#7ujIntro to Graphql

From the get go

- graphql is a **communication standard**
- graphql is not a programing language
- objective:

> "...for **describing** the capabilities and requirements of data models for client-server applications"

- **self-documented**:

> Ensure that all of your data is statically typed and these types inform what queries the schema supports.

- **included deprecation mechanism**

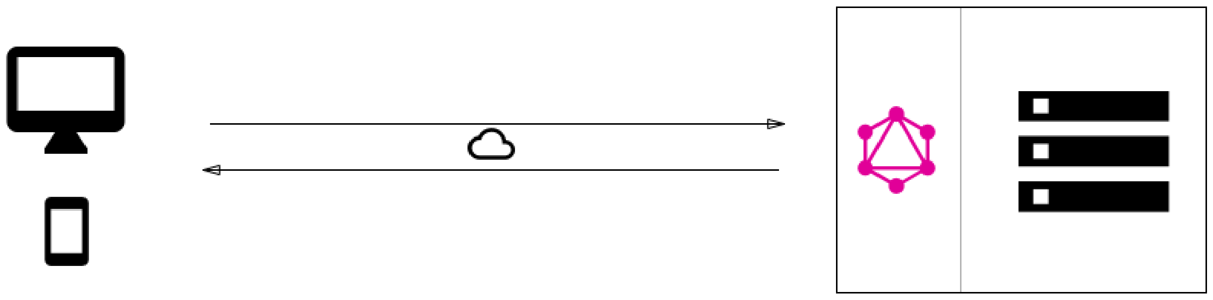
> Reduce the need for breaking changes, but utilize a built-in mechanism for deprecations when you need to.

- **data source Agnostic** "GraphQL does not mandate a particular programming language or storage system for application services that implement it"
- **you get what you ask for**:
  - GraphQL queries are **Field Sets**
  - field -> function **field resolver**

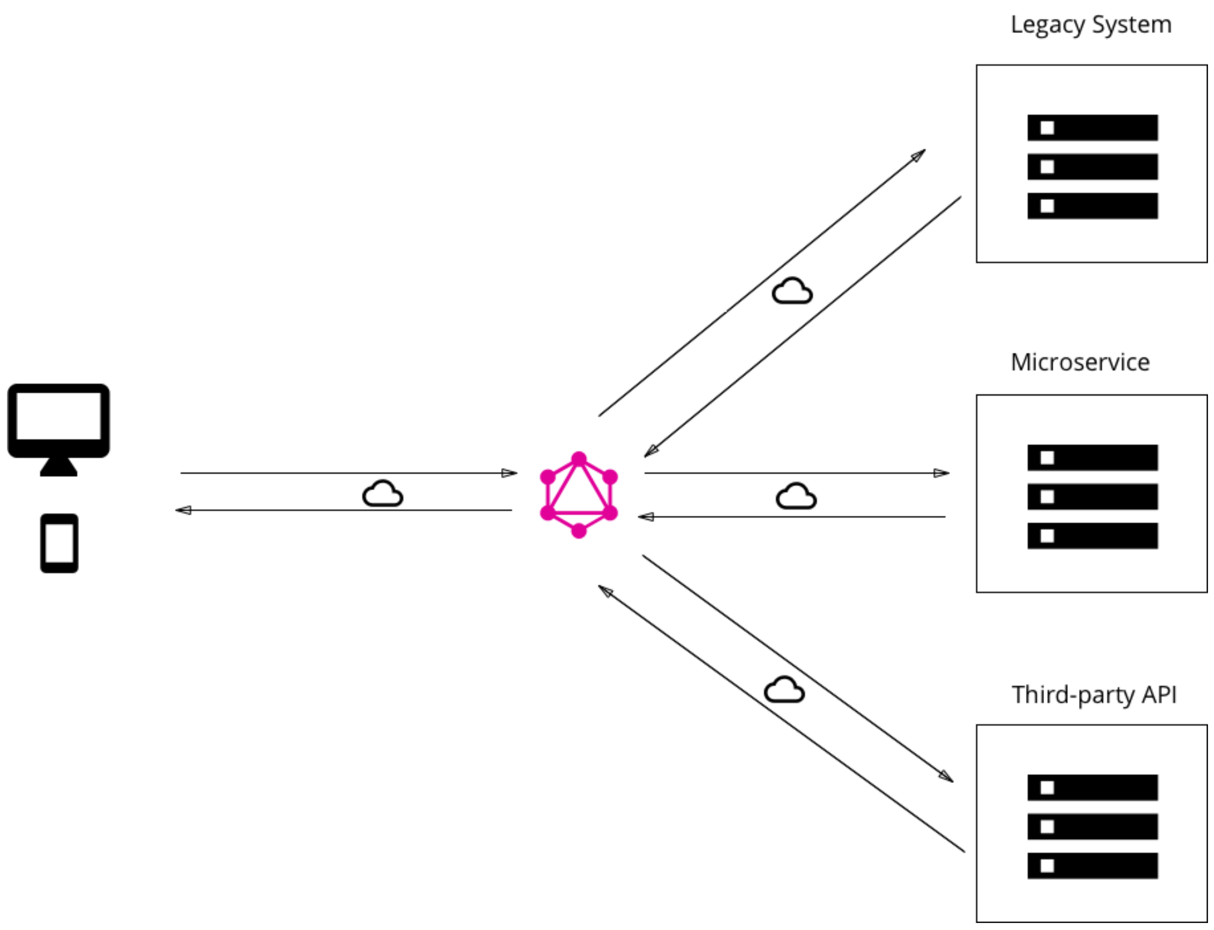
GraphQL **principles**:

