GRAPHQL

Stanko Krtalic Rusendic

github.com/Stankec
@monorkin

Hi! For those who don't know me, my name is Stanko. If you want to see my work or tweet at me here is the needed information.

I've been a Ruby engineer for the better part of the last three years. Lately I've been switching to functional oriented programming and exploring Rust, so perhaps some of you know me from the Lambda Zagreb meetup.

This brings me to my employer, FloatingPoint. These guys are actually paying me to hold this lecture and sponsorde my reasearch into this subject and the development of a few tools that will soon be published. (Soon here is expressed in Valve time)

01

REST PROBLEMS

So let's begin.
Over the years I've grown tired of REST.

Representational state transfer

I think by now everybody knows this… So REST is a way of organizing your application's resources at different enpoints so that they are easy to explore and retrieve.
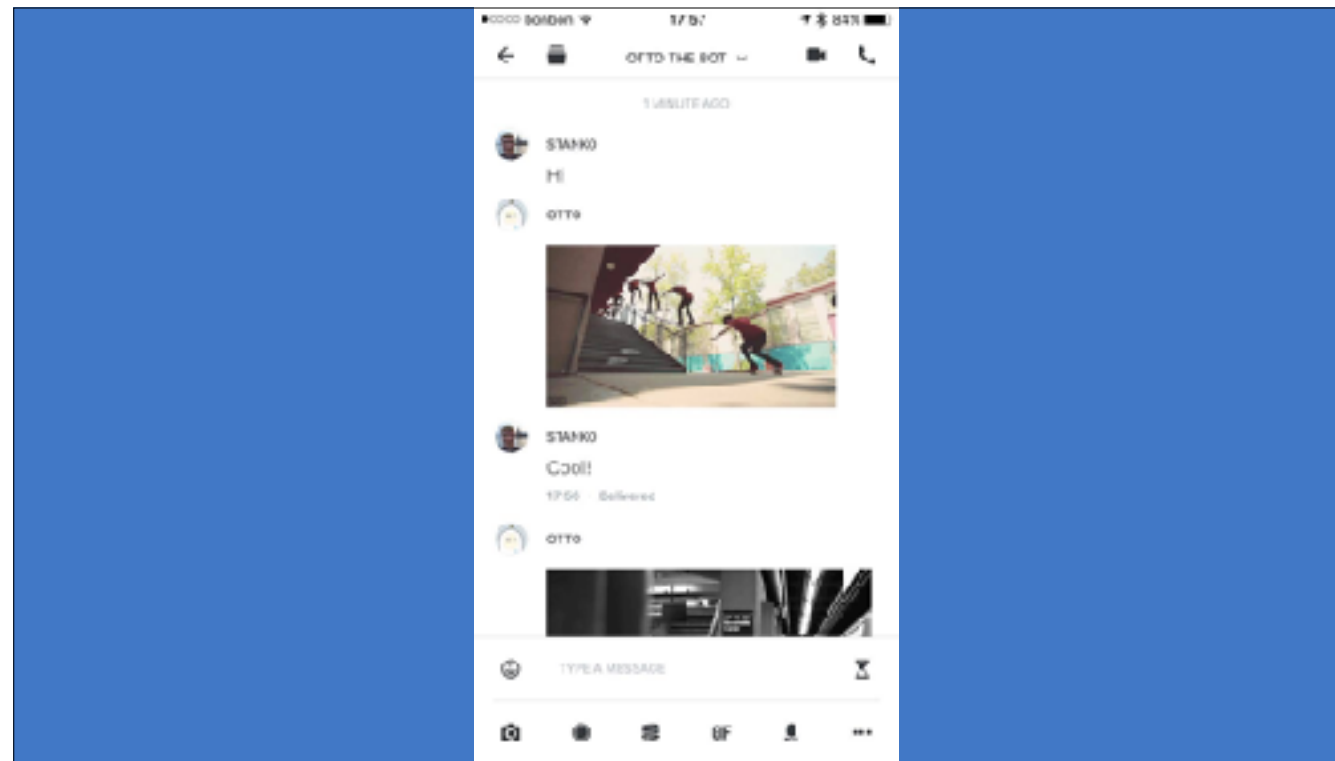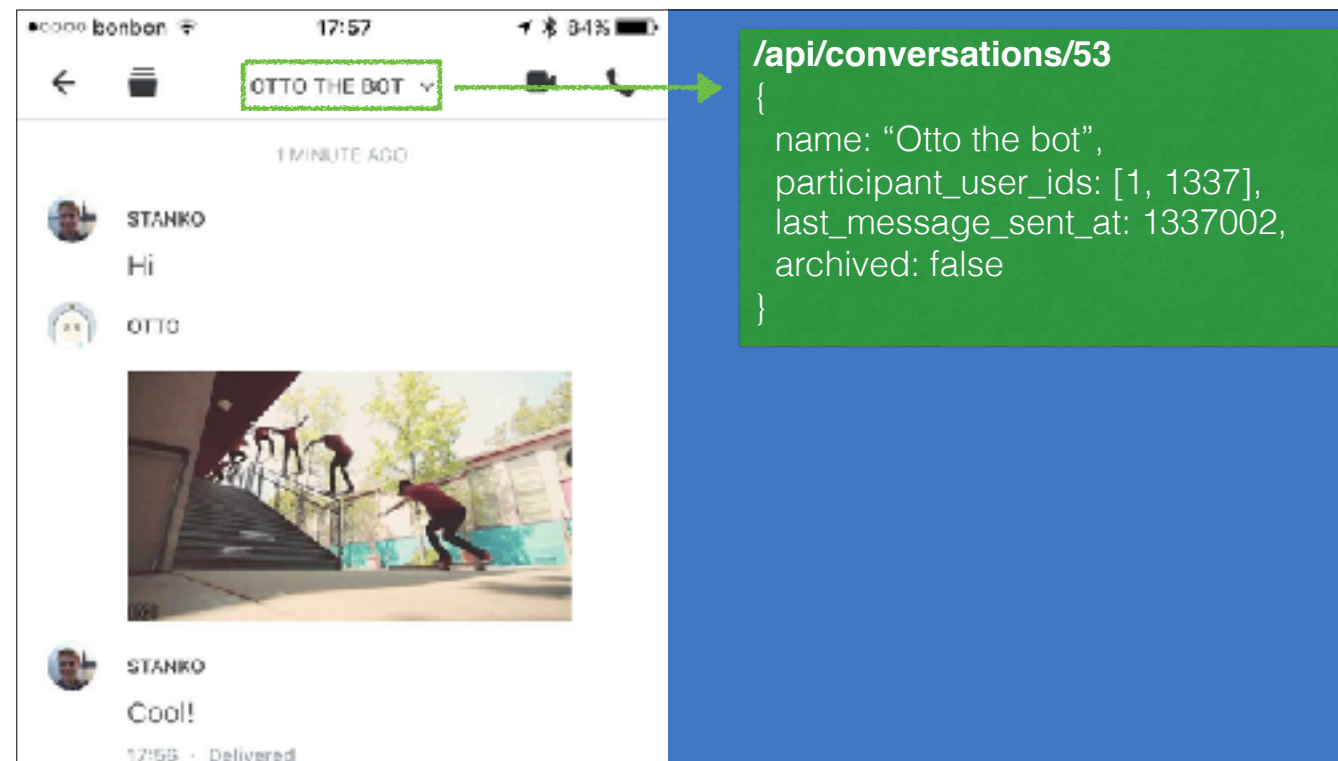
Let's go through how you would model a RESTful API for this simple chat application

IN THEORY

So the idea behind REST is to have resources at different endpoints that corespond to the resource's type and it's relation to other resources (e.g. it's children / associations).

