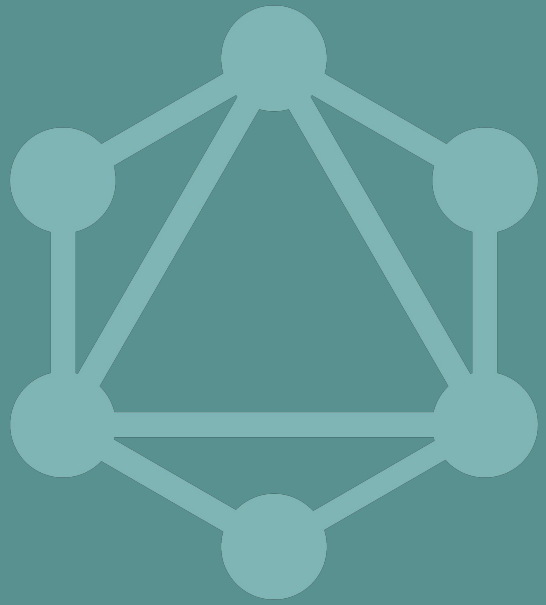
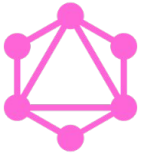
An Overview of GraphQL



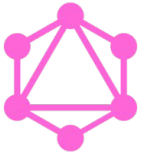
v0.1.0

April 30, 2019

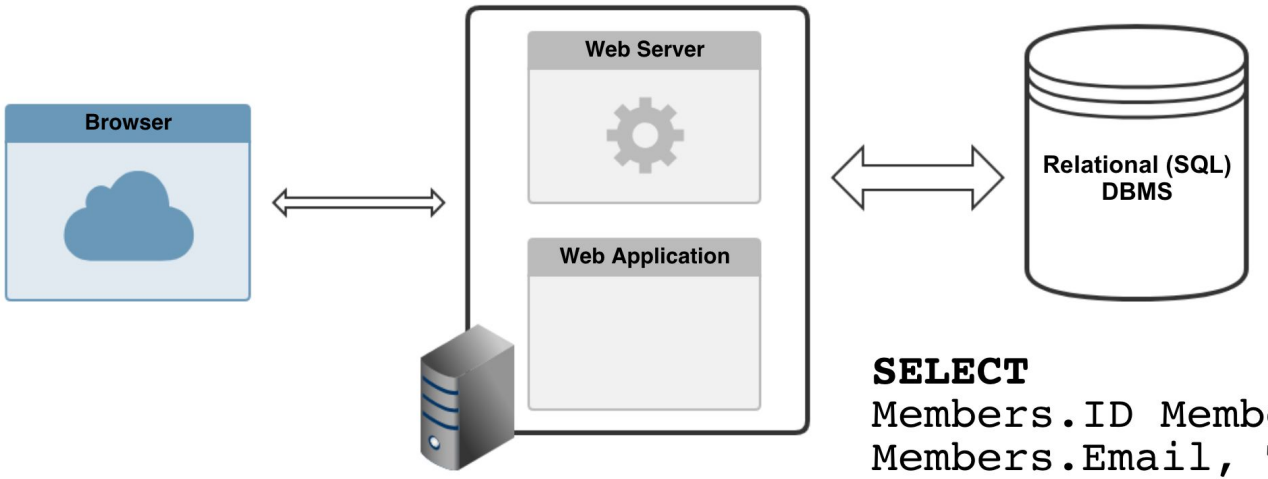


Some History

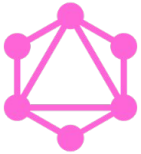
- **Why** and **How** is GraphQL created?



