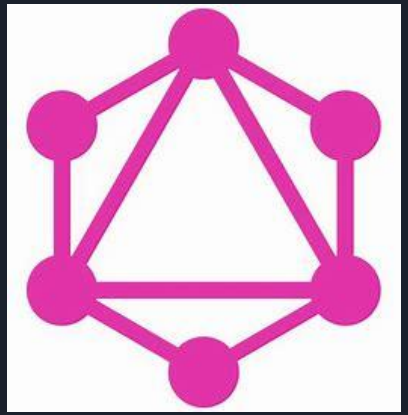
```
query Hello($nintendoId: NintendoId!){
    addresses(id: $nintendoId) {
        stateProvince
        cityName
        streetAddress
        postalCode
    }
}
```


```
{
  "data": {
    "addresses": [
      {
        "stateProvince": "Texas",
        "cityName": "San Antonio",
        "streetAddress": "1985 Nintendo Avenue Apt 1964",
        "postalCode": "78240"
      }
    ]
  }
}
```

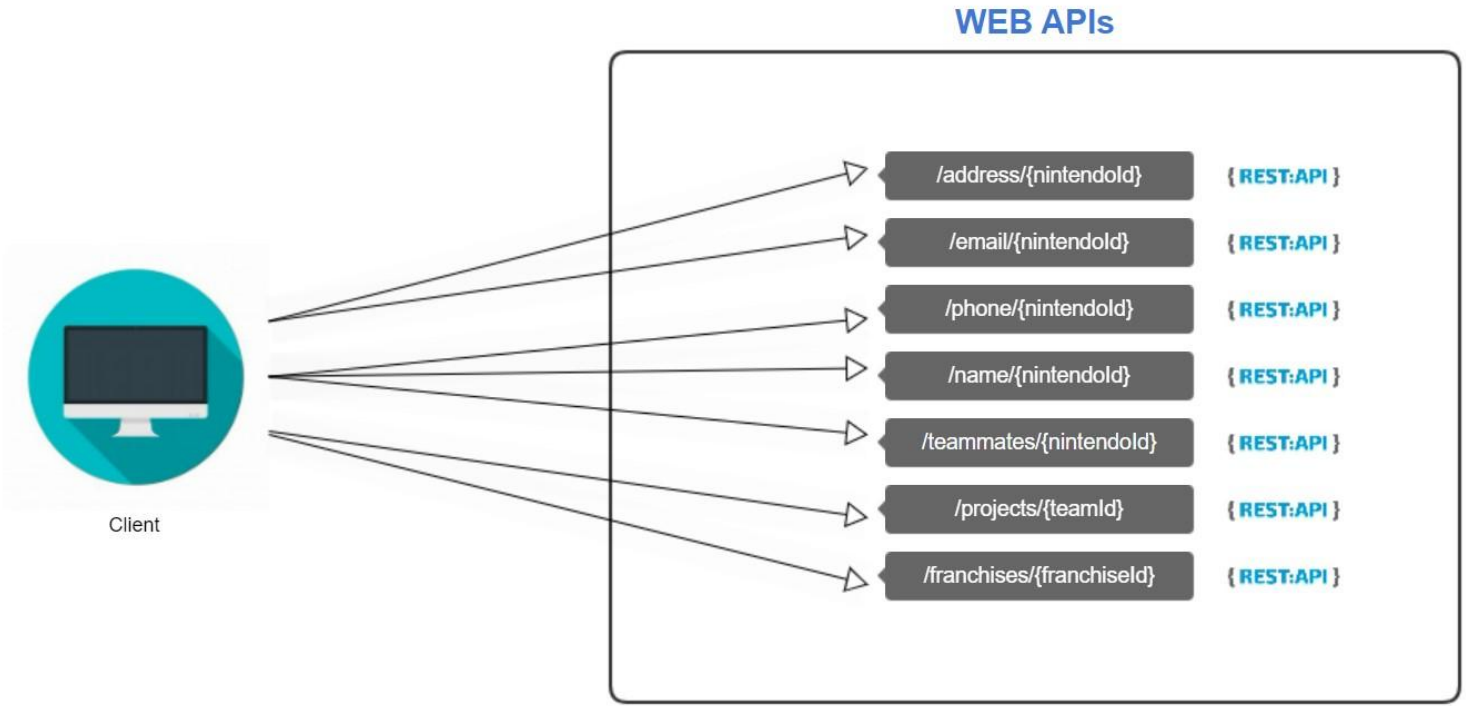

A query is a GraphQL Operation that allows you to retrieve specific data from the server

GraphQL - Only What You Need

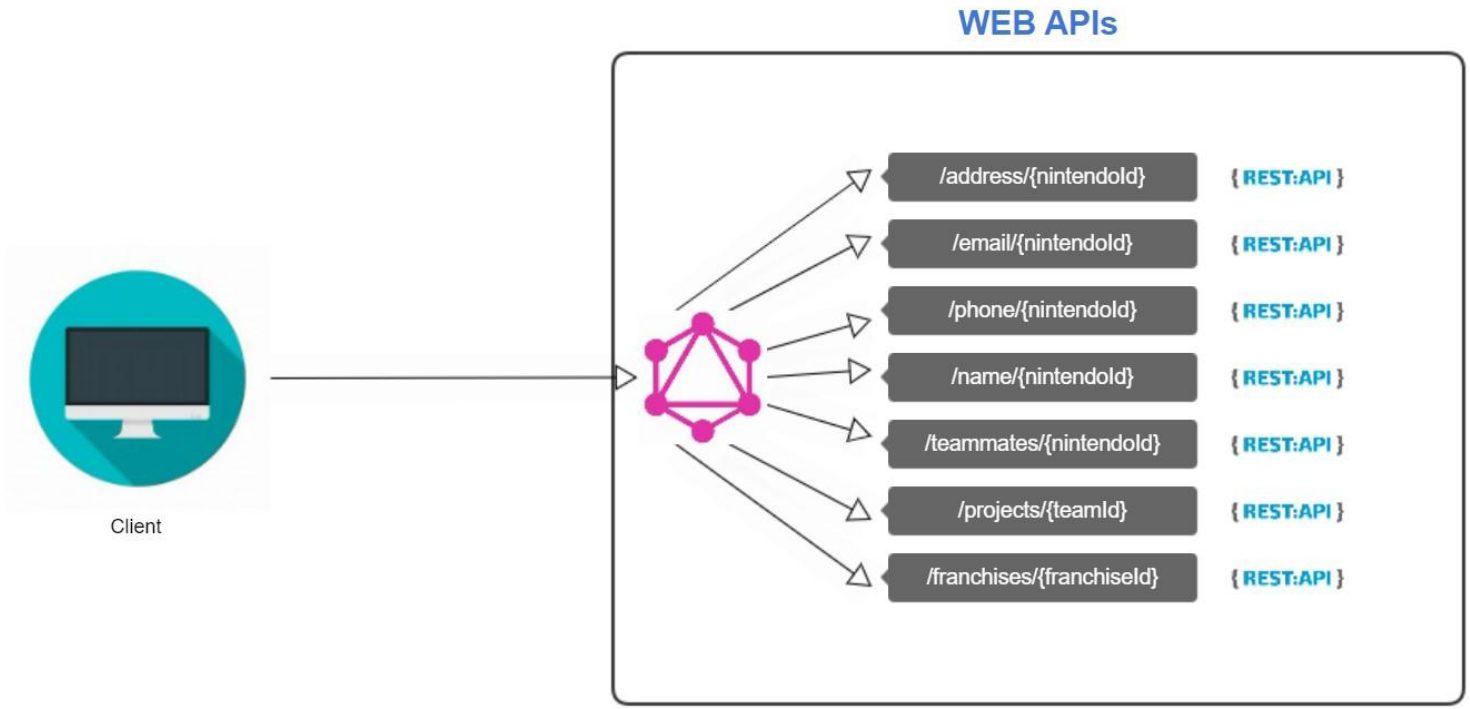
Returns exactly what you need and nothing more.