We can see that the chat application has a 'conversations' resource.
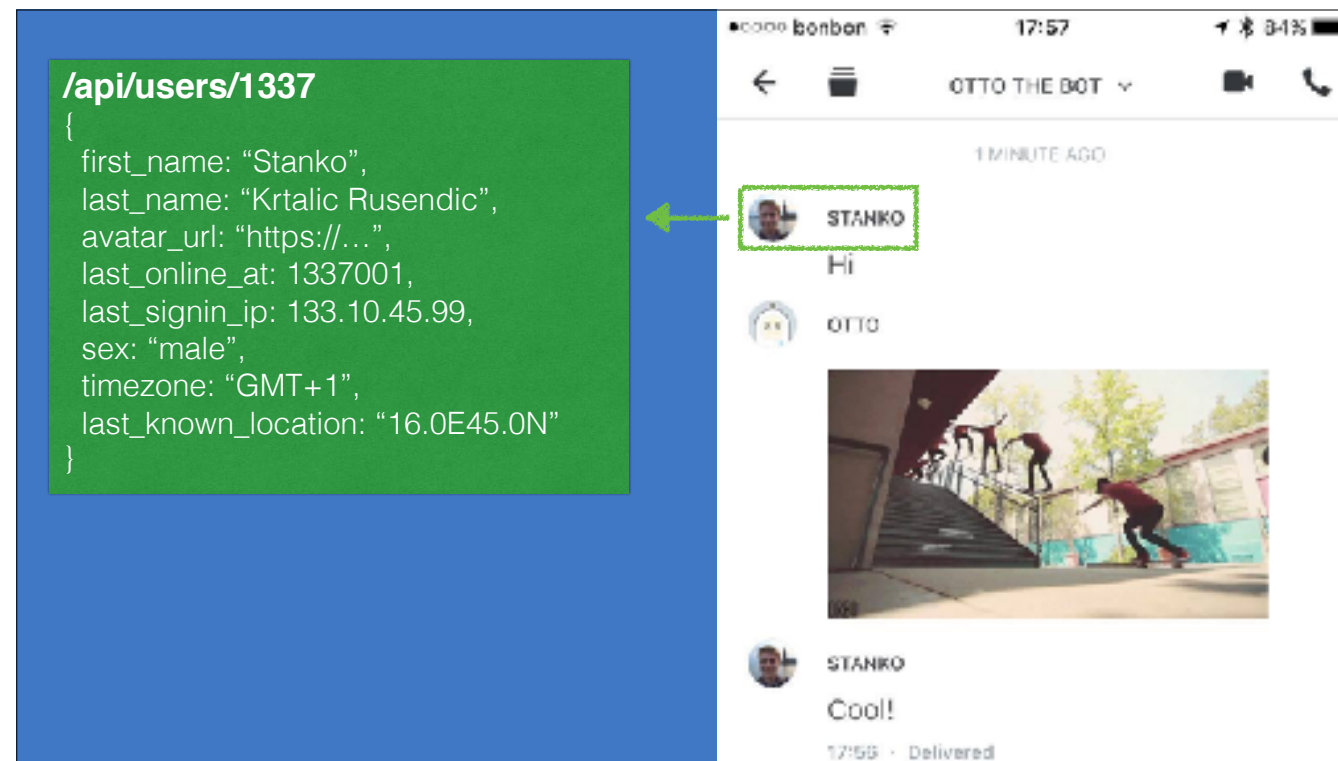
This means that you would have a conversations endpoint which serves the name of the conversation, references to it's participants and similar information…

/api/conversations/53/message/1

{
  message_type: "plain",
  body: "Hi",
  sender_id: "1337",
  created_at: 1337001,
  updated_at: 1337001,
  previous_version_ids: [],
  status: "delivered",
  seen_by_participant_ids: [1]
}
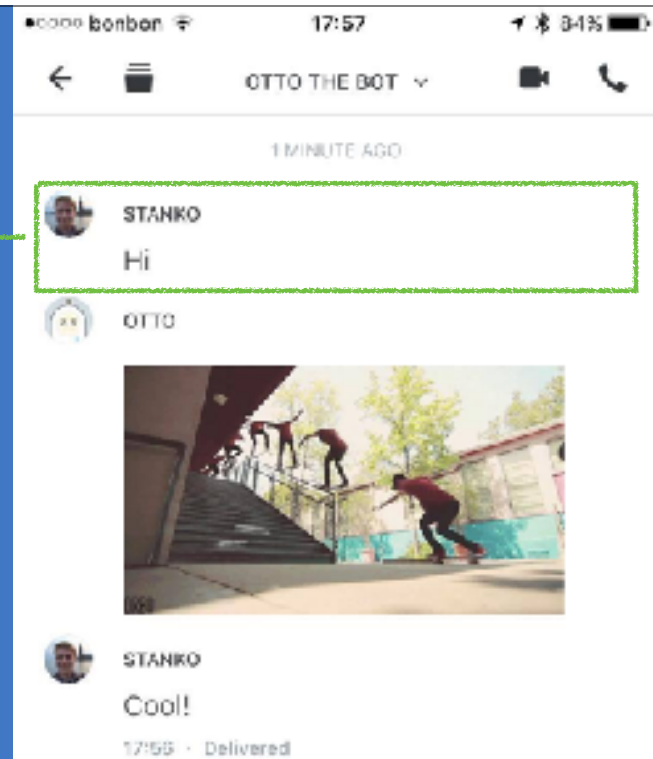
Then you would have an endpoint for each individual message… Note here that the messages endpoint is nested within the endpoint for the individual conversation, this is done so to indicate that the messages are strongly associated with the conversation (think of a parent-child association).

```
/api/users/1337
{
  first_name: "Stanko",
  last_name: "Krtalic Rusendic",
  avatar_url: "https://…",
  last_online_at: 1337001,
  last_signin_ip: 133.10.45.99,
  sex: "male",
  timezone: "GMT+1",
  last_known_location: "16.0E45.0N"
}
```
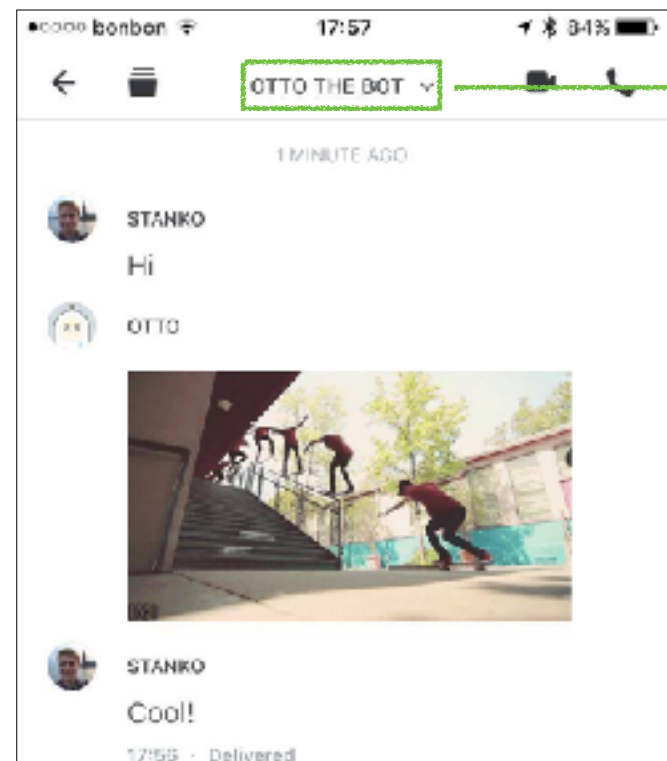
And then you would also have an endpoint to get the participants who are actually users, and since we want to be able to reference those even without it's associated conversation we put the endpoint in the root API path.

