



Describe your data

```
type Project {  
  name: String  
  tagline: String  
  contributors: [User]  
}
```

Ask for what you want

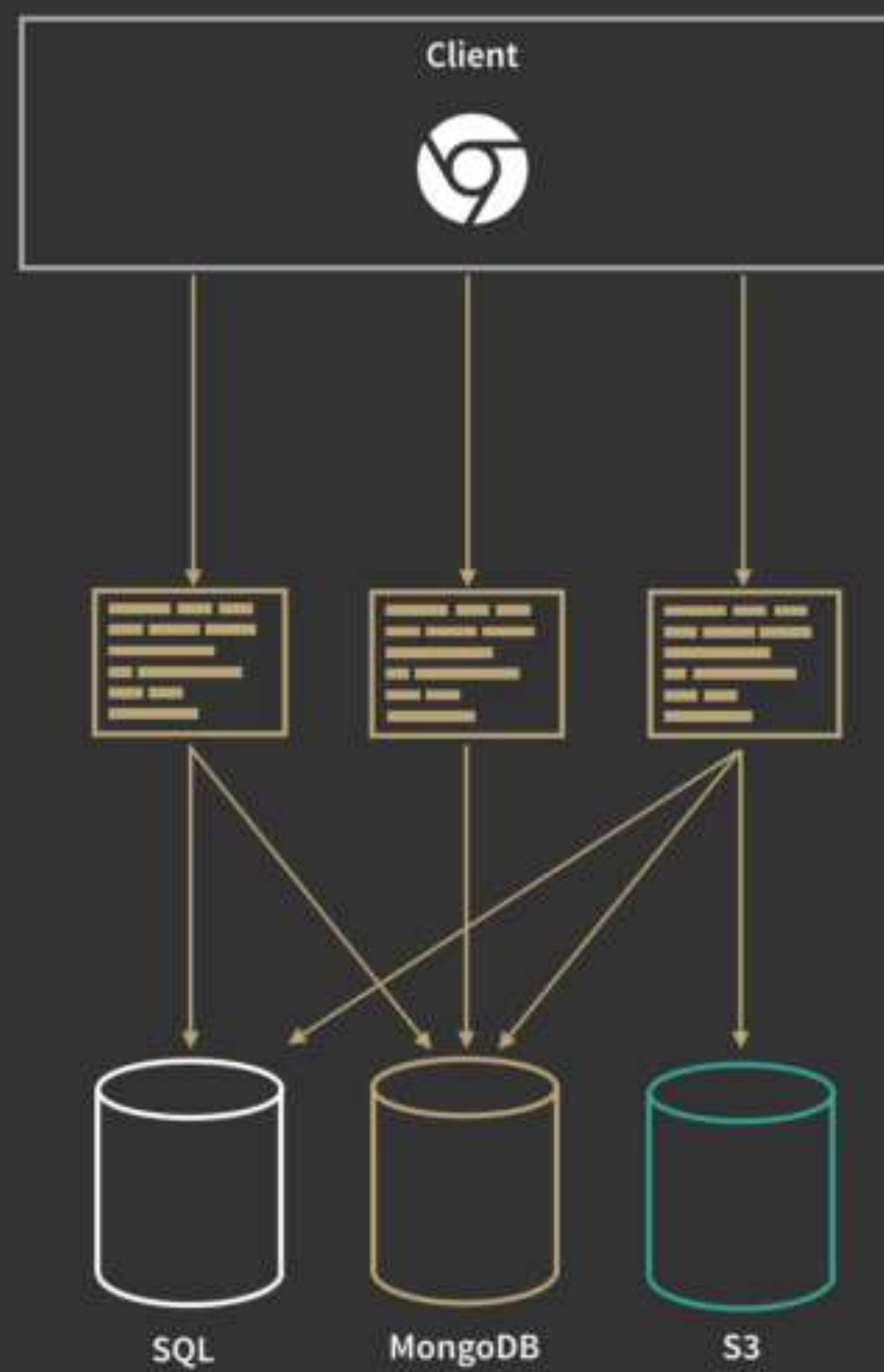
```
{  
  project(name: "GraphQL") {  
    tagline  
  }  
}
```