Fetches data across multiple different sources using a single query.



Our client would need to make multiple request.



Our client just needs to make one request.

GraphQL Schema

Query - operations used to retrieve data

Object Type - return types with their corresponding fields

Fields - attributes of an object. Can be primitive scalars (Int, String, etc), enums, custom scalars or other Object types

```
scalar NintendoId

type Query {
    # Retrieve all addresses associated to a Nintendo ID
    addresses(id: NintendoId!): [Address]
}

type Address {
    id: String!
    nintendoId: String!
    stateProvince: String
    cityName: String
    streetAddress: String
    postalCode: String
}
```

GraphQL Schema

Mutation - operations used to modify data

Input - arguments used for both Mutations and Queries

Custom Scalar - custom field types for validation

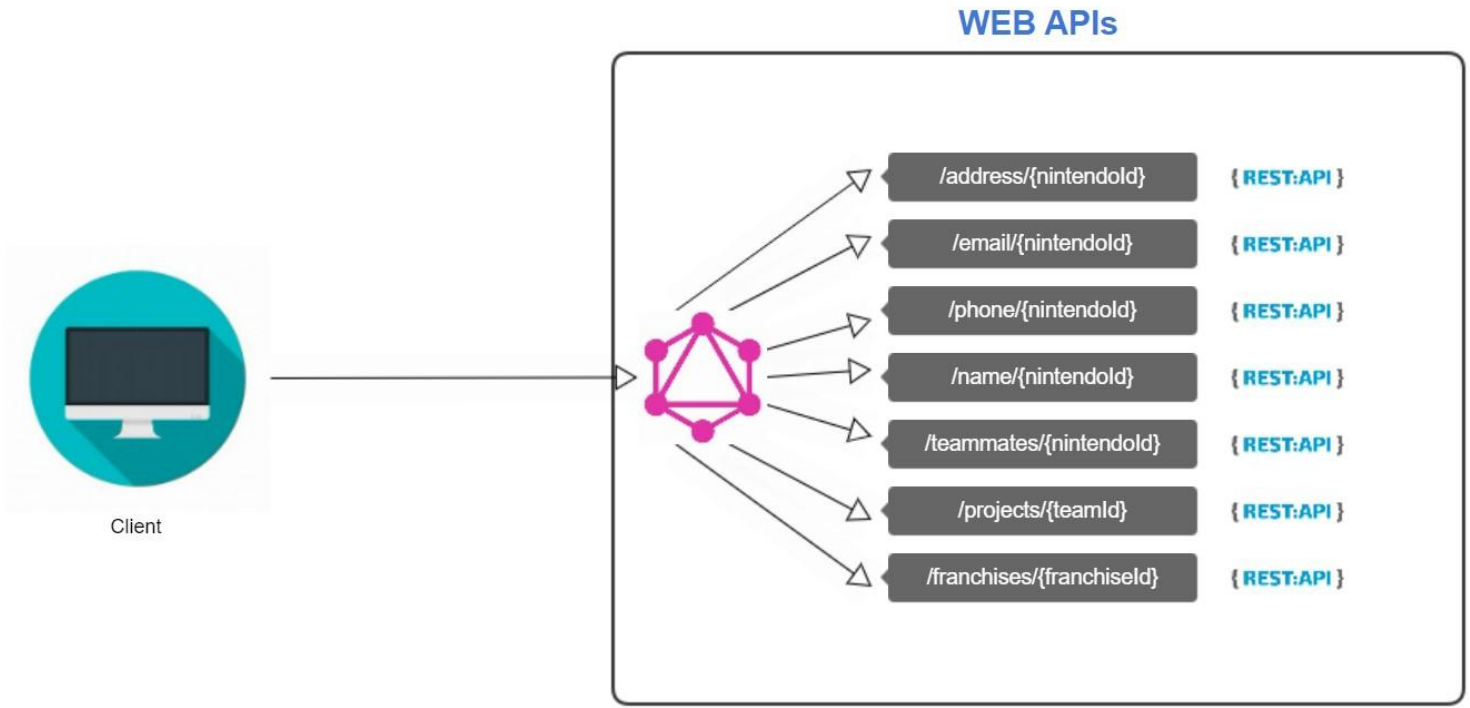
```
scalar NintendoGuid

type Mutation {
    newAddress(address: AddressInput!) : Address
    updateAddress(address: AddressInput!) : Address
    deleteAddress(id: NintendoGuid!) : String
}

input AddressInput {
    id: NintendoGuid
    nintendoId: String!
    stateProvince: String
    cityName: String
    streetAddress: String
    postalCode: String
}
```