IN REALITY

I just want to qoute my new favorite qoute here "In theory, theory and practice are the same. In practice, not so much…"

/api/conversations/53

{
  name: "Otto the bot",
  participant_user_ids: [1, 1337],
  last_message_sent_at: 1337002,
  last_message_sent_by: {
    first_name: "Stanko",
    last_name: "Krtalic Rusendic"
  }
  archived: false,
  last_message: {
    type: "plain",
    body: "Cool!",
    status: "delivered"
  }
}

Usually the conversations endpoint will serve a lot of information that doesn't actually belong to that resource. E.g. The sender of the latest message with this full data, the content's of the last message, …

/api/conversations/53/message/1

{
  message_type: "plain",
  body: "Hi",
  sender_id: "1337",
  created_at: 1337001,
  updated_at: 1337001,
  previous_version_ids: [],
  status: "delivered",
  seen_by_participants: [
    { first_name: "Otto" }
  ],
  sender: {
    first_name: "Stanko",
    avatar_url: "https://…"
  }
}
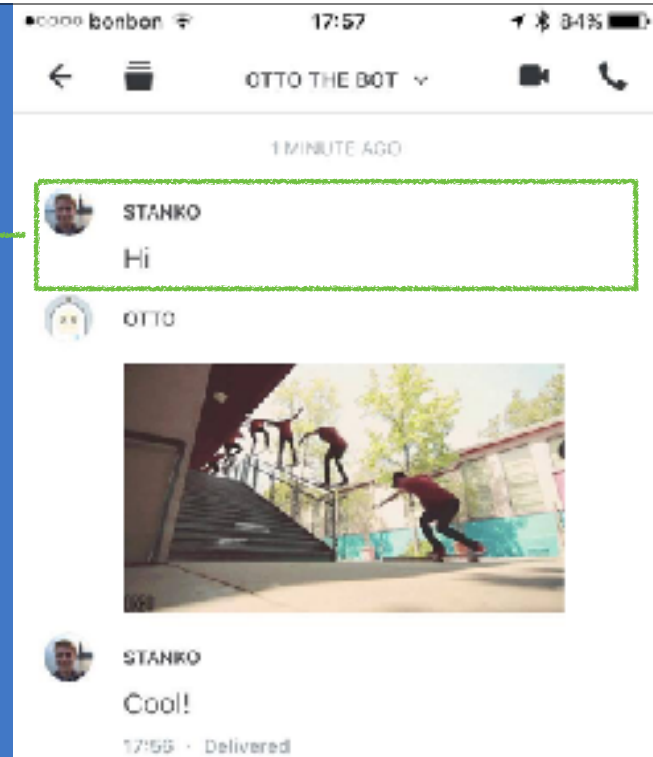
And messages will contain the sender and a lot of other meta-information…

REST is pointless if your endpoints
respond to a screen in your app

I won't go into as to why this happens, we can blame managment for ridiculous time constraints, we can blame frontend or mobile developers for being lazy and not implementing proper caching, we can blame backend developers for giving in too easily…

The point is, it doesn't matter why it happens - it happens all the time and it breaks REST.

If each endpoint returns a lot of data of which some is directly related to the resource and some isn't then you borke REST. This makes development harder because it's much easier to introduce bugs (e.g. inifinite resource loops user->articles->author), harder to understand and reason about the architecture without referencing the documentation and writing documentation sucks, but I'll get to that later. In general it makes it harder to separate concerns since one serializer is responsible for a lot of things.

There are some novel efforts to solve this problem:

* JSON::API introduces includes, meta-information, links, pointer, and what not. It's easier to say that it introduces a predictable structure for responses.
* HAL introduced linking and a general response structure

But they both only solved some symptoms, none actually solved the problem.
They both still require documentation to actually use. JSON::API is a bitch to implement properly (I haven't seen a full implementation in the wild, nor heard of it) and HAL requires a lot of requests to fetch a complex resource.