Get predictable results

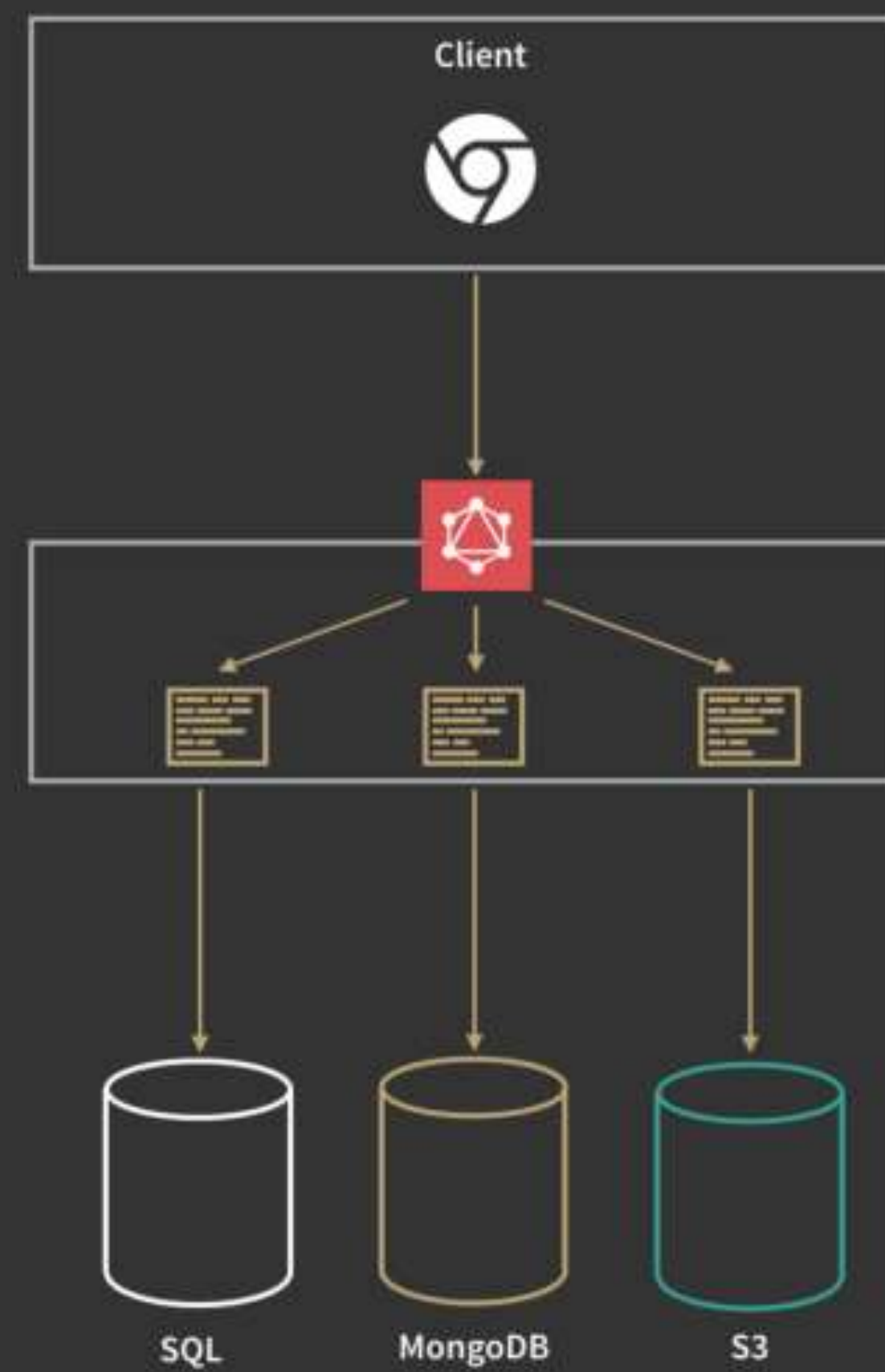
```
{  
  "project": {  
    "tagline": "A query language for APIs"  
  }  
}
```

Think GraphQL

R E S T



G R A P H Q L



 = Arbitrary code

GraphQL in a nutshell

GraphQL is a syntax that describes how to ask for data, and is generally used to load data from a server to a client. GraphQL has three main characteristics:

- It lets the client specify exactly what data it needs.
- It makes it easier to aggregate data from multiple sources.
- It uses a type system to describe data.

GraphQL Users



GraphQL History

GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015. On 7 November 2018, the GraphQL project was moved from Facebook to the newly-established GraphQL foundation, hosted by the non-profit Linux Foundation.


