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.

Floating Point

This brings me to my employer, FloatingPoint. If you are interested in working with some of the technologies I’ve mentioned, Phoenix or Clojure give us a shoutut or nag Nikica and Mihael during the drinkup after the lecture.

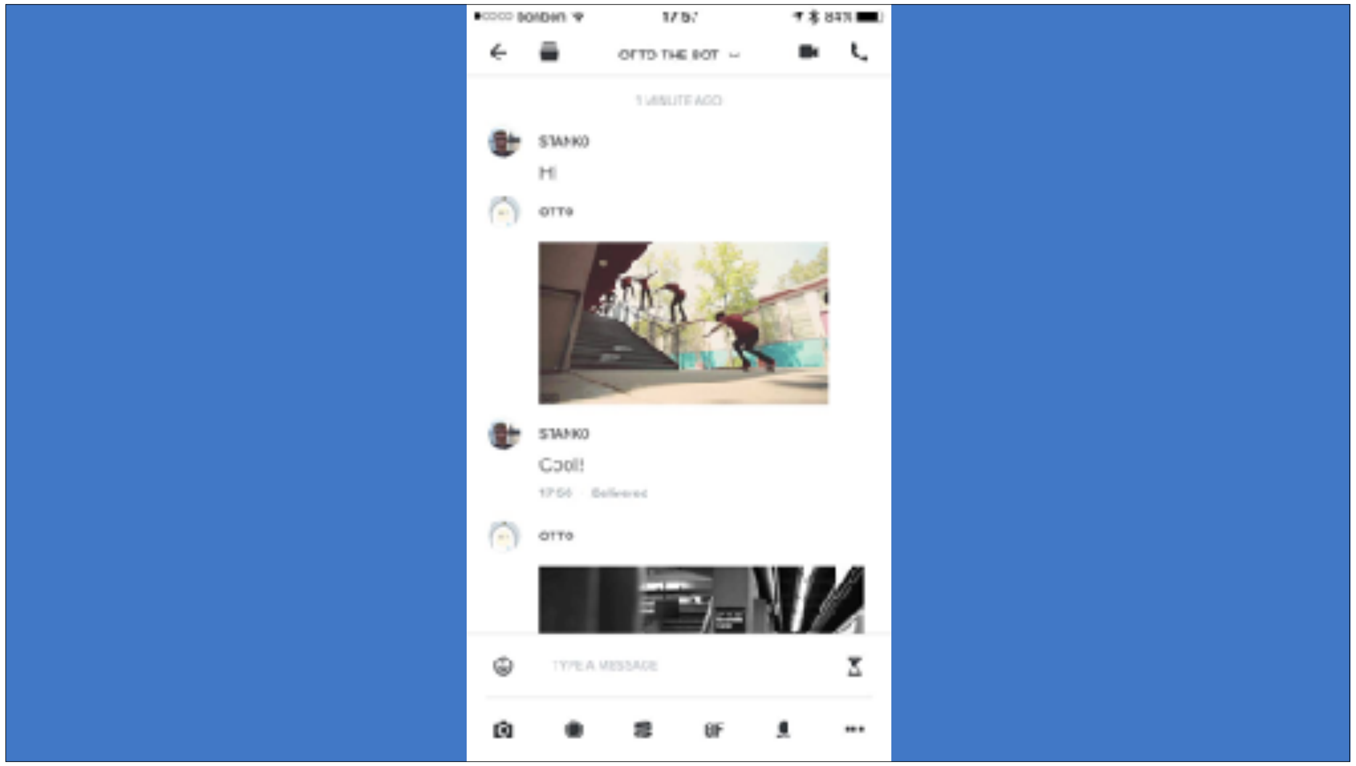
01

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.



STANKO
Hi

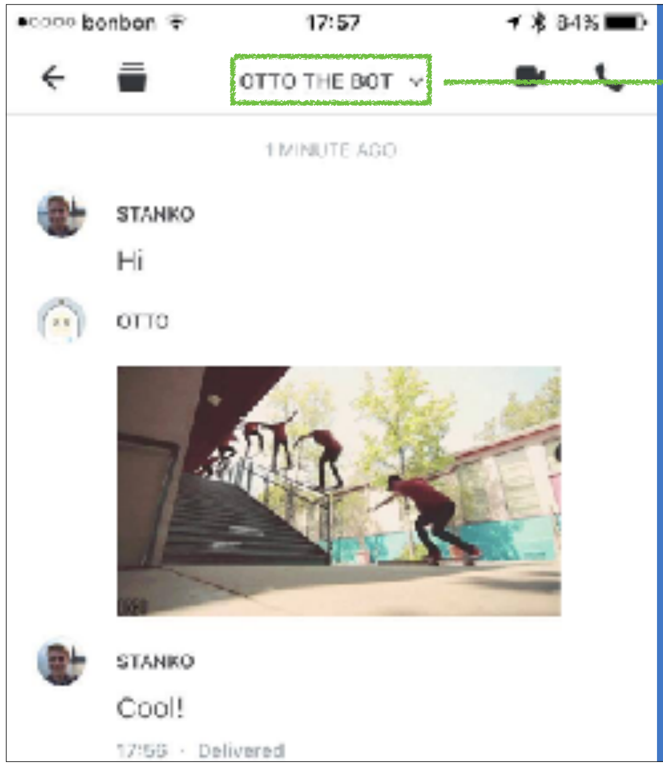
OTTO

STANKO
Cool!
17:56 · Delivered

OTTO

Let’s go through how you would model a RESTful API for this simple chat application

IDEA



1 MINUTE AGO

STANKO

Hi

OTTO

STANKO

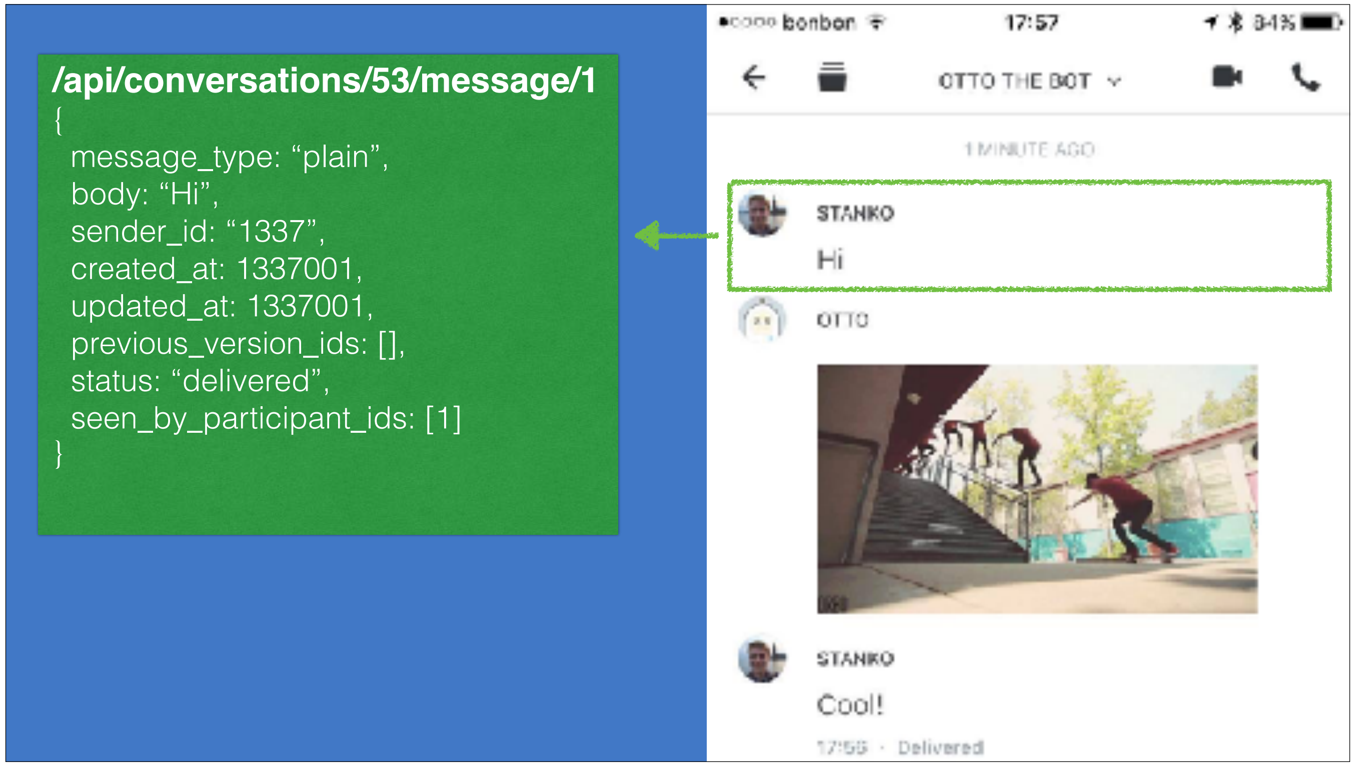
Cool!

17:56 · Delivered

/api/conversations/53

```
{
  name: “Otto the bot”,
  participant_user_ids: [1, 1337],
  last_message_sent_at: 1337002,
  archived: false
}
```

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]
}
```

STANKO
Hi

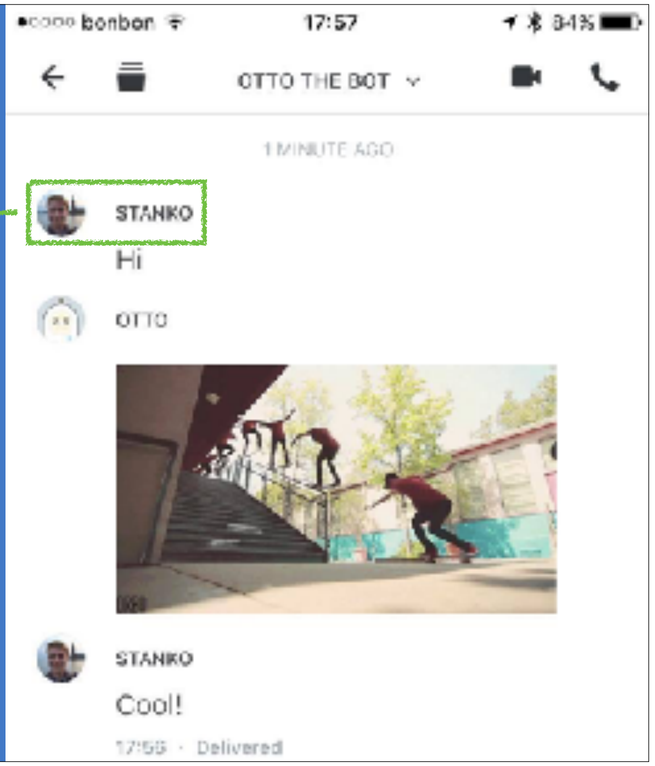
OTTO

STANKO
Cool!
17:56 · Delivered

Then you would have an endpoint for each individual message…

/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”
}
```