GraphQL Query Example

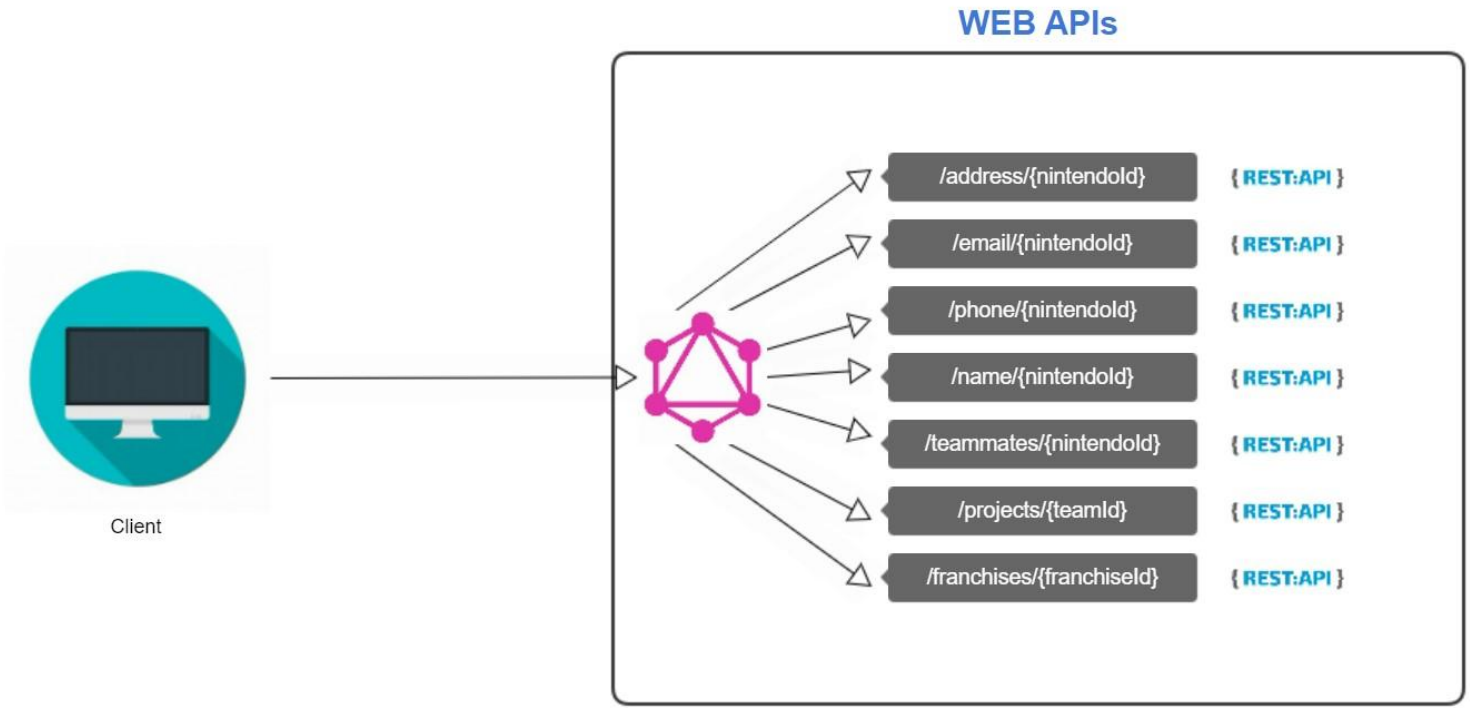
```
query Hello($nintendoId: NintendoId!){
    addresses(id: $nintendoId) {
        stateProvince
        cityName
        streetAddress
        postalCode
    }
}
```


```
{
  "data": {
    "addresses": [
      {
        "stateProvince": "Texas",
        "cityName": "San Antonio",
        "streetAddress": "1985 Nintendo Avenue Apt 1964",
        "postalCode": "78240"
      }
    ]
  }
}
```


A query is a GraphQL Operation that allows you to retrieve specific data from the server



We can continue to expand on the GraphQL API…

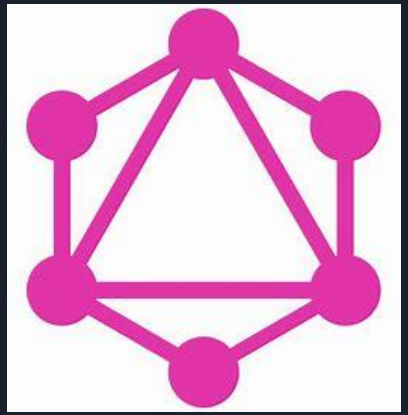


We can continue to expand on the GraphQL API… but we may run into some issues…

PROS AND CONS

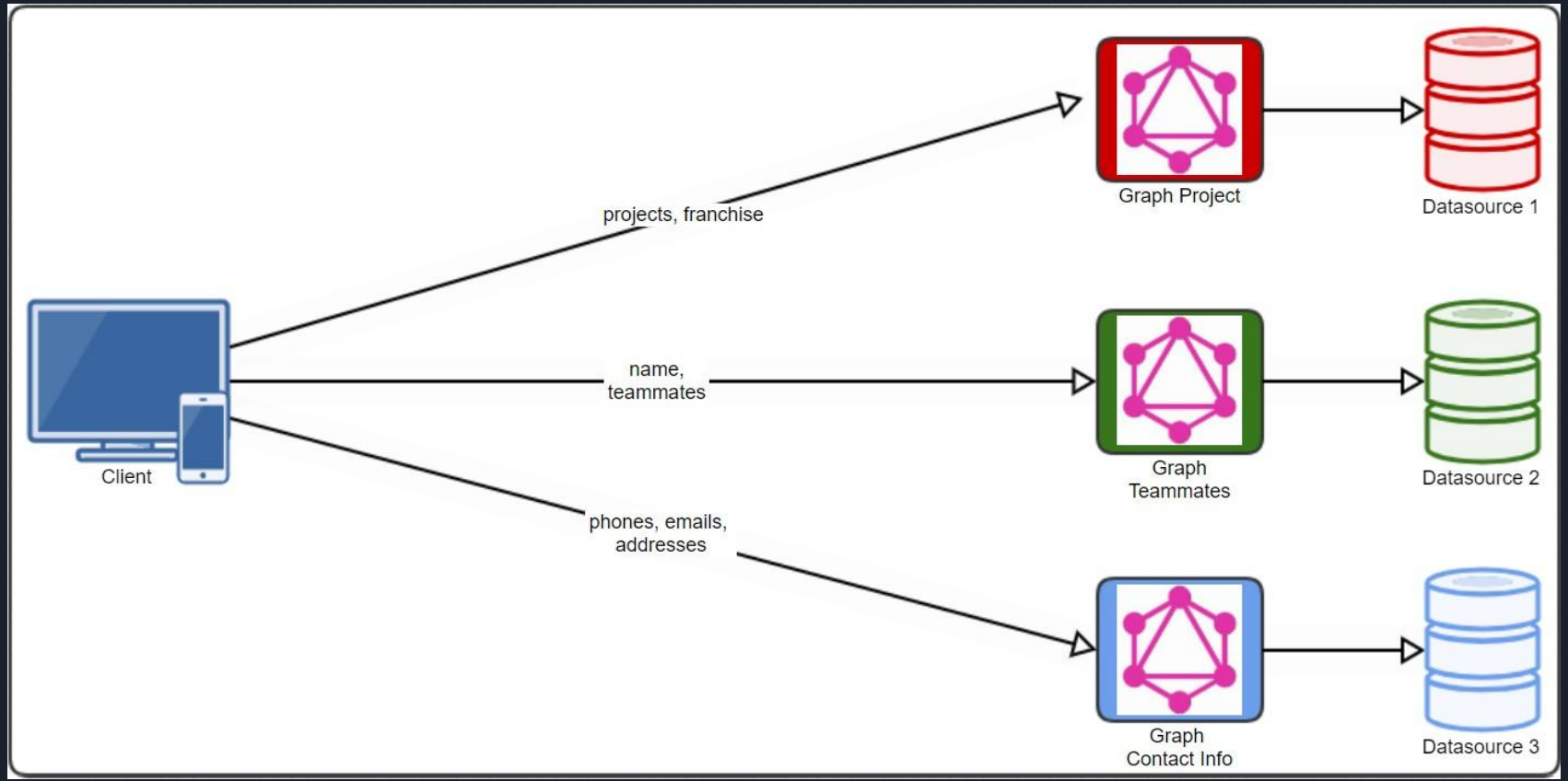
PROS

- Efficient, only fetch what you need
- One single API endpoint
- Self describing