Background


Imagine you need to display a list of posts, and under each post a list of likes, including user names. Easy enough, you tweak your posts API to include a likes array containing user objects

```
[
  {
    title: 'My First Post',
    likes: [
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      },
      {
        name: 'Mike',
        avatar: 'mike.jpg'
      }
    ]
  },
  {
    title: 'Another Great Post',
    likes: [
      {
        name: 'Julia',
        avatar: 'julia.jpg'
      },
      {
        name: 'Sacha',
        avatar: 'sacha.jpg'
      }
    ]
  }
]
```

My First Post


Liked by:


 Sacha

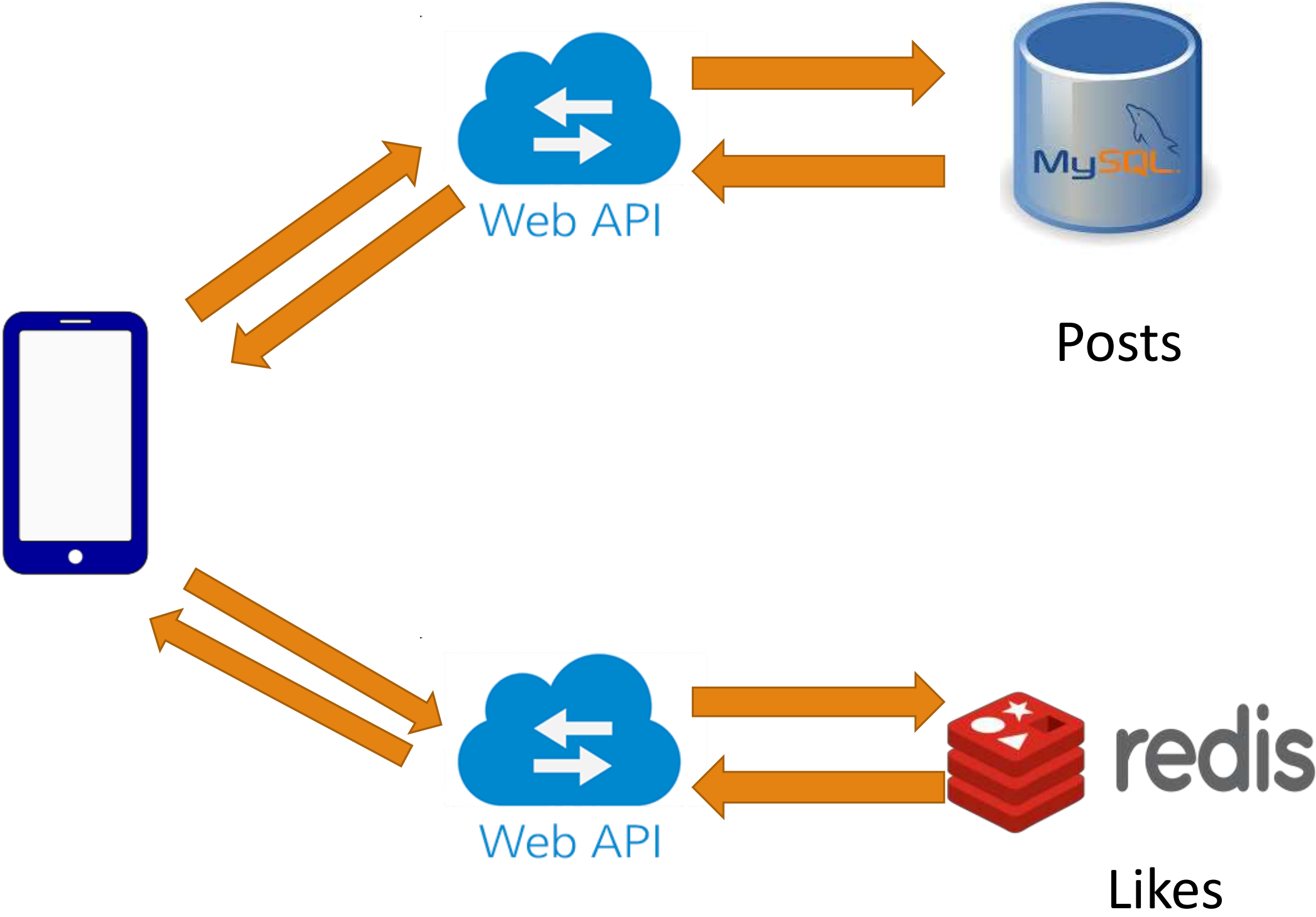
 Mike

Another Great Post

Liked by:

 Julia

 Sacha



Traditional REST APIs

The Solution

Instead of having multiple “dumb” endpoints, have a single “smart” endpoint that can take in complex queries, and then massage the data output into whatever shape the client requires.