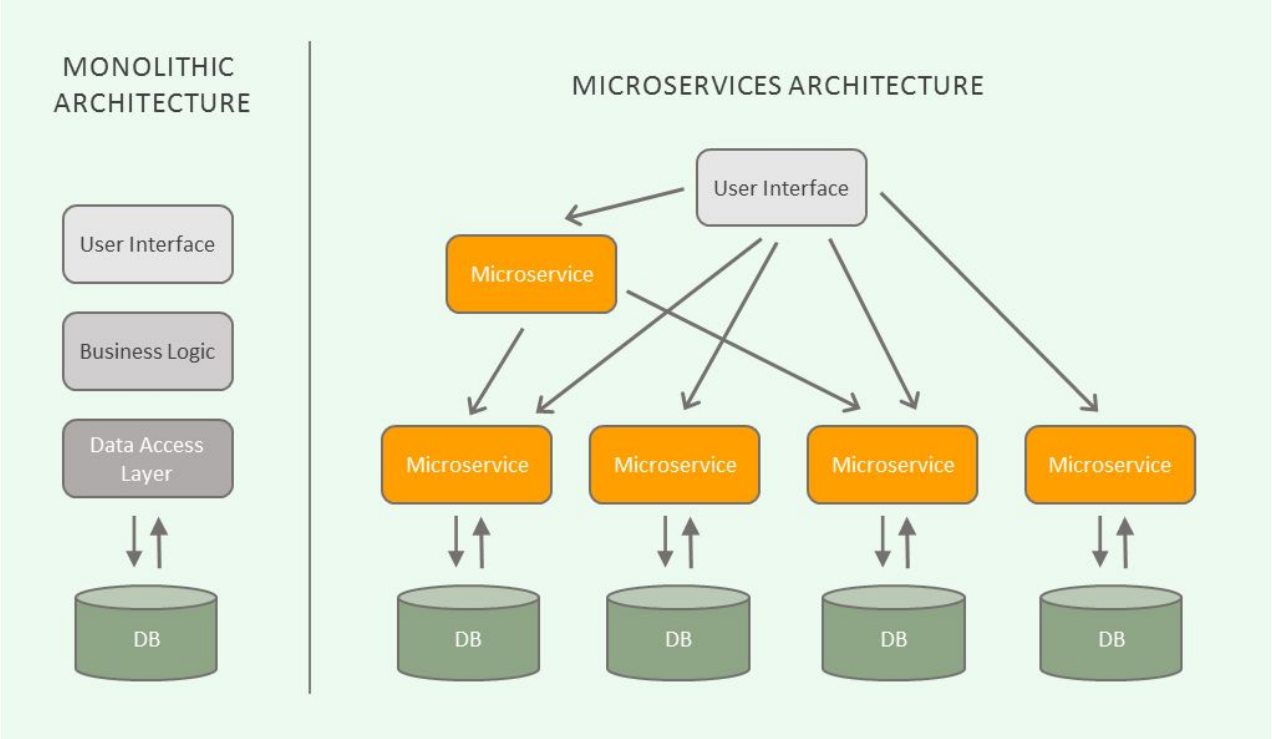
Monolithic Architecture

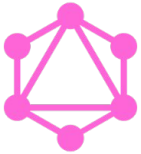



```
SELECT
Members.ID Members,Name,
Members.Email, Teams.Name
FROM Members JOIN Teams
ON Member.TeamID = Team.ID
```



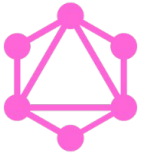
Microservices Architecture





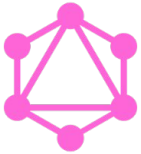
Where are Joins in Microservices?!

- If you are doing **joins** in your application, you are doing it **wrong**!
- Clients make more requests to get all data they want:
- An Example:
  - GET /api/v1/teams/1234
  - GET /api/v1/members?teamId=1234



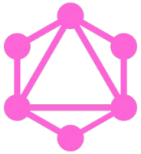
Background

- GraphQL is the response to challenges created by microservices.
- GraphQL was developed by Facebook in 2012 and open sourced in 2015.
- Other companies have created similar solutions (Netflix's Falcor).



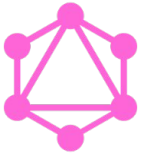
Who Uses GraphQL?

- Facebook
- GitHub
- Twitter
- Airbnb
- Pinterest
- Shopify
- Coursera
- https://graphql.org/users



What is GraphQL?

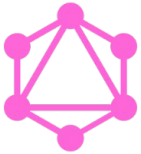
- GraphQL is a **query language** for your API.
- GraphQL is also a **runtime** for executing queries.