Also, they both return data that you actually don't need. To explain, if you have a users endpoint that returns all information about a user, and you only need a user's name, you will still have to get all other data the user has - which could be a lot.
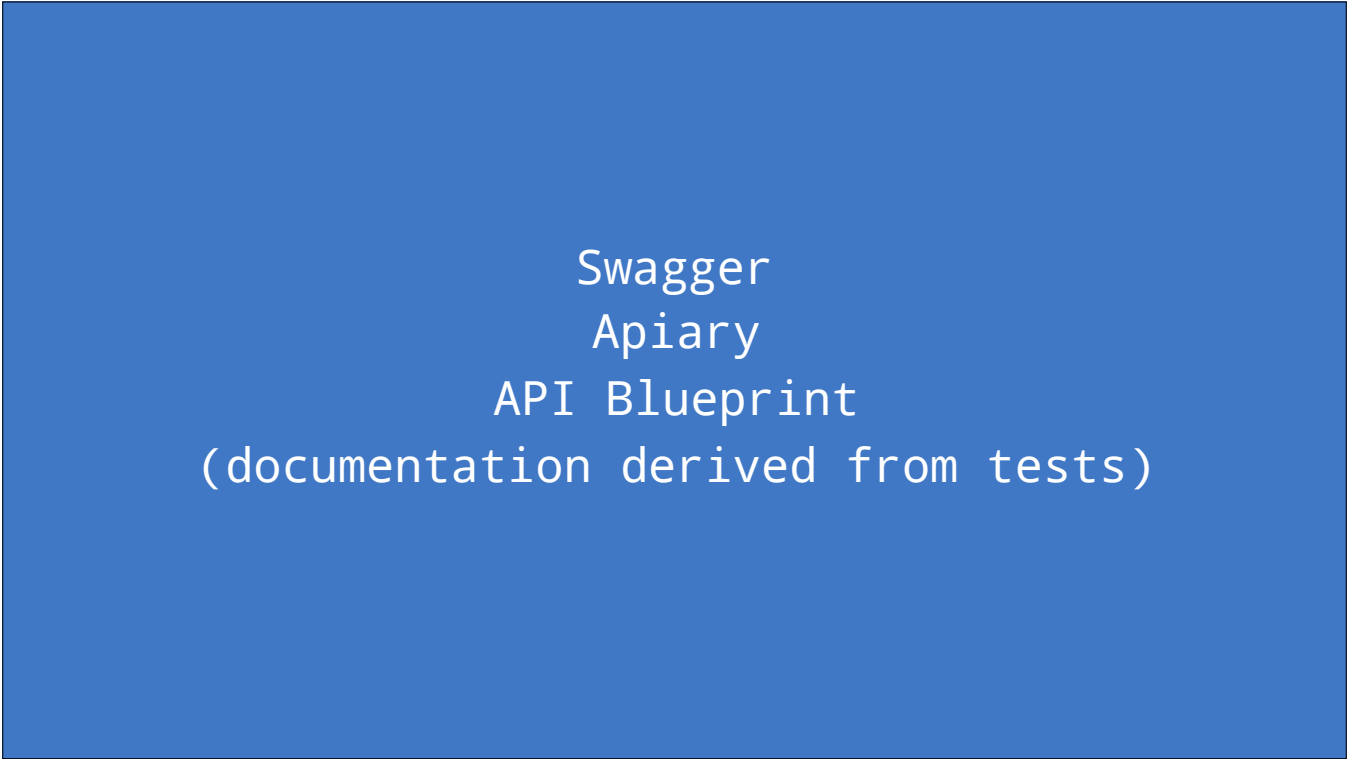
Another problem developers usually face is documentation.

I think everybody was in this situation at least once.

Documentation is boring.

Documentation will always be lacking,
or at least lag behind the
implementation

Swagger
Apiary
API Blueprint
(documentation derived from tests)

There are a lot of attempts to eleviate this issue.

Require manual labor
Laborious

But none have truely succeded.

03

GRAPHQL
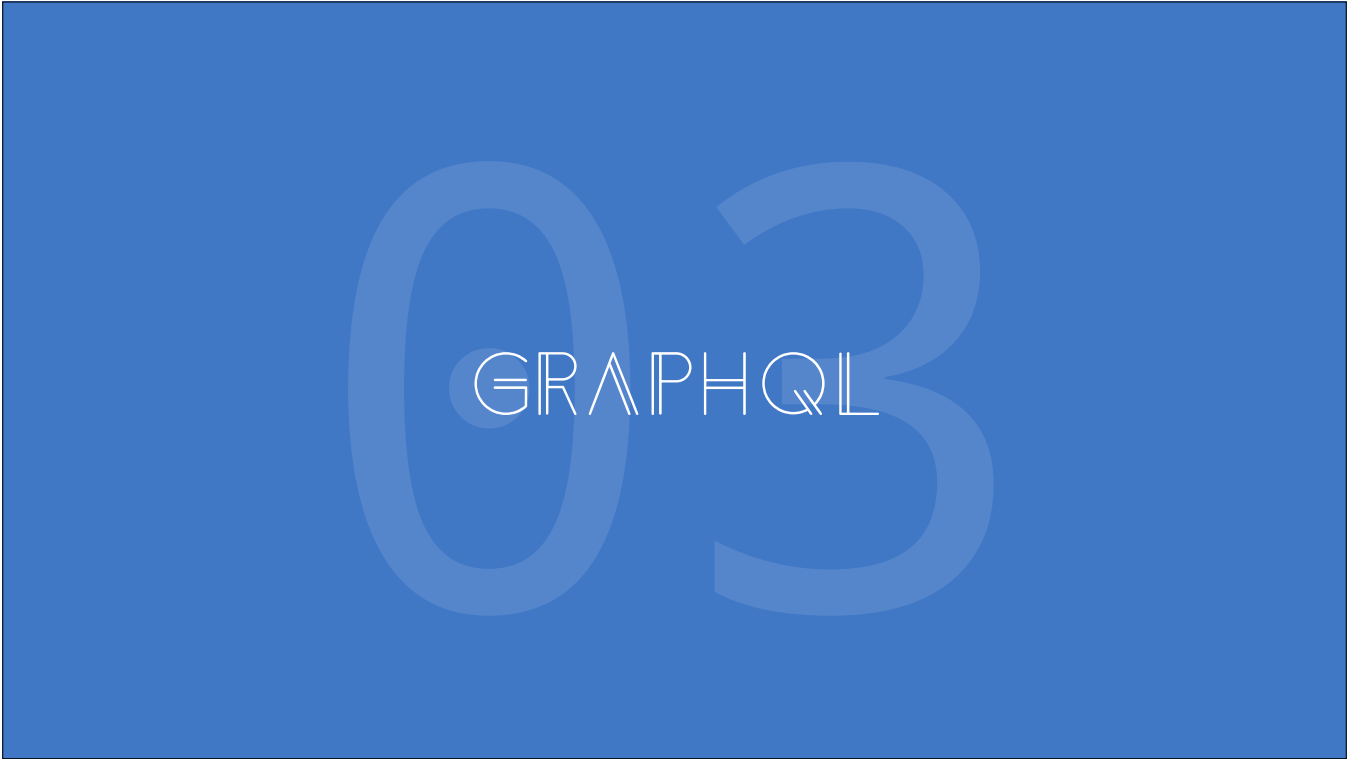
Think of your resources as
endpoints of a graph

```
query {
    posts {
        id
        title
        created_at
        author {
            full_name
        }
    }
}
```
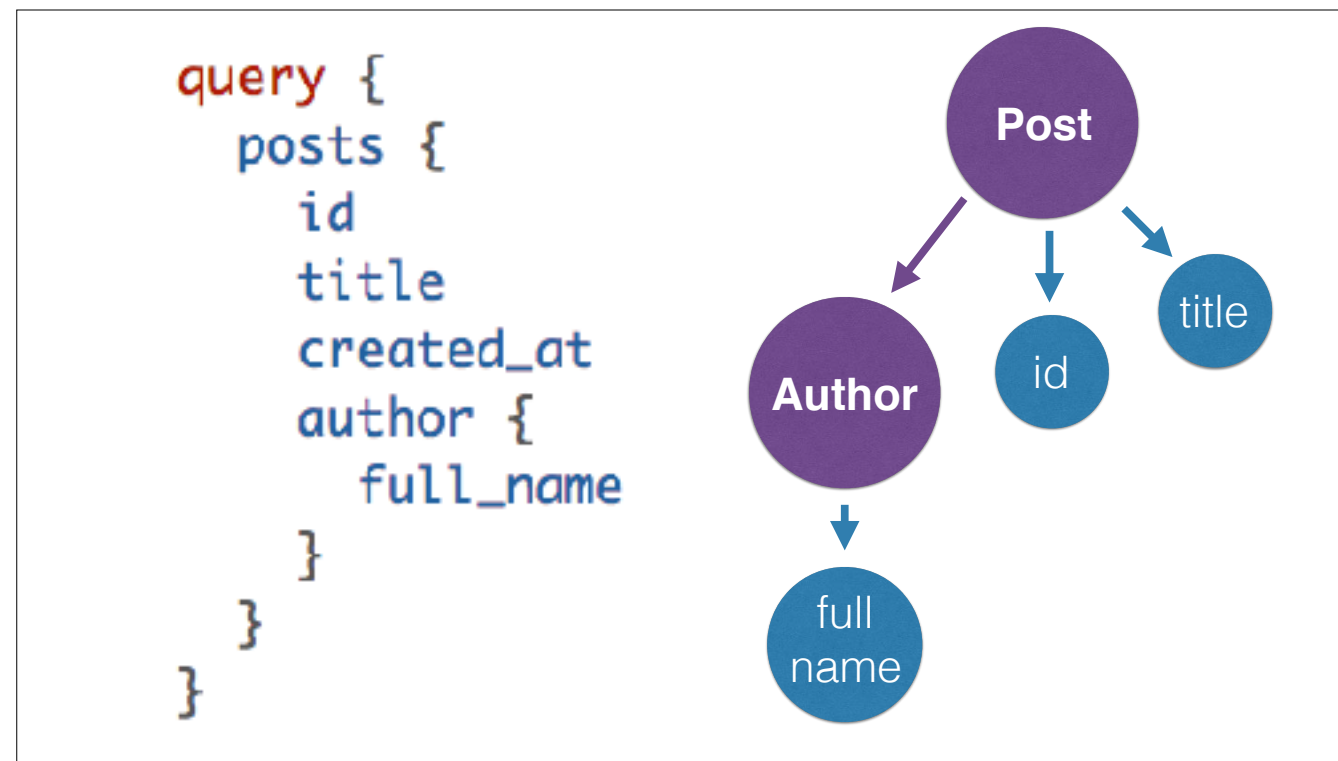
As said earlier, each resource or attirbute is an endpoint of the graph. Now we just need to specify which enpoints we need.

```
SELECT "posts".id, "posts".title, "posts".body, "posts".created_at
FROM "posts"
ORDER BY "posts".created_at DESC
```