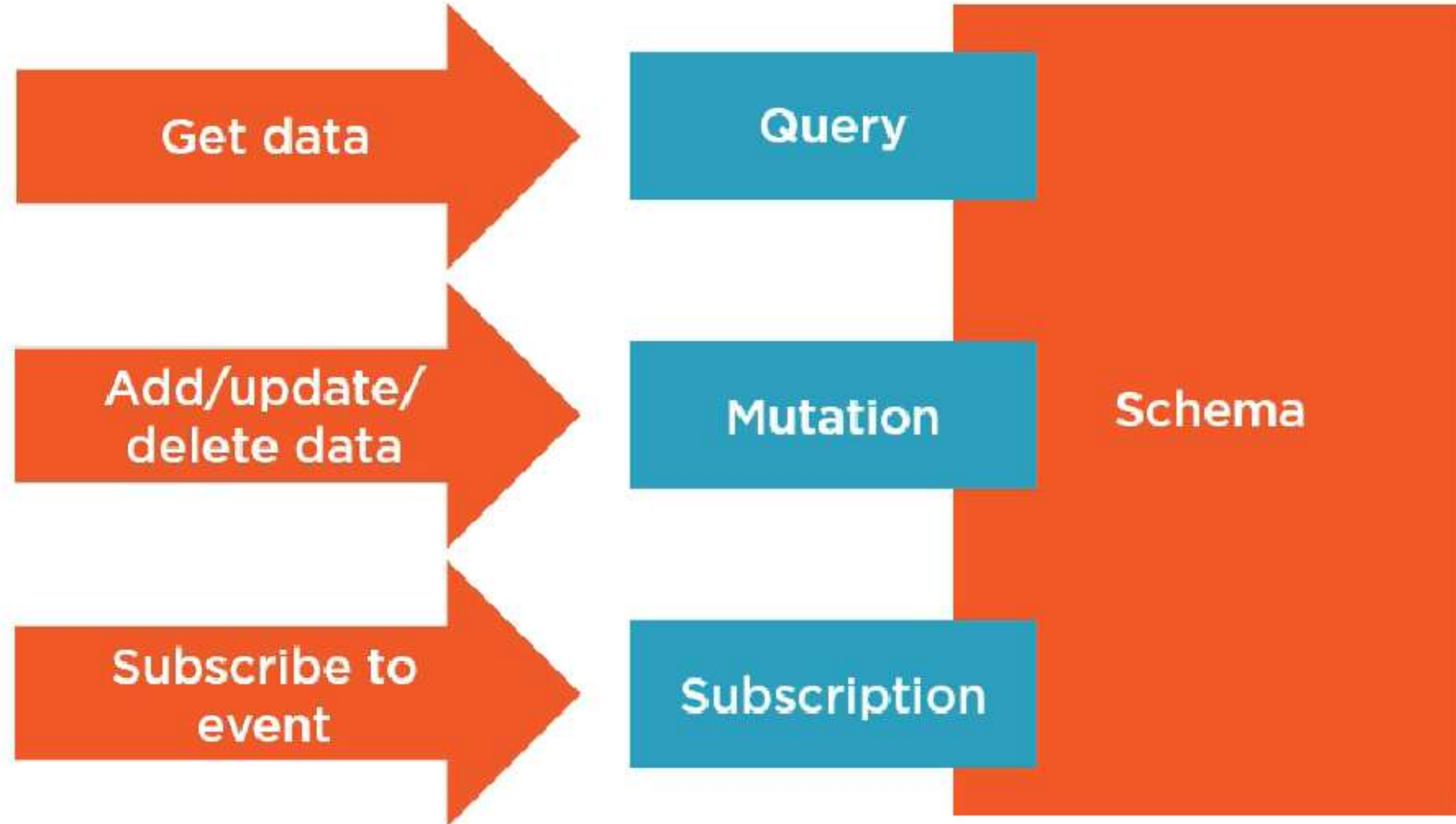
The GraphQL layer lives between the client and one or more data sources, receiving client requests and fetching the necessary data according to your instructions.

The Client can retrieve the data it needs in a single round-trip to the API gateway.



GraphQL magic

- A GraphQL schema is a strongly typed schema.
- GraphQL speaks to the data as a Graph, and data is naturally a graph.
- GraphQL has a declarative nature for expressing data requirements.
- GraphQL solved the multiple round-trip problem.



Components

Query

A GraphQL query is the client application request to retrieve data from database or legacy API's.