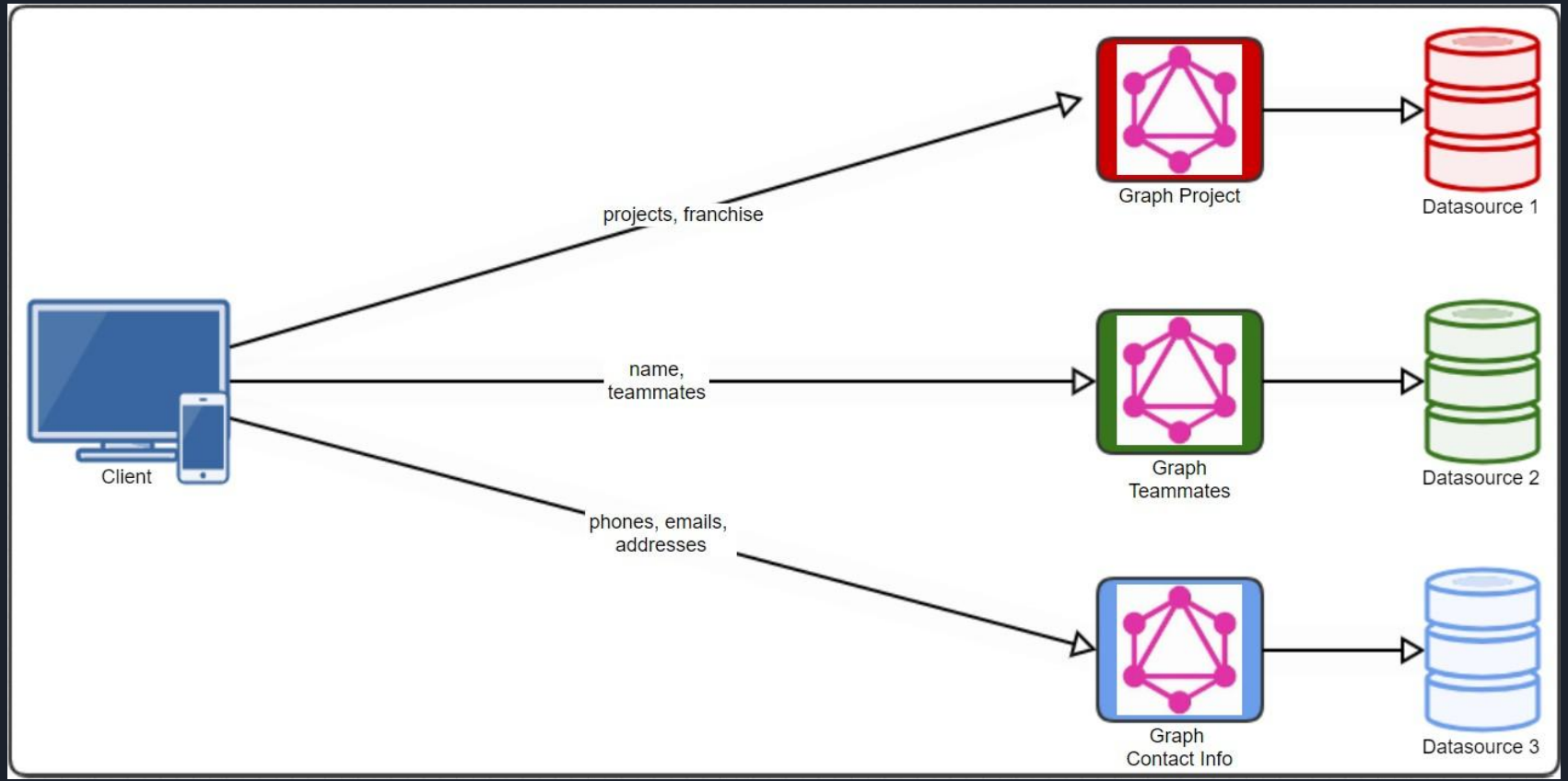


CONS

- Single endpoint usually means all code sits in one repository
- All code runs on a single server (not good for scalability)



