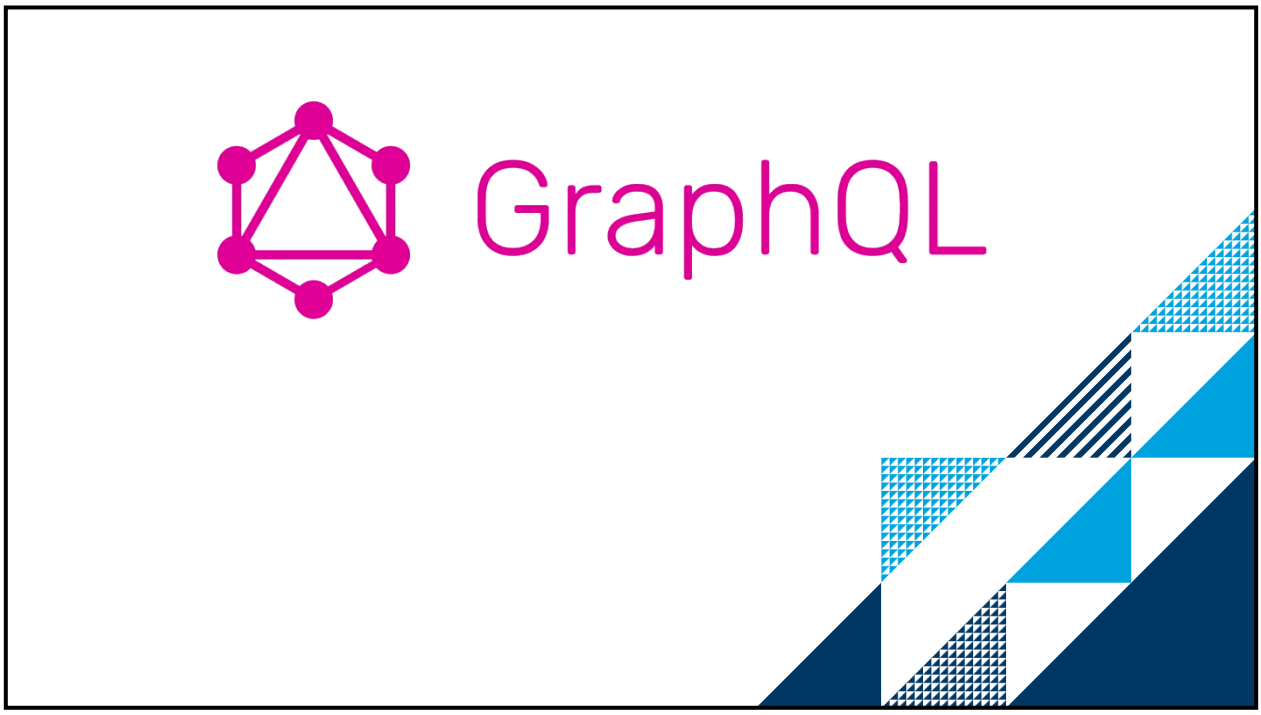



# What's up

**Day 1**

1. Introduction
2. GraphQL vs REST
3. Graphs
4. IDE Walkthrough
5. Fetch Data
6. Set up Environment
7. Lists & Nullability
8. Nest Objects
9. Arguments
10. Type System
11. Mutate Data and Use Variables
12. Realtime Data (optional)
13. Wrap-up

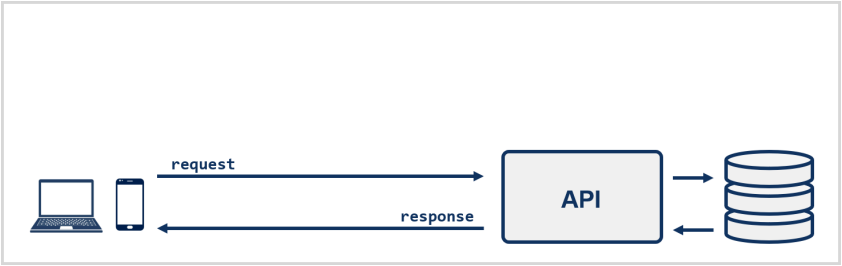
# What's up Cont.

Day 2

1. ORM & Database
2. Enums
3. Abstract Types
4. Custom Scalars
5. Directives
6. Introspection
7. Solving the N+1 problem
8. Authentication & Authorization
9. Wrap-up

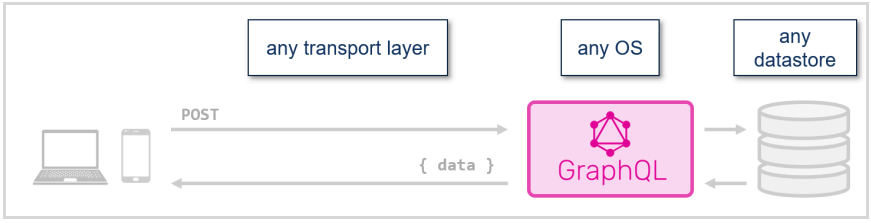
# Introduction

# What is GraphQL?




# Independency

As it is a specification:



# Language Support

- JavaScript
- C#
- Java / Kotlin
- Python
- C / C++
- 👉 https://graphql.org/code/

# Origin

Lee Byron - creator of GraphQL

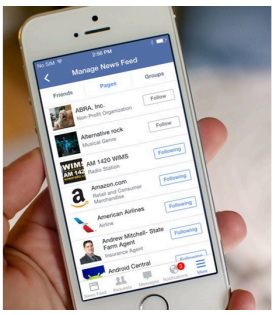
> *When we built Facebook's mobile applications, we needed a data-fetching API powerful enough to describe all of Facebook, yet simple enough to be easy to learn and use by our product developers. We developed GraphQL three years ago to fill this need. Today it powers hundreds of billions of API calls a day.*

👉 https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/

## History

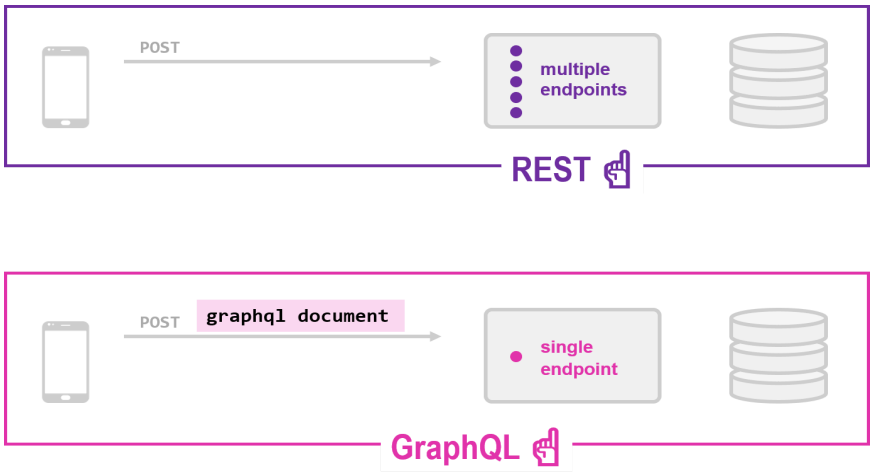
Internal Project

2012



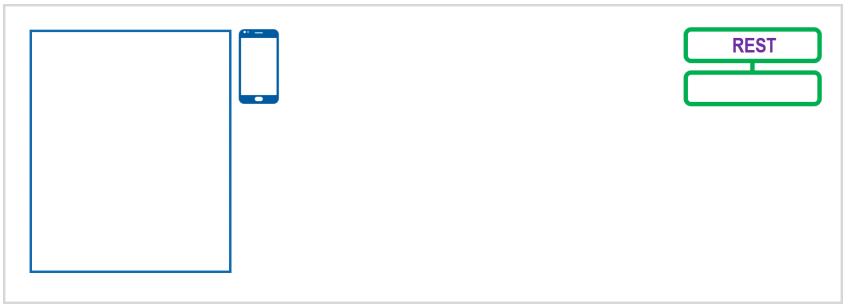

## GraphQL vs REST

## Compare

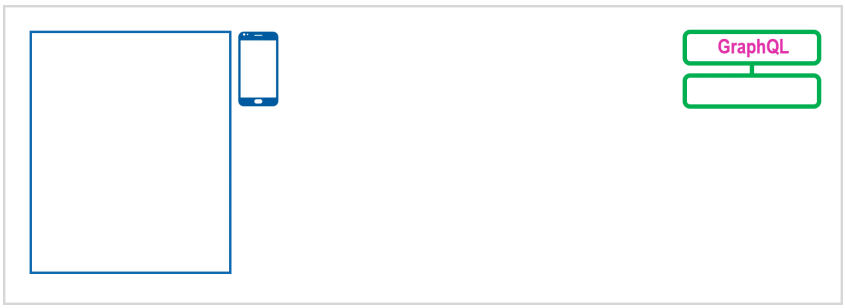



## Demo

## REST



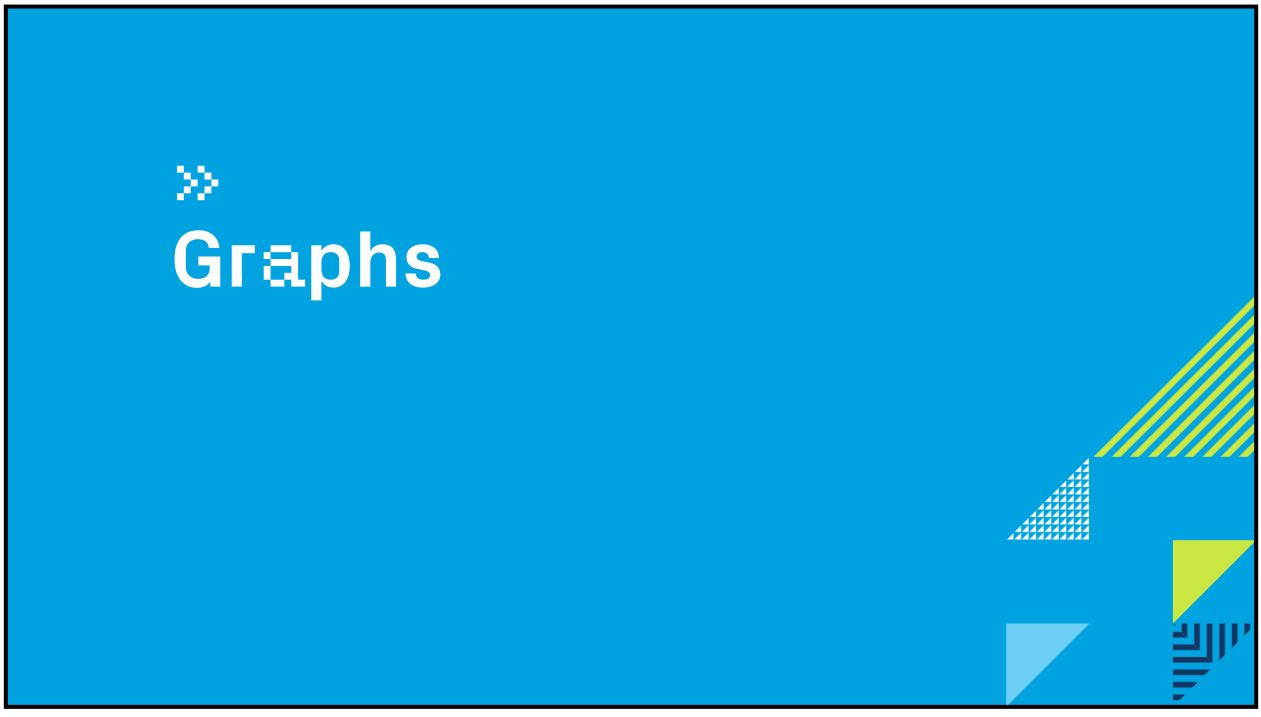

## GraphQL



## Take-away

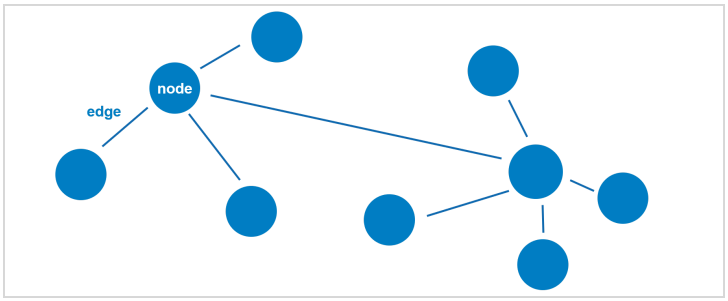
GraphQL:

1. Client determines what will be returned
2. No overfetching
3. No underfetching
4. Less versioning



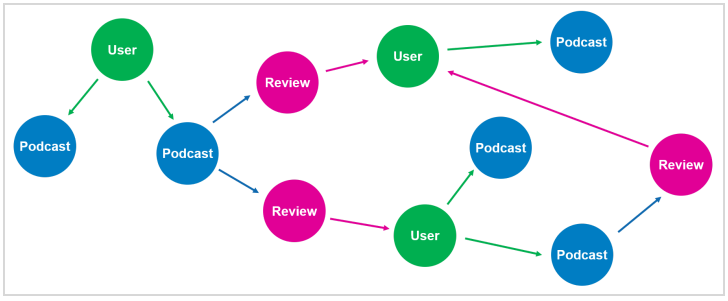
# What is a Graph?

- a *data structure*
- can be traversed recursively




# Edges

Each edge has a start and end: start ⟶ end

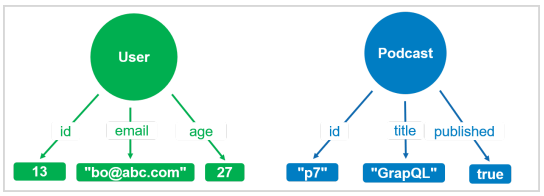



Above is a *Directed Acyclic Graph* (DAG)

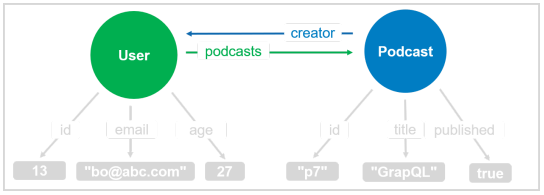
# Field Types

Object types have fields.