Resolver

Schema

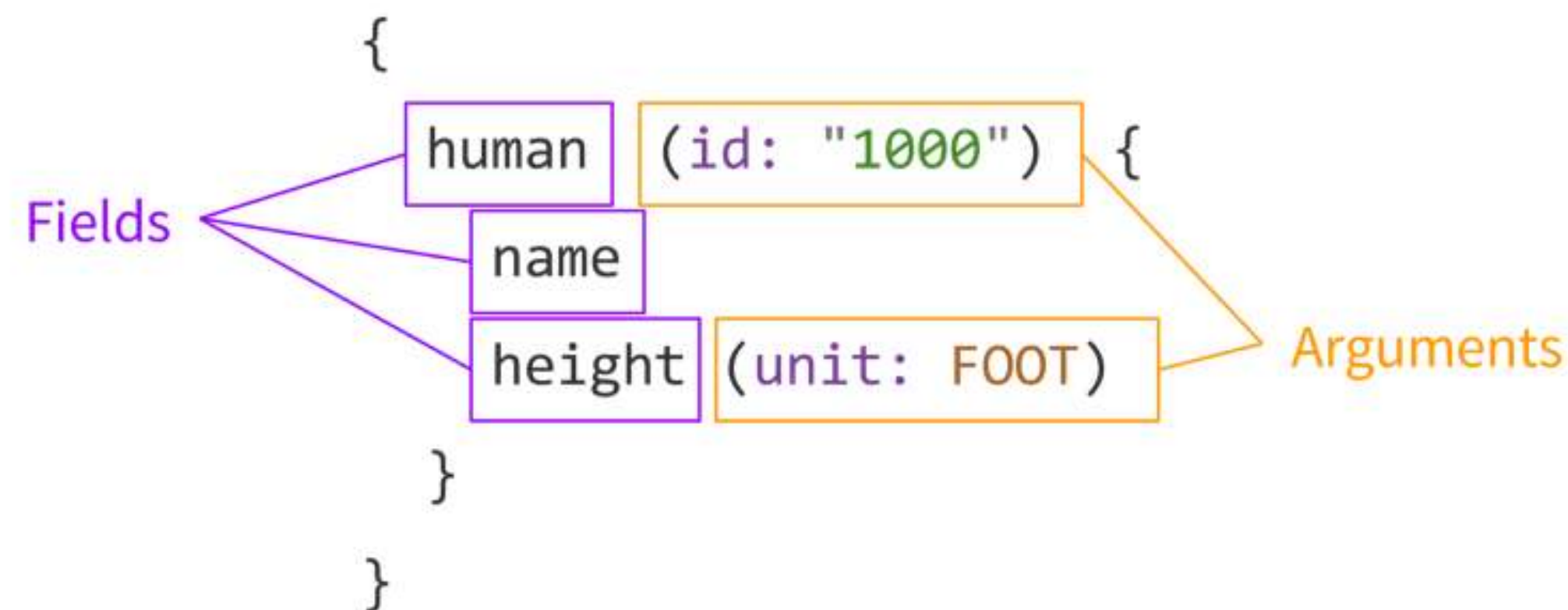
A GraphQL schema is at the center of any GraphQL server implementation and describes the functionality available to the clients which connect to it.

Resolvers

provide the instructions for turning a GraphQL operation into data. They resolve the query to data by defining resolver functions.

GraphQL queries

- GraphQL document: A string written in the GraphQL language that defines one or more operations and fragments.
- Operation: A single query, mutation, or subscription that can be interpreted by a GraphQL execution engine.
- Field: A unit of data you are asking for, which ends up as a field in your JSON response data.
- Arguments: A set of key-value pairs attached to a specific field. These are passed into the server-side execution of this field, and affect how it's resolved.



GraphQL queries Cont.

Operation type: This is either query, mutation, or subscription.

Operation name: For debugging and server-side logging reasons, it's useful to give your queries meaningful names.

Variable definitions: When you send a query to your GraphQL server, you might have some dynamic parts that change between requests, while the actual query document stays the same. These are the variables of your query.

Variables: The dictionary of values passed along with a GraphQL operation, that provides dynamic parameters to that operation.

The diagram shows a GraphQL query with three parts highlighted by purple boxes and labeled with purple text above them:

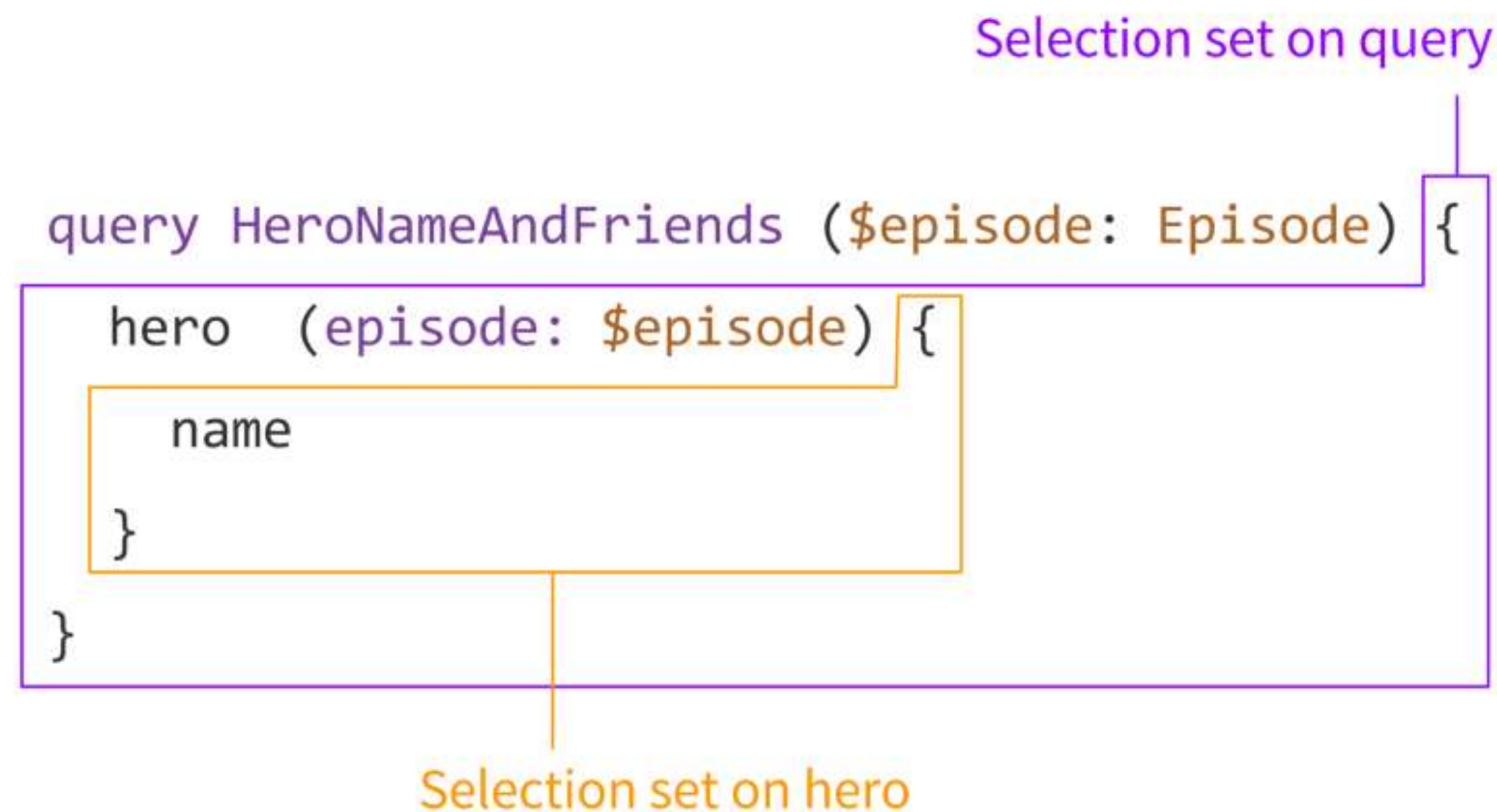
- Operation type:** points to the word `query`.
- Operation name:** points to `HeroNameAndFriends`.
- Variable definitions:** points to `($episode: Episode)`.

The full query text is:

```
query HeroNameAndFriends ($episode: Episode) {  
  hero(episode: $episode) {  
    name  
  }  
}
```

GraphQL queries Cont.

Selection set: A set of fields requested in an operation, or nested within another field. A GraphQL query must contain a selection set on any field that returns an object type, and selection sets are not allowed on fields that return scalar types, such as Int or String.



Fragments

Fragment definition: Part of a GraphQL document which defines a GraphQL fragment. This is also sometimes called a named fragment, in contrast to an inline fragment.

Fragment name: The name of each fragment has to be unique within a GraphQL document. This is the name you use to refer to the fragment in an operation or in other fragments.

Type condition: GraphQL operations always start at the query, mutation, or subscription type in your schema, but fragments can be used in any selection set.

```
fragment comparisonFields on Character {  
  name  
  appearsIn  
  friends {  
    name  
  }  
}
```

The diagram illustrates the components of a GraphQL fragment definition. The text is as follows:

- fragment**: The keyword to define a fragment.
- comparisonFields**: The **Fragment name**, which must be unique within the document.
- on Character**: The **Type condition**, specifying the type the fragment applies to.
- {**: The opening brace for the **Selection set**.
- name**, **appearsIn**, **friends {**, **name**, **}**: The fields and nested selection sets within the fragment.
- }**: The closing brace for the selection set.