GraphQL in Action!

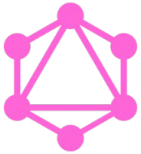
- https://www.graphqlhub.com

GraphQL Benefits



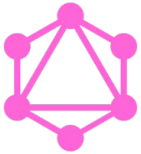
GraphQL is Simple!

- One of the challenges with REST API is input & output values!
  - Input: http verbs, url path, headers, query params, body
  - Output: status code vs. headers vs. response body
- With GraphQL you only send a single query.
  - It can be over tcp, http, protocol buffers, etc.
- Your **query shape** is also your **response shape**!



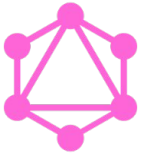
No Underfetching, No Overfetching!

- With REST API,
  - clients have to make a series call (underfetching)
  - a lot of unwanted data are also fetched (overfetching)
  - what does this mean for mobile apps?
- With GraphQL,
  - You only get what you exactly need in a **single request**!



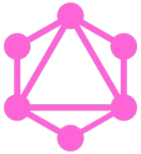
GraphQL is Typed

- One of the biggest challenges with REST API is interpretation!
  - http status codes!
  - undefined vs. null vs. empty
  - required vs. non-required values
- GraphQL is a **strongly typed** language!
  - nullable and non-nullable types



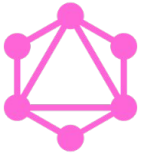
Your Data Model is a Graph!

- Each non-scalar (object) type is a node.
- The non-scalar types on your queries are the edges.
- Each query represents a path in your data model graph.



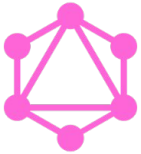
No API Versioning

- Deprecating and versioning REST APIs are challenging!
  - When we don’t have a type system, and
  - we don’t have any control over the response is returned,
    - any change could be a breaking change!
- GraphQL is an **ever-evolving** and **versionless** API paradigm.
  - Avoid backward-incompatible changes.
  - Old fields can be easily deprecated with a description.



Query Validation

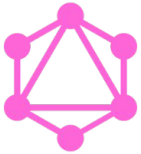
- GraphQL queries can be pre-determined and validated thanks to the GraphQL type system (schema).
- When implementing a back-end or a front-end GraphQL application, many problems and mistakes can be detected without waiting for runtime errors and debugging!



API Introspection

- We can discover a GraphQL API dynamically or programmatically.
- We can access the documentation of a GraphQL API using introspection.
- We can even introspect on the introspection system itself!

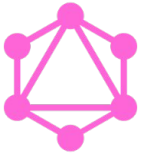
Schema Language



Schema Language

- It is a **type language** for defining a **type system**!
  - You basically define a set of types!
- Every schema starts as follows:

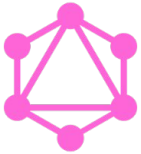
```
schema {
  query: Query
  mutation: Mutation
}
```



The Query Type

- Query is a special type that defines your queries!

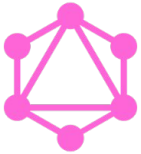
```
type Query {
  team(id: ID!): Team
  teams: [Team!]!
  members(teamId: ID!): [Member!]
}
```



The Mutation Type

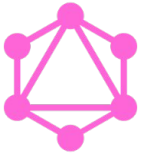
- Mutation is a special type that defines your mutation queries!

```
type Mutation {
  addTeam(name: String!): Team!
  addMember(teamId: ID!, name: String!, email: String): Member!
}
```



Scalar Types

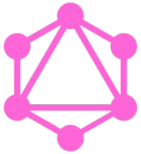
- ID
- Int
- Float
- String
- Boolean



Enumeration Types

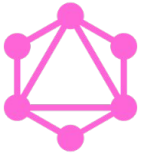
- Enumeration types are a special kind of scalar type!

```
enum Status {
  Pending
  Approved
  Cancelled
}
```



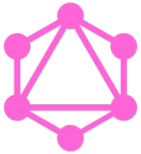
List and Non-Null

- myField: [String]
- myField: [String!]
- myField: [String]!
- myField: [String!]!