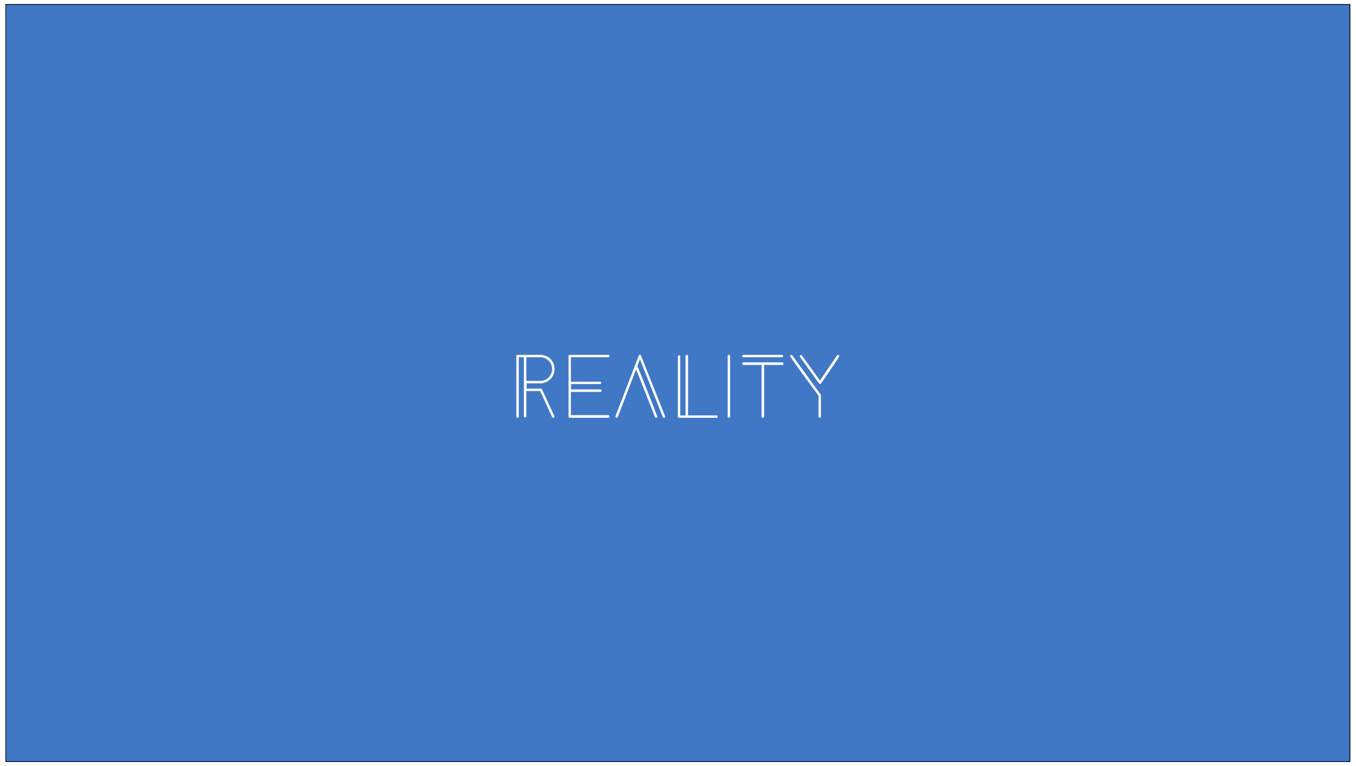
1 MINUTE AGO

STANKO
Hi

OTTO

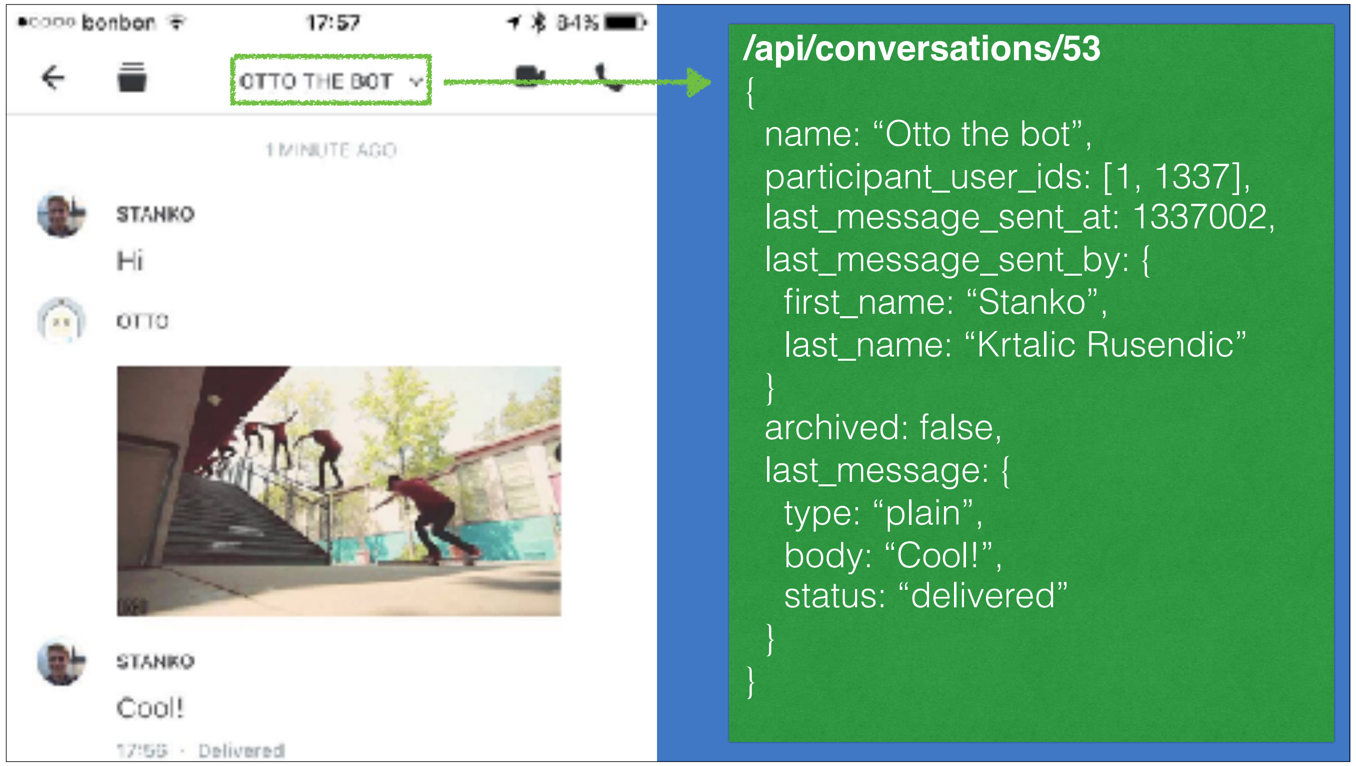
STANKO
Cool!
17:56 · Delivered

And then you would also create an endpoint to get users.



REALITY

In theory, theory and practice are the same. In practice, not so much…



STANKO
Hi

OTTO

STANKO
Cool!
17:56 · Delivered

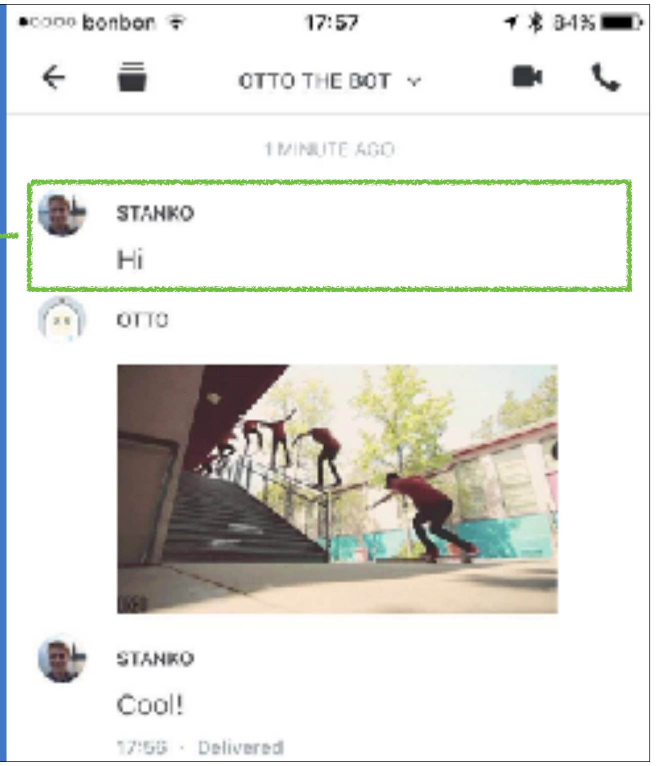
/api/conversations/53

```
{
  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”
  }
}
```