The previous example is no new concept. It's been around for over 30 years - Query languages.
They are especially usefull for this purpose. Everybody had contact with SQL. You basically do exactly what I described on the previous slide in SQL, you pick different resources and specify how you want them returned to you.

```
query {                        {
    posts {                        "data": {
        id                            "posts": [
        title                            {
        created_at                           "id": "1",
        author {                             "title": "Lilies of the Field",
            full_name                        "created_at": "2017-02-25T18:45:29Z",
        }                                    "author": {
    }                                            "full_name": "Stanko Krtalic Rusendic"
}                                            }
                                         },
```
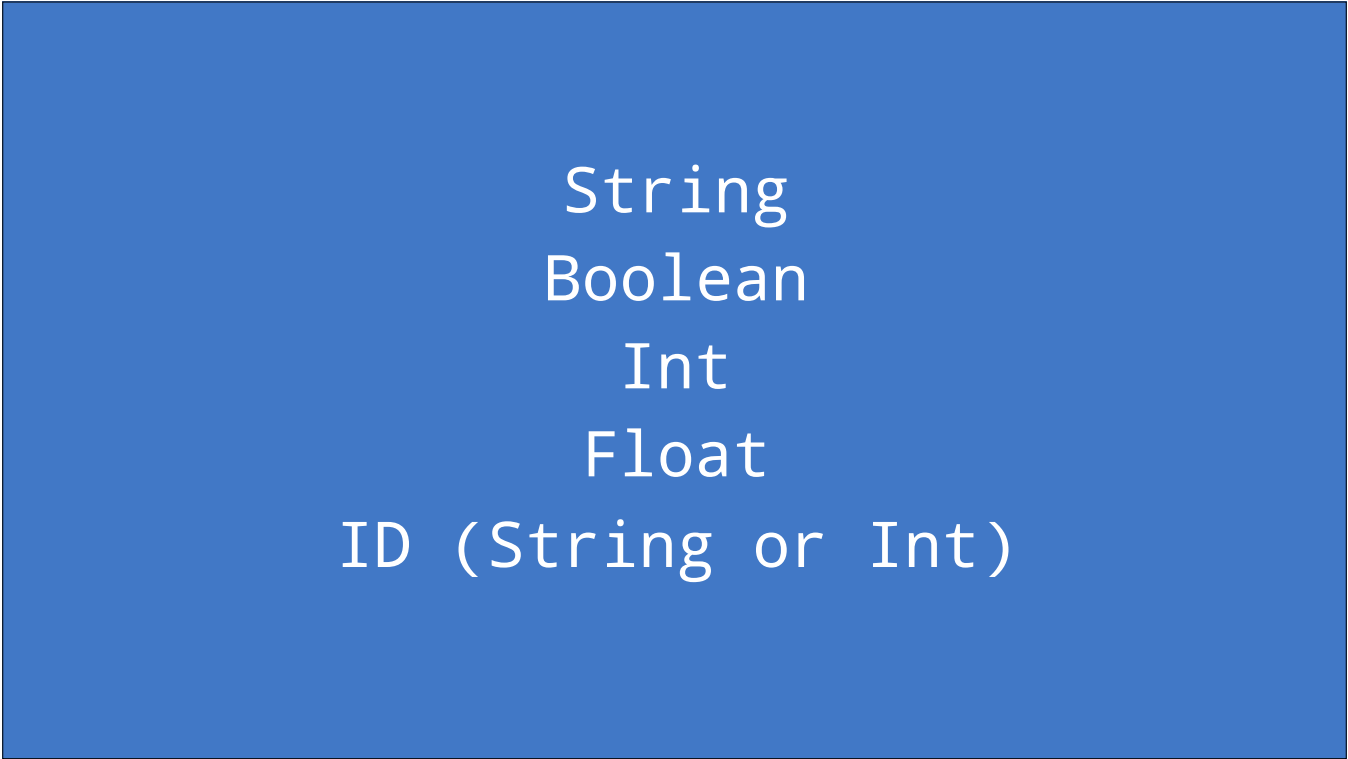
Here is a simple example of a GraphQL query.