Object Types

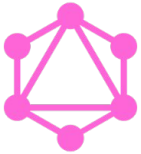
```
type Team {
  id: ID!
  name: String!
  members: [Member!]
}

type Member {
  id: ID!
  team: Team!
  name: String!
}
```



Arguments

- Every GraphQL field can have zero or more arguments.

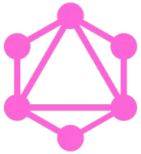
```
type Car {
  id: ID!
  model: String!
  length(unit: LengthUnit = METER): Float
  weight(unit: WeightUnit = Pound): Float
}
```



Input Types

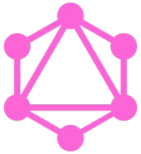
- Input types can only be used for arguments.

```
type Mutation {
  addMember(in: MemberInput!): Member!
}

input MemberInput {
  teamId: ID!
  name: String!
  email: String
}
```



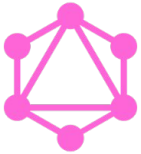
Interfaces

```
interface Member {
  id: ID!
  name: String!
}

type Employee implements Member {
  id: ID!
  name: String!
  employeeId: ID!
}
type Customer implements Member {
  id: ID!
  name: String!
  customerId: ID!
}
```



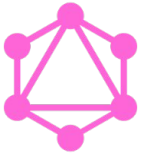
Union Types

```
union SearchResult = Profile | Page

Type Profile {
  id: ID!
  name: String!
}

type Page {
  id: ID!
  name: String!
  business: String!
}
```

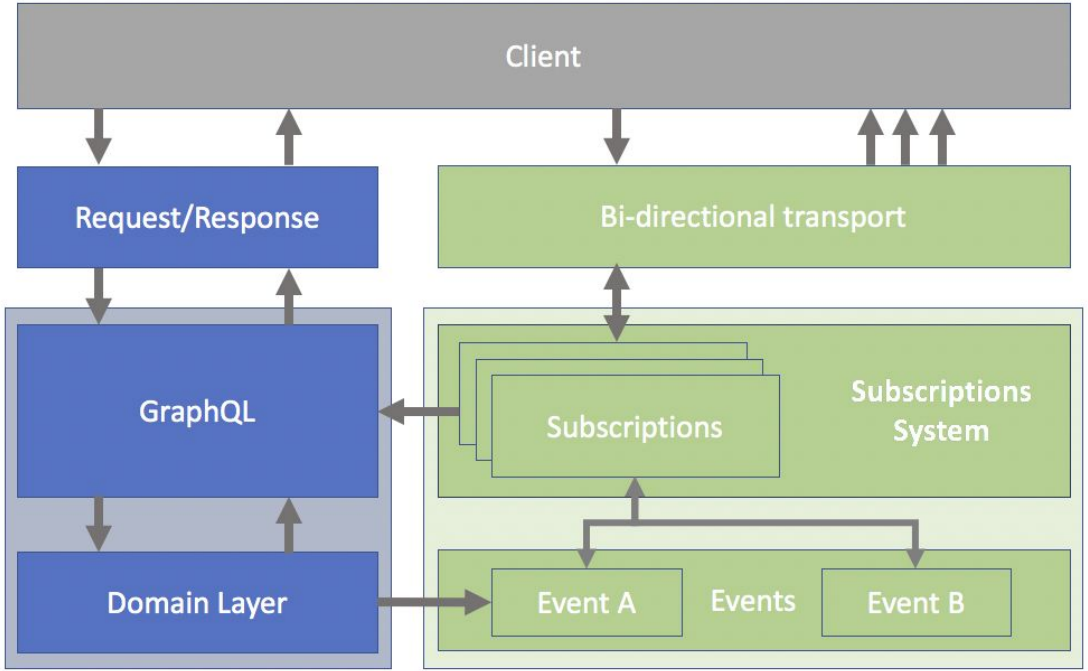


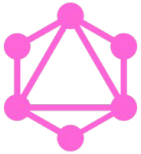
Subscriptions

- Clients can open a **long-lived** connection to back-end.
- When subscribing, clients specify:
  - What events they are interested in, and
  - What query should be executed when events occur.
- The server maps the inputs to an event stream and executes the query when the events trigger.
- This model avoids **overpushing/underpushing** but requires a GraphQL backend.