Ususally 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://…”
  }
}
```

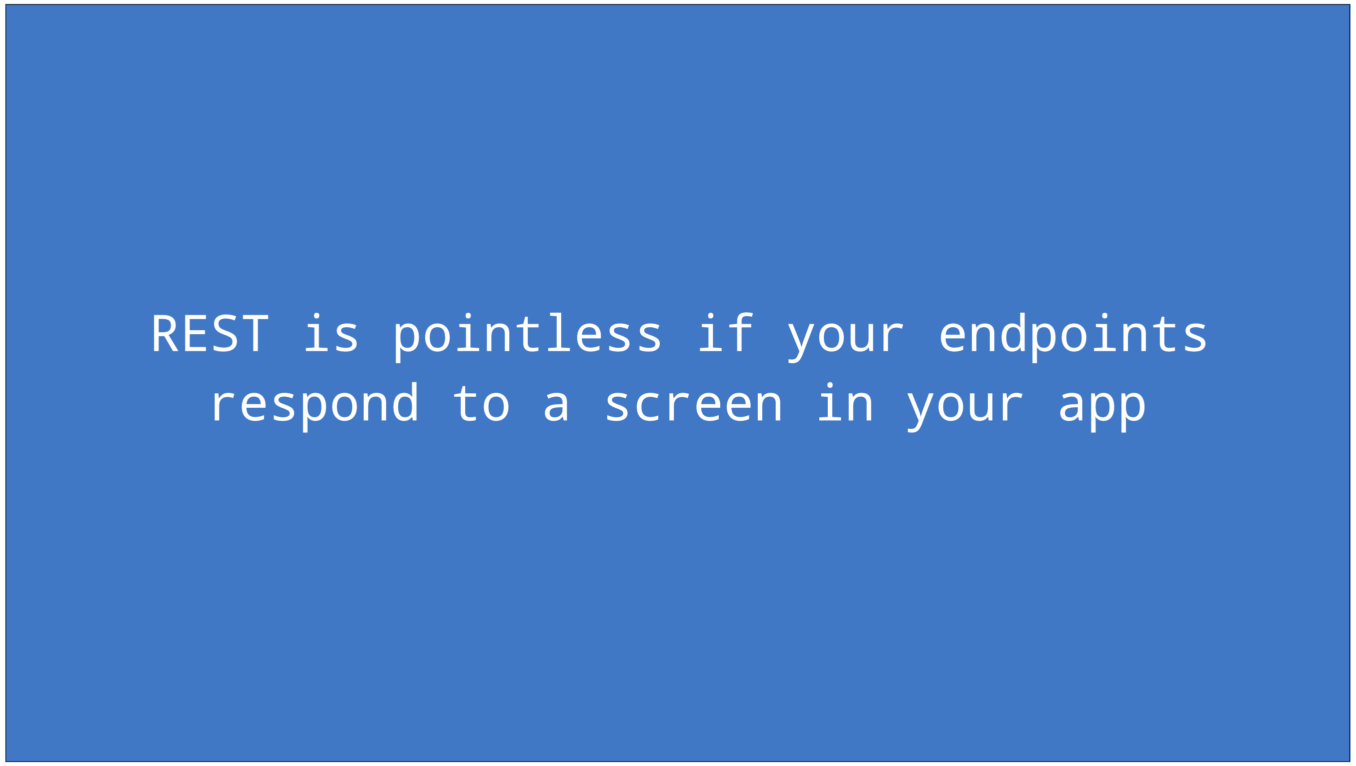


STANKO
Hi

OTTO

STANKO
Cool!
17:56 · Delivered

And messages will containe the sender and a lot of other metainformation…



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 forntend developers for being lazy, we can blame backend for giving in too easily, it doesn’t matter. It happens and when it happens 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). And in general it makes it harder to separate concerns since one serializer is responible for a lot of things.

JSON::API

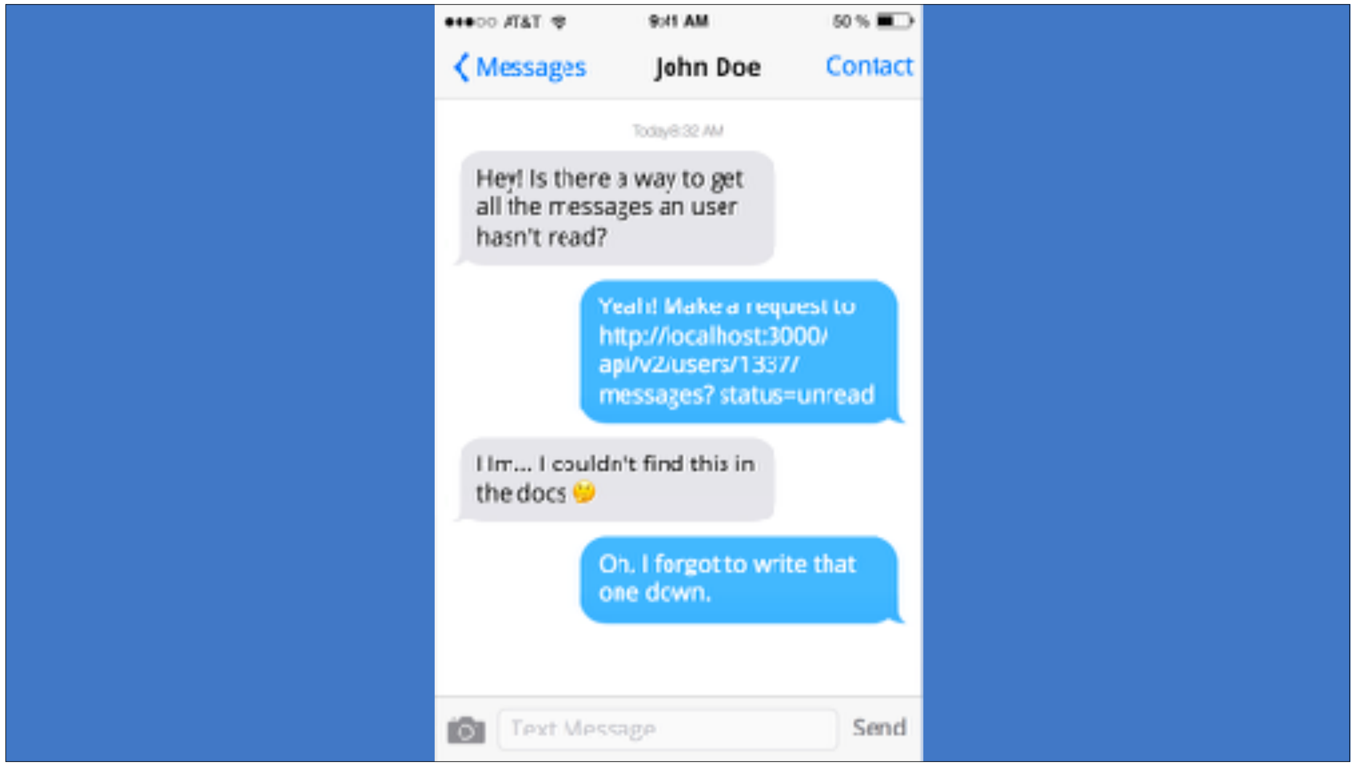
HAL

There are some novel efforts to solve this problem.
JSON::API introduces `includes`.
HAL introduced linking.
But in the end both are flawed, one is hard to implement and the other requires a lot of requests for a simple query.

02

DOCUMENTATION

Another problem developers usually face is documentation.



Today 8:32 AM

Hey! Is there a way to get all the messages an user hasn't read?

Yeah! Make a request to http://localhost:3000/api/v2/users/1337/messages? status=unread

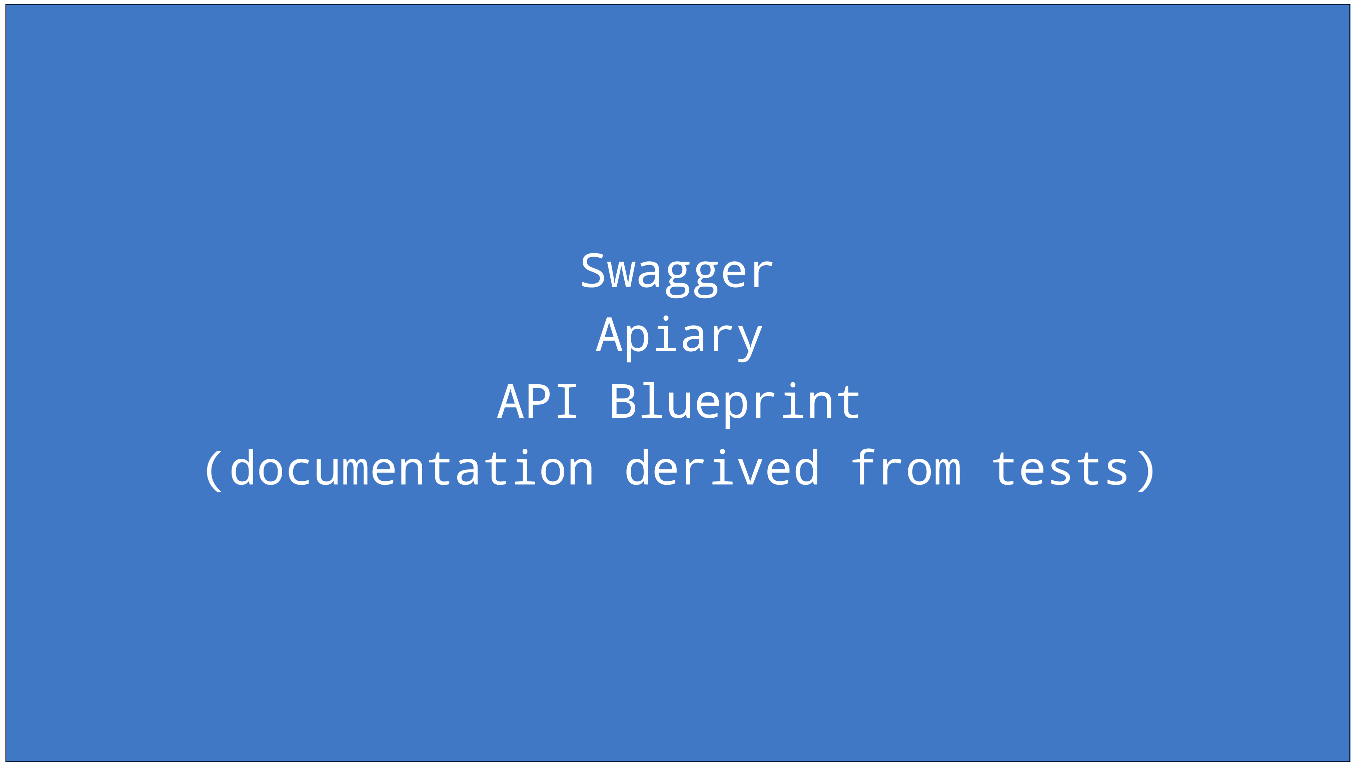
Hm... I couldn't find this in the docs 🤔

Oh, I forgot to write that one down.

I think everybody was in this situation at least once.

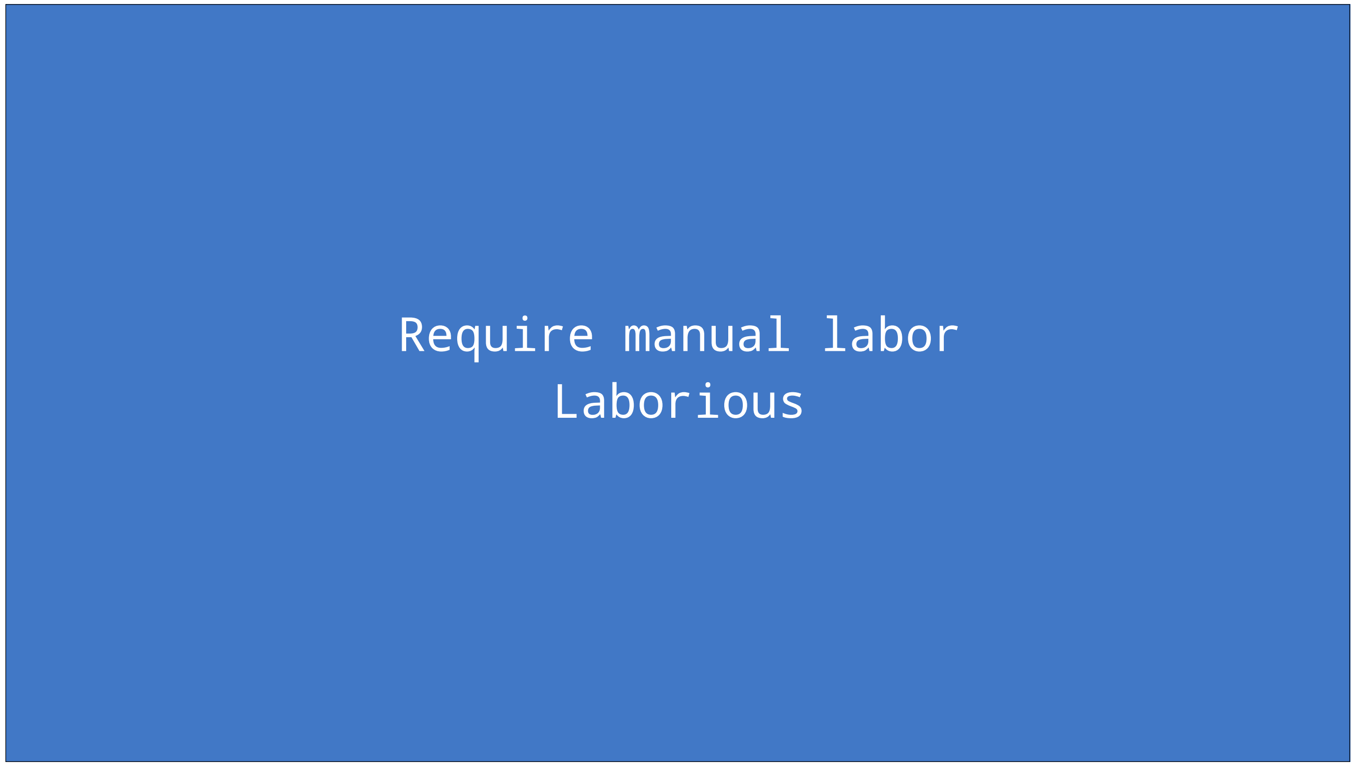
Documentation is boring.

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

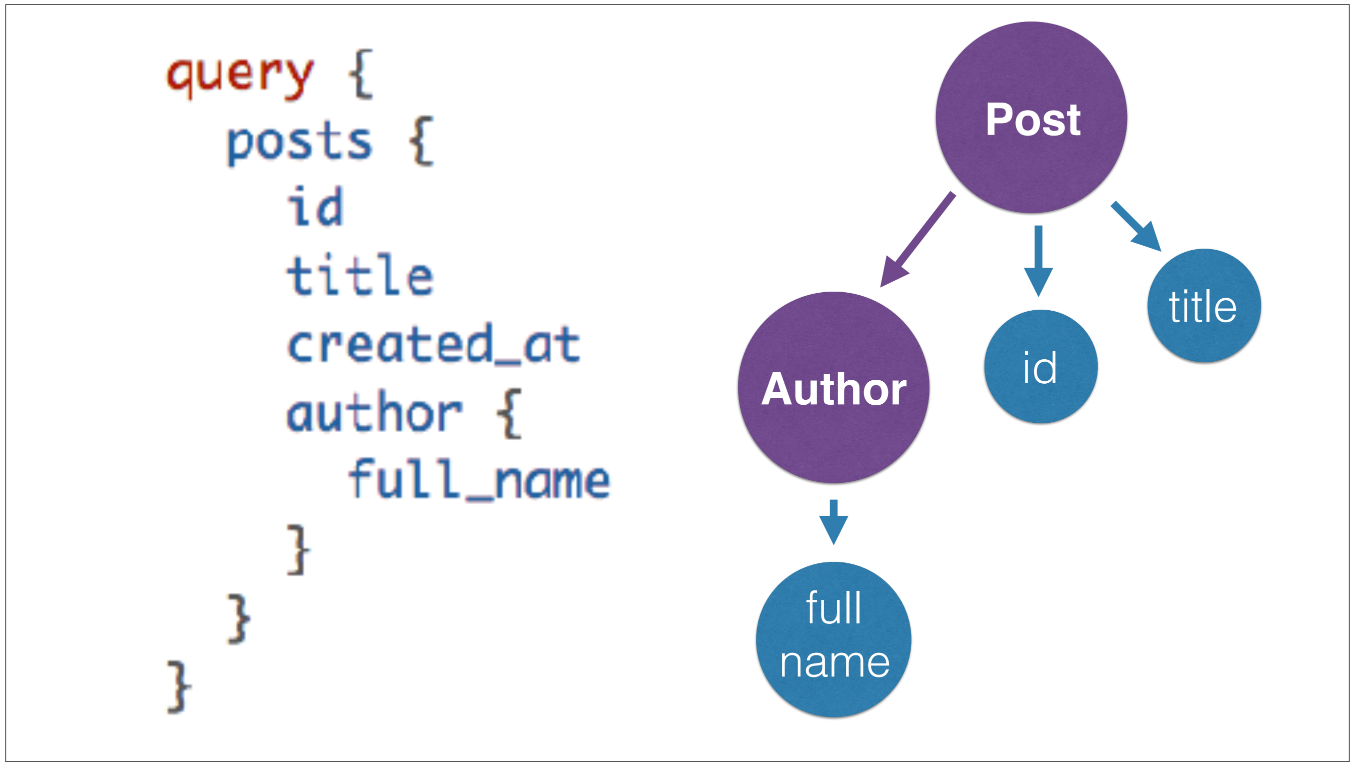
Laborious

But none have truely succeded.

03

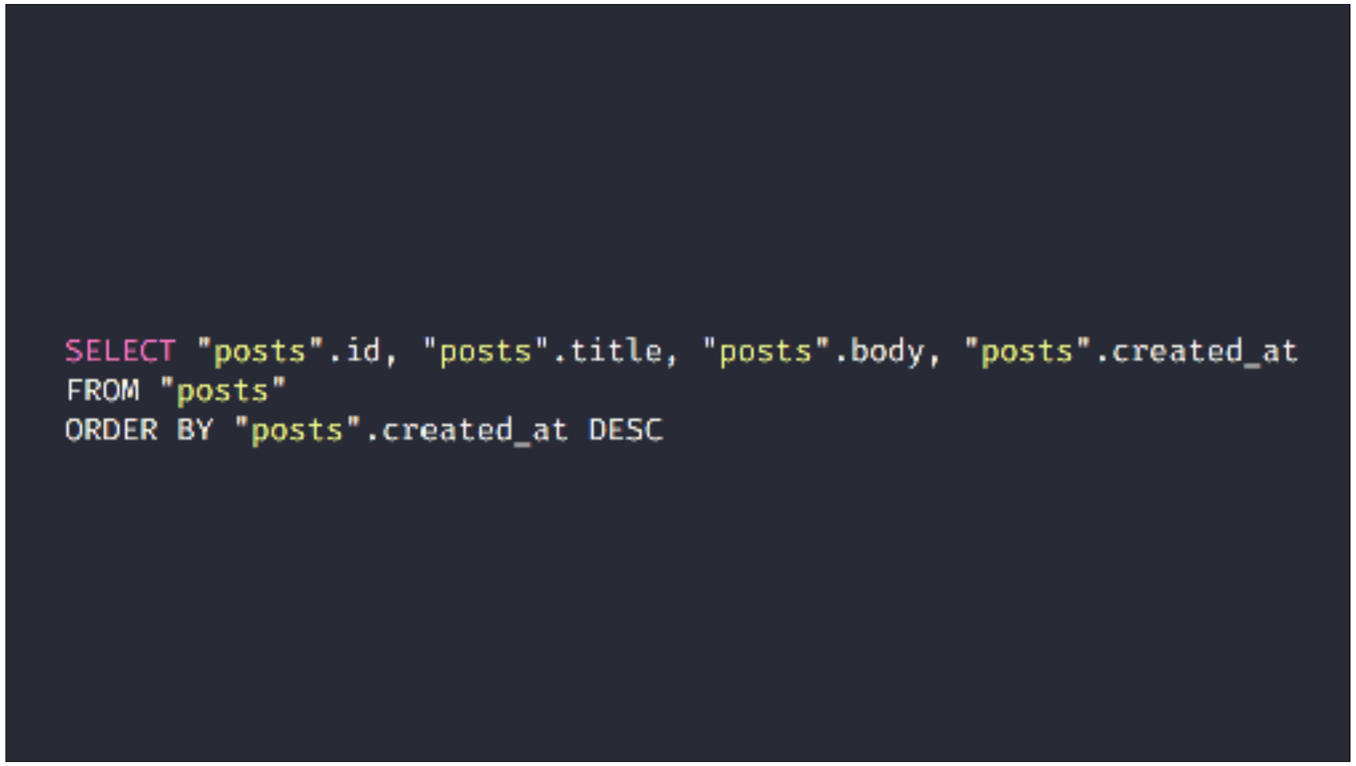
GRAPHQL

Think of your resources as endpoints of a graph



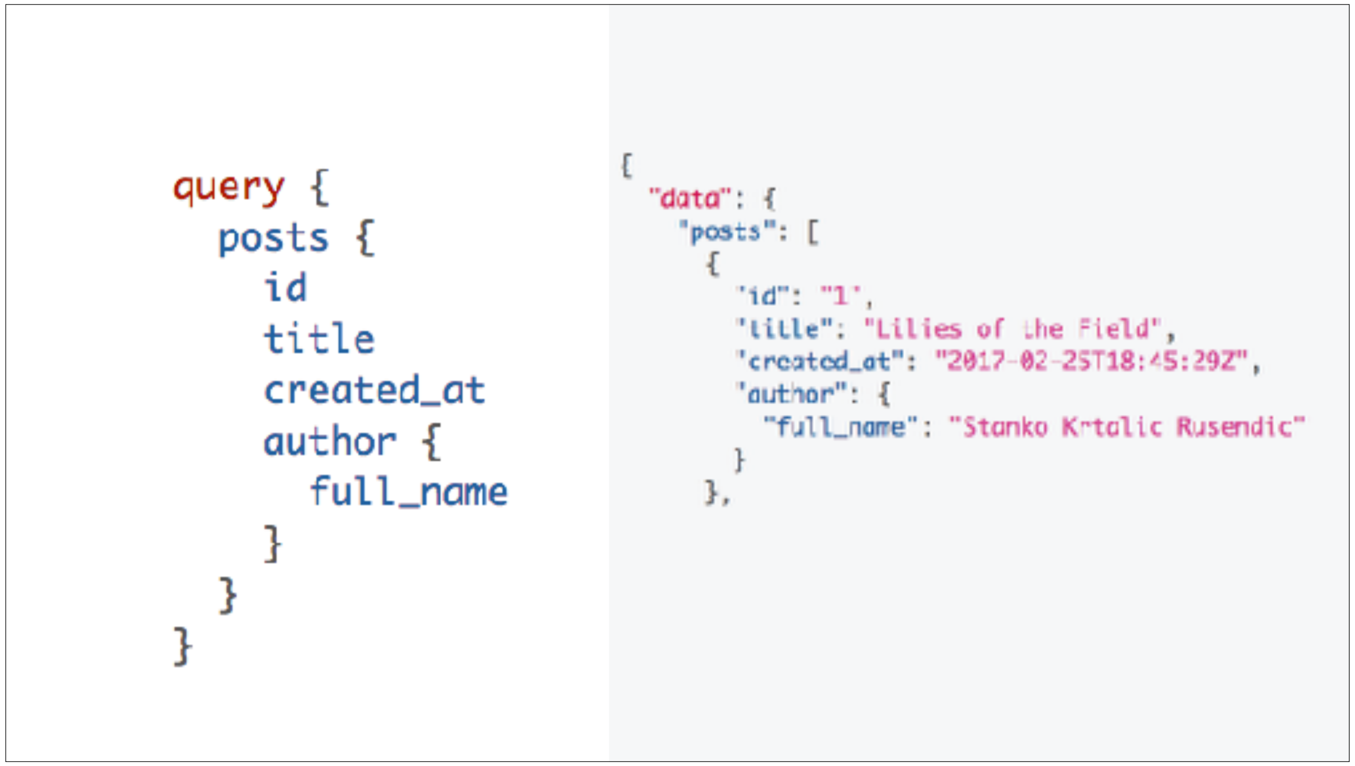
```
query {
  posts {
    id
    title
    created_at
    author {
      full_name
    }
  }
}
```

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
```