I want to demonstrate here that the response will be structured exactly like the query. You can map each attribute from the query to an attribute of the response.
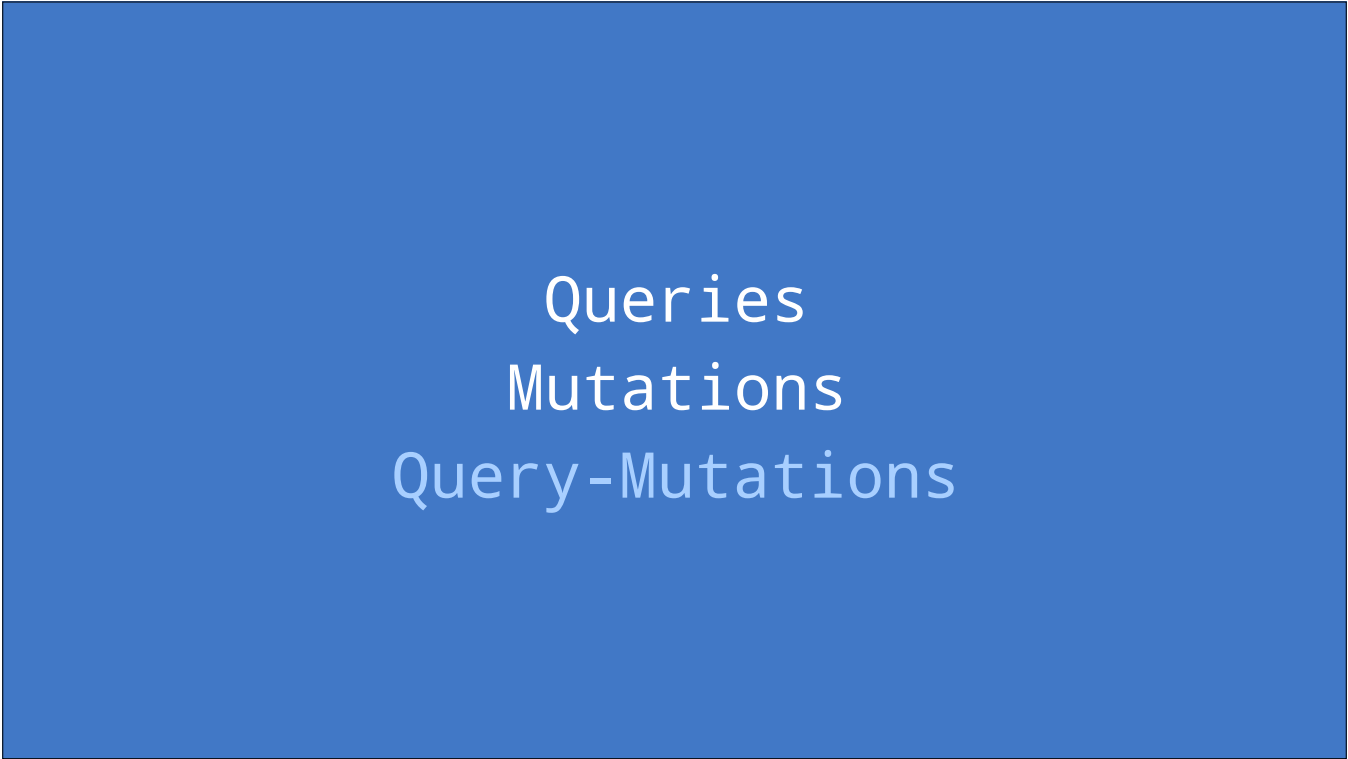
String
Boolean
Int
Float
ID (String or Int)

And before we dig deeper you need to know that GraphQL only supports these basic types.
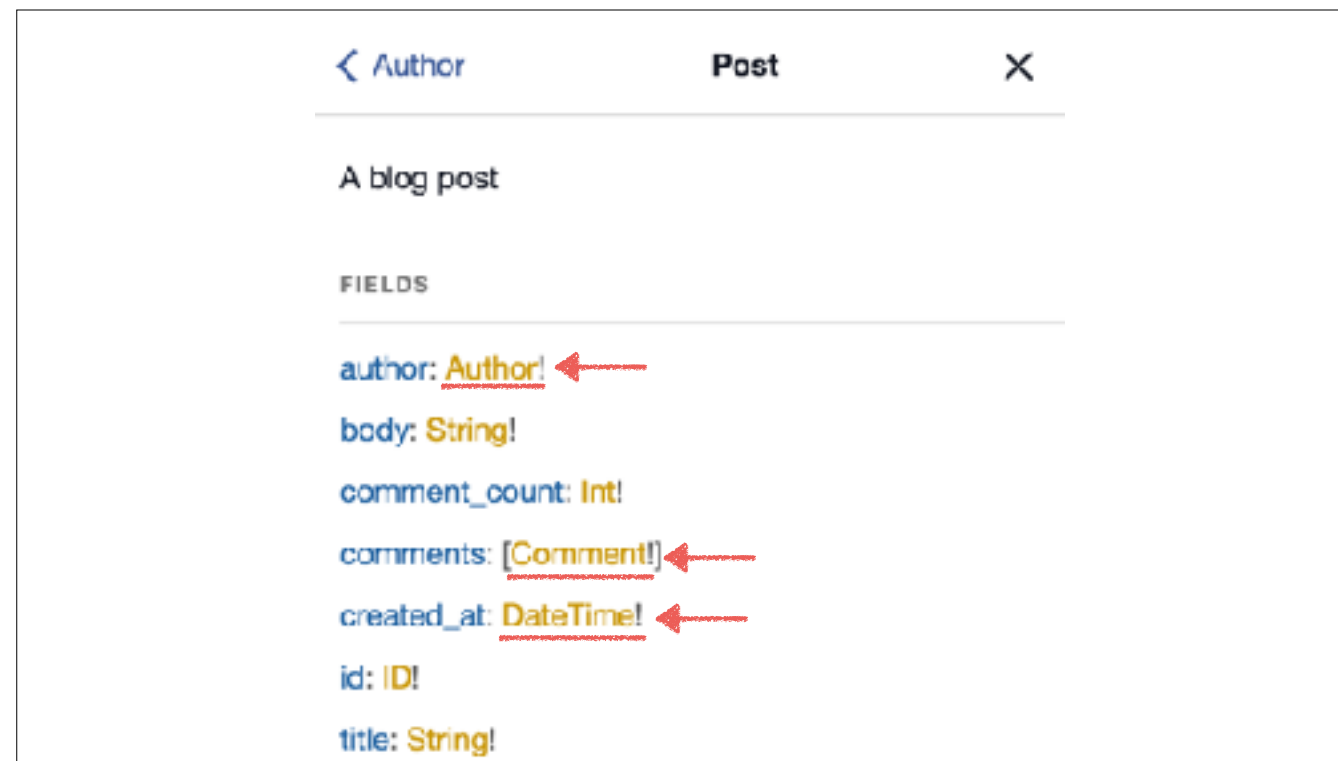
But you can define custom types.