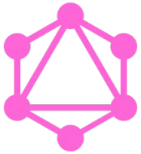
Subscriptions





Resolvers

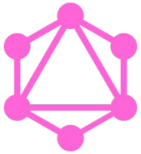
- For each field on each type, you define a **resolver function**.
- Resolver functions collectively implement your GraphQL API.



Resolver Example

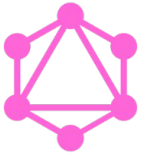
```
Query: {
  team(obj, args, context, info) { … }
}

Mutation: {
  addTeam(obj, args, context, info) { … }
}

Team: {
  id:   t => t.id
  name: t => t.name
}
```

Query Language



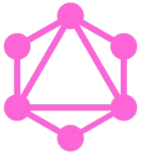
Fields

```
{
  teams {
    name
    members {
      name
    }
  }
}
```



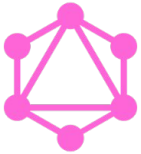
Arguments

```
{
  members(teamId: "1234") {
    name
    team {
      name
    }
  }
}
```



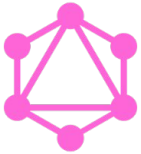
Aliases

```
{
  firstMember: members(teamId: "1234") {
    fullName: name
  }
  secondMember: members(teamId: "5678") {
    fullName: name
  }
}
```



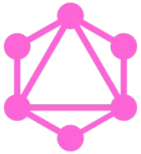
Operation Names

```
query TeamsAndMembers {
  teams {
    name
    members {
      name
    }
  }
}
```



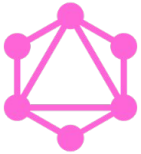
Mutations

```
mutation AddMember{
  addTeam(teamId: "1234", name: "Milad", email: "milad@example.com"){
    id
    name
  }
}
```



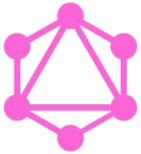
Variables

```
mutation AddMember($in: MemberInput!){
  addTeam(input: $in){
    id
    name
  }
}

{
  "in": {
    "teamId": "1234",
    "name": "Milad",
    "email":"milad@example.com"
  }
}
```



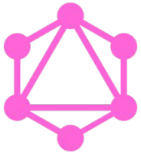
Directives

```
query TeamsAndMembers($withMembers: Boolean!) {
  teams {
    name
    members @include(if: $withMembers) {
      name
    }
  }
}

{
  "withMembers": false
}
```



Fragments

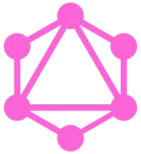
```
query {
  firstMember: members(teamId: "1234") {
    ... infoFields
  }
  secondMember: members(teamId: "1234") {
    ... infoFields
  }
}

fragment infoFields on Team {
  id
  name
}
```



Inline Fragments

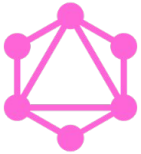
```
query FindFood {
  search(ingredient: "protein") {
    __typename
    ... on Fruit {
      name
    }
    ... on Meal {
      name
      calories
    }
    ... on Drink {
      brand
    }
  }
}
```

Best Practices



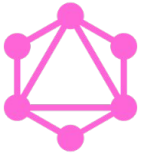
Think in Graph!

- Think of your data model (resources) and your API as a **graph**!
- “With GraphQL, you model your business domain as a graph”
- You need a common terminology for choosing names that make sense (intuitive APIs)!



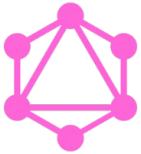
Transport Protocol

- HTTP
  - GET https://api.example.com/graphql?query={ ...}
  - POST https://api.example.com/graphql
- GZIP Encoding
  - Accept-Encoding: gzip
- JSON format for response



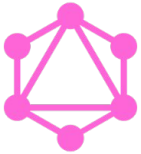
Pagination

- You can use any pagination model for your GraphQL schema
- Different pagination models enable different client capabilities.
- Implement pagination from day zero!