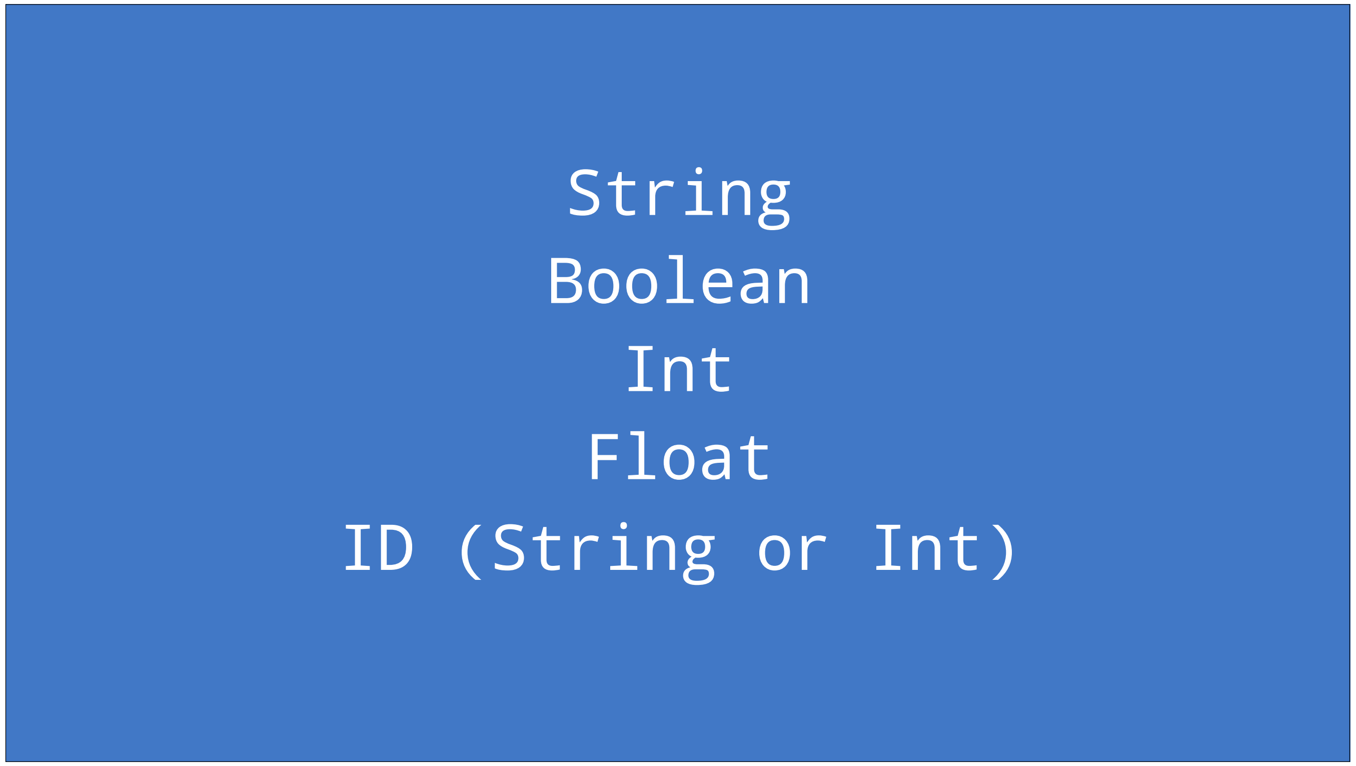
Query languages are especially usefull for this purpose. Everybody has had contact with SQL. You basically do exactly what I descriped a slide ago in SQL, you pick different resources and specify how you want them returned to you.



```
query {
  posts {
    id
    title
    created_at
    author {
      full_name
    }
  }
}
```

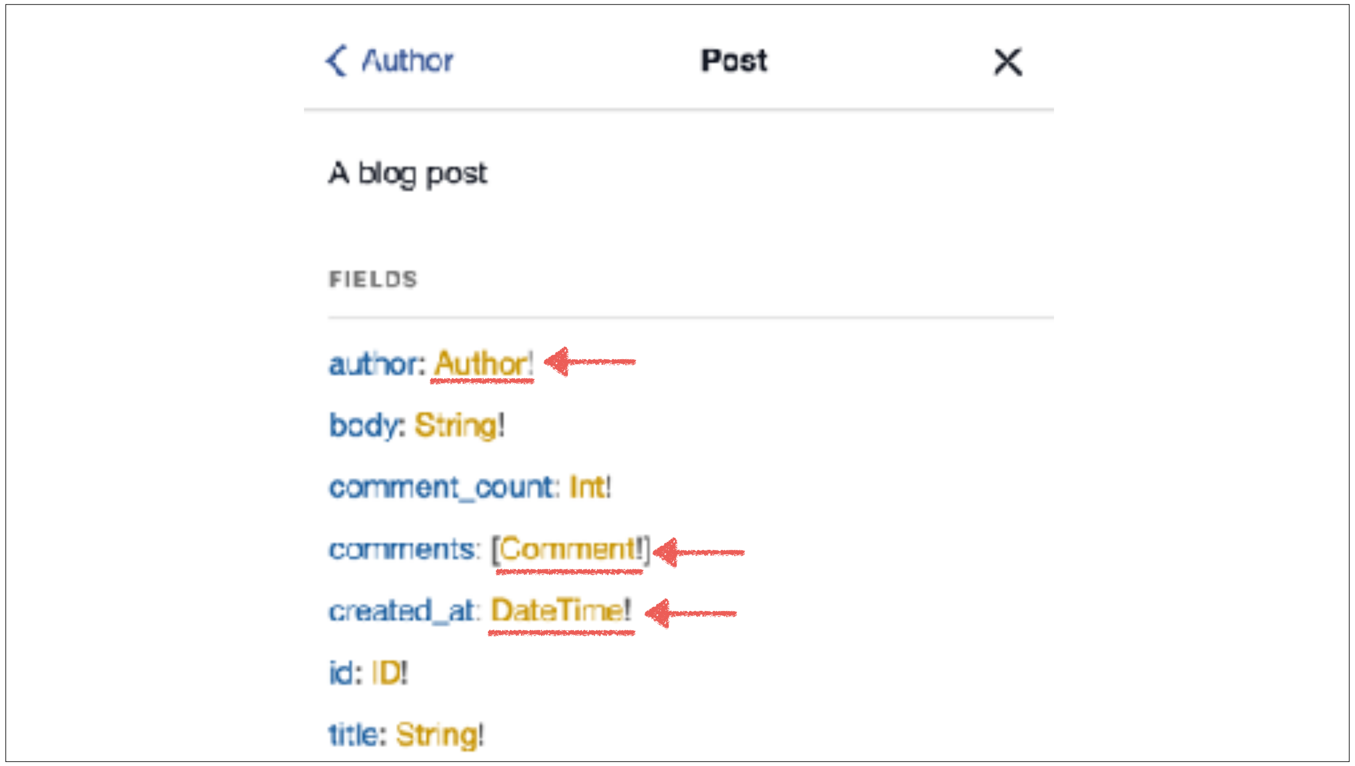
```
{
  "data": {
    "posts": [
      {
        "id": "1",
        "title": "Lilies of the Field",
        "created_at": "2017-02-25T18:45:29Z",
        "author": {
          "full_name": "Stanko Krtalic Rusendic"
        }
      },
```

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.



A blog post

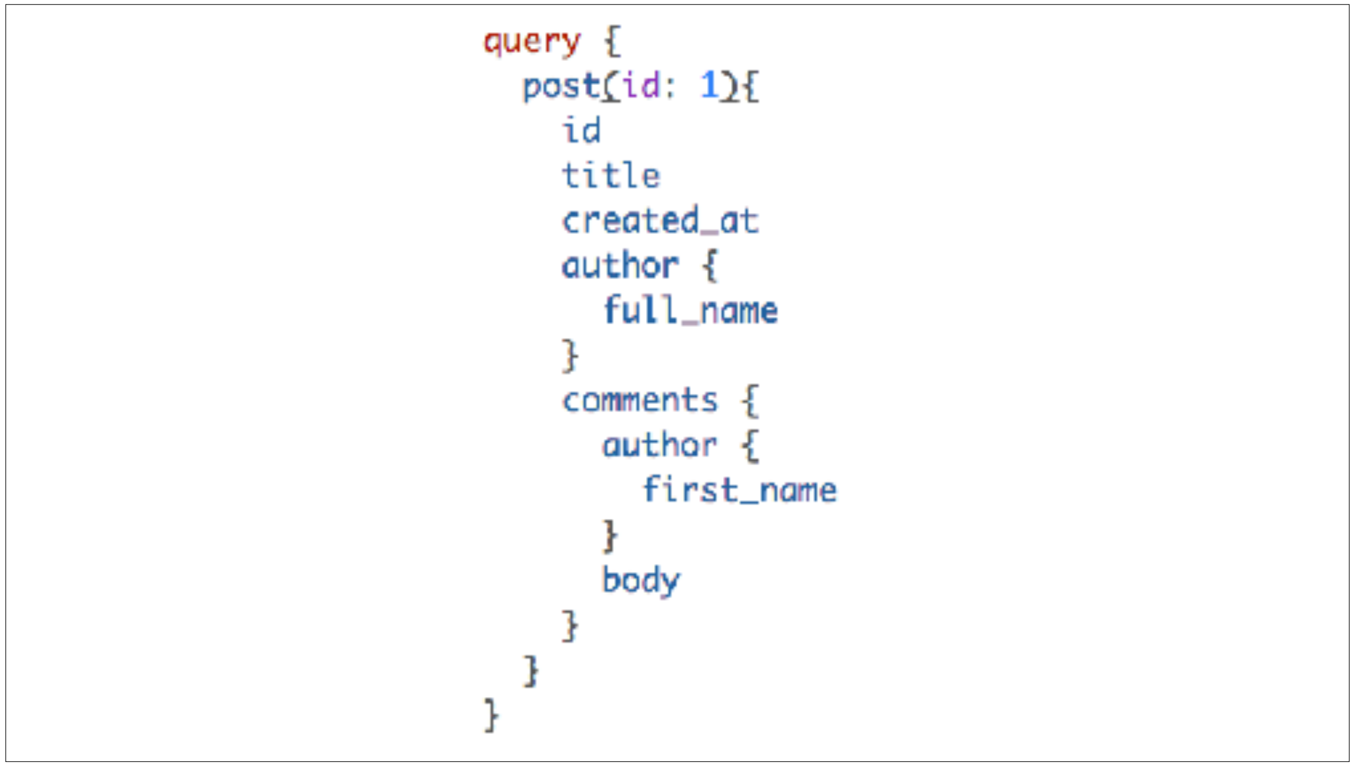
FIELDS

```
author: Author!
body: String!
comment_count: Int!
comments: [Comment!]
created_at: DateTime!
id: ID!
title: String!
```

But you can define custum types.

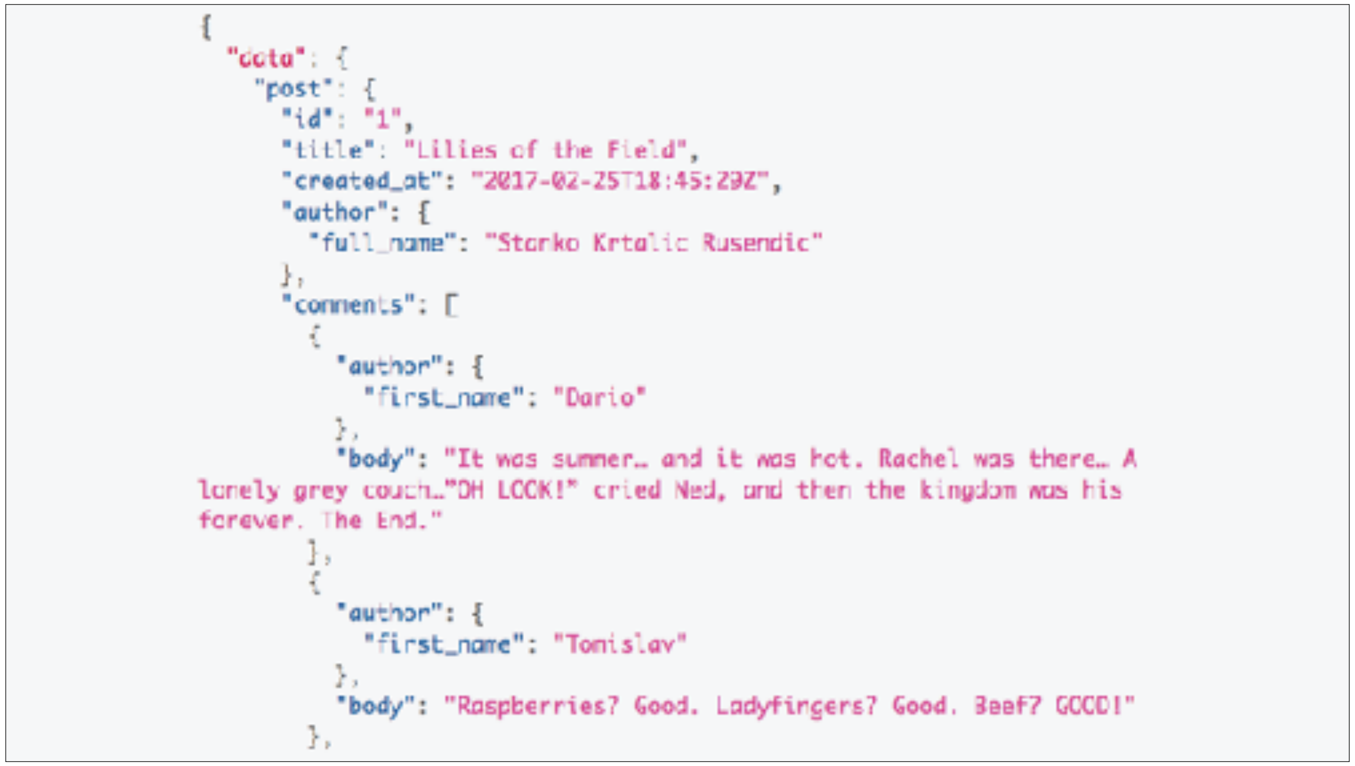
Queries
Mutations

There is also another important thing to note.
In general there are two kinds of methods in programming. Queries and mutations.