**scalar type fields**
endpoints in the graph




**object type fields**
define relations
between objects

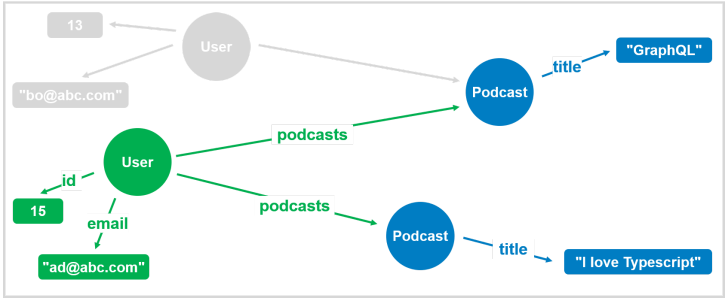



# Traversing

starts with an **entry point**

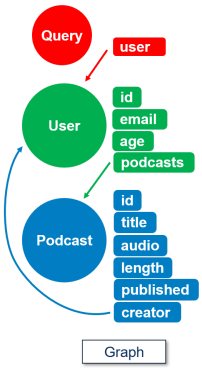
```
query {
  user(id: 15) {
    email
    podcasts {
      title
    }
  }
}
```

```
{
  "user": {
    "email": "ad@abc.com",
    "podcasts": [
      { "title": "GraphQL" },
      { "title": "I love Typescript" }
    ]
  }
}
```



# Graph to Schema

The graph is defined as a **Schema** on the server.




Graph

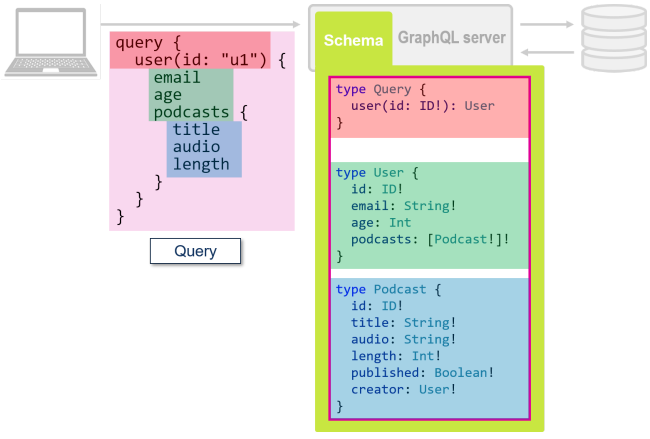
```
type Query {
  user(id: ID!): User
}

type User {
  id: ID!
  email: String!
  age: Int
  podcasts: [Podcast!]!
}

type Podcast {
  id: ID!
  title: String!
  audio: String!
  length: Int!
  published: Boolean!
  creator: User!
}
```

Schema

- at least one **entry point** is defined: `User`
- entry points live in a **root type**: `Query`

# GraphQL

Client's query is validated by the Schema.

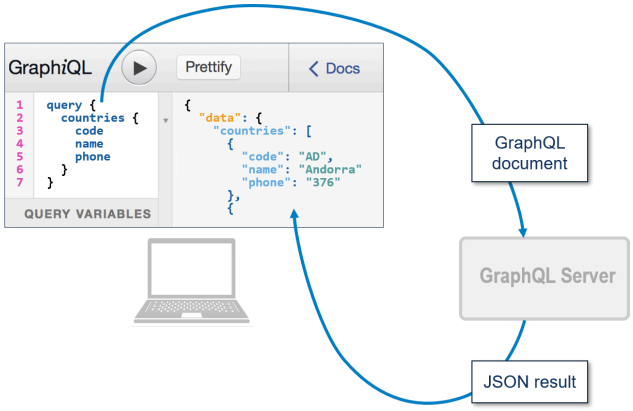



Query

# IDE Walkthrough

## IDE

Each GraphQL Server hosts a free IDE - **GraphiQL** is the default.




Send requests to the server and facilitate the retrieval of data.

# GraphiQL

GraphiQL has some basic features:

- autocompletion
- syntax checking
- live API documentation
- customizable

What makes the first 3 bullets possible?

# Playground

**Playground** is built on top of GraphiQL and has more features.

In a browser go to: https://countries.trevorblades.com/


Type a document in the left pane and click Play.

# Result

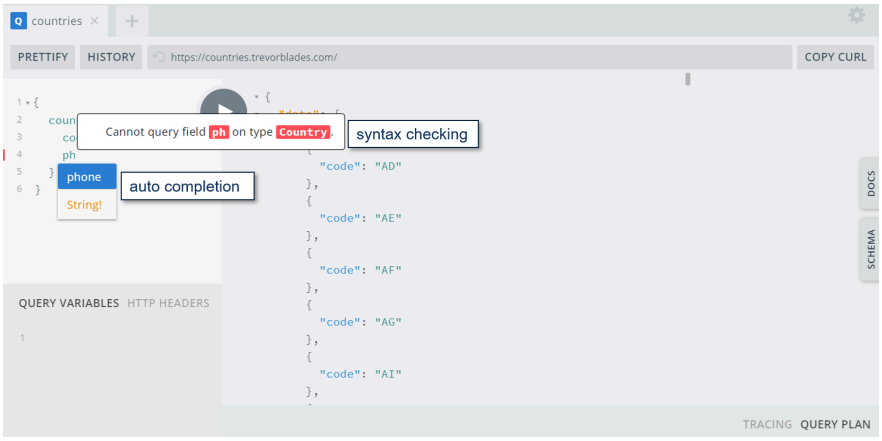
In the right pane the fetched data is shown. Format is JSON.

What stands out about the shape of the data?


```
{
  "data": {
    "countries": [
      {
        "code": "AD",
        "name": "Andorra",
        "phone": "376"
      },
      {
        "code": "AE",
        "name": "United Arab Emirates",
        "phone": "971"
      }
    ]
  }
}
```

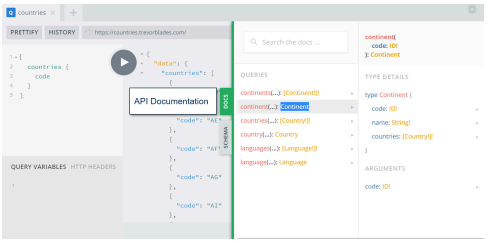

# Basic Features

# Prevent Errors




All thanks to a strongly typed schema on server!

# Self-documenting




Explore relationships between types and read details about individual types.

# Fetch Data

## Operation Types

GraphQL supports 3 types of operations.

- **Query** - fetch data from the data source 👈 focus in this module
- **Mutation** - change data in the data source: *insert*, *update*, *delete*
- **Subscription** - handling real-time data

Go to https://countries.trevorblades.com/

# query

means **fetch data** - it's the default operation type

Three ways to write a `query`:

**shorthand syntax**

```
QUERY
{
  countries {
    code
    name
    phone
  }
}
```

**operation type only**

```
QUERY
query {
  countries {
    code
    name
    phone
  }
}
```

**operation type + query type**

```
QUERY
query allCountries {
  countries {
    code
    name
    phone
  }
}
```

Which syntax helps clients to identify responses?

# Comment

```
QUERY
# I'm a comment and will not
# be sent to the server

query {
  countries {
    code
    name
    phone
  }
}
```

```
{
  "data": {
    "countries": [
      {
        "code": "AD",
        "name": "Andorra",
        "phone": "376"
      },
      {
        "code": "AE",
        "name": "United Arab Emirates",
        "phone": "971"
      },
      ...
    ]
  }
}
```

# Relational Data

QUERY

```
query {
  continents {
    code
    name
  }
}
```

QUERY

```
query {
  continents {
    code
    name
    countries {
      name
      capital
    }
  }
}
```

QUERY

```
query {
  continents {
    code
    name
    countries {
      name
      capital
      languages {
        name
        native
      }
    }
  }
}
```

Container objects are often written with plural nouns.
Retrieve fields on their members by using brackets `{ }`

# Arguments

QUERY

```
query oneCountry {
  country(code: "AF") {
    code
    capital
    currency
  }
}
```

```
{
  "data": {
    "country": {
      "code": "AF",
      "capital": "Kabul",
      "currency": "AFN"
    }
  }
}
```

- filter data via **arguments**
- use double quotes `""` for strings

# Multiple Datasets

QUERY

```
query {
  country(code: "AF") {
    name
  }
  languages {
    name
    native
  }
}
```

```
{
  "data": {
    "country": {
      "name": "Afghanistan"
    },
    "languages": [
      {
        "name": "Afrikaans",
        "native": "Afrikaans"
      },
      {
        "name": "Amharic",
        "native": "□□□□"
      },
      ...
    ]
  }
}
```

One request can hold multiple unrelated datasets.

# Aliases

Prevent a name conflict with an **Alias**.

QUERY

```
query {
  Andorra: country(code: "AD") {
    phone
    capital
  }
  Afghanistan: country(code: "AF") {
    phone
    capital
  }
}
```

```
{
  "data": {
    "Andorra": {
      "phone": "376",
      "capital": "Andorra la Vella"
    },
    "Afghanistan": {
      "phone": "93",
      "capital": "Kabul"
    }
  }
}
```

The two 'country' fields would have conflicted.
Use aliases to get both results in a single request.

# Fragments

To be 'DRY' use a **Fragment** - a defined set of fields which can be reused

### no fragments

QUERY

```
query {
  Andorra: country(code: "AD") {
    capital
    languages {
      name
      native
    }
  }
  Afghanistan: country(code: "AF") {
    capital
    languages {
      name          # 👀 repetition
      native        # 👀 of fields
    }
  }
}
```

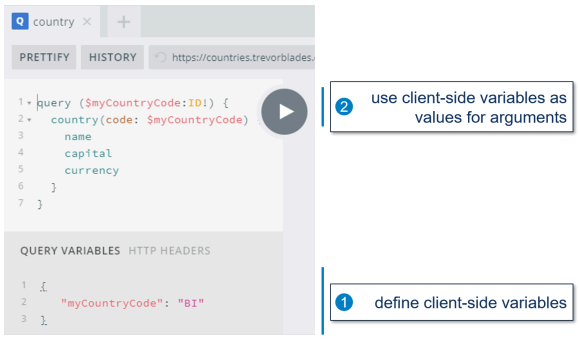
### with a fragment

QUERY

```
query {
  Andorra: country(code: "AD") {
    ...compareCountries   # 👀 reuse
  }
  Afghanistan: country(code: "AF") {
    ...compareCountries   # 👀 reuse
  }
}
fragment compareCountries on Country {
  capital
  languages {
    name            # 👀 define once
    native
  }
}
```

# Variables intro

Reuse same query written by the client - with different arguments.




Why do variables make sense?

# Variables intro

VARIABLE

```
{
  "myCountryCode": "BI"
}
```

1. Define variables (JSON):
   - Property names between double quotes: `"myCountryCode"`
   - Surrounded by brackets: `{}`

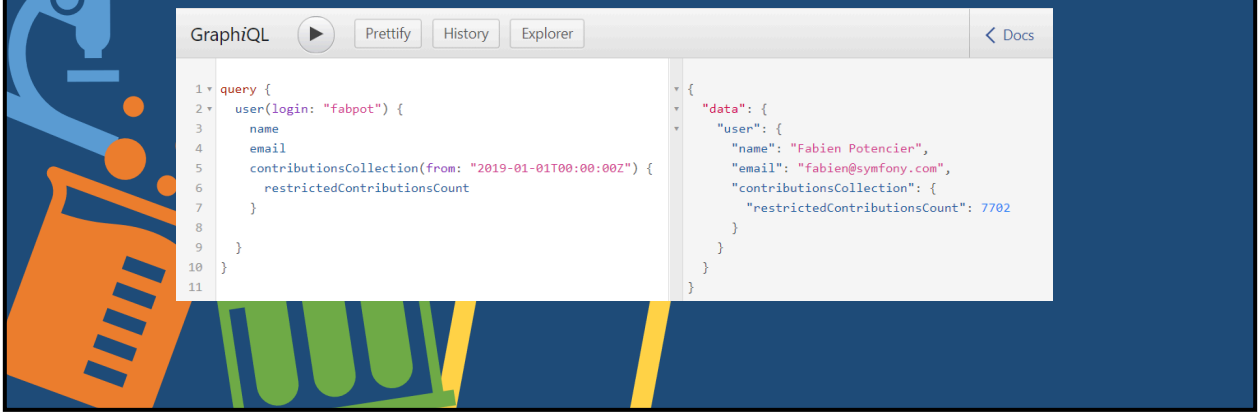
QUERY

```
query ($myCountryCode: ID!) {
  country(code: $myCountryCode) {
    name
    capital
    currency
  }
}
```

2. Use them in the query:
   - put a `$` in front of the variable names and pass them to the query: `query ($myCountryCode: ID!) {`
   - then use the variables as value for arguments: `code: $myCountryCode`

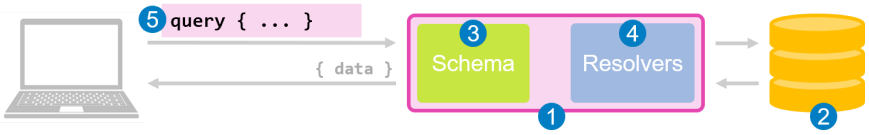
# Lab time!

Explore Github's GraphQL API



# Set up Environment

## Meet the players




1. **server** - Apollo Server running on Node.js, code written in TypeScript
2. **data source** - in-memory object
3. **schema** - data model
4. **resolvers** - business logic
5. **request** - client's request to the server

# New project


Built with: node.js + APOLLO

- In a new project folder, create a new project:

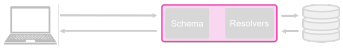
```
npm init --yes && npm pkg set type="module"
```

- Install dependencies:

```
npm install @apollo/server graphql
npm install --save-dev typescript @types/node
```

- Configure TypeScript
- Set `scripts` in `package.json` for easy running

# Hello world




Add a file `index.ts`

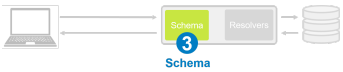
```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';

// 1. data source here 👇
const dataSource = [ /* ... */ ];

// 2. schema here 👇
const typeDefs = `...`;

// 3. resolvers here 👇
const resolvers = { /* ... */ };

const server = new ApolloServer({ typeDefs, resolvers });
const { url } = await startStandaloneServer(server, { listen: { port: 4000 } });
console.log(`🚀  Server ready at: ${url}`);
```

# Add data source


In `index.ts`, add a single in-memory object.

```
// 1. data source here 👇
const testPodcast = {
  id: 'p7',
  title: 'GraphQL',
  length: 54,
  price: 1.99,
  published: true,
};
```

# Schema




Can be defined in several ways.