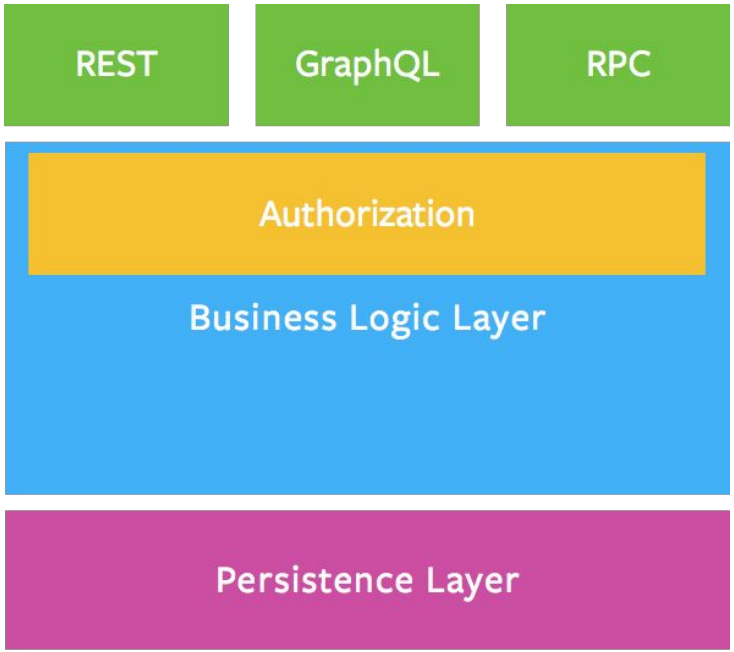


Authentication and Authorization

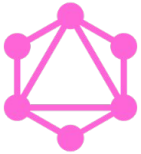
- Authentication and authorization should be implemented in business logic layer.
- The **business logic** layer should act as the **single source of truth** for enforcing business domain rules.



Authentication and Authorization

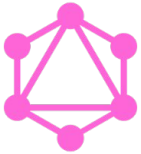


GraphQL Challenges



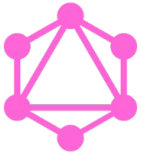
Caching

- Caching in REST is easy!
  - Resources are represented by uuid or guid.
  - The response for each resource has the same fields.
- Similarly, caching gRPC requests are fairly easy!
- In GraphQL, the response for the same query on the same resource id can have many different shapes!
- GraphQL community is putting a lot effort on this topic!



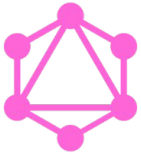
Rate Limiting and Profiling

- REST and gRPC are easy to measure and profile!
  - Each request has a known (usually fixed) cost.
- The cost of a GraphQL request depends on the query!
  - it may need one call to database, or
  - it may need tens of calls to different databases!



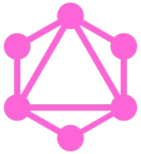
Schemas in Microservices!

- In microservices world, we want our microservices to fully and independently own their slice of schema.
- **Distributing** and **decentralizing** GraphQL schemas in microservices architecture is a fun challenge!
- **Load balancing** complexity depends on the transport layer.
- **Routing** cannot happen in transport layer!
    - The router should understand the GraphQL schema!



Schema Stitching

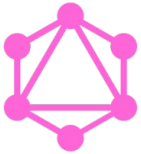
- Schema stitching is the art of composing a single unified GraphQL schema from multiple independent schemas.
  - creating a single connected graph from multiple disconnected graph!
- What is hard about stitching?
- The stitcher should take care of routing GraphQL queries.