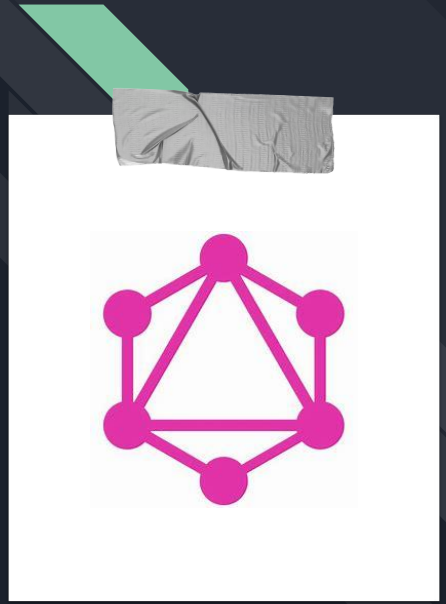
Divide and Conquer



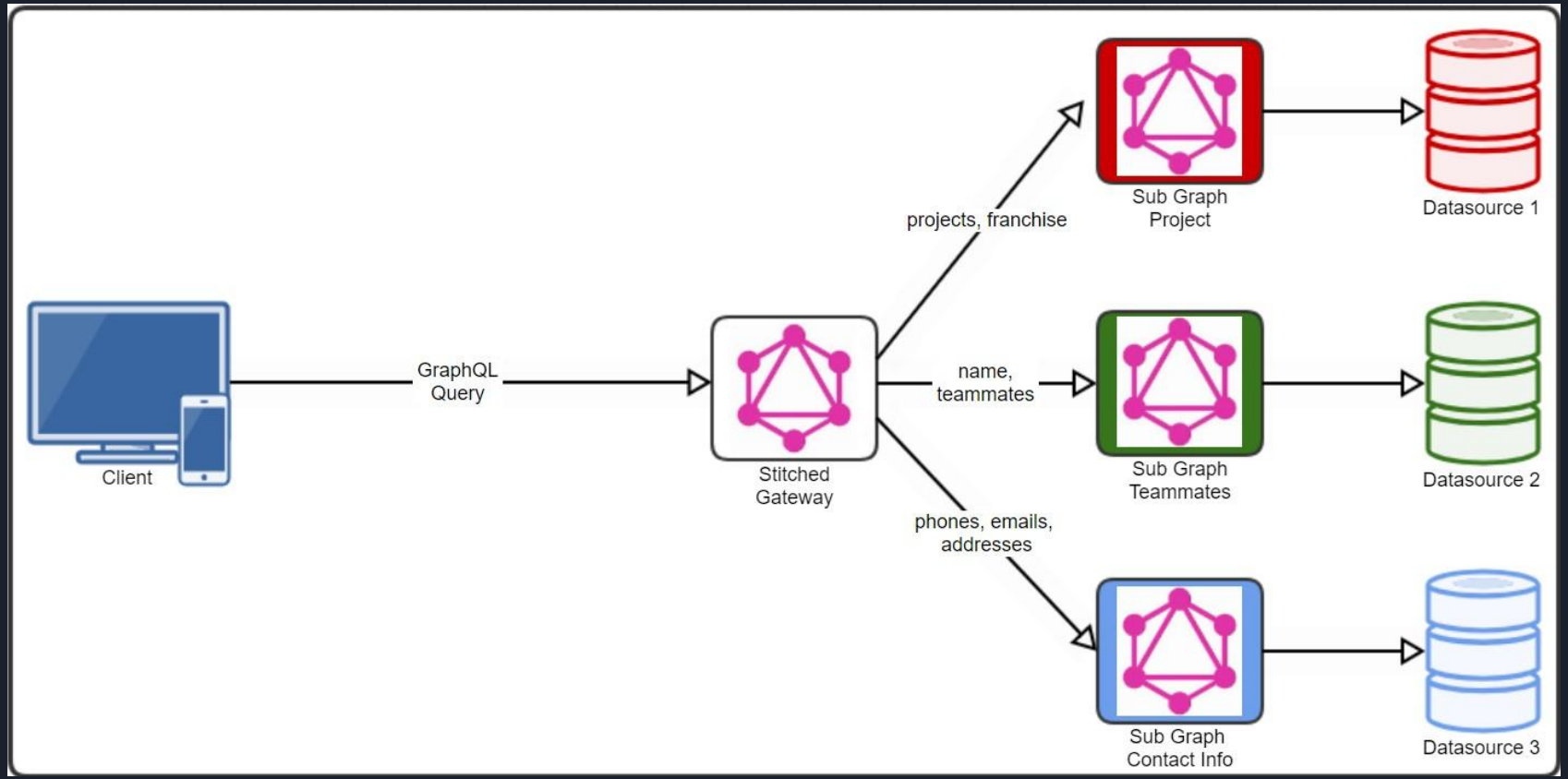
WAIT A MINUTE! We might end up with the same problem from earlier!!!

Schema Stitching

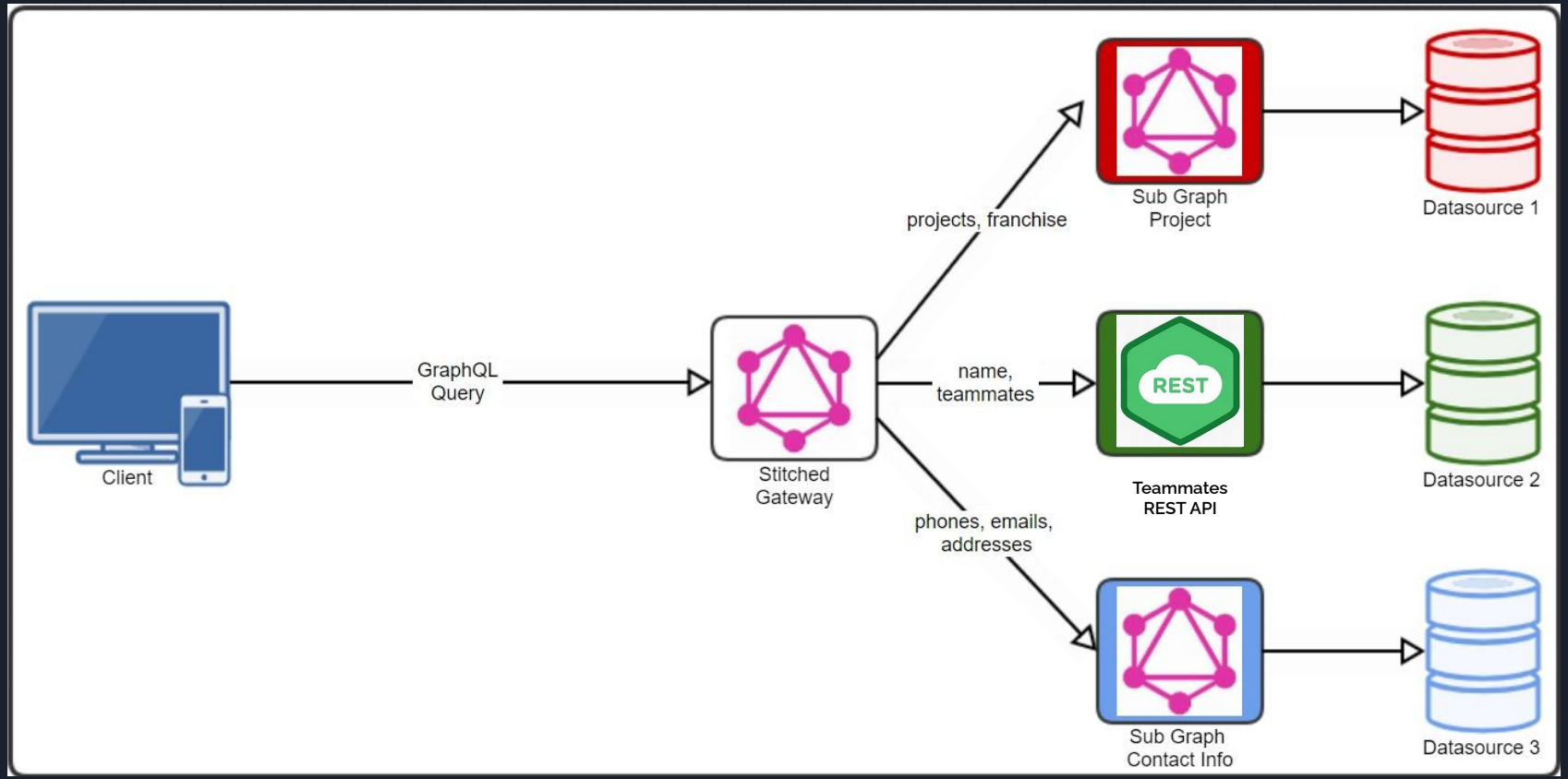
“Schema stitching is the process of creating a single GraphQL schema from multiple underlying GraphQL APIs.”



Source: [The ultimate guide to Schema Stitching in GraphQL (hasura.io)]



One GraphQL API to rule them all!!



One GraphQL API to rule them all!!