1. Product-centric: ***GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them***.
   - "Client First", me, 2023
   - "designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.", GraphQL Spec, 2021
2. Hierarchical
3. Strong-typing
4. Client-specified response
5. Introspective

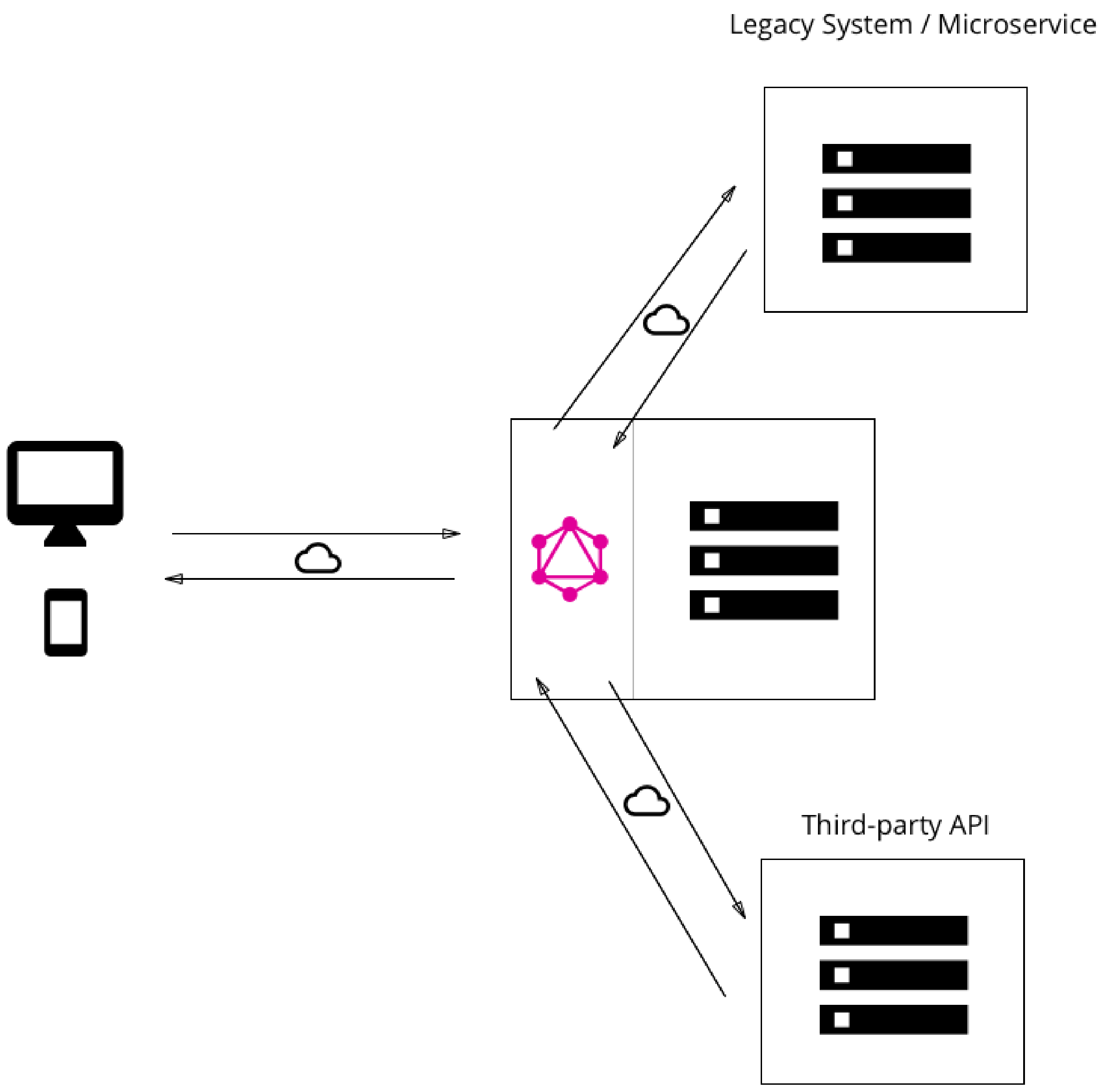
Architecture