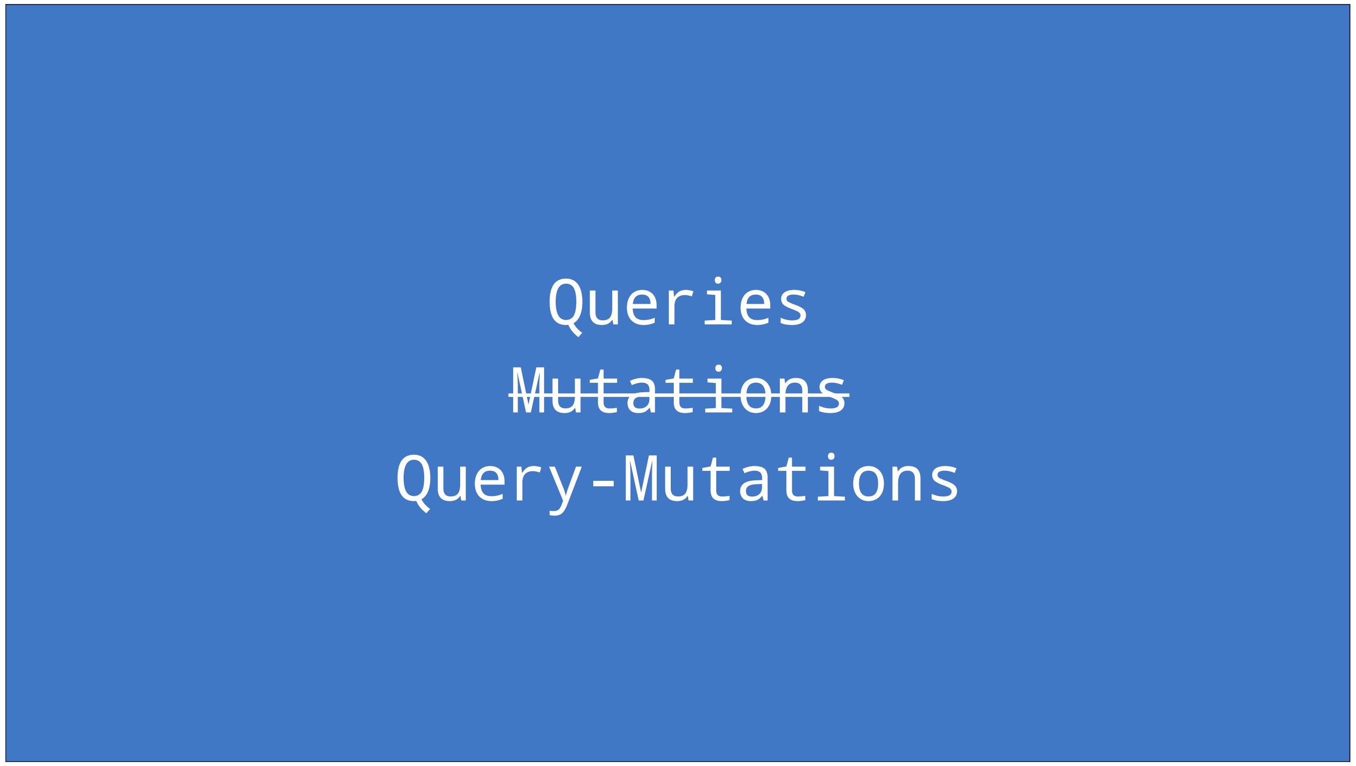


```
query {
  post(id: 1){
    id
    title
    created_at
    author {
      full_name
    }
    comments {
      author {
        first_name
      }
      body
    }
  }
}
```

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": "Stanko 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!"
        },
```

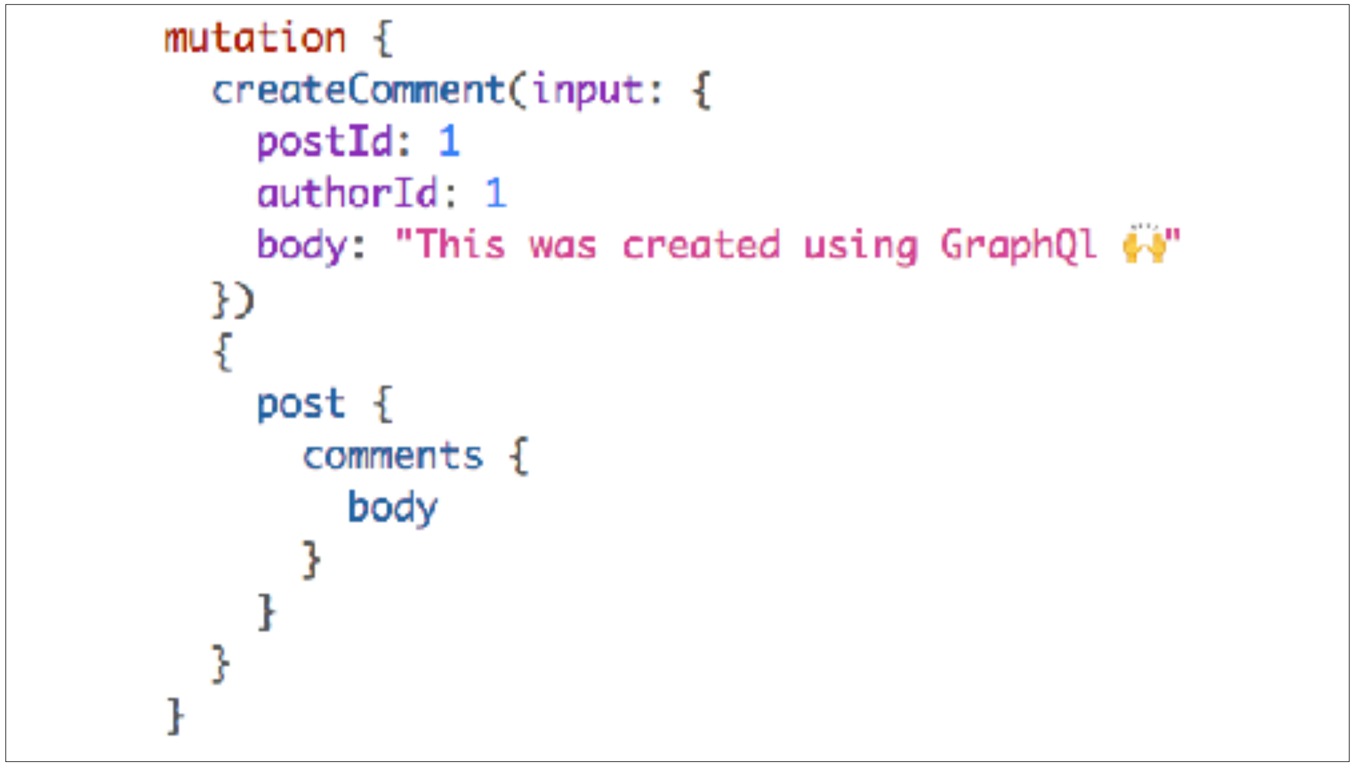


Queries

~~Mutations~~

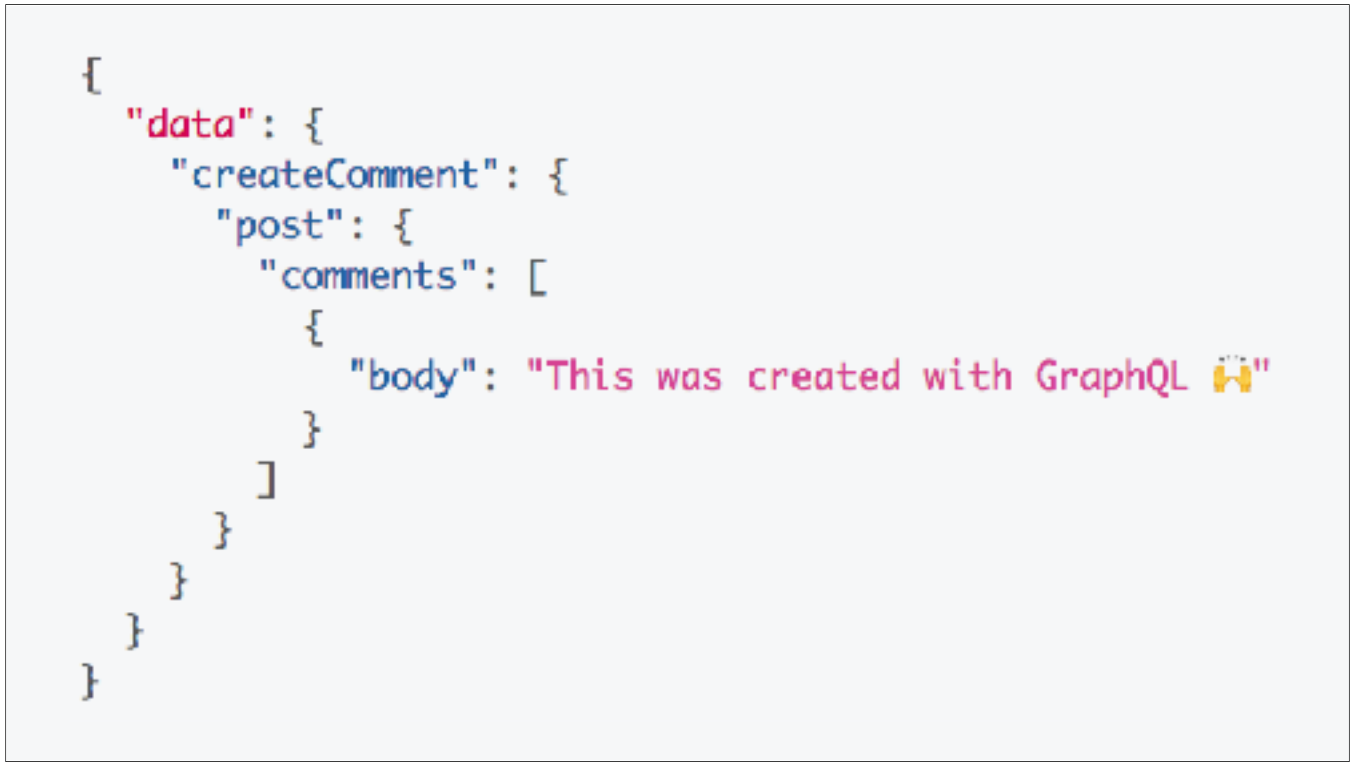
Query-Mutations

GraphQL doesn’t allow for pure mutations, only for query-mutations. The basic difference being that you have to specify which information you want in return after the query was executed.



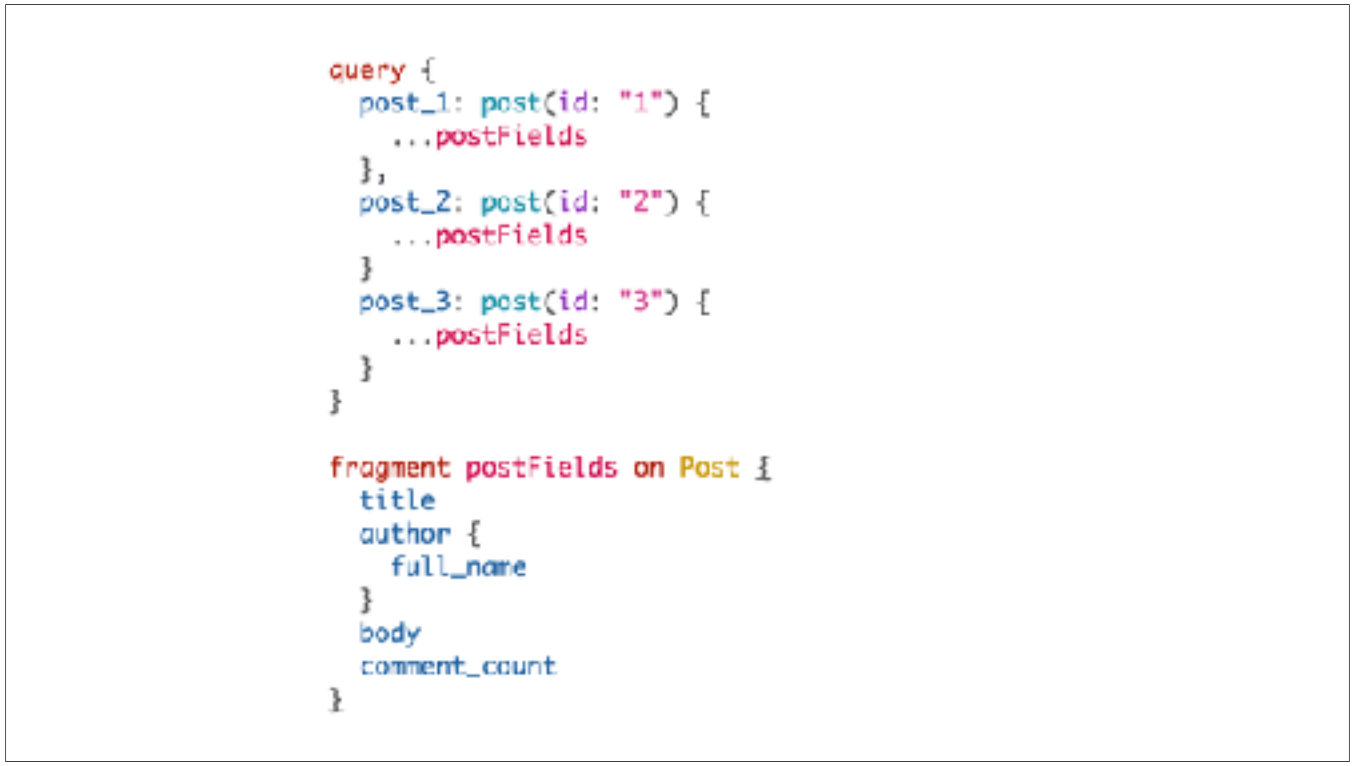
```
mutation {
  createComment(input: {
    postId: 1
    authorId: 1
    body: "This was created using GraphQl 🙌"
  })
  {
    post {
      comments {
        body
      }
    }
  }
}
```

And here is an example of a mutation. Note that you have to select something from the returned information of the mutation. In this example I specify that I want all the contents of all comments on the post I commented.



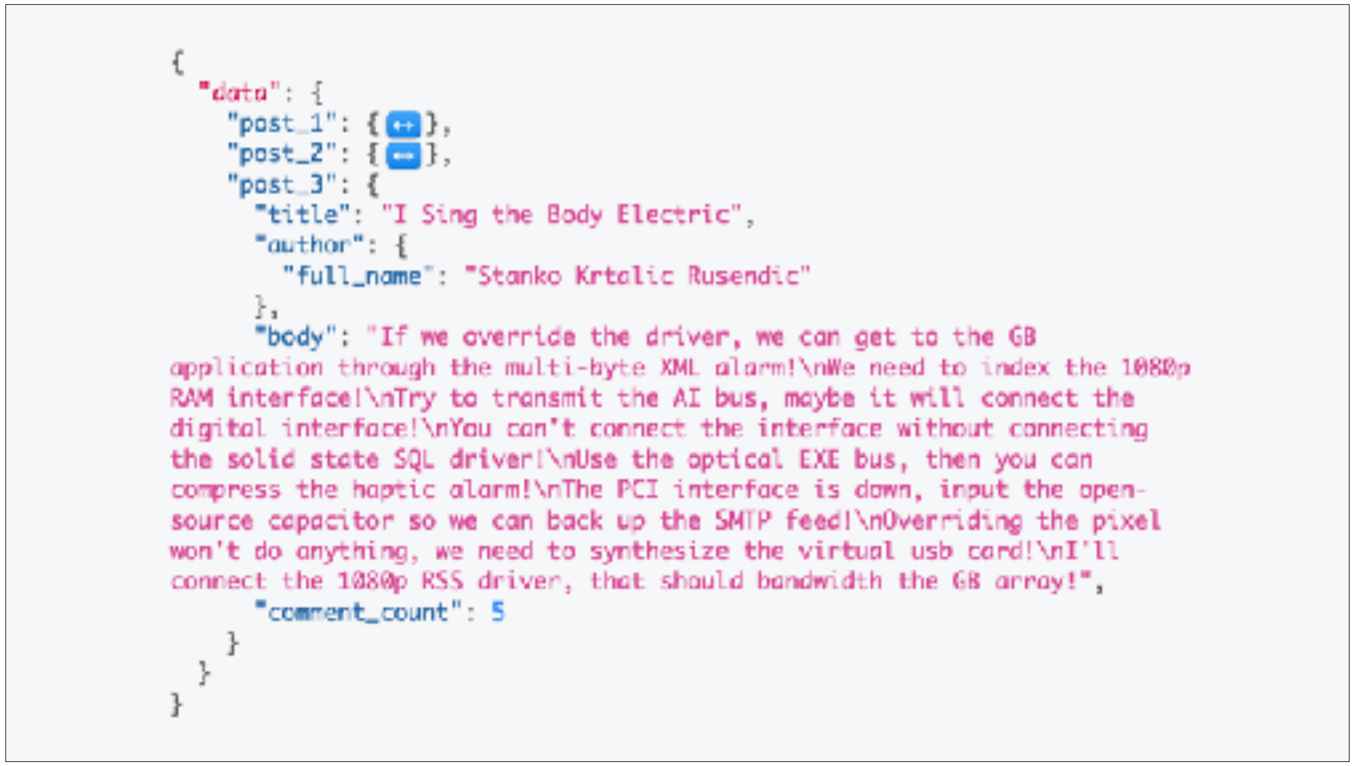
```
{
  "data": {
    "createComment": {
      "post": {
        "comments": [
          {
            "body": "This was created with GraphQL 🙌"
          }
        ]
      }
    }
  }
}
```

And that is exactly what I get back.



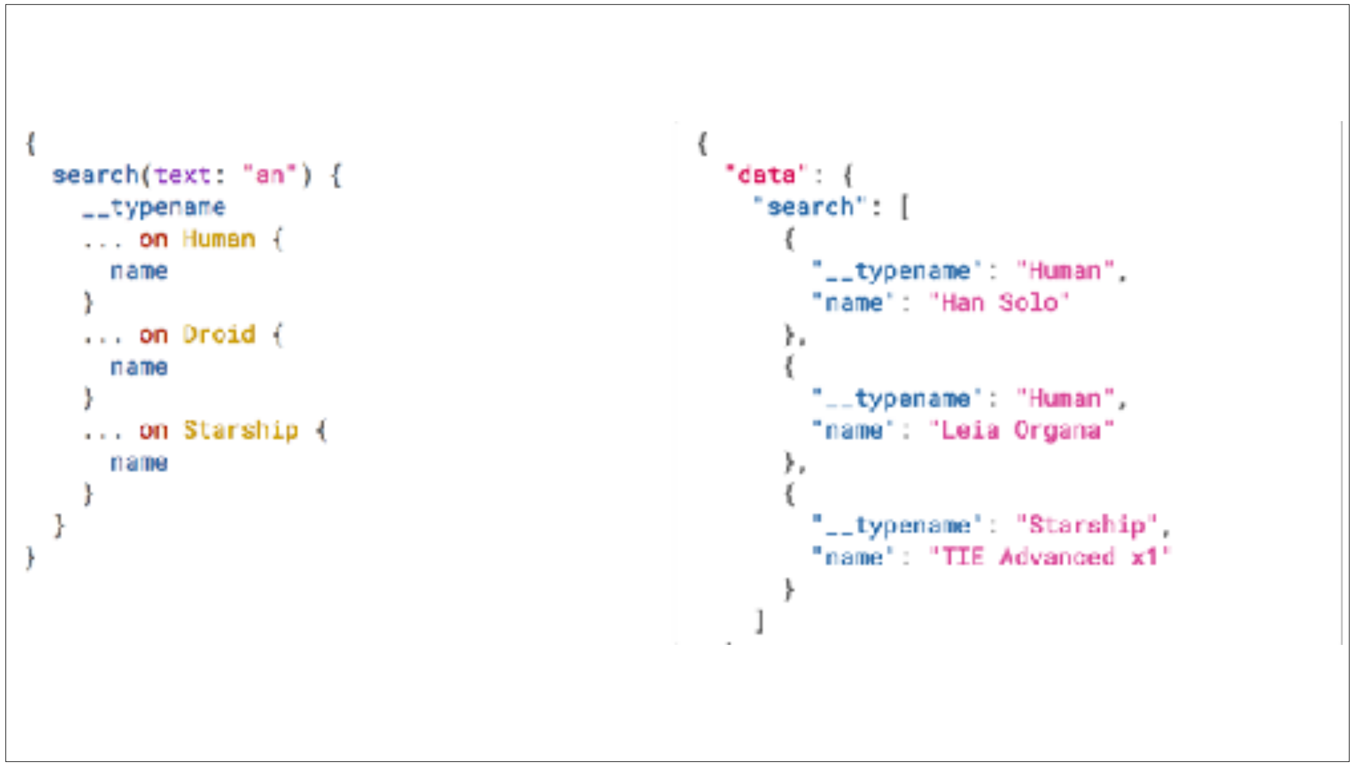
```
query {
  post_1: post(id: "1") {
    ...postFields
  },
  post_2: post(id: "2") {
    ...postFields
  }
  post_3: post(id: "3") {
    ...postFields
  }
}