Coding Examples

Stitching APIs

```
return stitchSchemas({
  // Our Stitched Gateway will have access to all queries/mutations defined by our Sub Graphs
  subschemas: [subSchemaTeam, subSchemaContact, subSchemaProject],
  // Defining extra types and queries on the Gateway Schema
  typeDefs: gatewaySchema,
  resolvers: {
    // Resolving the employeeData Query
    Query: {
      employeeData(obj, args, context, info) {
        return retrieveTeamInfo(args.id);
      }
    },
    Mutation: {···
    },
    // Resolving the NintendoEmployee object
    NintendoEmployee: {···
    },
    // Resolving the Teammate object
    Teammate: {···
    },
    // Resolving the Project object
    Project: {···
    },
    // Resolving the ContactInformation object
    ContactInformation: {···
    }
  }
});
}
```

Stitching API (graphql-tools.com)

All you need:

- ➔ Sub Schemas
- ➔ typeDefs and resolvers are optional (more on that later)

Sub Schemas

```
const subSchemaTeam = {
  //Defining our Remote Sub Graph's Schema Locally
  schema: teamSchema,
  // Remote Executor used to call Sub Graph
  executor: teamExec
}

const subSchemaContact = {
  // Fetching Remote Sub Graph's Schema using Introspection
  schema: await introspectSchema(contactExec),
  // Remote Executor used to call Sub Graph
  executor: contactExec
}

const subSchemaProject = {
  // Fetching Remote Sub Graph's Schema using Introspection
  schema: await introspectSchema(projectExec),
  // Remote Executor used to call Sub Graph
  executor: projectExec
}
```

Combining schemas (graphql-tools.com)

All you need:

- ➔ Schema
- ➔ Executor

Sub Schemas

```
const subSchemaTeam = {
  //Defining our Remote Sub Graph's Schema Locally
  schema: teamSchema,
  // Remote Executor used to call Sub Graph
  executor: teamExec
}

const subSchemaContact = {
  // Fetching Remote Sub Graph's Schema using Introspection
  schema: await introspectSchema(contactExec),
  // Remote Executor used to call Sub Graph
  executor: contactExec
}

const subSchemaProject = {
  // Fetching Remote Sub Graph's Schema using Introspection
  schema: await introspectSchema(projectExec),
  // Remote Executor used to call Sub Graph
  executor: projectExec
}
```

Combining schemas (graphql-tools.com)

Also

“Introspection” and “Local Storage” are the two most common approaches for defining your Sub Graph’s schemas…. But there are other ways as well.

Subschemas are loaded in on start up.

GraphQL Query - Simple Merge

```
query simpleMerge($nintendoId: NintendoId!){
  myName(nintendoId: $nintendoId) {
    firstName
    middleName
    lastName
  }
  myTeammates(nintendoId: $nintendoId) {
    teamId
    nintendoId
  }
  addresses(id: $nintendoId) {
    stateProvince
    cityName
    streetAddress
  }
  emails(id: $nintendoId) {
    purpose
    emailAddress
  }
  phones(id: $nintendoId) {
    type
    purpose
    number
  }
}
```


```
{
  "data": {
    "myName": {
      "firstName": "Mario",
      "middleName": "Jumpman",
      "lastName": "Nintendo"
    },
    "myTeammates": [
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0002"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0003"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin9999"
      }
    ],
    "addresses": [
      {
        "stateProvince": "Texas",
        "cityName": "San Antonio",
        "streetAddress": "1985 Nintendo Avenue Apt 1964"
      }
    ],
    "emails": [
      {
        "purpose": "WORK",
        "emailAddress": "mario@nintendo.com"
      },
      {
        "purpose": "PERSONAL",
        "emailAddress": "mario@switch.com"
      }
    ],
    "phones": [
      {
        "type": "MOBILE",
        "purpose": "BOTH",
        "number": "2104892777"
      }
    ]
  }
}
```


Our Stitched GraphQL Gateway will have access to all queries and mutations provided by our Sub Graph

Extending on the Stitched Schema

You have access to all this information but not in the way you want.

I can get my teammates info… but then I would have to circle back and make another call for each teammate I have.

```
scalar NintendoId
scalar NintendoTeamId

type Query {
    # Retrieve all Teammates associated to a Nintendo ID
    myTeammates(nintendoId: NintendoId!): [Teammate]
}

type Teammate {
    nintendoId: NintendoId!
    teamId: NintendoTeamId!
}
```

```
query Hello($nintendoId: NintendoId!){
    myTeammates(nintendoId: $nintendoId) {
      teamId
      nintendoId
    }
}
```


```
{
  "data": {
    "myTeammates": [
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0002"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0003"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin9999"
      }
    ]
  }
}
```

Extending on the Stitched Schema

You can extend your Type Objects.

```
type NintendoEmployee {
    nintendoId: String!
    teamId: String!
    teamInfo: Team
    name: Name
    contactInformation: ContactInformation
    teammates: [Teammate]
}

# Extending the Teammate type to allow look ups information for a teammate
extend type Teammate {
    details: NintendoEmployee
}
```

Extending on the Stitched Schema

You can add new queries, mutations, types, etc. on the Gateway

```
type Query {
    # Retrieve all employee information associated to a NintendoID
    employeeData(id: NintendoId!): NintendoEmployee
}
```

```graphql
query allEmployeeData($nintendoId: NintendoId!){
  employeeData(id: $nintendoId) {
    nintendoId
    teamId
    name {
      firstName
      lastName
    }
    contactInformation {
      address {
        country
        stateProvince
        cityName
        streetAddress
      }
      phone {
        type
        countryCode
        number
      }
      email {
        emailAddress
        purpose
      }
    }
    teammates {
      nintendoId
      teamId
      details {
        name {
          firstName
          lastName
        }
        contactInformation {
          email {
            emailAddress
          }
          phone {
            number
          }
        }
      }
    }
  }
}
```

QUERY VARIABLES

```json
{
  "nintendoId": "nin0001"
}
```

```json
{
  "data": {
    "employeeData": {
      "nintendoId": "nin0001",
      "teamId": "nintendo01",
      "name": {
        "firstName": "Mario",
        "lastName": "Nintendo"
      },
      "contactInformation": {
        "address": [
          { ↔ }
        ],
        "phone": [
          { ↔ }
        ],
        "email": [ ↔ ]
      },
      "teammates": [
        {
          "nintendoId": "nin0002",
          "teamId": "nintendo01",
          "details": {
            "name": {
              "firstName": "Luigi",
              "lastName": "Nintendo"
            },
            "contactInformation": {
              "email": [
                {
                  "emailAddress": "luigi@nintendo.com"
                },
                {
                  "emailAddress": "luigi@switch.com"
                }
              ],
              "phone": [
                {
                  "number": "2104892777"
                }
              ]
            }
          }
        },
        {
          "nintendoId": "nin0003",
          "teamId": "nintendo01",
          "details": {
            "name": {
              "firstName": "Peach",
              "lastName": "Toadstool"
```

```
query simpleMerge($nintendoId: NintendoId!){
  myName(nintendoId: $nintendoId) {
    firstName
    middleName
    lastName
  }
  myTeammates(nintendoId: $nintendoId) {
    teamId
    nintendoId
  }
  addresses(id: $nintendoId) {
    stateProvince
    cityName
    streetAddress
  }
  emails(id: $nintendoId) {
    purpose
    emailAddress
  }
  phones(id: $nintendoId) {
    type
    purpose
    number
  }
}
```


```
{
  "data": {
    "myName": {
      "firstName": "Mario",
      "middleName": "Jumpman",
      "lastName": "Nintendo"
    },
    "myTeammates": [
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0002"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin0003"
      },
      {
        "teamId": "nintendo01",
        "nintendoId": "nin9999"
      }
    ],
    "addresses": [
      {
        "stateProvince": "Texas",
        "cityName": "San Antonio",
        "streetAddress": "1985 Nintendo Avenue Apt 1964"
      }
    ],
    "emails": [
      {
        "purpose": "WORK",
        "emailAddress": "mario@nintendo.com"
      },
      {
        "purpose": "PERSONAL",
        "emailAddress": "mario@switch.com"
      }
    ],
    "phones": [
      {
        "type": "MOBILE",
        "purpose": "BOTH",
        "number": "2104892777"
      }
    ]
  }
}
```