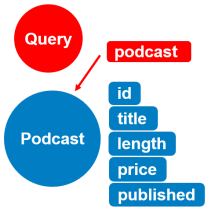
Most common is through the **Schema Definition Language** (SDL):

- language-agnostic
- easy to read for humans

This notation is also part of and is used in the *GraphQL Specification*.

# Determine types


- `type Podcast` defines a `Podcast` object
- `type Query` (root type) has got one entry point to return a `Podcast`




Graph

```
type Query {
  podcast: Podcast
}

type Podcast {
  id: ID!
  title: String!
  length: Int!
  price: Float!
  published: Boolean!
}
```

Schema

# Types intro


GraphQL supports 5 different scalars (primitive values):
**`ID`**, **`String`**, **`Int`**, **`Float`** and **`Boolean`**

SCHEMA

```
type Query {          # Object Type
  podcast: Podcast!   # podcast = entry point of type Podcast 👇
}

type Podcast {            # Object Type
  id: ID!                 # ID      = unique identifier, string formatte
  title: String!          # String  = UTF-8 character sequence
  length: Int!            # Int     = whole number
  price: Float            # Float   = floating-point value
  published: Boolean!     # Boolean = true or false
}
```

**`!`** = field is **non-nullable**:
GraphQL promises to always return a value when the field is queried.

# Code

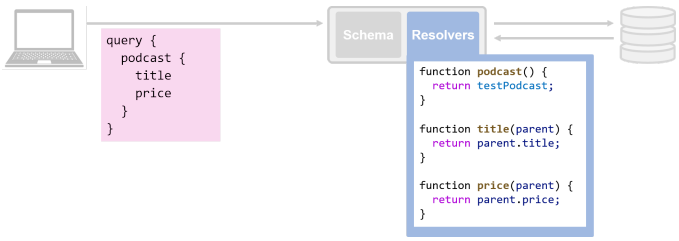

In APOLLO defined within an ES6 tagged template literal.

```
//  2. schema here 👇
const typeDefs = `
  type Query {
    podcast: Podcast!
  }

  type Podcast {
    id: ID!
    title: String!
    length: Int!
    price: Float
    published: Boolean!
  }
`;
```

You can also use a `gql` tagged template literal, for which IDEs often provide better syntax highlighting.

# Resolver

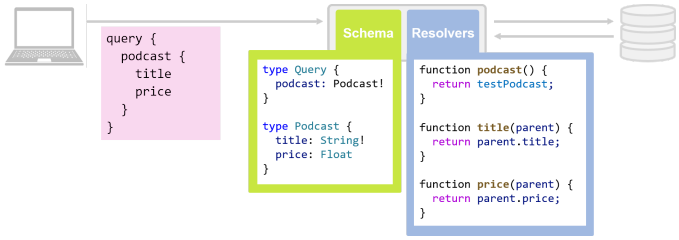

Every field on every type is backed by a *function* called a **resolver**.




A resolver *resolves* a value for a type by fetching data from a data source.

# Resolver

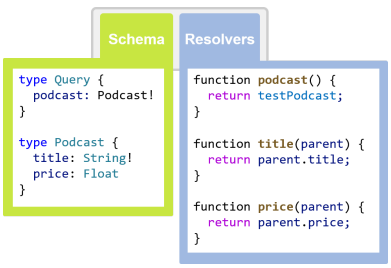

Resolvers are mapped to the Schema.




Final response (JSON) is generated by:
*executing resolvers in a sequence defined by the Schema*

# Resolver


Returns either an **object** or a **scalar** (`String`, `Int`, etc.)



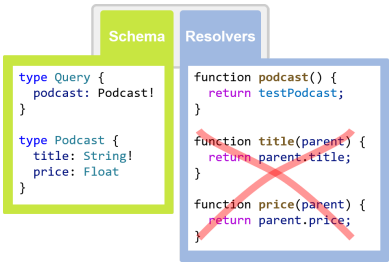

- when returning an object... execution continues to next child
- when returning a scalar... execution completes

Each resolver can have 4 optional arguments - first one is **parent**:
*it contains the return value of the resolver of its parent*

# Default Resolver


Resolving a *scalar* is straight-forward:
`Podcast` object is assumed to have a `title` and `price` property.

Most GraphQL libraries build **default resolvers** for these cases.




Developers can omit them.

# Code


Resolvers are stored in a **resolver map**.

RESOLVER

```
// 3. resolvers here 👇
const resolvers = {                 // resolver map
  Query: {
    podcast: () => testPodcast,     // the resolver
  },
};
```

Fields in resolver map match types in schema.

SCHEMA

```
type Query {
  podcast: Podcast!
}
...
```

# Test


Start the server:

```
npm start   # this should run tsc followed by node dist/index.js
```

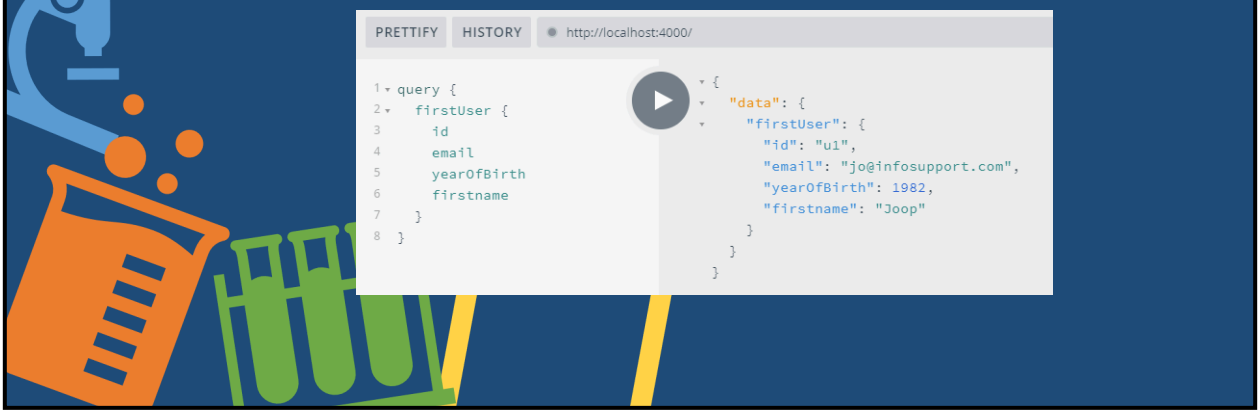
In the browser go to: http://localhost:4000.
Run a couple of queries.

```
query {
  podcast {
    title
    price
    published
  }
}
```

```
{
 "data": {
    "podcast": {
      "title": "GraphQL",
      "price": 1.99,
      "published": true
    }
  }
}
```

# Lab time!

Build a GraphQL endpoint for a Blogging App



# Lists & Nullability

## Lists

Most nodes in a graph live in lists.
**list** = sequence of values

SCHEMA

```
type Query {
  podcast: Podcast # first entry point: single Podcast object
  podcasts: [Podcast!]! # second entrypoint: list of Podcast objects
}
```

- list modifier is written as square brackets `[]`
- kind of similar to *array* notation in other languages

# Lists and Nullability

*A modifier is a type that reference and changes another type.*

GraphQL specs define 2 modifiers:

- `!` = non-nullable
- `[]` = list

What is allowed per combination?

| | List of non-`null` values | Empty list | List including `null` values | `null` |
|---|---|---|---|---|
| `[Podcast]` | ✔ | ✔ | ✔ | ✔ |
| `[Podcast]!` | ✔ | ✔ | ✔ | |
| `[Podcast!]` | ✔ | ✔ | | ✔ |
| `[Podcast!]!` | ✔ | ✔ | | |

# List Example


Data source is an *array* with podcast objects.

```
const testPodcasts = [
  {
    id: "p7",
    title: "GraphQL",
    length: 54,
    price: 1.99,
    published: true,
  },
  // ...
];
```

# List Example


In the Schema the list modifier is used.

SCHEMA

```
type Query {
  podcasts: [Podcast!]!  # 👀 [] = list modifier
}                        # 👀 !  = non-nullable modifier
type Podcast {
  id: ID!
  title: String!
  length: Int!
  price: Float
  published: Boolean!
}
```

# List Example


Resolver for the entry point `podcasts` is defined.

RESOLVER

```
const resolvers = {
  Query: {
    podcasts: () => testPodcasts, // 👀 podcasts is plural
  },
};
```

For the scalar fields (`id`, `title`, etc.) default resolvers are built automatically.

# List Example


Test in Playground:

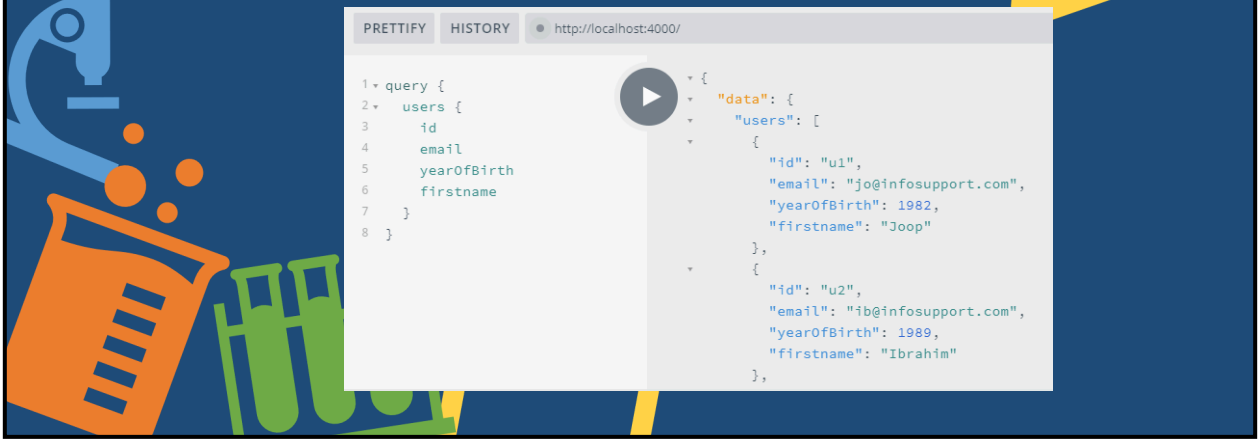
QUERY

```
query {
  podcasts {
    id
    title
    published
  }
}
```

```
{
  "data": {
    "podcasts": [
      {
        "id": "p7",
        "title": "GraphQL",
        "published": true
      },
      {
        "id": "p8",
        "title": "We love TypeScript",
        "published": true
      },
      ...
    ]
  }
}
```

# Lab time!

Retrieve a List of users &
reason about Nullability




## Another List

New root type `users` of type *User List*

**data source**

```
const testUsers = [
  {
    id: 'u1',
    email: 'u1@live.com',
  },
  {
    id: 'u2',
    email: 'u2@live.com',
  },
];
```

**schema**

```
type Query {
  podcasts: [Podcast!]!
  users: [User!]!
}
```

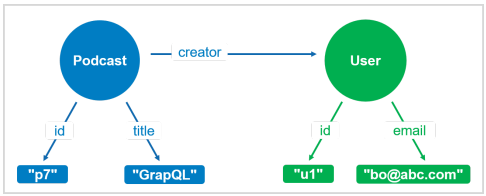
```
type User {
  id: ID!
  email: String!
}
```

**resolver**

```
const resolvers = {
  Query: {
    podcasts: () => testPodcasts,
    users: () => testUsers,
  },
};
```

# Nest the Objects

Relation between Podcast and User can be made:




GraphQL API is great at handling queries with **nested objects** such as:

QUERY

```
query {
  podcasts {
    title
    published
    creator {
      id
      email
    }
  }
}
```

# Nesting Example


Data source defines a `creator` field.

```
const testPodcasts = [
  {
    // ...
    creator: 'u1',
  },
  {
    // ...
    creator: 'u2',
  },
];
```

# Nesting Example


`Podcast` type is extended with a `creator` field.

SCHEMA

```
type Query {
  podcasts: [Podcast!]!
  users: [User!]!
}

type Podcast {
  # ...
  creator: User! # each podcast has a creator of type User
}

type User {
  id: ID!
  email: String!
}
```

# Nesting Example


`creator` represents an object, so a resolver needs to be coded:

RESOLVER

```
const resolvers = {
  Query: {
    podcasts: () => testPodcasts,
    users: () => testUsers,
  },
  Podcast: {
    creator: (parent) => fetchUserById(parent.creator);
  }
};
```

- creator resolver is called for every podcast in the list
- its parent argument contains all details of the current podcast

Why should resolvers be kept *thin* as best practice?

# Nesting Example


A resolver can optionally accept four positional arguments:

RESOLVER

```
const resolverMap = {
  // ...
  resolvername(parent, args, context, info) {
    fetchDataFromDB();
  }
  // ...
};
```

**Resolver Arguments**

| Argument | Description |
|---|---|
| `parent` | Result from the previous/parent type |
| `args` | Arguments provided to the field |
| `context` | Object shared among all resolvers that are executing an operation |
| `info` | Field-specific information relevant to the query (used rarely) |

# Nesting Example


QUERY

```
query {
  podcasts {
    creator {
      email
    }
  }
}
```

When field contains object type ...
... fields on that object must be resolved

When subfield contains object type ...
... fields on that object must be resolved

This object field pattern can continue to an arbitrary depth.

**Resolver Chain:**

# Lab time!

Add blogs &
allow nesting blogs and users in flexible ways!