Fragments

Fragment spread: When you use a fragment inside an operation or another fragment, you do so by putting ... followed by the fragment name. This is called a fragment spread, and it can appear in any selection set that matches that named fragment's type condition.

Inline fragment: When you just want to execute some fields depending on the type of a result, but you don't want to split that out into a separate definition, you can use an inline fragment.

```
query HeroForEpisode {  
  hero {  
    name  
    ...comparisonFields  
    ... on Droid {  
      primaryFunction  
    }  
  }  
}
```

The diagram illustrates the components of the provided GraphQL query. A purple box highlights the `...comparisonFields` line, with a purple line pointing to the label "Fragment spread". Another purple box highlights the `... on Droid {` block, with a purple line pointing to the label "Inline fragment". An orange box highlights the `on Droid` text, with an orange line pointing to the label "Type condition".

Directives

Directive: An annotation on a field, fragment, or operation that affects how it is executed or returned.

Directive arguments: These work just like field arguments, but they are handled by the execution engine instead of being passed down to the field resolver.

```
query HeroForEpisode($var: Boolean!) {  
  hero @skip(if: $var) {           # on a field  
    name  
    ...comparisonFields @include(if: $var) # on a fragment spread  
    ... on Droid @skip(if: $var) {       # on an inline fragment  
      primaryFunction  
    }  
    ... @include(if: $var) {           # on an inline fragment  
      name                             # with no type condition  
    }  
  }  
}
```

Directive

Directive arguments

@include(if: Boolean) Only include this field in the result if the argument is true.

@skip(if: Boolean) Skip this field if the argument is true.

Pagination

There are a number of ways we could do pagination:

- We could do something like `friends(first:2 offset:2)` to ask for the next two in the list.
- We could do something like `friends(first:2 after:$friendId)`, to ask for the next two after the last friend we fetched.
- We could do something like `friends(first:2 after:$friendCursor)`, where we get a cursor from the last item and use that to paginate.

```
{
  hero {
    name
    friends(first:2) {
      totalCount
      edges {
        node {
          name
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
```


Mutation

Same as query behavior but aims to Insert-Update-Delete

```
mutation CreateReviewForEpisode($ep: Episode!  
  createReview(episode: $ep, review: $review  
    stars  
    commentary  
  )  
{  
}
```

VARIABLES

```
{  
  "ep": "JEDI",  
  "review": {  
    "stars": 5,  
    "commentary": "This is a great movie!"  
  }  
}
```

```
{  
  "data": {  
    "createReview": {  
      "stars": 5,  
      "commentary": "This is a great movie!"  
    }  
  }  
}
```


Samples

1-Sample Query

<pre>{ hero { name # Queries can have comments! friends { name } } }</pre>	<pre>{ "data": { "hero": { "name": "R2-D2", "friends": [{ "name": "Luke Skywalker" }, { "name": "Han Solo" }] } } }</pre>
--	---

2-Query with args

<pre>{ human(id: "1000") { name height(unit: FOOT) } }</pre>	<pre>{ "data": { "human": { "name": "Luke Skywalker", "height": 5.6430448 } } }</pre>
--	---

3-Aliase

<pre>{ empireHero: hero(episode: EMPIRE) { name } jediHero: hero(episode: JEDI) { name } }</pre>	<pre>{ "data": { "empireHero": { "name": "Luke Skywalker" }, "jediHero": { "name": "R2-D2" } } }</pre>
--	--

4-Fragments

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
}
```

Samples Cont.

4-Fragments

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
}
```

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "appearsIn": ["NEWHOPE", "EMPIRE", "JEDI"],
      "friends": [ {"name": "Han Solo"}, {"name": "Leia Organa"}, {"name": "C-3PO"}, {"name": "R2-D2"} ]],
      "rightComparison": {
        "name": "R2-D2",
        "appearsIn": ["NEWHOPE", "EMPIRE", "JEDI"],
        "friends": [ {"name": "Luke Skywalker"}, {"name": "Han Solo"}, {"name": "Leia Organa"} ]]
      }
    }
  }
}
```

Samples Cont.

5-Inline Fragments

```
query HeroForEpisode($ep: Episode!) {  
  hero(episode: $ep) {  
    name  
    ... on Droid {  
      primaryFunction  
    }  
    ... on Human {  
      height  
    }  
  }  
}
```

VARIABLES

```
{  
  "ep": "JEDI"  
}
```

```
{  
  "data": {  
    "hero": {  
      "name": "R2-D2",  
      "primaryFunction": "Astromech"  
    }  
  }  
}
```

Samples Cont.

6-Directives

```
query Hero($episode: Episode, $withFriends:
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```


Samples Cont.