Extending on the Stitched Schema

```
return stitchSchemas({
  // Our Stitched Gateway will have access to all queries/mutations defined by our Sub Graphs
  subschemas: [subSchemaTeam, subSchemaContact, subSchemaProject],
  // Defining extra types and queries on the Gateway Schema
  typeDefs: gatewaySchema,
  resolvers: {
    // Resolving the employeeData Query
    Query: {
      employeeData(obj, args, context, info) {
        return retrieveTeamInfo(args.id);
      }
    },
    Mutation: {···
    },
    // Resolving the NintendoEmployee object
    NintendoEmployee: {···
    },
    // Resolving the Teammate object
    Teammate: {···
    },
    // Resolving the Project object
    Project: {···
    },
    // Resolving the ContactInformation object
    ContactInformation: {···
    }
  }
});
}
```

Type Definition + Resolvers = Custom GraphQL Fun!

Delegate to Schema

```
NintendoEmployee: {
  name: {
    resolve(nintendoEmployee, args, context, info) {
      return delegateToSchema({
        schema: subSchemaTeam,
        operation: 'query',
        fieldName: 'myName',
        args: { nintendoId: nintendoEmployee.nintendoId },
        context, info})
    }
  },
```

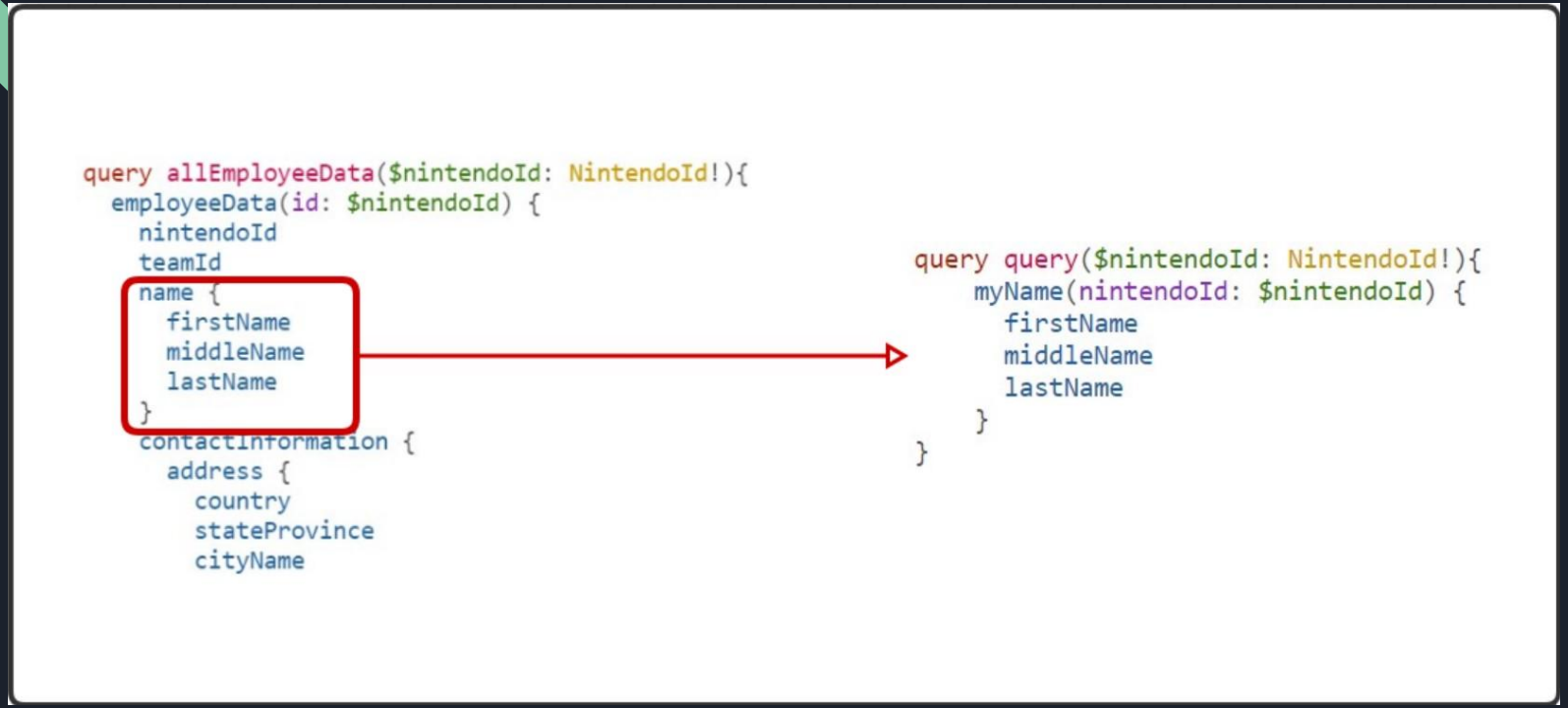
Schema delegation (graphql-tools.com)