# Arguments

# Arguments

**Arguments** - *dynamic parameters*

QUERY

```
query {
  podcast(id: "p8") {
    id
    title
    length(inSeconds: true)
  }
}
```

```
{
  "data": {
    "podcast": {
      "id": "p8",
      "title": "We love TypeScript",
      "length": 3600
    }
  }
}
```

Use cases:

- filtering
- pagination in queries
- payload for mutations

# Arguments

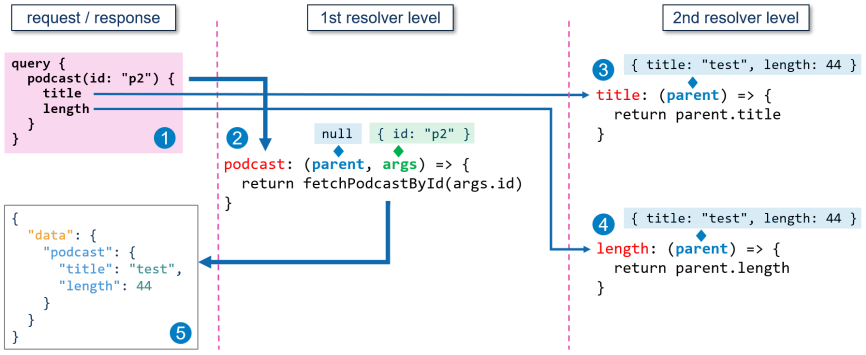
- They are always passed **by name**.
- can be either **required** or **optional**.
- optional arguments can have default values defined in the schema.
- are available in the resolver via its 2nd parameter `args`

QUERY

```
query {
  podcasts(titleStartsWith: "W") {
    title
    length
  }
}
```

```
podcasts(parent, args) {
  /* .... */
}
```

# Arguments in Resolver Chain




# Arguments

Arguments can be passed in two ways:

```
query {
  podcasts(
    titleStartsWith: "W",
    titleEndsWith: "t"
  ) {
    title
    length
  }
}
```

passed as **scalars**

```
query {
  podcasts(input: {
    titleStartsWith: "W",
    titleEndsWith: "t"
  }) {
    title
    length
  }
}
```

passed as **input type**

```
{
  "data": {
    "podcast": {
      "title": "We love TypeScript",
      "length": 60
    }
  }
}
```

# Argument as Scalar

Example: `userById` entry point is added

### schema

SCHEMA

```
type Query {
  # id is required
  userById(id: ID!): User!
}
```

### query

QUERY

```
query {
  userById(id: "u2") {
    id
    email
  }
}
```

### resolver

RESOLVER

```
const resolvers = {
  Query: {
    // 👀 args = all passed arguments
    userById(parent, args) {
      return fetchUserById(args.id)
    }
  },
  // ...
};
```

# Default value

can be assigned to an *optional* argument
no value provided in the query 👉 default value is used

### schema

SCHEMA

```
type Query {
  userById(id: ID!): User!
}
```

`id` as required argument

### schema

SCHEMA

```
type Query {
  userById(id: ID = "u1"): User!
}
```

`id` as optional argument with default value `"u1"`

# Arguments via Input Type

Example: `podcasts` entry point gets simple pagination

### schema

SCHEMA

```
type Query {
  podcasts(input: PodcastsInput): [Podcast!]!
}
```

SCHEMA

```
input PodcastsInput {  # use keyword input
  first: Int = 2
  offset: Int = 2
}
```

### resolver

RESOLVER

```
const resolvers = {
  Query: {
    podcasts(parent, { input }) {
      const { first, offset } = input;
      return fetchPodcasts({
        first,
        offset
      });
    },
  },
};
```

With above implementation, which queries can be sent to the server?

# Lab time!

Filter blogs and users in flexible ways!

# Type System

## GraphQL Type System

GraphQL has as **strict static** type system:

| | Strict | Relaxed |
|---|---|---|
| Static | C#, Java, GraphQL | Visual Basic |
| Dynamic | Python | JavaScript |

Advantages:

- early detection of errors
- live documentation
- front- and backend teams can work independently
- smart tooling in IDEs: autocompletion & linting

# GraphQL Types

GraphQL's type system classifies 8 different types.

| Primitive values | Object types |
| --- | --- |
| • Scalar i o | • Input Object type i |
| • Enum i o | • Object type o |
| **Abstract Types** | **Modifiers** |
| • Interface o | • Non-null i o |
| • Union o | • List i o |

- i **input type** - argument for query or mutation
- o **output type** - all results from the server

# Primitive Values

Are the **leaves** in terms of graph theory.

Three types:

1. **built-in scalars** - `String`, `Int`, `Float`, `Boolean`, `ID`
2. **enums** - enumerate all possible values in a field
3. **custom scalars** - implementation depends on language, most frequently used: `DateTime`, `JSON`, `HTML`

# Built-in Scalars

- `String` - UTF-8 character seq. to represent free-form human-readable text
- `Int` - signed 32-bit integer, a whole numeric value between -($2^{31}$) and $2^{31}$-1
- `Float` - signed double-precision fractional values as specified by IEEE 754
- `Boolean` - true or false
- `ID` - unique identifier, not intended to be human-readable

# Object Types

Two categories of *Object types*:

- **Object Type** - meant for output:

SCHEMA

```
type Podcast {          # 👀 keyword 'type'
  id: ID!
  title: String!
  price: Float
}
```

- **Input Object Type** - meant for input:

SCHEMA

```
input PodcastsInput {   # 👀 keyword 'input'
  first: Int = 2
  offset: Int = 2
}
```

# Object Types

**Never reuse an Object Type for input!**
**Never use an Input Object Type for output!**

1. *Object Type* can contain fields that express circular references:

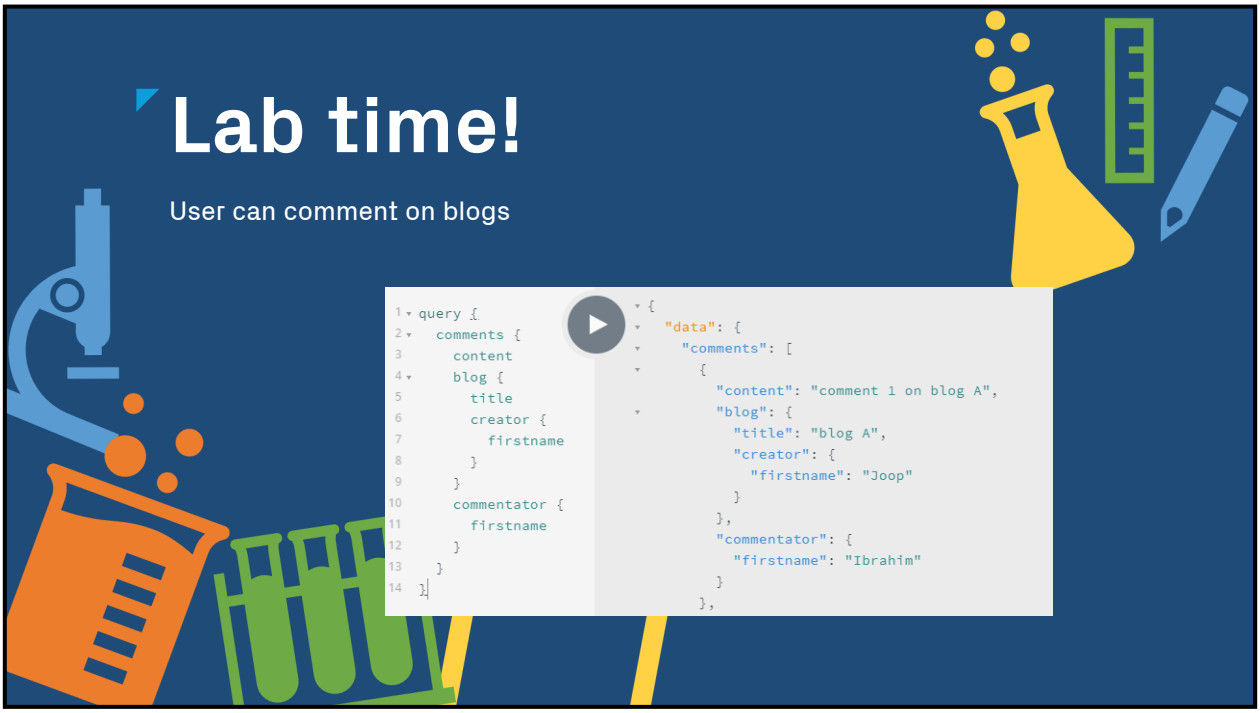
SCHEMA

```
type Podcast {
  title: String!
  creator: User!
}

type User {
  email: String!
  podcasts: [Podcast!]!
}
```

2. *Object Type* can refer to interfaces and unions
3. *Input Object Type* supports default values

# Lab time!

User can comment on blogs



# Mutate Data

## and Use Variables

## Mutation

**`Mutation`** - root type for actions that *alter* data.

```
mutation {

}
```
QUERY

operation type only

```
mutation AddUser {

}
```
QUERY

operation type + query name

- mutations - always use POST requests (no PUT, PATCH or DELETE)
- queries - use either POST or GET requests

Why is the mutation keyword mandatory?

# Mutation


- `Query` - root type for entry points that **fetch** data
- `Mutation` - root type for entry points that **alter** data

SCHEMA

```
# 👇 CRUD entry points mapped to the 2 root types

type Query {
  users: [User!]!                                  # read data
}

type Mutation {
  createUser(input: CreateUserInput!): User!     # create data
  updateUser(input: UpdateUserInput!): User!     # update data
  deleteUser(input: DeleteUserInput!): User!     # delete data
}
```

# Mutation


Each mutation consists of two parts:

QUERY

```
mutation {
  createUser(
    email: "john@live.com",
    city: "London"
  ) {
    id
    email
    city
  }
}
```

top part

bottom part

- **top part** - responsible for mutation
- **bottom part** - query the changed state of the server (!)

# Arguments

```
mutation {
  createUser(input: {
    email: "john@live.com",
    city: "London"
  }) {
    id
    email
  }
}
```

Best practice: always use one argument: **Input Object type**.

What is the alternative?

# Mutation Examples

**Insert**

QUERY

```
mutation {
  createUser(input: {
    email: "john@live.com",
    city: "London"
  }) {
    id
    email
    city
  }
}
```

**Update**

QUERY

```
mutation {
  updateUser(input: {
    id: "u2",
    city: "LA"
  }) {
    id
    email
    city
  }
}
```

**Delete**

QUERY

```
mutation {
  deleteUser(id: "u2") {
    id
    email
    city
  }
}
```

Make mutations as specific as possible:

`createUser()`, `likePodcast()`, `updateReview()`, `completeTask()`, ...

Naming convention:

createUser

type of operation — Object Type

# Input Types


Define **Input Types**, two naming conventions:

**schema**

SCHEMA

```
input CreateUserInput {
  email: String!
  city: String = 'London'
}

type Mutation {
  createUser(input: CreateUserInput): User
}
```


name of argument is **`input`**

# Mutation


RESOLVER

```
const resolvers = {
  // Query: { ... },
  Mutation: {
    updateUser(parent, args) {
      // 1. read args and validate args
      // 2. issue? throw error
      // 3. update data source
      // 4. return updated user 👈 don't forget!
    },
  }
};
```

- Second argument `args` contains `input` object with properties: `id`, ...
- Resolver must always return the mutated object (!)

Why should the updated object always be returned?

# Example

### data source

```
const testUsers = [
  { id: 'u1', /* ... */ },
  // ...
];
```

### resolver

RESOLVER

```
const resolvers = {
  // Query: { ... },
  Mutation: {
    updateUser(parent, args) {
      // 1. args.input contains:
      //    id, email, city
      // 2. throw error if
      //    user id doesn't exist
      // 3. update user in data source
      // 4. return updated user
    },
  }
};
```

### schema

SCHEMA

```
input UpdateUserInput {
  id: ID!
  email: String
  city: String
}
```

SCHEMA

```
type Mutation {
  updateUser(input: UpdateUserInput): User
}
```

SCHEMA

```
type User {
  id: ID!
  email: String!
  city: String!
}
```

# Why Input Object type?

### Advantage 1: nesting

Supports evolving the GraphQL schema over time:

- Deprecate sections if necessary
- Add new names within a conflict-free section

QUERY

```
mutation {
  updatePodcast(input: {
    id: 3,
    patch: {
      name: "What I like",
      length: 46
    }
  }) {
    name
    length
  }
}
```

# Why Input Object type?

**Advantage 2: easier client-side**

QUERY

```
mutation (
  $id: ID!,
  $name: String!,
  $length: Int
) {
  updatePodcast(input: {
    id: $id,
    name: $name,
    length: $length
  }) {
    name
    length
  }
}
```

**more maintainable:**

QUERY

```
mutation($input: UpdatePodcastInput) {
  updatePodcast(input: $input) {
    name
    length
  }
}
```

Reduces complexity of frontend code (especially with 10+ arguments)

# Input variables

Reuse the same client mutation with different arguments.

**no variables**

QUERY

```
mutation {
  updateUser(input: {
    id: "u1",
    email: "guan@yahoo.com"
    city: "Barcelona"
  }) {
    id
    name
    city
  }
}
```

VARIABLE

```
{
}
```

**with variables**

QUERY

```
mutation($input: UpdateUserInput) {
  updateUser(input: $input) {
    id
    name
    city
  }
}
```

VARIABLE

```
{
  "input": {
    "id": "u1",
    "email": "guan@yahoo.com",
    "city": "Barcelona"
  }
}
```

# Input variables

VARIABLE

```
{
  "input": {
    "id": "u1",
    "email": "guan@yahoo.com",
    "city": "Barcelona"
  }
}
```

1. Define `input` variable in JSON:
   - all property names between double quotes!!
   - nest using brackets
   - last property no trailing comma!