Ok, now some theory and design decisions.
In general there are two basic kinds of methods in programming.
Queries and Mutations.
And, of course, the grey-area - Query-Mutations.

```
 1 # Query method - given an id it returns the coresponding user
 2 def get_user(id)
 3 end
 4
 5 # Mutation method - given the inputs it creates a new user, but returns nothing
 6 def create_user(first_name, last_name, email)
 7 end
 8
 9 # Query-Mutation - given the inputs it returns the modified user object
10 def update_user(first_name, last_name, email)
11 end
12
```

Ok, now some theory and design decisions.

In general there are two basic kinds of methods in programming.

Queries and Mutations.

And, of course, the grey-area - Query-Mutations.

Queries

~~Mutations~~

Query-Mutations

GraphQL doesn't allow for pure mutations, only for query-mutations. This will be important in the next few slides.

```
query {
  post(id: 1){
    id
    title
    created_at
    author {
      full_name
    }
    comments {
      author {
        first_name
      }
      body
    }
  }
}
```

Here is another example of a query

```
{
    "data": {
        "post": {
            "id": "1",
            "title": "Lilies of the Field",
            "created_at": "2017-02-25T18:45:29Z",
            "author": {
                "full_name": "Starko Krtalic Rusendic"
            },
            "comments": [
                {
                    "author": {
                        "first_name": "Dario"
                    },
                    "body": "It was summer... and it was hot. Rachel was there... A
lonely grey couch..."OH LOOK!" cried Ned, and then the kingdom was his
forever. The End."
                },
                {
                    "author": {
                        "first_name": "Tomislav"
                    },
                    "body": "Raspberries? Good. Ladyfingers? Good. Beef? GOOD!"
                },
```

Please note that the response is structured identically to the request.

```
mutation {
    createComment(input: {
        postId: 1
        authorId: 1
        body: "This was created using GraphQl 🙌"
    })
    {
        post {
            comments {
                body
            }
        }
    }
}
```

And here is an example of a mutation.

Note that you have to select something from the returned information of the mutation - hence only query-mutations are allowed.

In this example I specify that I want all the contents of all comments on the post I commented.

```json
{
  "data": {
    "createComment": {
      "post": {
        "comments": [
          {
            "body": "This was created with GraphQL 🙌"
          }
        ]
      }
    }
  }
}
```

And that is exactly what I get back.

```graphql
query {
  conversation(id: 53) {
    name
    messages {
      author {
        fullName
        avatar {
          url
        }
      }
      body
      createdAt
      status
    }
  }
}
```
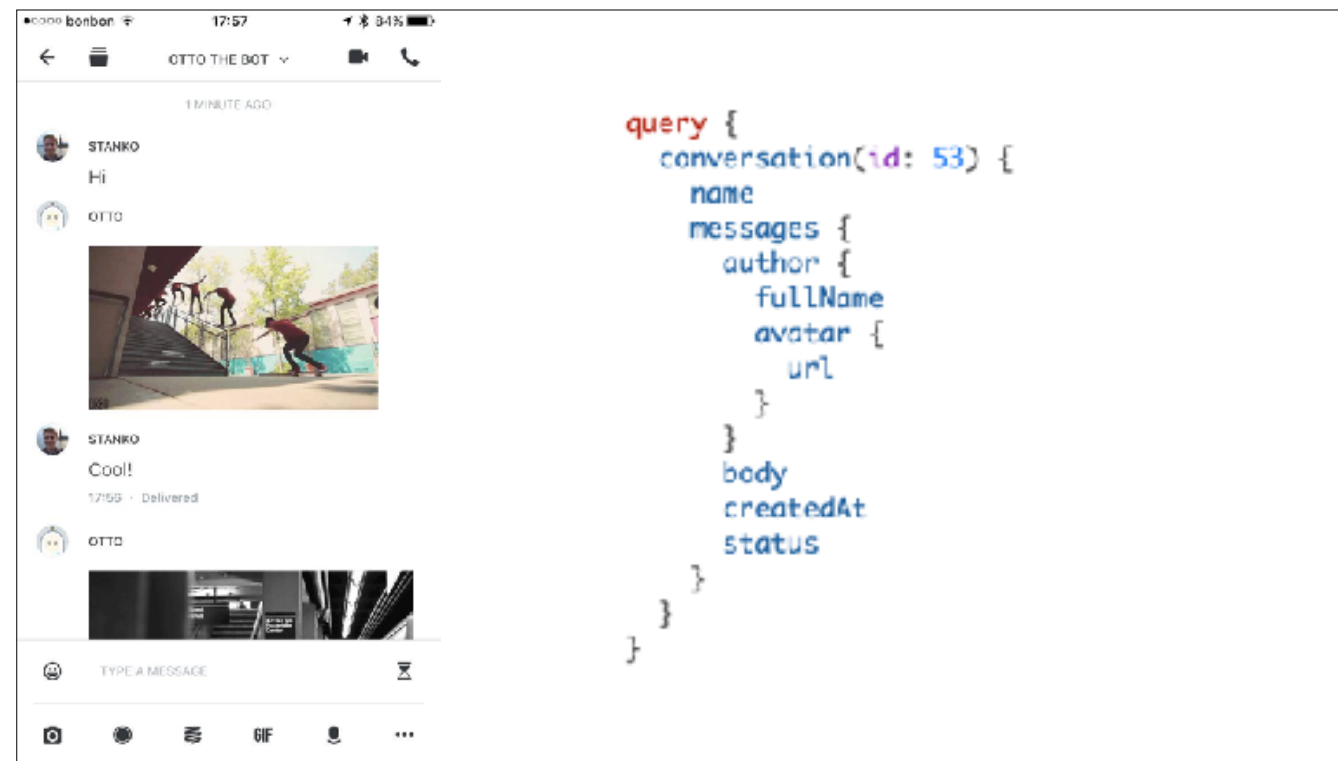
This whole screen becomes one query

Everything is served from
one endpoint

http://localhost/graphql

```
query {
  post_1: post(id: "1") {
    ...postFields
  },
  post_2: post(id: "2") {
    ...postFields
  }
  post_3: post(id: "3") {
    ...postFields
  }
}

fragment postFields on Post {
  title
  author {
    full_name
  }
  body
  comment_count
}
```

But GraphQL offers many more features.
E.g. these are aliases and fragments.