GraphQL server and db in same vm serving mobile and web clients [1]



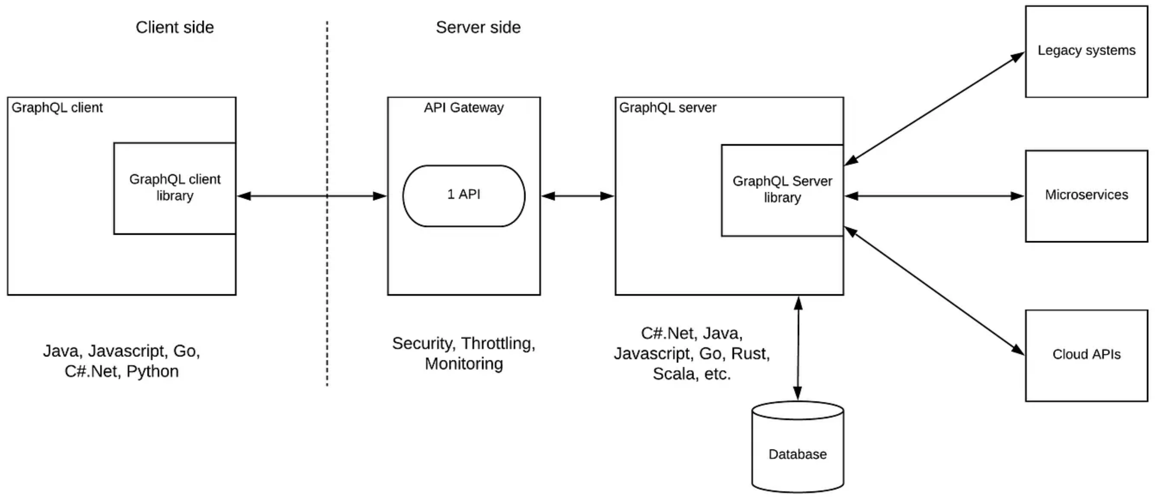
graphql server in dedicated 'orchestator' node in microservice arch with 3 different data sources [3]



graphql server with db in same vm while also orchestrating with two external data sources [3]

Authentication?

- TODO elaborate



[5]