QUERY

```
mutation($input: UpdateUserInput!) {
  updateUser(input: $input) {
    id
    email
    city
  }
}
```

2. Use `input` variable in mutation:
   - pass variable to mutation as `$input` (type: `UpdateUserInput`)
   - use `$input` as value for `input` argument

# Output Type

**Expert tip** - Use a separate type for outputs as well:

SCHEMA

```
type Mutation {
    updateUser(input: UpdateUserInput!): UpdateUserPayload!
}
```

SCHEMA

```
input UpdateUserInput {
  id: ID!
  email: String
  city: String
}
```

SCHEMA

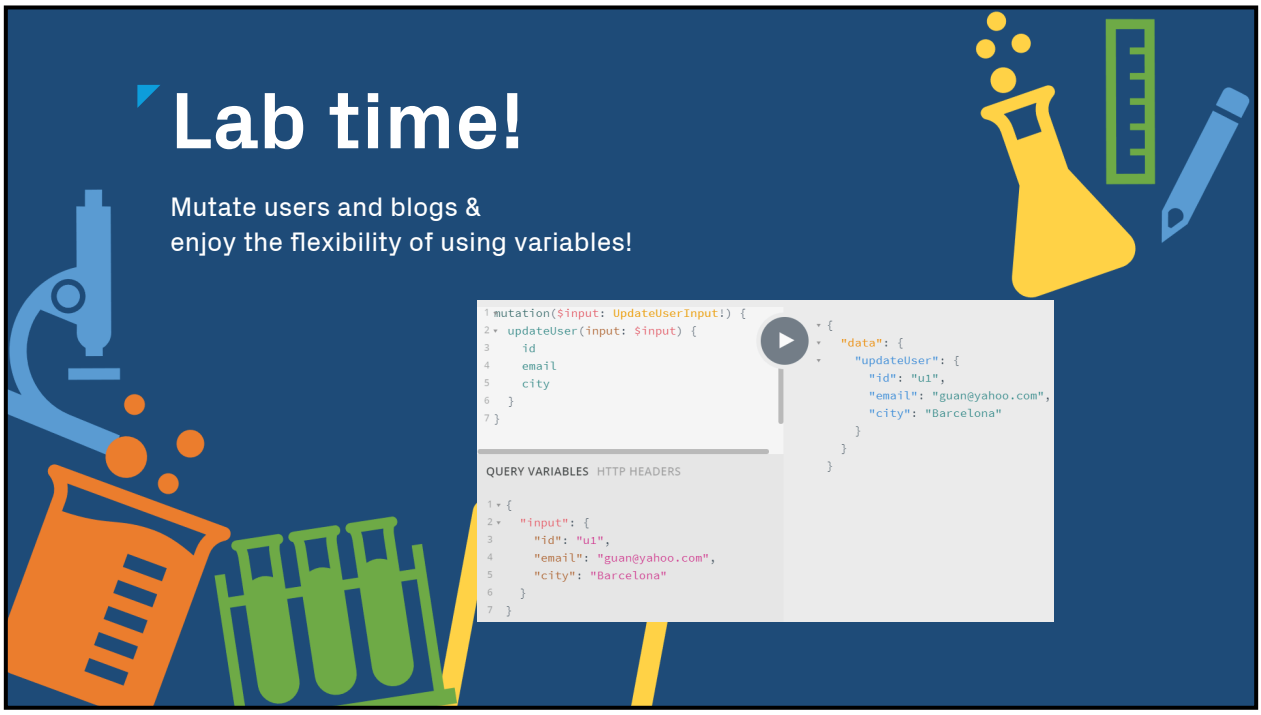
```
type User {
  id: ID!
  email: String!
  city: String!
}
```

SCHEMA

```
type Error {
  number: Int!
  message: String!
}
```

SCHEMA

```
type UpdateUserPayload {
  user: User          # when successful, return updated user
  errors: [Error!]!   # return user errors
}
```




# Subscriptions

**Notify a client *in real-time* about particular events**
(new object created, fields updated, ...)

- subscription is 3rd operation type in GraphQL
- underlying technology: *web sockets*

**web sockets**
bi-directional real-time communication between server and client

# Use Cases

Use subscriptions:

- **small, incremental changes to large objects**
  planner application with a lot of initial data / fields change rarely
- **low-latency, real-time updates**
  online trading application where clients need updates ASAP

Subscriptions are often not necessary!

What will be used in scenarios where subscriptions are overkill?

# Publish-Subscribe

In APOLLO use **PubSub** class:
exposes a simple publish and subscribe API

**publish–subscribe** - asynchronous messaging pattern:
**publishers** (senders of messages) and **subscribers** (receivers) don't know about the existence of one another


# Setup Subscriptions

Implementing GraphQL subscriptions require these steps:

1. Setup Subscription Server
2. Declare Subscription in Schema
3. Setup **PubSub**
4. Publish events to PubSub (in Mutation resolver)
5. Implement Subscription resolver
6. Test

# 1. Setup Subscription Server

Subscriptions use the *ws protocol* instead of *HTTP*:
Add a 2nd GraphQL endpoint specifically for subscriptions.

In APOLLO:
version 2: Nothing to do! 😄
version 3: More to do 😕

- integrate with **Express**, or similar
- build executable schema
- more dependencies

# 2. Subscription


```
type Query {
  users: [User!]!
}

type Mutation {
  createUser(email: String!): User!
}

type Subscription {       # 👀 subscription root type
  userCreated: User!      # client can subscribe to userCreated
}

type User {
  id: ID!
  email: String!
}
```

# 3. Setup PubSub

In APOLLO available via the PubSub class:

```
import { PubSub } from 'graphql-subscriptions';
const pubsub = new PubSub();
```

Share pubsub instance via context argument of constructor.

```
const server = new ApolloServer({
  schema,               // 👆 executable schema (typdefs + resolvers)
  context: { pubsub }   // 👆 available in all resolvers
  // ...
});
```

Also Subscription server needs executable schema + pubsub in context!

# 4. Publish Events

Pick a mutation of which a client might want to be notified.

3rd argument `context` contains the pubsub instance

```
Mutation: {
  createUser: (parent, args, ctx) => {
    const user = DB.createUser({ email: args.email });
    ctx.pubsub.publish('USER_CREATED', { userCreated: user });   // 👀
    return user;
  },
},
```

The `publish()` method of pubsub expects 2 arguments:

- **topic** - name of the channel
- **payload** - object with data to send to the client

# 5. Subscription


Every subscription needs a resolver.
resolver name mirrors schema (as usual):

```
Mutation: {
  // ...
},
Subscription: {
  userCreated: {    // 👀 an object - not a function
    // ...
  }
},
```

Note that a subscription resolver is not a function but **an object**!

# 5. Subscription


Define a `subscribe()` method witin the resolver object.

```
Subscription: {
  userCreated: {
    subscribe: (parent, args, ctx) => {
      return ctx.pubsub.asyncIterator('USER_CREATED');
    }
  }
},
```

`asyncIterator` - emits published messages to all subscribed clients

This function has one string argument: the **topic**.

# 6. Test

```
subscription {
  userCreated {
    id
    email
  }
}
```

1

Listening...

2

```
{
  "data": {
    "userCreated": {
      "id": "u1622047215245",
      "email": "test@infosupport.com"
    }
  }
}
```

4

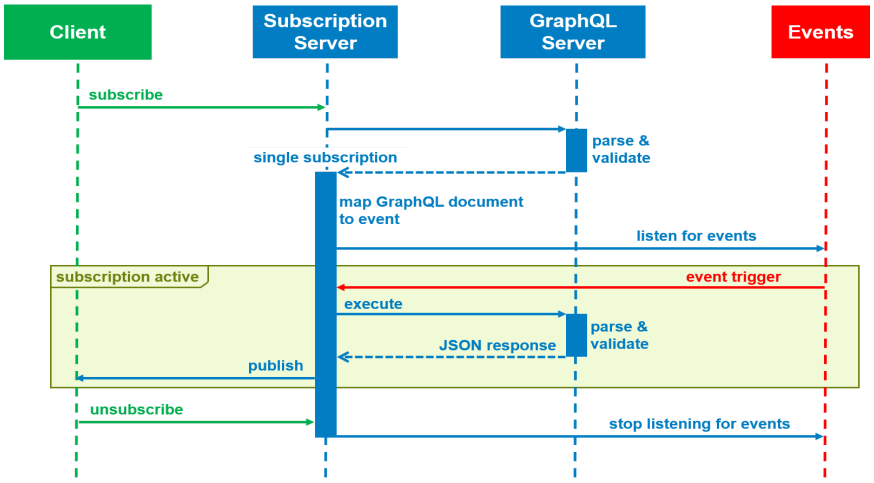
```
mutation {
  createUser(email:"test@infosupport.com") {
    id
    email
  }
}
```

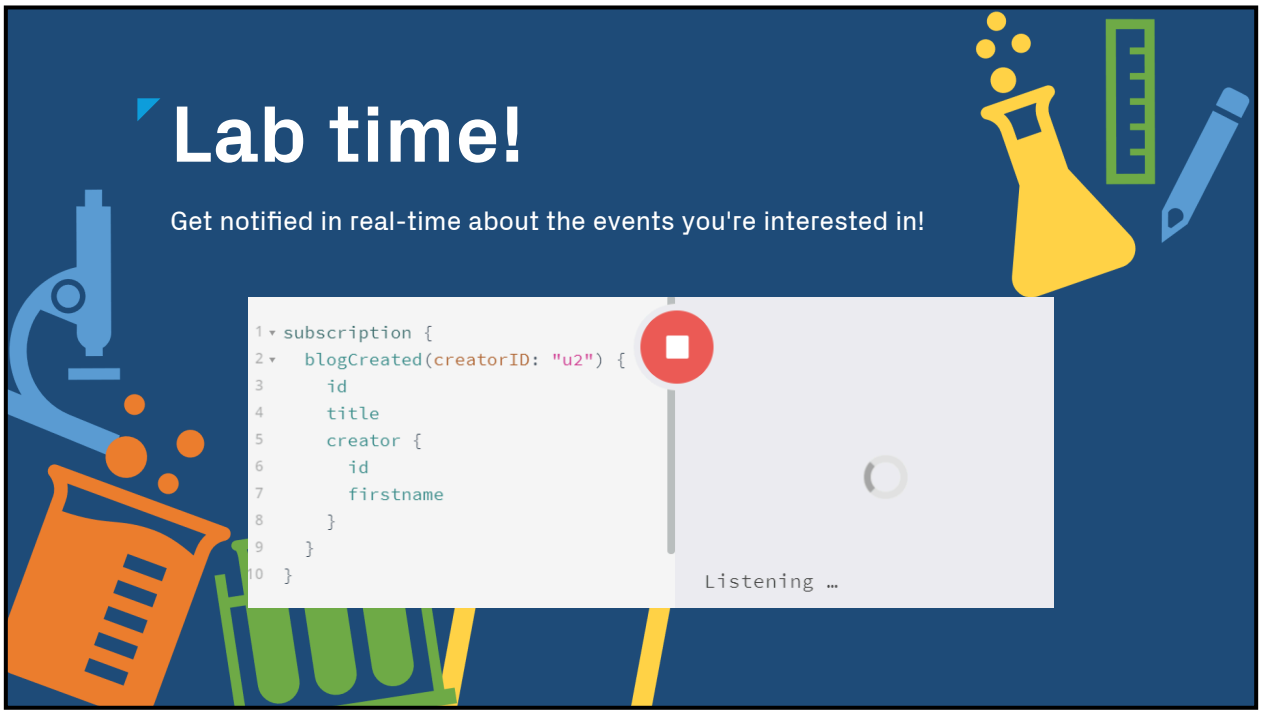
3

```
{
  "data": {
    "createUser": {
      "id": "u1622047215245",
      "email": "test@infosupport.com"
    }
  }
}
```

5

# Full Flow






# Questions?

Last chance!

# Continued Training

- **GraphQL Advanced** - 1 day
  Learn advanced GraphQL topics
- **Node.js** - 3 days
  Server-side JavaScript done right
- **React** - 3 days
  Develop Applications Using the ReactJS Framework
- **Angular** - 4 days
  Building Professional Single Page Applications with Angular
- **TypeScript** - 4 days
  Develop maintainable JavaScript applications with TypeScript

More? Check out our web development courses

# The end

Don't forget to fill out an evaluation at:

➡ http://eval (from inside Info Support network)

or

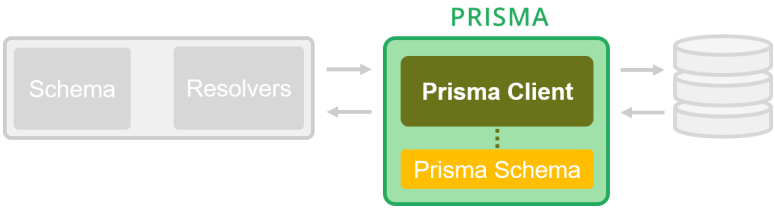
➡ https://tinyurl.com/infosupporteval

# ORM & Database

## Prisma & SQLite


- **Prisma** - open source ORM for Nodejs
- **SQLite** - small, fast, self-contained DB engine

# Prisma




- **Prisma Client** - auto-generated query builder
- **Prisma Schema** - data models & relations

# Prisma CLI

Command Line tool to build all assets.