Stitching by Convention

```
Type Team {
  id: ID!
  name: String!
}

Type Member {
  id: ID!
  teamId: ID!
  name: String!
  email: String
}
```

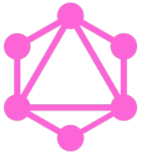
```
Type Team {
  id: ID!
  name: String!
  members: [Member!]
}

Type Member {
  id: ID!
  teamId: ID!
  team: Team!
  name: String!
  email: String
}
```



Stitching by Configuration

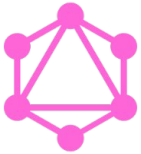
```
Type Team {
  id: ID!
  name: String!
}

Type Member {
  id: ID!
  teamId: ID!
  name: String!
  email: String
}
```

```
extend type Team {
  members: [Member!]
}

extend type Member {
  team: Team!
}
...
Resolvers = {
  Team: {
    members: ...
  }
  Member: {
    team: ...
  }
}
```

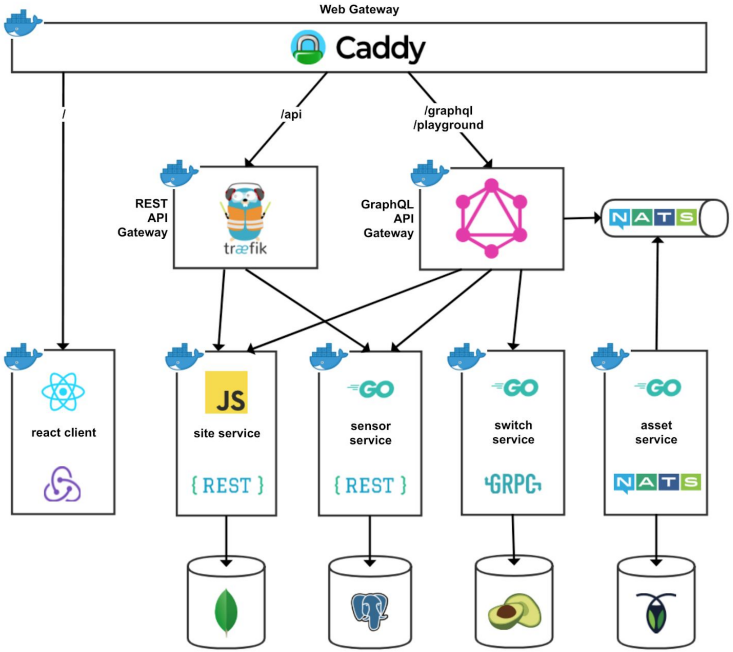


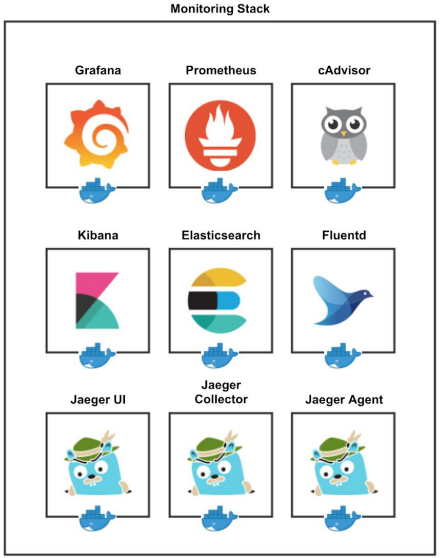
Decentralizing by Choreography!

```
Type Team {
  id: ID!
  name: String!
  org: Organization!  @Link org-service
  members: [Member!]  @Link member-service
}

Type Member {
  id: ID!
  teamId: ID!
  team: Team!  @Link team-service
  name: String!
  email: String
}
```

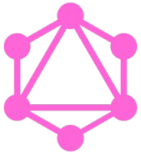


A Practical Approach github.com/moorara/microservices-demo



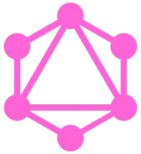


GraphQL Implementations



Runtime (Back-End)

- Go: **graphql**, graphql-go, gqlgen, ...
- Node.js: **graphql-js**, graphql-tools, apollo-server, ...
- Rust, Elixir, Clojure, Ruby, Python, Scala, Java, C#, PHP, ...



Query Language (Front-End)

- Relay (react-relay)
  - High learning curve
  - Very opinionated (React and React Native ecosystem)
  - Query validation, optimization, and compiling
- Apollo (apollo-client)
  - Framework-agnostic (React, Vue, Ember, iOS, Android, ...)
  - Focused on ease of use and very flexible
  - Subscription support via WebSockets