~~ææ~~ ### Stitching

TODO elaborate

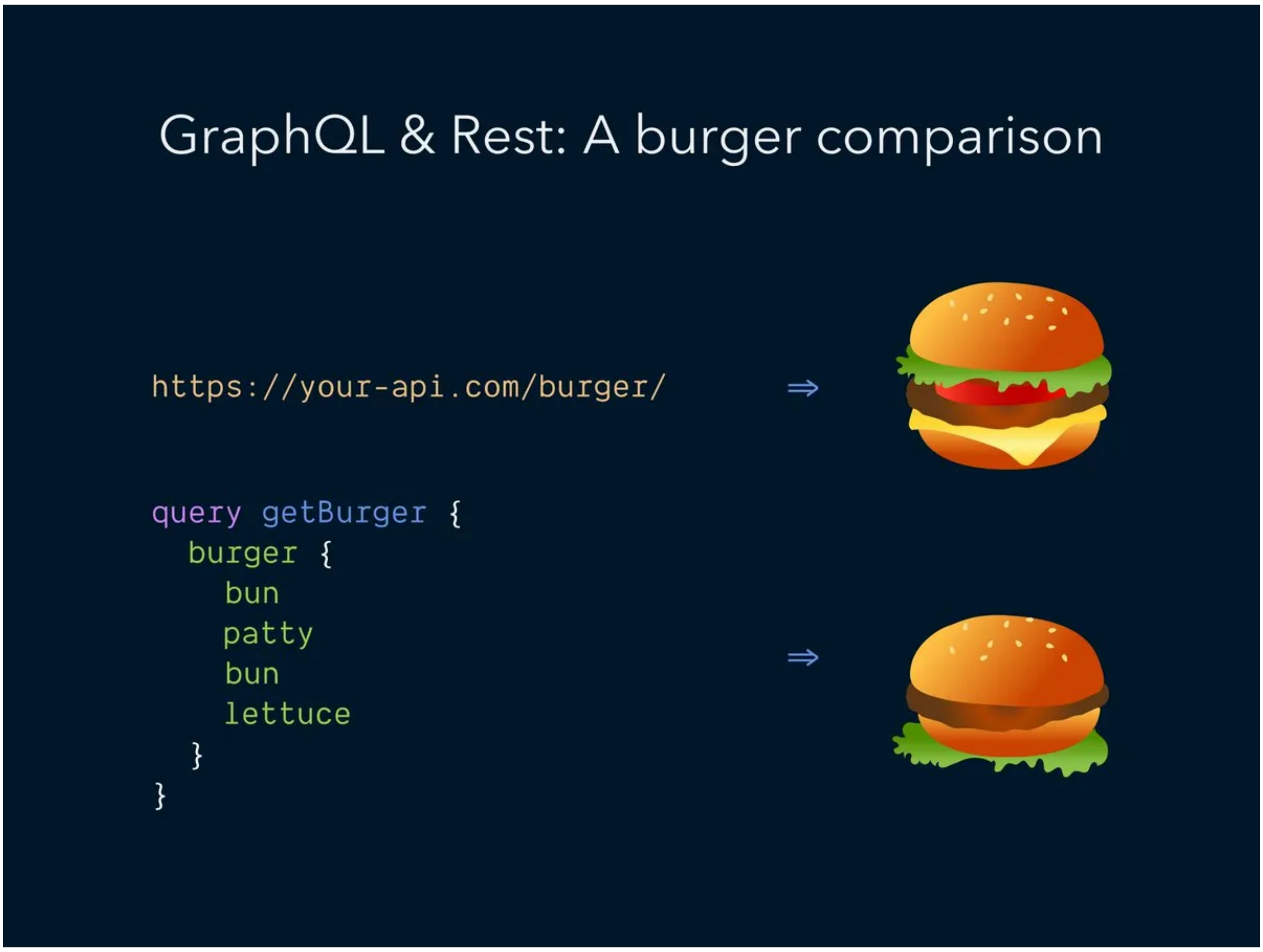
-
-

Performance

Dado que:

- ~~1 field -> ææ1 resolver function~~
- data batching on the server in stead of client -> less http calls for same data
- catered query for client -> allows for mutiple different clients, same endpoint fullfills different needs

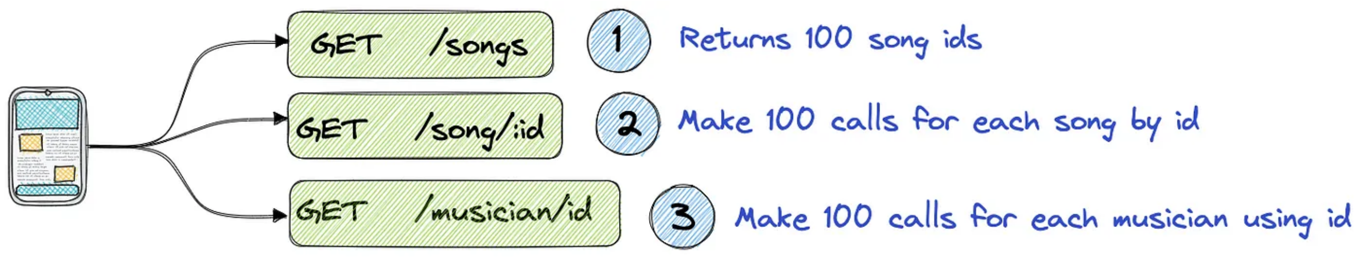
Entonces: ==> performance improvements in frontend:

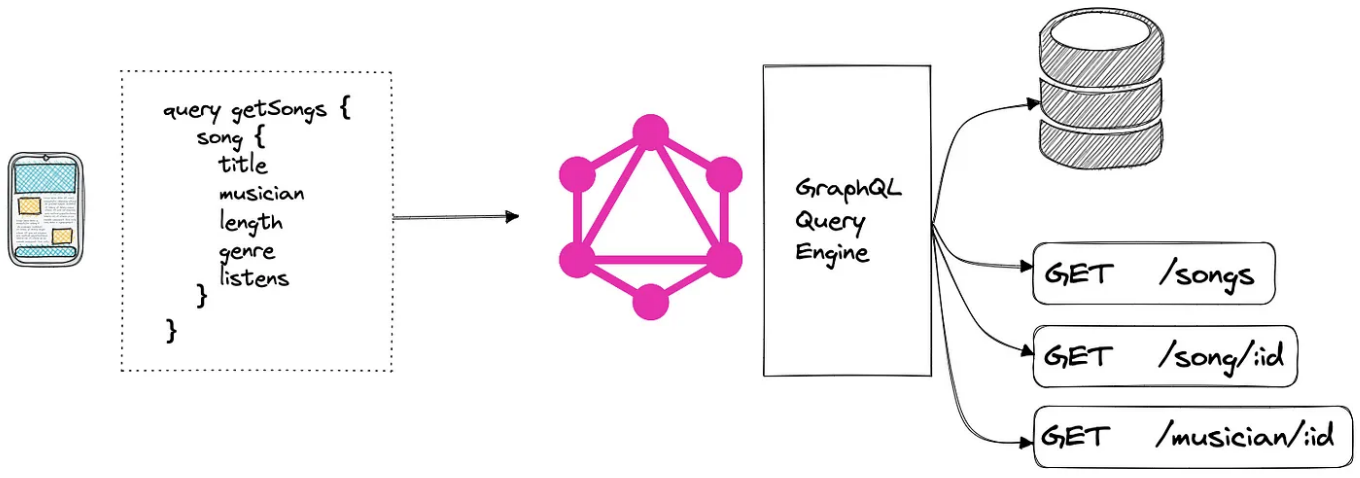


GraphQL & Rest: A burger comparison

```
https://your-api.com/burger/        ⇒

query getBurger {
  burger {
    bun
    patty
    bun
    lettuce
  }
}                                   ⇒
```


GET /songs
GET /song/:id
GET /musician/id
1 Returns 100 song ids
2 Make 100 calls for each song by id
3 Make 100 calls for each musician using id



---

if

> repeatedly load data from your database.

Then,

> implement batching technique or DataLoader.

Language

Schema Definition:

- type

  ```
  type Person {
      name: String
      age: Int
      picture: Url
  }
  ```

- interface

  ```
  interface Book {
      title: String!
      author: Author!
  }
  type Textbook implements Book {
  ```

```
    title: String! # Must be present
    author: Author! # Must be present
    courses: [Course!]!
}
```

- union

```
union SearchResult = Book | Author

type Query {
    search(contains: String): [SearchResult!]
}

query GetSearchResults {
    search(contains: "Shakespeare") {
        __typename
        ... on Book {
        title
        }
        ... on Author {
        name
        }
    }
}
```

```
{
    "data": {
        "search": [
        {
            "__typename": "Book",
            "title": "The Complete Works of William Shakespeare"
        },
        {
            "__typename": "Author",
            "name": "William Shakespeare"
        }
        ]
    }
}
```

- enum

```
enum CardinalDirection {
NORTH
EAST
SOUTH
WEST
}
```

- input objects
- non-null ``` name: String! ``
- Field Arguments

```
type Person {
    name: String
    picture(size: Int): Url
}
{
    name
    picture(size: 600)
}
```

- query: a read-only fetch.

```
type Query {
    books: [Book!]!
    }
    query GetBooks {
    books {
        title
        author
    }
}
```

- mutation: a write followed by a fetch.

```
mutation {
    likeStory(storyID: 12345) {
        story {
            likeCount
        }
    }
}
mutation {
    sendEmail(message: "Hello,\n  World!\n")
}
```

- subscription: a long-lived request that fetches data in response to source events.
    - web sockets

- support for EDD

Fields and Field Resolvers

- Selection Set
- Field Alias

Fragments

- primary unit of composition
- recycle and reuse common pieces of queries
- inline fragments ???

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

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

Instrospection

```
{
  __type(name: "Droid") {
    name
    fields {
      name
      type {
        name
        kind
      }
    }
  }
}
{
  "data": {
    "__type": {
      "name": "Droid",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "name",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "friends",
          "type": {
            "name": null,
            "kind": "LIST"
          }
        },
        {
          "name": "friendsConnection",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "appearsIn",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "primaryFunction",
          "type": {
            "name": "String",
            "kind": "SCALAR"
          }
        }
      ]
    }
  }
}
```

Sources

1. GraphQL Spec October2021
2. howtographql.com: Big Picture (Architecture)

3. Solution Architects Guide to GraphQL
4. Introduction to GraphQL
5. https://chanakaudaya.medium.com/graphql-based-solution-architecture-patterns-8905de6ff87e
6. GraphQL.org: Instrospection
7. Apollo Server: Union and Interfaces
8. 12 Microservices Patterns I Wish I Knew Before the System Design Interview