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

But GraphQL offers many more features. E.g. these are called 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.



```
{
  "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
    }
  }
}
```

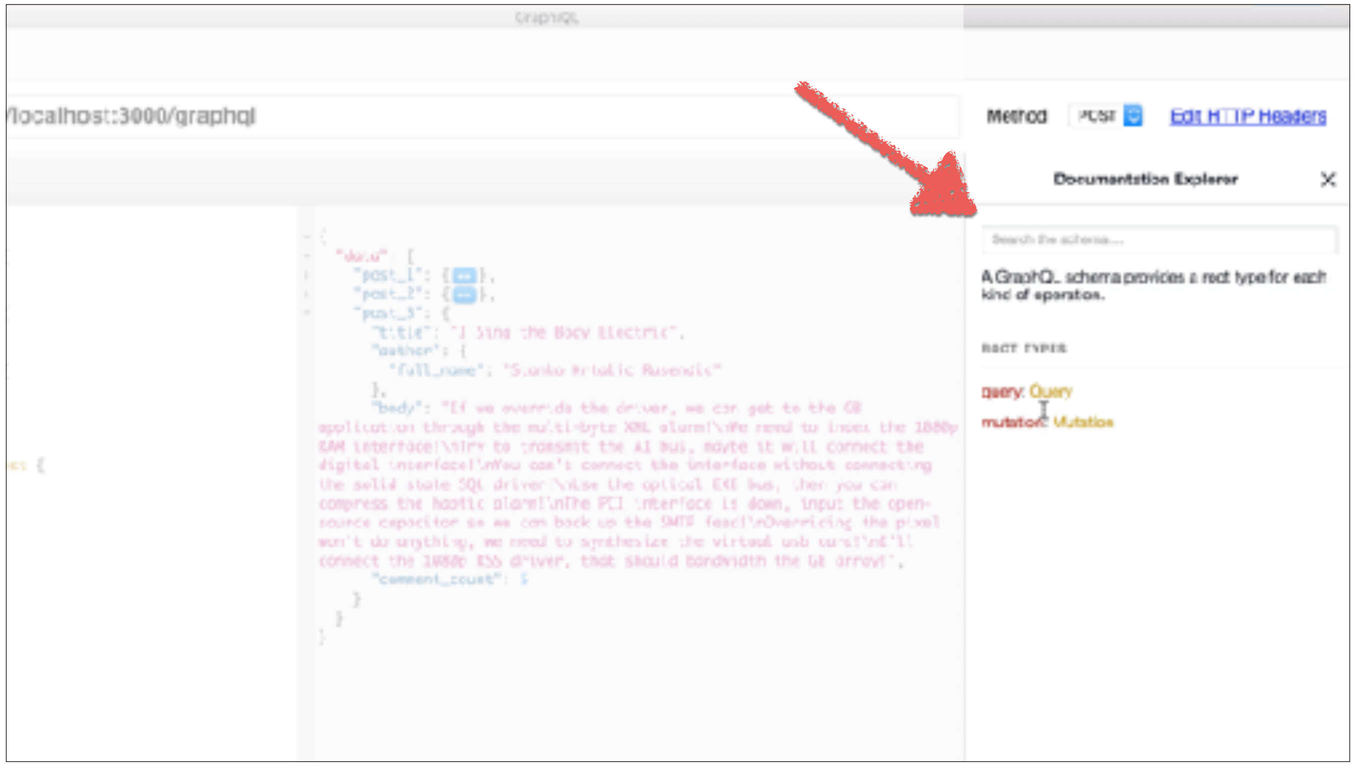
This is the result of the query. The blu arrows are just collapsed data. All responses are by structure identical to the one displayed.



```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
    }
    ... on Droid {
      name
    }
    ... on Starship {
      name
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo"
      },
      {
        "__typename": "Human",
        "name": "Leia Organa"
      },
      {
        "__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.