- install CLI
  `npm install prisma --save-dev`
- initialize Prisma project
  `npx prisma init`
- in ***.env*** file set path to SQLite db file
  `DATABASE_URL="file:./dev.db"`

# Prisma Schema 1/2

- update the db provider

```
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
```

- add models to the schema by introspecting the database


```
npx prisma db pull
```

# Prisma Schema 2/2

- make all fields lowercase
- express *semantics* similar to the GraphQL Schema

```
model Podcasts {
  # 👀 Old
  Users     Users      @relation(fields: [userid], references: [id])
  Comments  Comments[]


  # 👀 New
  creator   Users      @relation(fields: [userid], references: [id])
  comments  Comments[]
}
```

# Prisma Client

- install Prisma Client
  `npm install @prisma/client`
- read Prisma schema en generate Prisma Client
  `npx prisma generate`

# Import Prisma Client

***server.js***

```
import pkg from '@prisma/client';   // 👈 import PrismaClient
const { PrismaClient } = pkg;
const prisma = new PrismaClient();  // 👈 make prisma instance

// ...

const server = new ApolloServer({
  typeDefs,
  resolvers: { ... },
  context: {
    prisma,                         // 👈 share prisma among all resolve
    pubsub,
  },
});
```

# Prisma in Resolvers

1. Replace `db` with `prisma` in argument list and in function body
2. Make resolver asynchronous by adding `async` and `await`
3. When fetching, use ***`findMany()`*** or ***`findUnique()`***
4. When mutating, use ***`create()`***, ***`update()`*** or ***`delete()`***

# Fetching

***`findMany()`*** and ***`findUnique()`***

Both support optional object argument with **`where`** property:

```
podcastById: async (_parent, { id }, { prisma }) => {
  return await prisma.podcasts.findUnique({
    where: {
      id: +id,      // unary plus: convert string to number
    }
  });
}
```

Extendable with nesting objects like:

```
findMany({ where: { firstname: { contains: namePart } } })
findMany({ take: 5 })   // first 5
findMany({ take: -5 })  // last 5
```

# Mutating

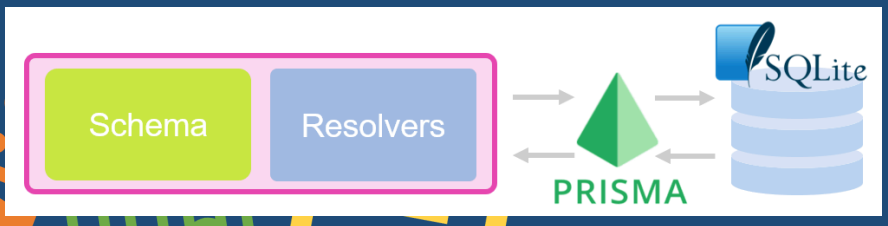
Object argument is mandatory:

- `create({ data: ... })`
- `update({ data: ..., where: ... })`
- `delete({ where: ... })`

```
updateUser: async (_parent, { id, input }, { prisma }) => {
  const user = await prisma.users.update({
    data: input,
    where: {
      id: +id,
    }
  });

  return user;
}
```

# Lab time!

App gets real database ...



# Enums

## Enum

`Enum` = enumeration type

- special kind of primitive
- restricted to a defined set of allowed values (e.g. EASY, MEDIUM, HARD)

QUERY

```
query {
  podcastByLevel(level: EASY) {
    id
    title
    length
  }
}
```

# Enum in Type System

| Primitive values | Object types |
| --- | --- |
| • Scalar | • Input Object type |
| • Enum | • Object type |
| **Abstract Types** | **Modifiers** |
| • Interface | • Non-null |
| • Union | • List |

What do you think, can an enum be used for input, output or both?

# Enum Advantages

1. **prevent typos**:
   is the Podcast's *level* defined as "Medium" or "medium"?
2. **discoverability**:

String

```
query {
  podcastByLevel(level: "") {
    id
    title
    length
  }
}
```

Enum

```
query {
  podcastByLevel(level: M) {
    id
    title
    length
  }
}
```

MEDIUM

Level

3. promotes **type reuse** across the Schema

# Enum Advantages

4. **no need to code validation logic** to check for valid values:

```
Query: {
  async podcastByLevel(_parent, { level }) {
    const validLevels = ["easy", "medium", "hard"];    // not needed
    if (!validLevels.includes(level)) {                // when using Enum
      throw new Error("Invalid level!");
    }

    return await podcasts.findMany({
      where: {
        level,
      },
    });
  };
}
```

# Enum in Schema


Enums are named using the all-caps convention.

```
enum Status { # 👀 define enum
  HIRED
  LONGTERMLEAVE
  MATERNITYLEAVE
  EXPIRED
}

type Query {
  users(first: Int!, status: Status!): [User!]! # 👀 use enum for input
}

type User {
  email: String!
  status: Status! # 👀 use enum for output
}
```

# Enum Value Resolution

How enum values are resolved is an implementation detail.

In APOLLO each enum value is resolved to its string equivalent:

*Status.HIRED* resolves to *"HIRED"*
*Status.EXPIRED* resolves to *"EXPIRED"*
...

Apollo implementation also supports **enum resolvers**.

# Enum in Resolver


Data source uses the values *'hired'*, *'ltl'* and *'expired'*.

```
const resolvers = {
  Status: {                // 👀 enum resolver
    HIRED: 'hired',
    LONGTERMLEAVE: 'ltl',
    EXPIRED: 'expired'
  },
  Query: {
    async usersByStatus(_parent, { status }) {
      return await users.findMany({ where: { status }});
    }
  },
};
```

value of Enum:

- `"ltl"` within the resolver code
- `LONGTERMLEAVE` in the public API

# Example

SCHEMA

```
enum Level {
  EASY
  MEDIUM
  HARD
}
```

SCHEMA

```
type Query {
  podcasts(level: Level!): [Podcast!]!
}
```

SCHEMA

```
type Podcast {
  title: String!
  level: Level!
}
```

RESOLVER

```
const resolvers = {
  Level: {
    EASY: 'easy',
    MEDIUM: 'medium',
    HARD: 'hard',
  },
  Query: {
    async podcastByLevel(_parent, { level }) {
      return await podcasts.findMany({
        where: { level  }
      });
    }
  },
};
```

# Lab time!

Further improve the API by using enums!

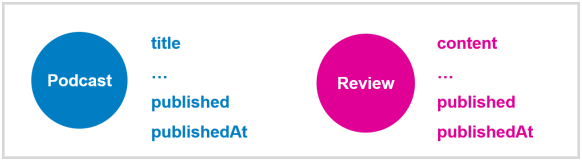
```
query {
  usersByStatus(status: ACTIVE) {
    firstname
    status
    blogs {
      title
      content
    }
  }
}
```

```
{
  "data": {
    "usersByStatus": [
      {
        "firstname": "Joop",
        "status": "ACTIVE",
        "blogs": [
          {
            "title": "blog A",
            "content": "Content of Blog A"
          },
```

# Abstract Types

## Different Objects

Assume two Object types:




Both have the fields *published* and *publishedAt*.
Both are kind of *publishable* items.

# One List

Podcast: title, published, publishedAt

Review: content, published, publishedAt

Client needs *one list of all publishable items*:

### Solution 1

```
query {
  podcasts {
    title
    published
    publishedAt
  }
  reviews {
    content
    published
    publishedAt
  }
}
```

### Solution 2

```
query {
  items {
    content
    title
    published
    publishedAt
  }
}
```

In Schema, add an extra `Result` object type:
4 fields; `title` and `content` are optional

Why are above solutions not great?

# Abstract Types

Podcast: title, published, publishedAt

Review: content, published, publishedAt

In GraphQL an **abstract type** can be used to
**return different concrete types for a single field**:

This allows for smart queries like:

```
query {
  items {
    published
    publishedAt
    ... on Podcast {
      title
    }
    ... on Review {
      content
    }
  }
}
```

# Abstract Types

GraphQL specification defines two different abstract types:

- `interface`
- `unions`

Use them to:

- reduce complexity of the schema
- reduce the number of queries and mutations
- describe data in a more precise way

# Abstract Types in Type System

| Primitive values | Object types |
| --- | --- |
| • Scalar | • Input Object type |
| • Enum | • Object type |
| **Abstract Types** | **Modifiers** |
| • Interface | • Non-null |
| • Union | • List |

Can abstract types be used for input, output or both?

# Interface

`Interface` - a *contract*:
list of fields which may be implemented by object types

1. interface has fields, but is never instantiated
2. concrete object types implement the interface
3. fields may return interface types

# Define Interface


`interface` has fields, but is never instantiated.

```
interface Publishable {
  published: Boolean!
  publishedAt: Int
}
```

- cannot be used in a query directly

Can an interface be used as type for an input argument?

# Implement Interface


Concrete object types implement the interface.
They must include all fields of the interface.

```
type Review implements Publishable {
  content: String!
  published: Boolean!      # 👀
  publishedAt: Int         # 👀
}

type Podcast implements Publishable {
  title: String!
  published: Boolean!      # 👀
  publishedAt: Int         # 👀
}
```

# Return


Fields may return interface types.
Returned object may be any member of that interface.

```
type Query {
  items: [Publishable]
}
```

# GraphQL Specs


## Resolving an Abstract Type

> *When completing a field with an abstract return type, that is an Interface or Union return type, first the abstract type must be resolved to a relevant Object type. This determination is made by the internal system using whatever means appropriate.*

# Resolve Interface


In APOLLO a `__resolveType` function must be defined.

```
const resolvers = {
  Publishable: {
    __resolveType(item, _context, _info) {
      if (item.length) return "Podcast";     // 👀 name of type
      if (item.content) return "Review";     // 👀 name of type
      return null;                           // 👀 GraphQLError is shown
    },
  },
  Query: {
    items: () => [...testPodcasts, ...testReviews],
  },
};
```

# Inline Fragment


Access data on the underlying concrete type.

**Inline Fragments** use `... on`

In an earlier module we saw **Named Fragents**. These also use the spread operator `...` without `on`.

```
1  query {
2    items {
3      published
4      publishedAt
5      ... on Podcast {
6        title
7      }
8      ... on Review {
9        content
10     }
11   }
12 }
```

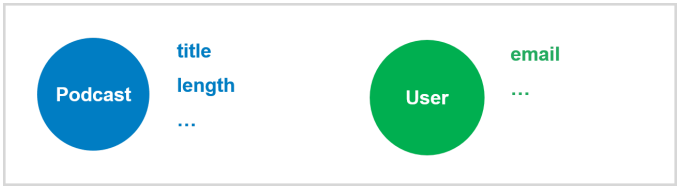
# `__typename`


Use meta field `__typename` to get the name of the concrete type.

```
1  query {
2    items {
3      published
4      publishedAt
5      __typename
6      ... on Podcast {
7        title
8      }
9      ... on Review {
10       content
11     }
12   }
13 }
```

# Union

**union** - another abstract type, used when:
**concrete object types don't have any significant fields in common**




# Define Union

SCHEMA

```
union Result = Podcast | User
```

A **resolver** is then needed to determine the actual type:

RESOLVER

```
__resolveType(obj, _ctx, _info) {
  if (obj.email)
    return 'User';
  if (obj.title)
    return 'Podcast';
  return null;          // GraphQLError is thrown
}
```

# Errors

In GraphQL sending back errors to the client is different compared to REST.

- consider HTTP error codes (404, 500 etc.) server- and framework-level errors
- send domain-level errors to clients in an object type:


These will all have an HTTP status code of 200 (= successful request)!

# Domain Errors 1/3


**Expert tip** - Use a combination of a *union* and an *interface*.

Mutation returns a special object type `CreatePodcastPayload`.

SCHEMA

```
type Mutation {
  createPodcast(input: CreatePodcastInput): CreatePodcastPayload!
}

input CreatePodcastInput { ... }    # title and length

union CreatePodcastError = TooManyPodcasts | MissingFields    # 👀 union

type Podcast { ... }                # id, title and length

type CreatePodcastPayload {
  errors: [CreatePodcastError]!     # always return a list with errors
  podcast: Podcast                  # optionally return the created podc
}
```

# Domain Errors 2/3


The object types defined in the union all implement an Error *interface*.

SCHEMA

```
interface Error { # 👀 interface
  message: String!
  code: Int!
}

type TooManyPodcasts implements Error {
  message: String!
  code: Int!
}

type MissingFields implements Error {
  message: String!
  code: Int!
}
```

# Domain Errors 3/3


Query the full payload in a mutation:

QUERY

```
mutation {
  createPodcast(input: { title: "a new title", length: 78 }) {
    podcast {
      title
      length
    }
    errors {
      ... on Error {
        __typename
        code
        message
      }
    }
  }
}
```

# Lab time!

More flexibility with abstract types


# Custom Scalars

## Scalars

GraphQLs type system supports two *atomic* types:

- Scalar (built-in: `Int`, `Float`, `String`, `Boolean`, `ID`)
- Enum

Certain scalars are 'missing', like: `Byte` and `DateTime`

That's why GraphQL supports *custom scalars*.

## AST and Coercion

Implement custom scalars by:

1. using a well-supported library (**graphql-scalars** for Nodejs)
2. building and using self-coded custom scalars

Both scenarios require basic knowlegde about:

- Abstract Syntax Tree (AST)
- Coercion

# Abstract Syntax Tree

**query**

```
query MyTest {
  userById(id: "u2") {
    id
    email
  }
}
```

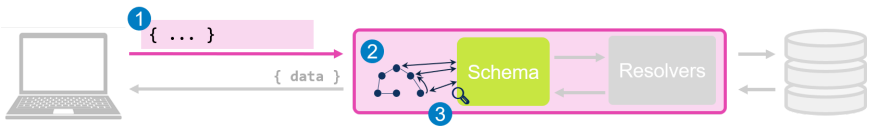
QUERY

**AST** (JSON representation)

```
{
  "kind": "Document",
  "definitions": [
    {
      "kind": "OperationDefinition",
      "operation": "query",
      "name": {
        "kind": "Name",
        "value": "MyTest",
        "loc": {
          "start": 6,
          "end": 12
        }
      },
      // ...
```

- graph representation of source code
- used by other compilers as well (C++, TypeScript, Babel, ...)

# AST in GraphQL




1. client's request is a single string
2. GraphQL parses this string into an AST
3. GraphQL processes the query:
   a. traverses the AST
   b. executes each part against the schema (performing validation)
   c. maps schema types onto the respected *branches* in the AST (metadata)

What might be the result of the validation?

# Coercion

A scalar provides **type-safety** about:

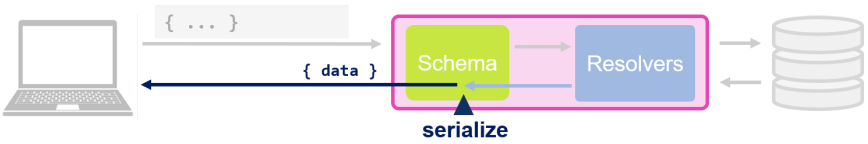
- returned values
- values used as input

Per scalar, **three functions** are implemented to guarantee type-safety:

- Result Coercion
- Literal Input Coercion
- Value Input Coercion

**coercion** - each function validates and *converts* a provided value

# 1. Result Coercion

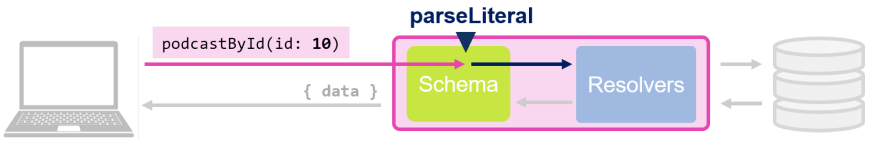



**serialize** - convert outgoing scalar type to JSON when sent back to client

Details depend on implementation!