Let's say you want the first three posts with the exact same data returned. You would specify a fragment with the data you wish to have returned and simply include it in the query.
Also notice that I named the queries for each post and used the identical query method with different arguments.

```
{
    "data": {
        "post_1": { ⊕ },
        "post_2": { ⊖ },
        "post_3": {
            "title": "I Sing the Body Electric",
            "author": {
                "full_name": "Stanko Krtalic Rusendic"
            },
            "body": "If we override the driver, we can get to the GB
application through the multi-byte XML alarm!\nWe need to index the 1080p
RAM interface!\nTry to transmit the AI bus, maybe it will connect the
digital interface!\nYou can't connect the interface without connecting
the solid state SQL driver!\nUse the optical EXE bus, then you can
compress the haptic alarm!\nThe PCI interface is down, input the open-
source capacitor so we can back up the SMTP feed!\nOverriding the pixel
won't do anything, we need to synthesize the virtual usb card!\nI'll
connect the 1080p RSS driver, that should bandwidth the GB array!",
            "comment_count": 5
        }
    }
}
```

This is the result of the query. The blu arrows are just collapsed data. All responses are by structure identical to the one displayed.

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

VARIABLES

```json
{
  "episode": "JEDI"
}
```

```json
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

You can also have variables, for values that you use often.

```
query Hero($episode: Episode, $withFriends: Boolean!)  {
  hero(episode: $episode) {                              "data": {
    name                                                   "hero": {
    friends @include(if: $withFriends) {                     "name": 'R2-D2"
      name                                                 }
    }                                                    }
  }                                                    }
}

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

Or if you want to conditionally return some field you can use directives.
In this example, only if `withFriends` is set to `true` will the `friends` field be returned.

```
{                                       {
  search(text: "an") {                    "data": {
    __typename                              "search": [
    ... on Human {                            {
      name                                      "__typename": "Human",
    }                                           "name": "Han Solo"
    ... on Droid {                            },
      name                                    {
    }                                           "__typename": "Human",
    ... on Starship {                           "name": "Leia Organa"
      name                                    },
    }                                         {
  }                                             "__typename": "Starship",
}                                               "name": "TIE Advanced x1"
                                              }
                                            ]
                                        }
```

There is also support for polymorphic selections. If some types have different attributes that you want selected but are returned in the same query. Here, `__typename` is a meta-field that returns the type of object that is returned.
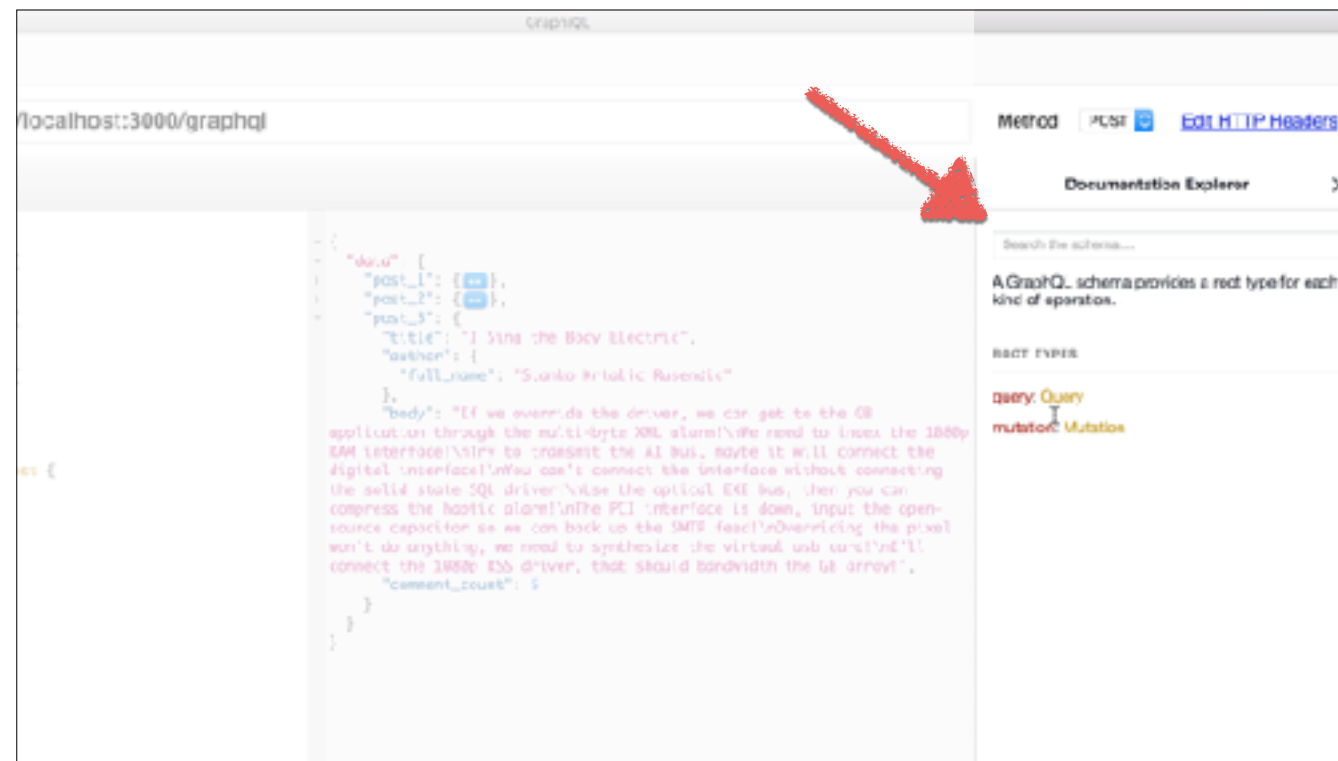
And perhaps the best feature! Autogenerated documentation!

Though, to be honest this is much better than just documentation. This is a list of all available query and mutation methods that can be fetched from the server at any time!

This allows for runtime checks!

Schema definition language

GraphQL also comes with a rich schema definition language that enables you to define you API's interface beforehand. This the backend and frontend teams don't have to wait for each other!

```
type Character {
  name: String!
  appearsIn: [Episode]!
}

type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Define custom types

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
union SearchResult = Human | Droid | Starship
```

Define enums or unions

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

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

You can even define interfaces!

GraphQL is an OpenSource project run by Facebook. It's the cornerstone for most of their infrastructure so it'll be here for a while.
There if a pletera of parsers and query builders available in different languages, and even whole frameworks!

GraphQL-Ruby, Graphqlpy and Absinthe are implementations of the format in Ruby, Python and Elixir. And Relay is a framework for React that consumes GraphQL exclusively.

This presentation is
available at

github.com/stankec/lectures

Any questions?
Feel free to ask question, I use GraphQL on three production projects and would never go back :)

Questions?

Any questions?

Feel free to ask question, I use GraphQL on three production projects and would never go back :)