```
{
  "data": {
    "post_1": {↔},
    "post_2": {↔},
    "post_3": {
      "title": "I Sing the Body Electric",
      "author": {
        "full_name": "Stanko Krtolic Rusendic"
      },
      "body": "If we override the driver, we can get to the CB 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 UE array!",
      "comment_count": 5
    }
  }
}
```

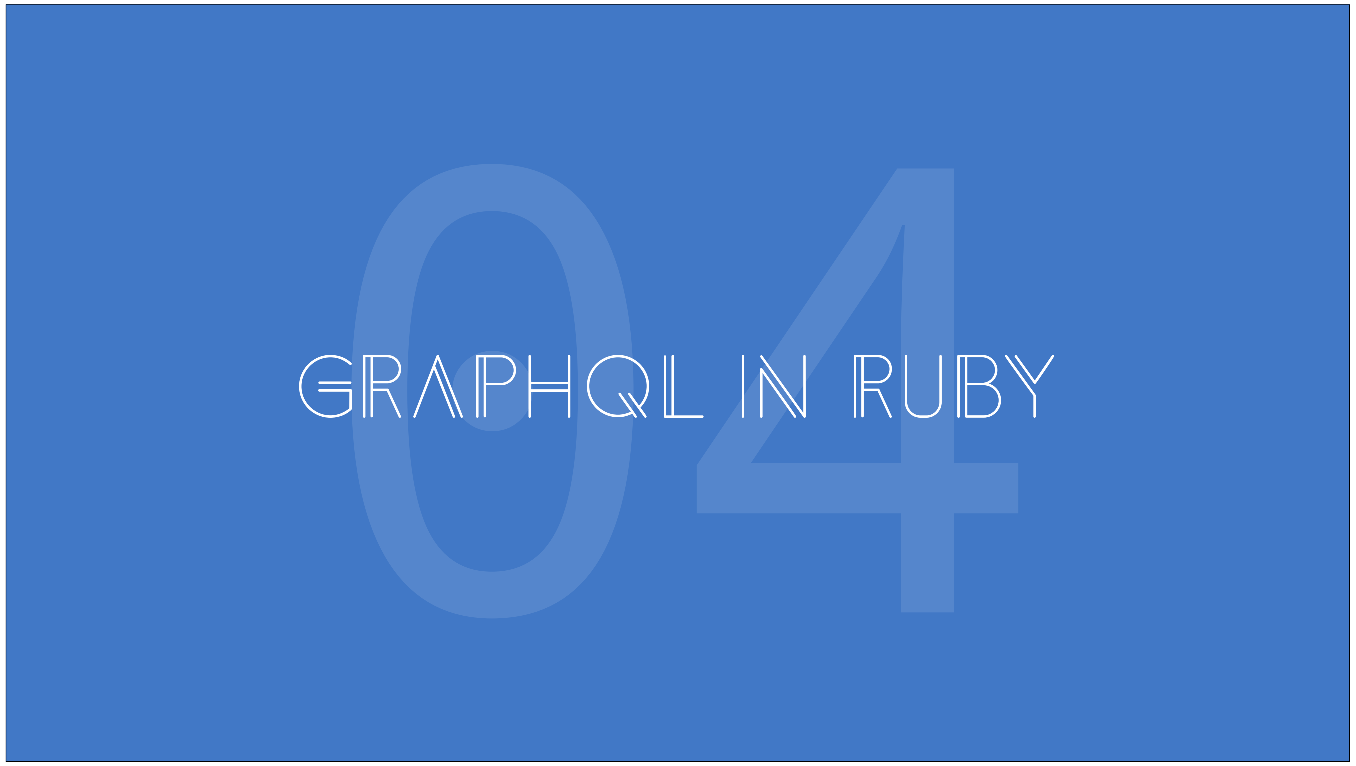
A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

query: Query

mutation: Mutation

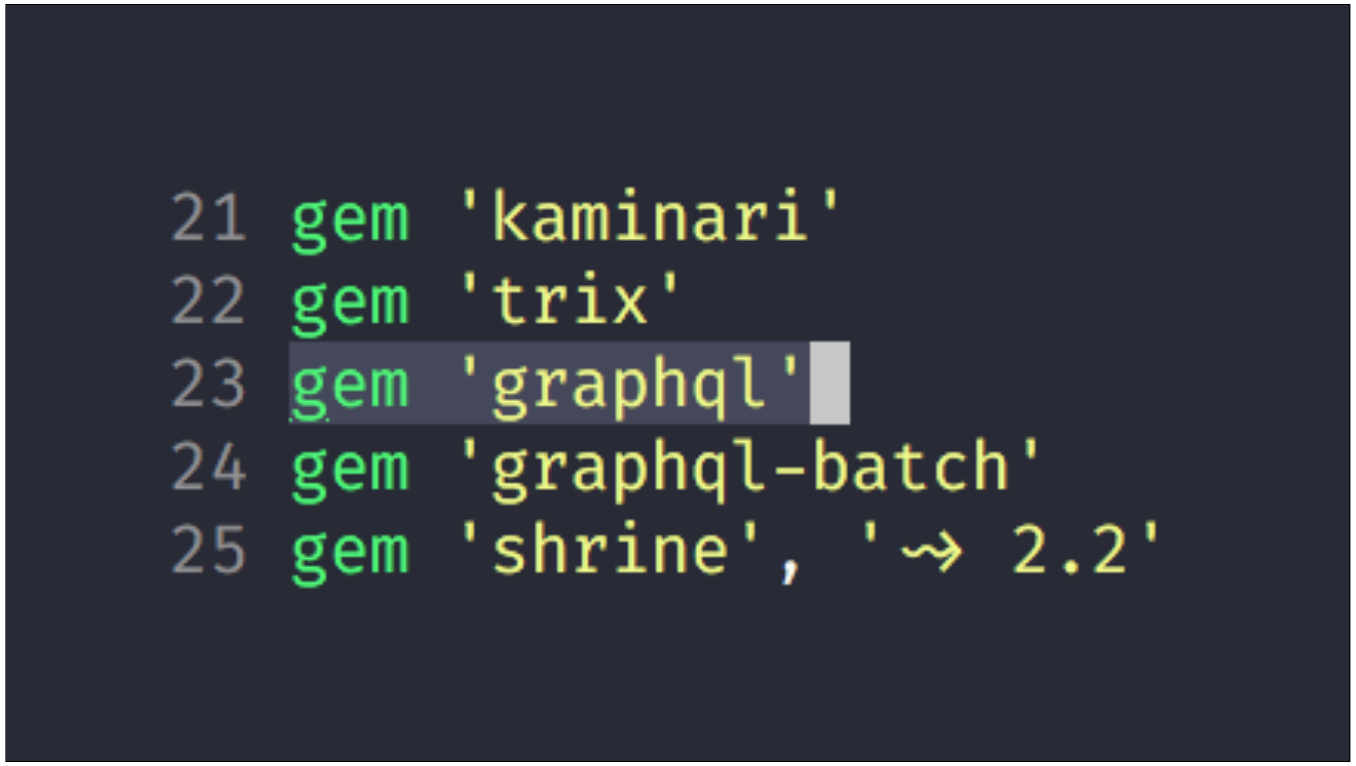
And perhaps the best feature! Autogenerated documentation!



04

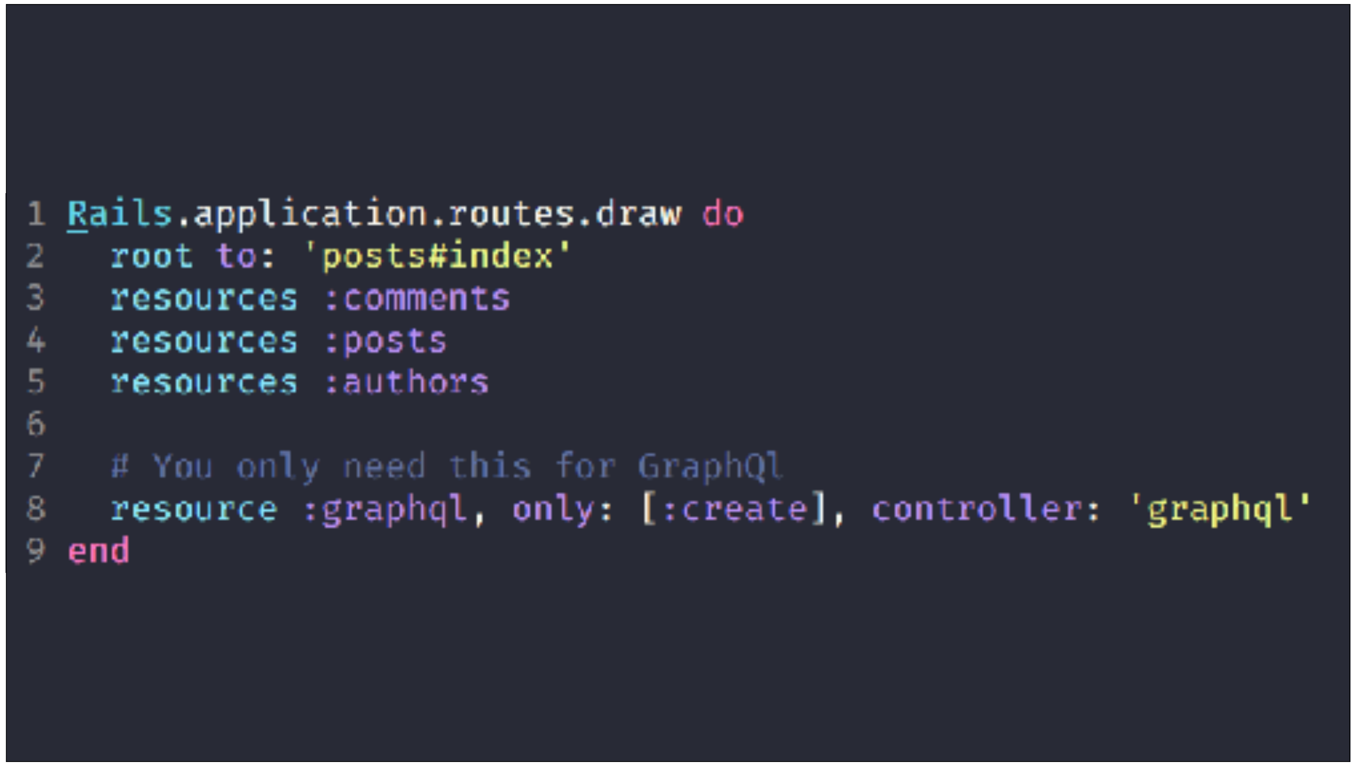
GRAPHQL IN RUBY

Ok, but how do we do it in ruby?



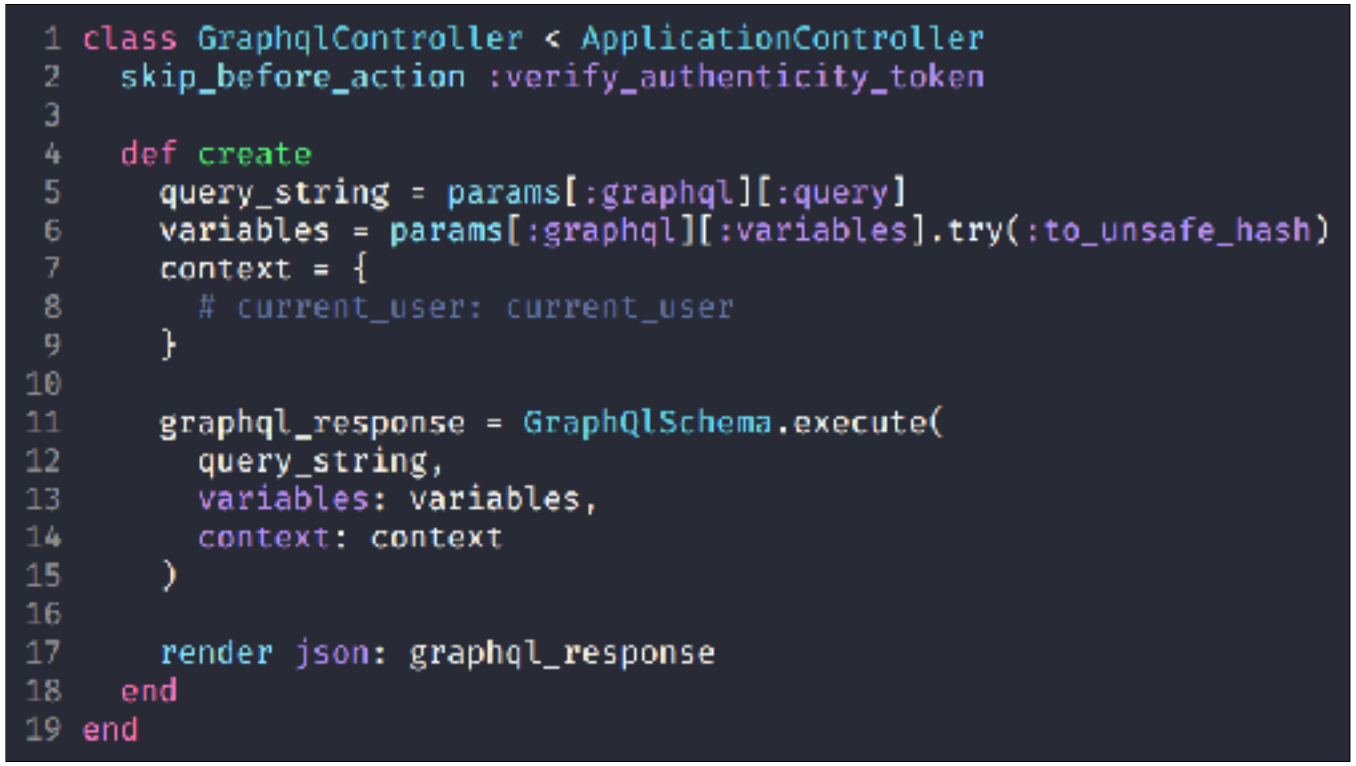
```ruby
gem 'kaminari'
gem 'trix'
gem 'graphql'
gem 'graphql-batch'
gem 'shrine', '~> 2.2'
```

You add the gem



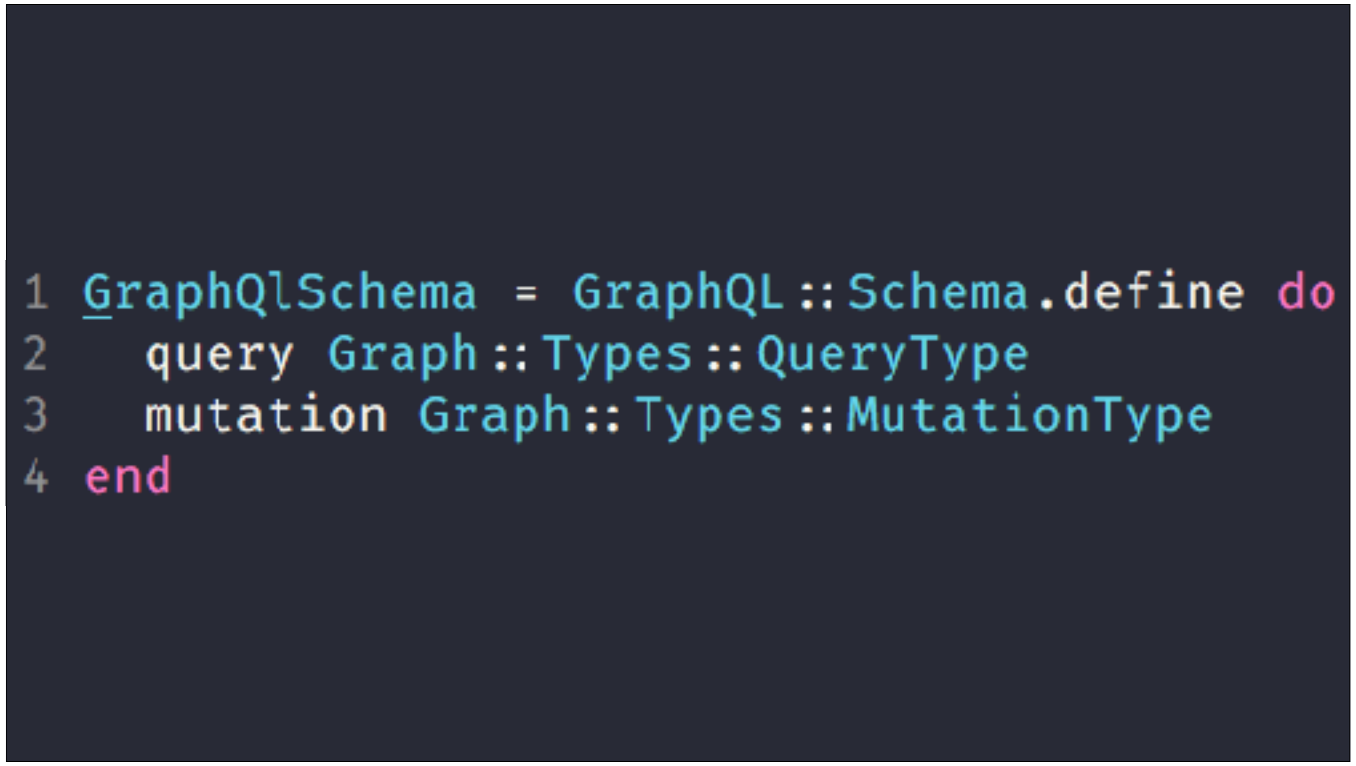
```ruby
Rails.application.routes.draw do
  root to: 'posts#index'
  resources :comments
  resources :posts
  resources :authors