For example in *GraphQL.js* (Facebook's reference implementation):
**Boolean** - only `true`, `false`, `1` or `0` are serialized
(eg. 1 becomes true, "apple" raises an error)

# 2. Literal Input Coercion

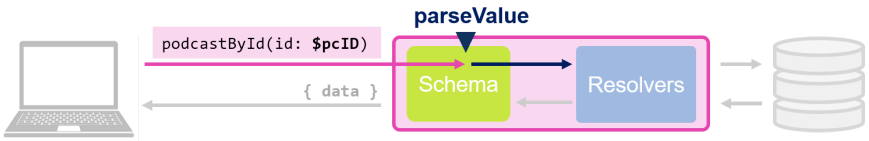



**parseLiteral** - parse client input that was passed **inline** in the query

Details much less platform dependent as input is cleary defined.

For example in GraphQL spec:
**Int** - only integer is accepted (eg. 12 becomes 12, 12.3 raises an error)
**ID** - only string or integer is accepted

# 3. Value Input Coercion




**parseValue** - parse client input that was passed through **variables**

For built-in scalars: same rules apply as parsing input passed inline.

# Custom Scalar

In SDL:

```
scalar MyScalar
```
SCHEMA

Shape of Scalar:

- `name`
- `description`
- `serialize(value)` - used for output
- `parseLiteral(ast)` - used for literal input (note that `ast` is used!)
- `parseValue(value)` - used for variable input

Where do you think the description will become visible?

# Code 1/3

APOLLO

Podcast has nullable *`publishedDate`* field (eg. '2021-06-14') in data source.

Define new scalar **`Date`**.

```
import { GraphQLScalarType, Kind } from "graphql";

const dateScalar = new GraphQLScalarType({
  name: "Date",
  description: "Superhandy Date custom scalar type",
  serialize(value) {
    /* ... */
  },
  parseValue(value) {
    /* ... */
  },
  parseLiteral(ast) {
    /* ... */
  },
});
```

# Code 2/3

APOLLO

Implement the three methods.

```
serialize(value) {
  return value;
},
parseValue(value) {
  if (typeof value === 'string' && value.length === 10) {
    return new Date(value);
  }
  return null;
},
parseLiteral(ast) {                  // Abstract Syntax Tree string
  if (ast.kind === Kind.STRING) {    // provides the helper Kind
    return new Date(ast.value);
  }
  return null;           // Invalid hard-coded value (not an integer)
},
```

# Code 3/3

APOLLO

Add the defined `dateScalar` to the resolvers map.

RESOLVER

```
const resolvers = {
  Date: dateScalar, //  👀 defined scalar
  Query: {
    podcasts(parent) {
      return testPodcasts;
    },
  },
};
```

# Update Schema

SCHEMA

```
scalar Date # 👀 Define new scalar type
type Query {
  podcasts: [Podcast!]!
  podcastsPublishedSince(startDate: Date!): [Podcast!]! # 👀 Use for INP
}

type Podcast {
  id: ID!
  title: String!
  publishedDate: Date # 👀 Use for OUTPUT
}
```

# Add Resolvers

RESOLVER

```
const resolvers = {
  Date: dateScalar,
  Query: {
    podcasts: parent => testPodcasts,
    podcastsPublishedSince(parent, args) {
      return testPodcasts.filter(podcast => {
        return new Date(podcast.publishedDate) > args.startDate;
      });
    },
  },
};
```

## Test

APOLLO

```
query ($startDate: Date!) {
  podcastsPublishedSince(startDate: $startDate) {
    title
    publishedDate
  }
}
```

parseValue()

```
{
  "startDate": "2021-06-22"
}
```

```
{
  "data": {
    "podcastsPublishedSince": [
      {
        "title": "We love TypeScript",
        "publishedDate": "2021-06-28"
      }
    ]
  }
}
```

serialize()

## Lab time!

Custom scalars for input validation and output formatting

Custom Scalars

# Directives

## Why Directives?

Directives enable a GraphQL server to provide custom capabilities:

- caching
- data formatting
- improve performance
- logging
- data mocking
- security
- ...

# Directive

**`Directive`** - function that augments some other functionality.

- defined using the **`@`** symbol
- kind of aspect-oriented programming (AOP)


custom directive **`@camelCase`**
can be applied to any string field

# Directive Types

- **schema directives** - executed when building the schema

SCHEMA

```
type Podcast {
  title: String!
  price: Float @money    # 👀 defined in the SDL
}
```

- **operation directives** - executed when resolving the query

QUERY

```
query {
  podcasts {
    id
    name @uppercase    # 👀 appears in the client query
  }
}
```

# Directive Arguments

- 0 or more arguments
- can be optional:

SCHEMA

```
type Image {
  id: ID!
  height: Int! @deprecated
  small: String! @deprecated(reason: "Use picture instead!")
  picture: String!
}
```

Are arguments always *named* arguments?

# Default Directives

To be spec-compliant, an implementation must support 4 directives:

| Directive | Description |
|---|---|
| `@deprecated(reason: String)` | marks the schema definition of a field (or enum value) as deprecated, optional argument: reason |
| `@specifiedBy(url: String!)` | specify the behavior of a *custom scalar type*, mandatory argument: url |
| `@skip(if: Boolean!)` | if true, the decorated field (or fragment) in an operation is not resolved |
| `@include(if: Boolean!)` | if true, the decorated field (or fragment) in an operation is resolved |

Which are map-to-schema directives and which map-to-operation?

# Default Operation Directives

- **`@include(if: Boolean)`**
  Allow client to include field(s) based on value of mandatory **`if`** argument.

```
query ($showCreator: Boolean!) {
  podcasts {
    title
    creator @include(if: $showCreator) {
      email
    }
  }
}
```

```
{
  "showCreator": true
}
```

```
{
  "data": {
    "podcasts": [
      {
        "title": "GraphQL",
        "creator": {
          "email": "u1@live.com"
        }
      },
      ...
    ]
  }
}
```

when showCreator is true

```
{
  "data": {
    "podcasts": [
      {
        "title": "GraphQL"
      },
      ...
    ]
  }
}
```

when showCreator is false

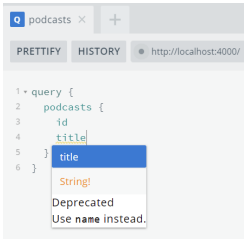
- **`@skip(if: Boolean)`**
  The other way around ...

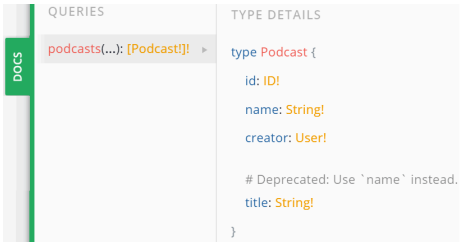
# Default Schema Directives 1/2

- **`@deprecated(reason: String)`**

SCHEMA

```
# Define in schema where applicable 👇
type Podcast {
  # ...
  title: String! @deprecated(reason: "Use \`name\` instead.")
  name: String!
}
```



# Default Schema Directives 2/2

- `@specifiedBy(url: String!)`

**url** - declare how custom scalar types behaves

Custom scalar type *UUID*:

```
scalar UUID @specifiedBy(url: "https://tools.ietf.org/html/rfc4122")
```
SCHEMA

- added to the spec ~June 2021
- APOLLO not implemented yet

# Directives Improving Latency

- `@defer(if: Boolean, label: String)`
- `@stream(initialCount: Int!, label: String)`

RFC for these 2 directives is in the Proposal Stage:

https://github.com/graphql/graphql-wg/blob/main/rfcs/DeferStream.md


(these directives are *not* in Apollo Server)

# @stream

contrawork/
**graphql-helix**

Mimick slow retrieval of users:

**Resolver**

RESOLVER

```
Query: {
  users: async function* () {
    for (const user of db.testUsers) {
      await new Promise((resolve) => setTimeout(resolve, 1000));  // 👀
      yield user;
    }
  },
}
```

# @stream

contrawork/
**graphql-helix**

Assume retrieval of 5 users.

QUERY

```
query {
  users {
    id
    email
  }
}
```

after 5 seconds,
all users retrieved at once

QUERY

```
1 query {
2   users @stream(initialCount: 1) {
3     id
4     email
5   }
6 }
```

after 1 second,
first user is retrieved, afterwards
each second a user is retrieved

## @defer

contrawork/
**graphql-helix**

Assume retrieval of creator's details is delayed.

```
query {
  podcasts {
    id
    title
    length
  }
}
```

fast retrieval of podcasts

```
query {
  podcasts {
    id
    title
    length
    creator {
      id
      email
    }
  }
}
```

retrieval of *all* data delayed due to slow retrieval of creators data

```
query {
  podcasts {
    id
    title
    length
    creator {
      ... x
    }
  }
}

fragment x on User {
  id
  email
}
```

similar, now using a name fragment

```
query {
  podcasts {
    id
    title
    length
    creator {
      ... x @defer
    }
  }
}

fragment x on User {
  id
  email
}
```

retrieve podcasts (fast), defer retrieval of creators

## Experimental Support

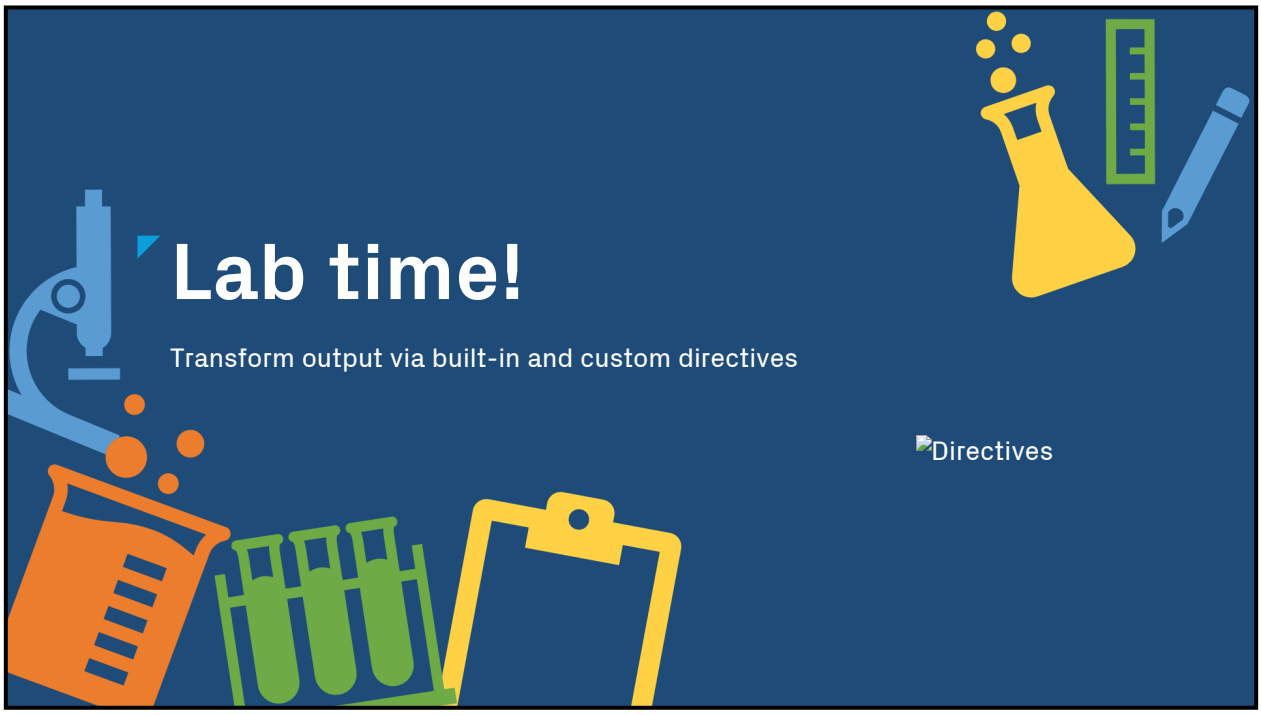
contrawork/
**graphql-helix**

***package.json***

```
  "dependencies": {
    "graphql": "15.4.0-experimental-stream-defer.1",
    ...
  }
```

Implementation uses a multipart response:

```
res.writeHead(200, {
  Connection: "keep-alive",
  "Content-Type": 'multipart/mixed; boundary="-"',
  "Transfer-Encoding": "chunked",
});
```




# Introspection in GraphQL

GraphQL's schema can be queried by GraphQL!

Introspection queries allow:

- clients to get a complete view of all resources
- rich IDEs like *GraphiQL* to provide self-documentation
- the community to build more tools for GraphQL

Why is introspection possible in GraphQL?

# Query

Server exposes 3 introspection queries on the Query operation type:

- `__schema`
- `__type`
- `__typename`

All fields in the introspection system are prefixed with 2 underscores: `__`

## __schema

Built-in schema definition:

SCHEMA
```
__schema: __Schema!
```

SCHEMA
```
type __Schema {
  types: [__Type!]!
  queryType: __Type!
  mutationType: __Type
  subscriptionType: __Type
  directives: [__Directive!]!
}
```

QUERY
```
query {
  __schema {
    types {
      name
      description
    }
  }
}
```

QUERY
```
query {
  __schema {
    directives {
      name
      description
    }
  }
}
```

## Description

Improve documentation by adding descriptions.

SCHEMA
```
"""User can be a Podcast Creator or Commentor"""
type User {
  id: ID!

  """User's firstname"""
  firstname: String!
  # ...
}
```

Apollo: embed your description in: `""" """`
graphql-js: start your description with `#`

As introspection queries return these descriptions, IDEs will do as well.

## __type

Query for a type specified by the **name** argument.

QUERY

```
query {
  __type(name: "User") {
    kind
    name
    description
    fields(includeDeprecated: true) {
      name
      description
    }
  }
}
```

## __typename

Available through all types when querying.

- Get name of concrete type in scenarios where abstract types are used.
- Apollo Client caches entities by using a combination of **id** and **__typename**.