Delegate to Schema

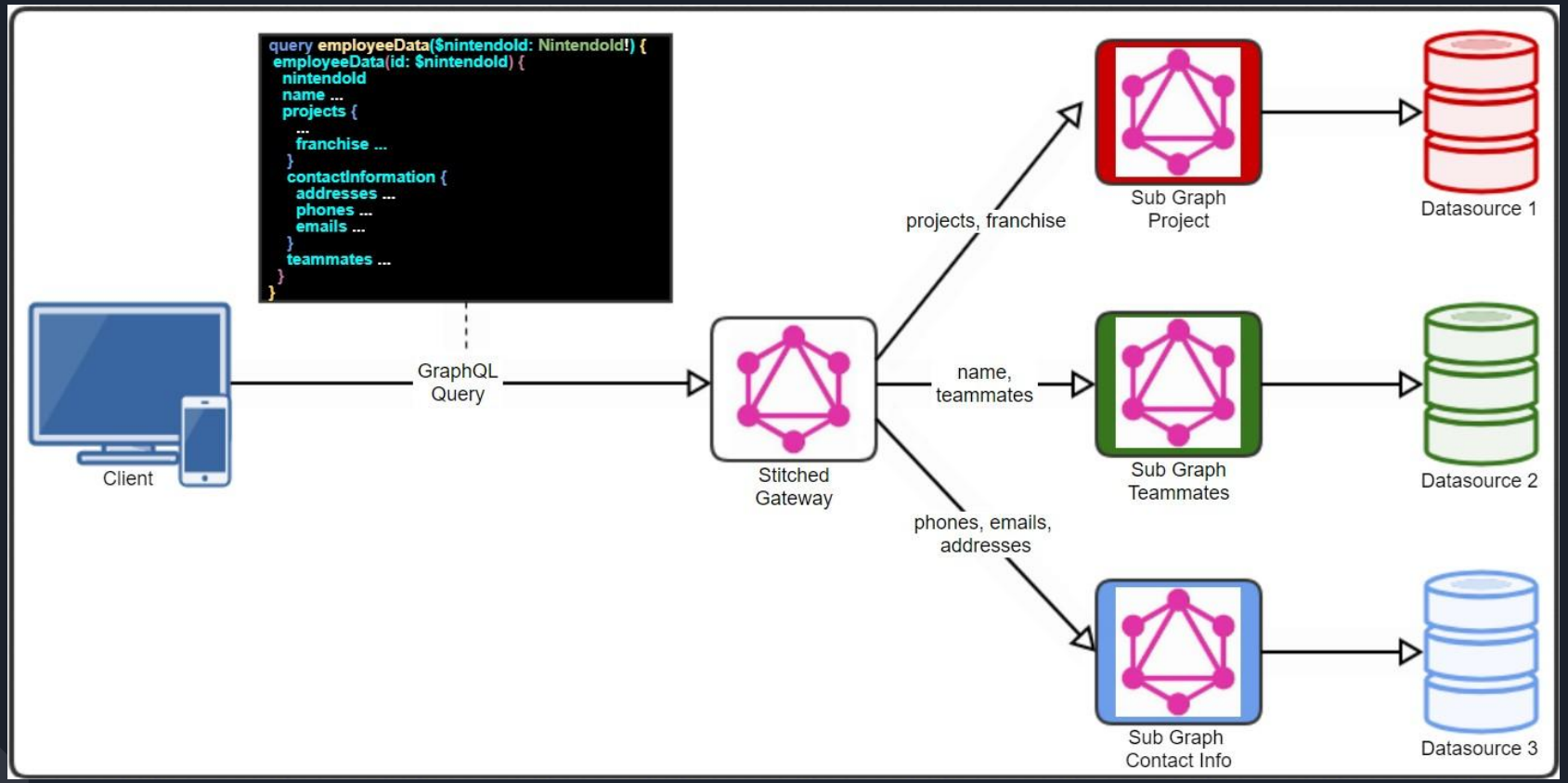
```
type Query {
    # Retrieve a Name associated to a Nintendo ID
    myName(nintendoId: NintendoId!): Name

    # Retrieve all Teammates associated to a Nintendo ID
    myTeammates(nintendoId: NintendoId!): [Teammate]
}
```

Schema delegation (graphql-tools.com)

Delegate to Schema



Schema delegation (graphql-tools.com)


query employeeData($nintendoId: NintendoId!) {
employeeData(id: $nintendoId) {
nintendoId
name ...
projects {
...
franchise ...
}
contactInformation {
addresses ...
phones ...
emails ...
}
teammates ...
}
}
Client
GraphQL
Query
Stitched
Gateway
projects, franchise
name,
teammates
phones, emails,
addresses
Sub Graph
Project
Sub Graph
Teammates
Sub Graph
Contact Info
Datasource 1
Datasource 2
Datasource 3

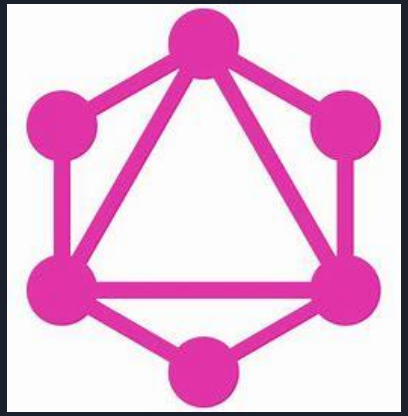
NINTENDO:
Fun with GraphQL Schema Stitching

Codebase: Fun-with-GraphQL-Schema-Stitching

PROS AND CONS - GraphQL

PROS

- Efficient, only fetch what you need
- One single API endpoint
- Self describing



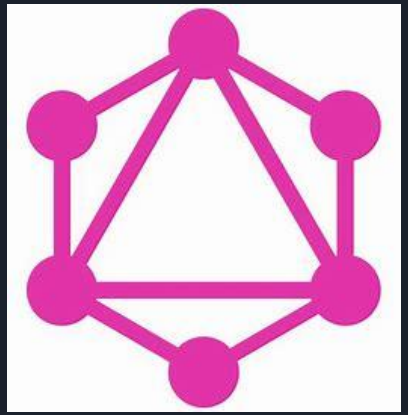
CONS

- Single endpoint usually means all code sits in one repository
- All code runs on a single server (not good for scalability)

PROS AND CONS - Schema Stitching

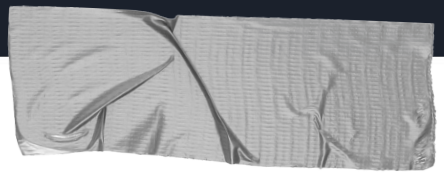
PROS

- Integrate with multiple GraphQL backends
  - One Graph to Rule them all
  - Easily hook in other HTTP protocols
- Extend existing fields to make new operations
- Backend can be divided across domains/teams
  - Easier to manage (no fighting among teams)
- Each individual server becomes scalable (Gateway is stateless)



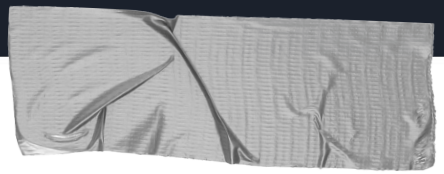
CONS

- Support API Gateway schema (if extending)
- Gateway requires resolvers (if extending)
- Some schemas can be hard to stitch
- Resolvers require coordination and possible redeployment to support new backend fields
- Increased latency



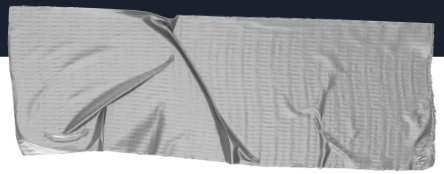
Resources

- ➔ How to GraphQL
- ➔ GraphQL Tools
- ➔ The Ultimate Guide to Schema Stitching
- ➔ Fun with GraphQL Schema Stitching
- ➔ GraphQL Security - Best practices



Resources

- ➔ Custom GraphQL Scalar Types
  - ◆ Helps with input schema validation
- ➔ GraphQL Depth Limit
  - ◆ Defense against unbounded queries
- ➔ Pagination using Relay Connection



Resources

- Type Merging
  - Stitching Types from different schemas
- Schema Directives
- Batch Delegation
- Data fetching
  - Data Loader and caching

Questions?