7-Mutation

```
mutation CreateReviewForEpisode($ep: Episode!  
  createReview(episode: $ep, review: $review  
    stars  
    commentary  
  )  
}
```

VARIABLES

```
{  
  "ep": "JEDI",  
  "review": {  
    "stars": 5,  
    "commentary": "This is a great movie!"  
  }  
}
```

```
{  
  "data": {  
    "createReview": {  
      "stars": 5,  
      "commentary": "This is a great movie!"  
    }  
  }  
}
```


GraphQL Schema

A GraphQL schema is at the center of any GraphQL server implementation and describes the functionality available to the clients which connect to it.

The core building block within a schema is the “type”. Types provide a wide-range of functionality within a schema, including the ability to:

- Create relationships between types (e.g. between a Book and an Author).
- Define which data-fetching (querying) and data-manipulation (mutating) operations can be executed by the client.
- If desired, self-explain what capabilities are available to the client (via introspection).

GraphQL Schema Types

Scalar types

Scalar types represent the leaves of an operation and always resolve to concrete data. The default scalar types which GraphQL offers are:

Int: Signed 32-bit integer

- Float: Signed double-precision floating-point value
- String: UTF-8 character sequence
- Boolean: true or false
- ID (serialized as String): A unique identifier, often used to refetch an object or as the key for a cache. While serialized as a String, ID signifies that it is not intended to be human-readable

GraphQL Schema Types Cont.

Object types

The object type is the most common type used in a schema and represents a group of fields. Each field inside of an object type maps to another type, allowing nested types and circular references.

```
type TypeName {  
  fieldA: String  
  fieldB: Boolean  
  fieldC: Int  
  fieldD: CustomType  
}  
  
type CustomType {  
  circular: TypeName  
}
```

GraphQL Schema Types Cont.

The Query type

A GraphQL query is for fetching data and compares to the GET verb in REST-based APIs.

```
type Query {  
  getBooks: [Book]  
  getAuthors: [Author]  
}
```

Schema

```
query {  
  getBooks {  
    title  
  }  
  
  getAuthors {  
    name  
  }  
}
```

Request

```
{  
  "data": {  
    "getBooks": [  
      {  
        "title": "..."  
      },  
      ...  
    ],  
    "getAuthors": [  
      {  
        "name": "..."  
      },  
      ...  
    ]  
  }  
}
```

Response

GraphQL Schema Types Cont.

The Mutation type

Mutations are operations sent to the server to create, update or delete data.

```
type Mutation {  
  addBook(title: String, author: String): Book  
}
```

Schema

```
mutation {  
  addBook(title: "Fox in Socks", author: "Dr. Seuss") {  
    title  
    author {  
      name  
    }  
  }  
}
```

Request

```
{  
  "data": {  
    "addBook": {  
      {  
        "title": "Fox in Socks",  
        "author": {  
          "name": "Dr. Seuss"  
        }  
      }  
    }  
  }  
}
```

Response

GraphQL Schema Types Cont.

The Interface type

An Interface is an abstract type that includes a certain set of fields that a type must include to implement the interface.

```
interface Character {  
  id: ID!  
  name: String!  
  friends: [Character]  
  appearsIn: [Episode]!  
}
```

```
type Human implements Character {  
  id: ID!  
  name: String!  
  friends: [Character]  
  appearsIn: [Episode]!  
  starships: [Starship]  
  totalCredits: Int  
}
```

```
type Droid implements Character {  
  id: ID!  
  name: String!  
  friends: [Character]  
  appearsIn: [Episode]!  
  primaryFunction: String  
}
```

GraphQL Schema Types Cont.

The Subscriptions type

GraphQL subscriptions are a way to push data from the server to the clients that choose to listen to real time messages from the server. Subscriptions are similar to queries in that they specify a set of fields to be delivered to the client, but instead of immediately returning a single answer, a result is sent every time a particular event happens on the server.

why GraphQL is great for real-time apps:

- Live-queries (subscriptions) are an implicit part of the GraphQL specification. Any GraphQL system has to have native real-time API capabilities.
- A standard spec for real-time queries has consolidated community efforts around client-side tooling, resulting in a very intuitive way of integrating with GraphQL APIs.

GraphQL Advantages

- Declarative Data Fetching
- No Over fetching with GraphQL
- Single Source of Truth
- GraphQL embraces modern Trends
- GraphQL Schema Stitching
- GraphQL Introspection
- Strongly Typed GraphQL
- GraphQL Versioning
- A growing GraphQL Ecosystem

GraphQL Disadvantages

- GraphQL Query Complexity
- GraphQL Caching
- GraphQL security