QUERY

```
query {
  viewer {
    login
    repositories(first: 3) {
      nodes {
        id
        name
        __typename
      }
    }
  }
}
```

# Handy query 1

Which queries can a client send to an API?

List all root queries:

QUERY

```
query allAvailableQueries {
  __schema {
    queryType {
      fields {
        name
        description
      }
    }
  }
}
```

# Handy query 2

Which possible values does an enumerator has?

QUERY

```
query enumerationValues {
  __type(name: "UserStatus") {
    kind
    name
    description
    enumValues {
      name
      description
    }
  }
}
```

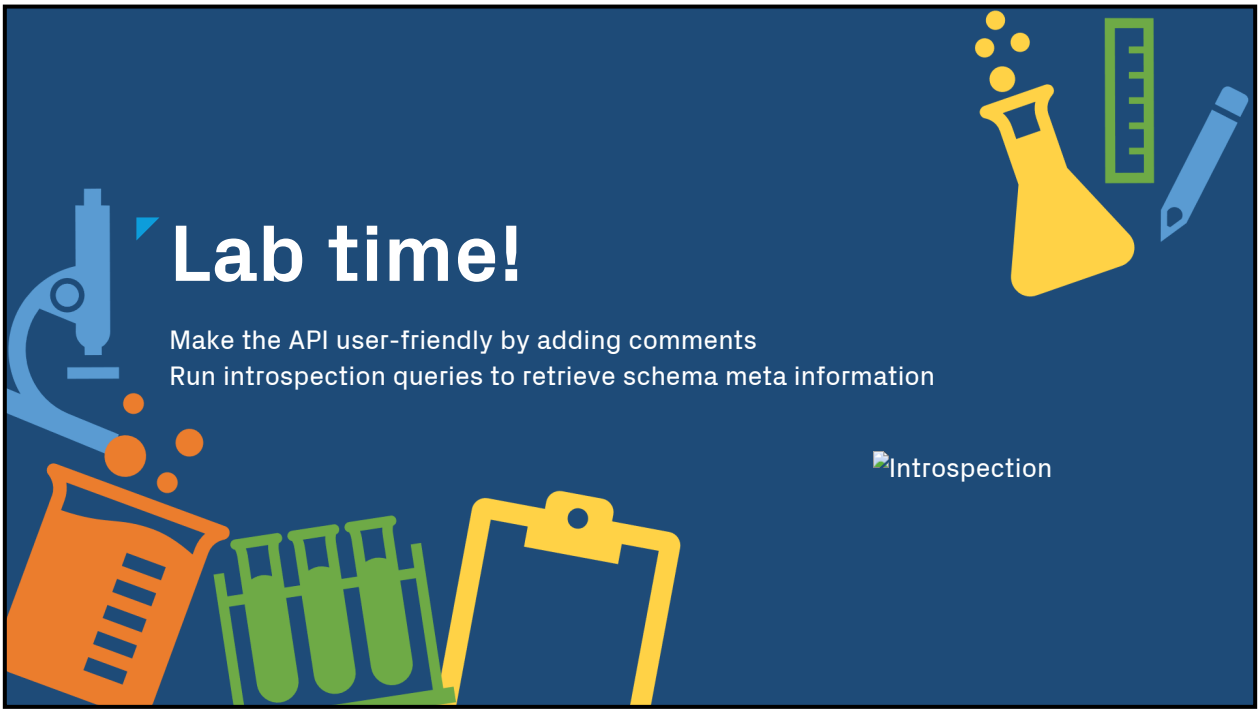
# Handy query 3

Which concrete object types are:

- implemented by an **`interface`**, or
- associated with a **`union`**

QUERY

```
query abstractTypes {
  __type(name: "Publishable") {
    kind
    name
    description
    possibleTypes {
      name
      kind
      description
    }
  }
}
```



# Solving the N+1 problem

## N+1 problem

For every **1** query that return **n** results,
**n** additional queries are needed!

```
query {
  podcasts {
    id
    title
    creator {
      id
      email
    }
  }
}
```

run **1** query: get all podcasts

```
SELECT * FROM Podcasts
```

run **n** queries: per podcast get creator

```
SELECT * FROM Users WHERE id = 1
SELECT * FROM Users WHERE id = 2
SELECT * FROM Users WHERE id = 1
SELECT * FROM Users WHERE id = 2
SELECT * FROM Users WHERE id = 3
```

# Dataloader

Most popular solution for the n+1 problem

- use in data fetching layer
- batching queries & caching results
- exists since 2010, actively maintained
- implementations for many program languages: .NET, Java, Python, ...

https://github.com/graphql/dataloader

# Dataloader concepts

```
query {
  podcasts {
    id
    title
    creator {
      id
      email
    }
  }
}
```

the creator resolver:

1. returns a **promise** instead of a value
2. calls a **data loader** instead of the data source


the data loader:

- collects user ids required to fetch creators
- batch load the data in a more efficient way

# Dataloader methods


Main methods:

- ***load(identifier)***
  input 👆 identifier of data the caller is interested in
  output 👉 promise which will be fulfilled later
- ***perform(batchfunction)***
  use all accumulated keys and loads the data in most efficient way

`perform()` runs automagically *after* all promises are enqueued
implementation details vary per environment

# Implement Dataloader

1. In constructor of Dataloader provide an asynchronous **`batchfunction`**:

```
const creatorBatchFunction = async (keys) => {

  // 👀 part 1: single database request: translates to WHERE id IN (1, 2
  const creators = await prisma.users.findMany({
    where: { id: { in: keys } }
  });

  // 👀 part 2: return users in specific structure expected by data load
};
```

2. Share dataloader instance in GraphQL context
3. Use dataloader instance in resolver to load the unique identifiers

# Alternative 1: Use a JOIN

- Pro: 1 roundtrip instead of 2
- Con: when creator data is not requested in client queries, it's still retrieved

```
function podcastsFormatter(podcasts) {
  return podcasts.map((podcast) => ({
    ...podcast,
    creator: { id: podcast.userid, email: podcast.email },
  }));
}
```

```
podcasts: async (_parent, _arg, { prisma }) => {
  const q = 'SELECT * FROM Podcasts AS p JOIN Users AS u ON p.userid = u
  const result = await prisma.$queryRaw(q);
  return podcastdFormatter(result);
};
```

# Alternative 2: ORM features

Prisma has a data loader aboard.

***`findUnique()`*** queries are automagically batched when they:

1. occur in the same tick, and
2. use the same where and include parameters

details:
https://www.prisma.io/docs/guides/performance-and-optimization/query-optimization-performance

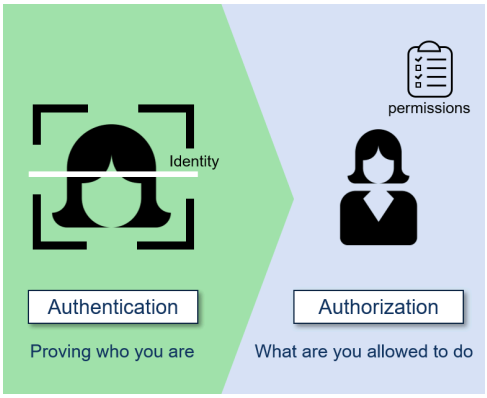
# Lab time!

Use a dataloader when retrieving creators of blogs to prevent many many database calls

Dataloader

# Authentication & Authorization

# Different things




Which **authentication** methods do you know?

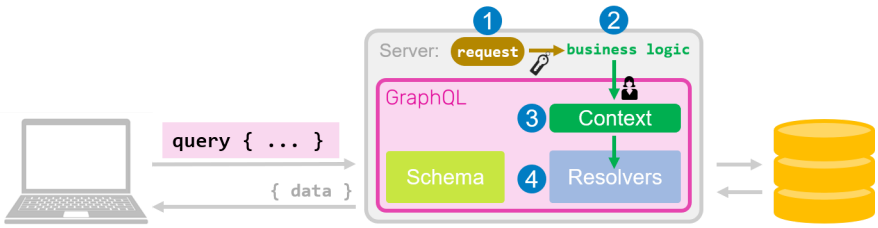
# Separation of concerns

Separate business logic and authentication logic.

Also GraphQL resolvers shouldn't know about authentication logic. Why?

- Best practice:
  authentication implementation provides the **currentUser** to the resolvers via the **GraphQL context**

# Authentication flow




1. contains *Authorization* header with info like a token
2. trades the token for the user
3. context contains the (current) user
4. each resolver function can access the context via its 3rd argument

# Authentication implementation

```
import ApolloServer from 'apollo-server';

const server = new ApolloServer({
  typeDefs,
  resolvers,
  async context({ req }) {
    const token = req.headers.authorization || ''; // 👀 get token
    const user = getUser(token);                    // 👀 trade token for
    return { user };                                // 👀 add user to con
  },
});

server.listen().then(() => console.log('🚀 Server ready at port 4000'));
```

# Authorization

At the end of the authentication flow the context contains info like:

- `{ isAuthenticated: true }` or
- `{ id: '22', roles: ['reviewer', 'editor'] } }`

This will be input for the authorization logic.

❓ Where would you implement authorization logic in GraphQL?

# Authorization ▶ in context

No *public* APIs?

👉 Authorize once in the context

```
context: ({ req }) => {
 const token = req.headers.authorization || '';
 const user = getUser(token);

  // 👇 Block the user (possibly with user roles/permission checks)
  if (!user) throw new AuthenticationError('you must be logged in');

 return { user };
},
```

# Authorization ▶ in resolvers

Both *restricted* and *public* APIs?

👉 Authorize within each resolver

```
async allPodcasts(_parent, _args, { prisma, user }) {
  if (!user || !user.roles.includes('admin')) {
    return null; // or throw an error
  }
  return await prisma.podcasts.findMany();
};
```

❓ What is a disadvantage of writing authorization logic in resolvers?

# Authorization ▶ *before* resolvers

Good alternatives to authorization logic directly in resolvers:

- **guard** - wrap resolvers which need protection in a reusable function
- **custom directives** - an `@auth` schema directive
- **data model** - models in the context, each generated via a function:

```
export const generatePodcastModel = ({ user }) => ({
  async getAll() {
    if (!user || !user.roles.includes('admin')) return null;
    return await prisma.comments.findMany();
  },
});
```

# Authorization ▶ in other api

If GraphQL wraps another API with built-in Authentication/Authorization, then simply pass the request object.

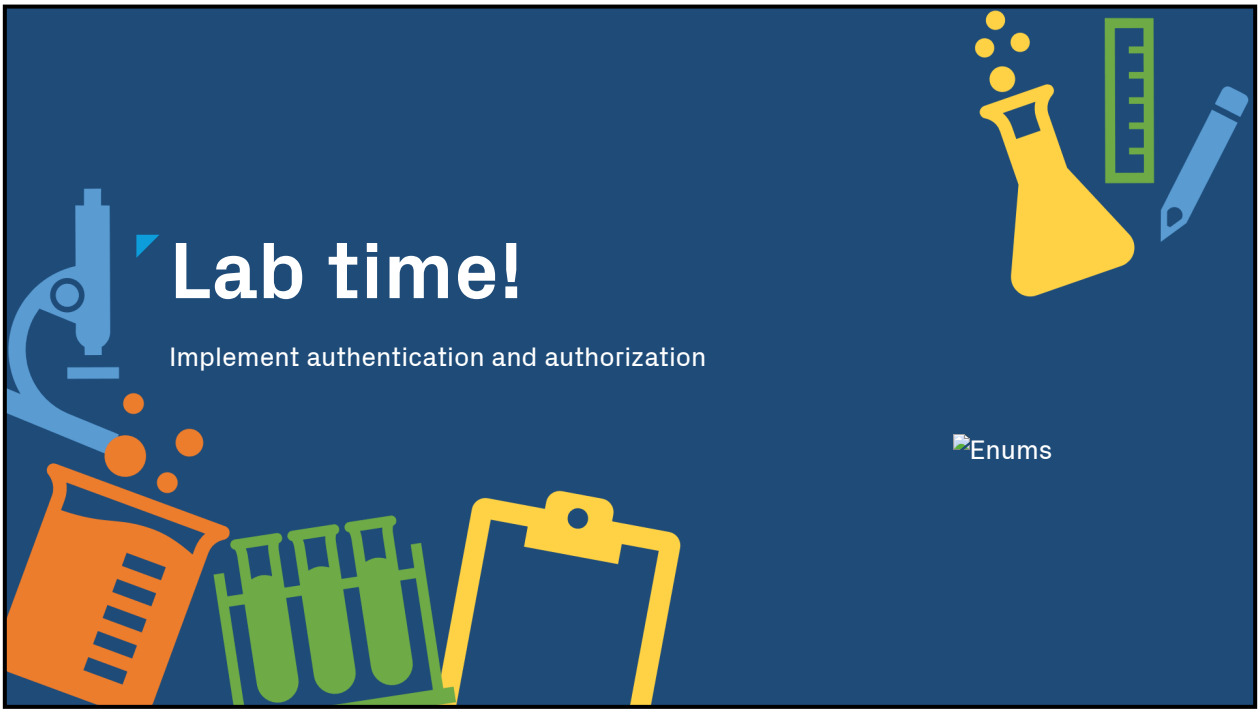
```
context: ({ req }) => ({
  user,
  models: { User: generateUserModel({ req }), /* ... */ }
}),
```

Other endpoint then can use whatever header or cookie it needs.

```
export const generateUserModel = ({ req }) => ({
  getAll: () => fetch('http://other-api.com/users', { headers: req.heade
});
```

# Lab time!

Implement authentication and authorization



# Wrap-up

# Questions?

Last chance!

# Continued training

- **Node.js** - 3 days
  Server-side JavaScript done right
- **React** - 3 days
  Develop Applications Using the ReactJS Framework
- **Angular** - 4 days
  Building Professional Single Page Applications with Angular
- **TypeScript** - 4 days
  Develop maintainable JavaScript applications with TypeScript
- **Pragmatic JavaScript** - 4 days
  Applying JavaScript in practice

More? Check out our **web development courses**

# The end

Don't forget to fill out an evaluation at:

➡ http://eval (from inside Info Support network)

or

➡ https://tinyurl.com/infosupporteval