  # You only need this for GraphQl
  resource :graphql, only: [:create], controller: 'graphql'
end
```

Add the route



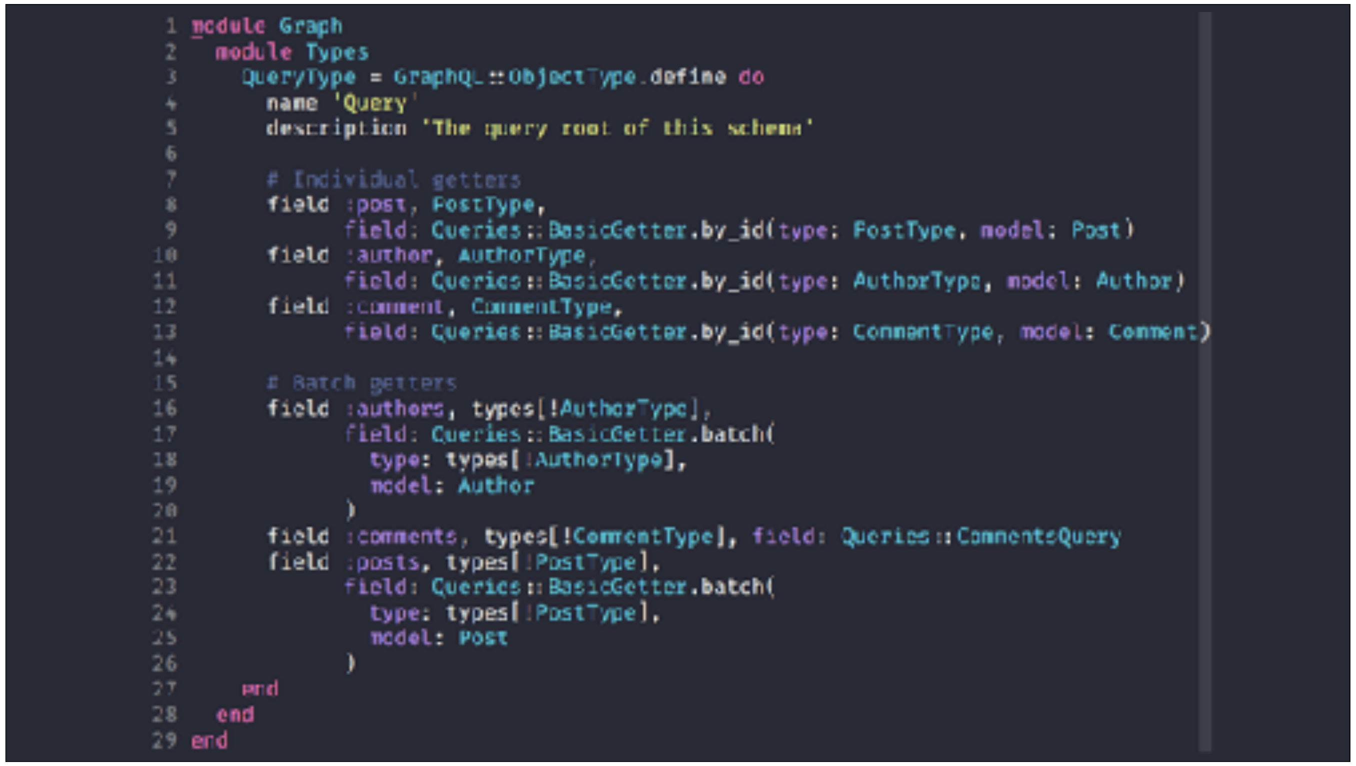
```ruby
class GraphqlController < ApplicationController
  skip_before_action :verify_authenticity_token

  def create
    query_string = params[:graphql][:query]
    variables = params[:graphql][:variables].try(:to_unsafe_hash)
    context = {
      # current_user: current_user
    }

    graphql_response = GraphQlSchema.execute(
      query_string,
      variables: variables,
      context: context
    )

    render json: graphql_response
  end
end
```

Create a simple controller.
Note that here you cans specify a context, e.g. the logged in user, or some headers that have been passed, or whatever you woud like. This will later be available globally to all queries, mutations, resolvers, and whatnot.



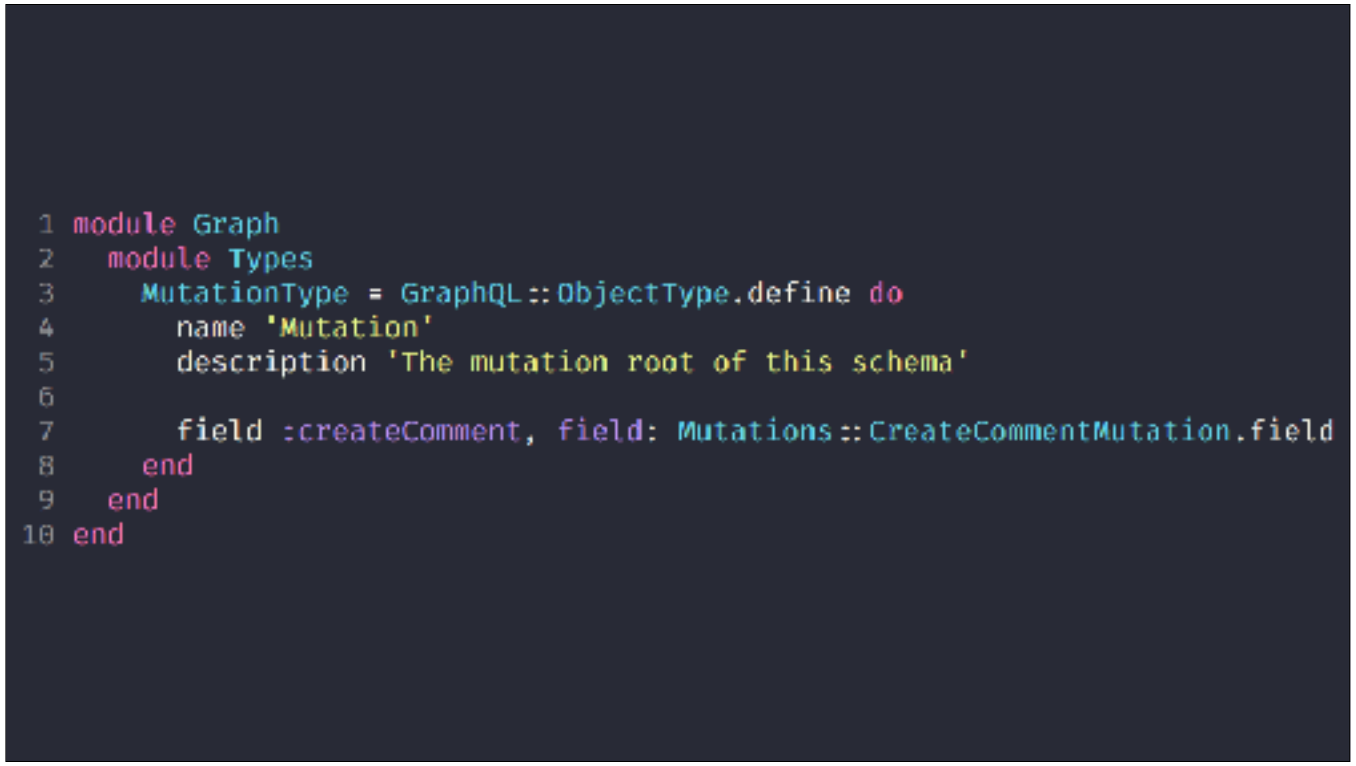
```
GraphQlSchema = GraphQL::Schema.define do
  query Graph::Types::QueryType
  mutation Graph::Types::MutationType
end
```

Implement a schema.
A schema is the entrypoint of your GraphQL resolver. It specifies what your app can and can’t do.
Each scherma has to have at least a query or a mutation type.
Those define what queries and mutations are available.



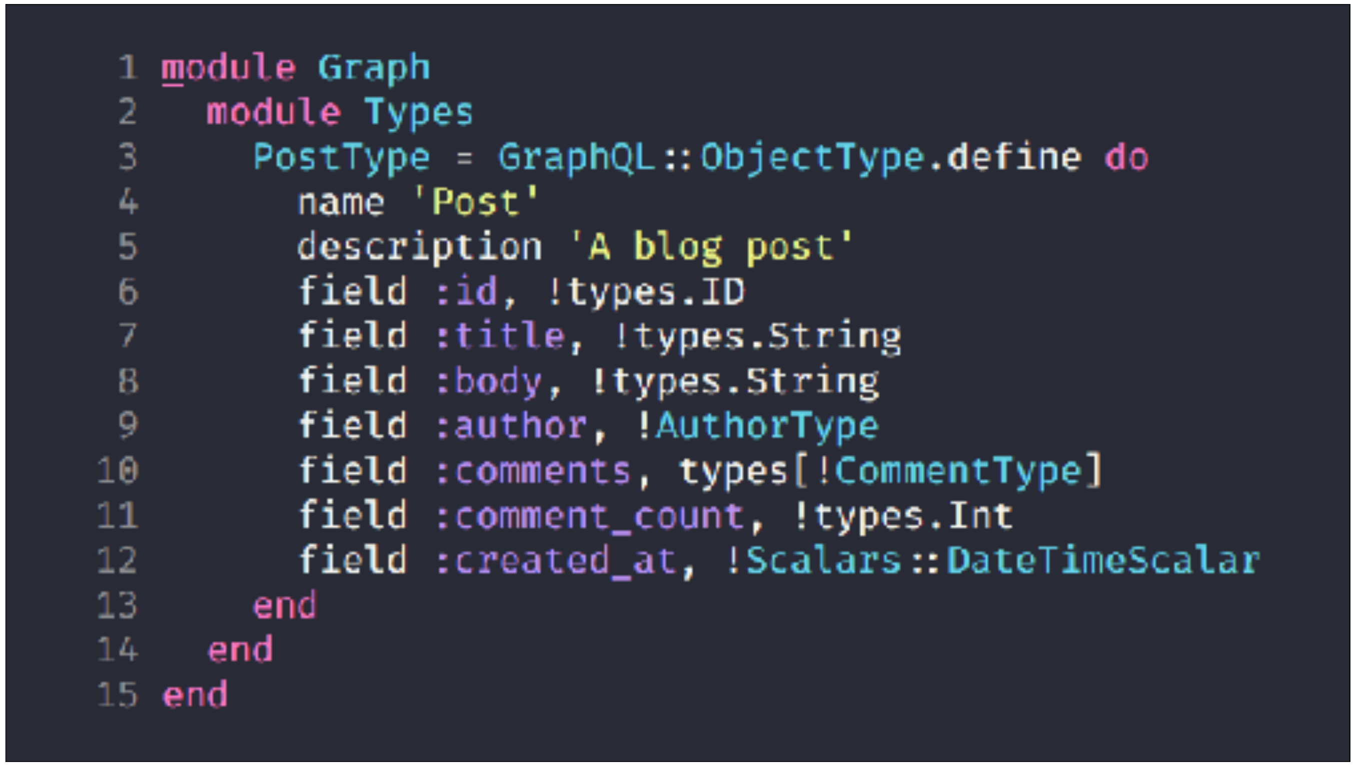
```ruby
module Graph
  module Types
    QueryType = GraphQL::ObjectType.define do
      name 'Query'
      description 'The query root of this schema'

      # Individual getters
      field :post, PostType,
            field: Queries::BasicGetter.by_id(type: PostType, model: Post)
      field :author, AuthorType,
            field: Queries::BasicGetter.by_id(type: AuthorType, model: Author)
      field :comment, CommentType,
            field: Queries::BasicGetter.by_id(type: CommentType, model: Comment)

      # Batch getters
      field :authors, types[!AuthorType],
            field: Queries::BasicGetter.batch(
              type: types[!AuthorType],
              model: Author
            )
      field :comments, types[!CommentType], field: Queries::CommentsQuery
      field :posts, types[!PostType],
            field: Queries::BasicGetter.batch(
              type: types[!PostType],
              model: Post
            )
    end
  end
end
```

An example of a query type.
It specifies what kind of queries exist.



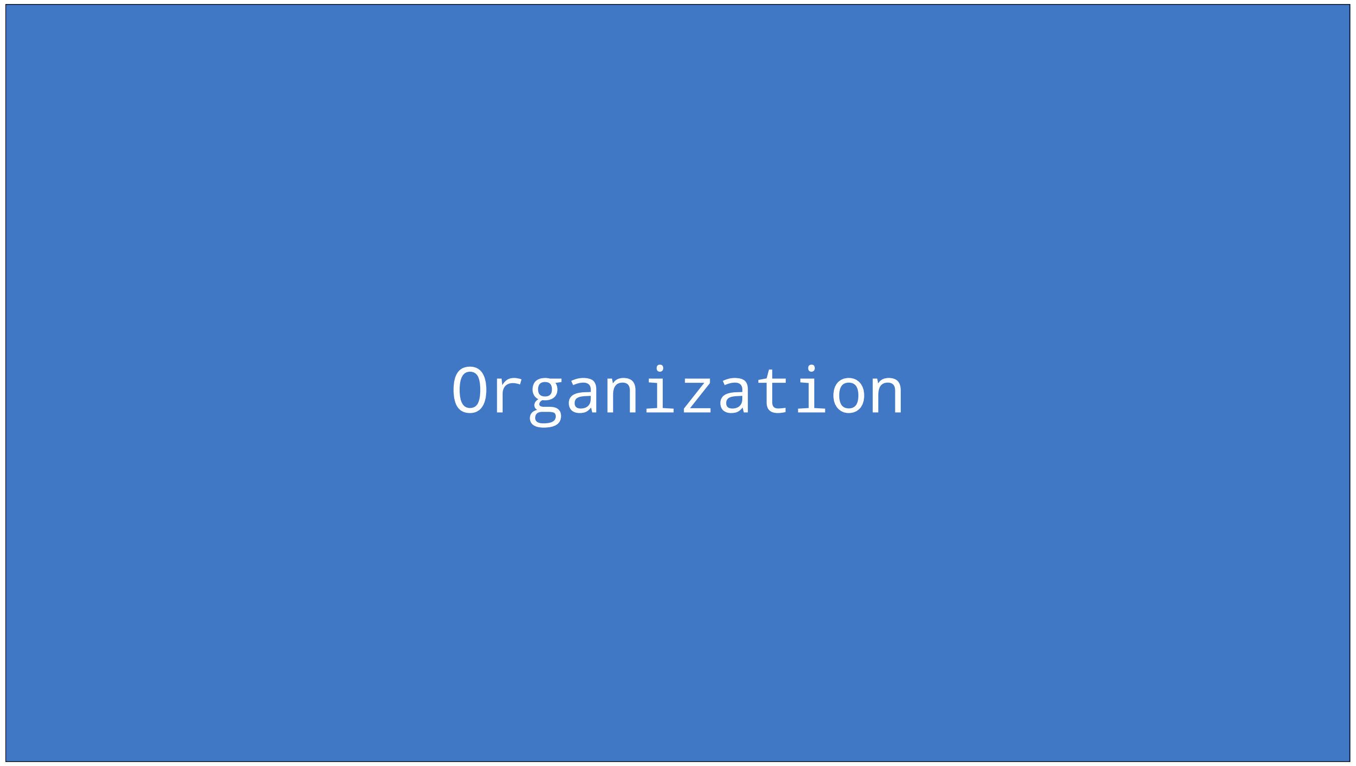
```ruby
module Graph
  module Types
    MutationType = GraphQL::ObjectType.define do
      name 'Mutation'
      description 'The mutation root of this schema'

      field :createComment, field: Mutations::CreateCommentMutation.field
    end
  end
end
```

And a mutations type which spercifies what kind of mutations exist.



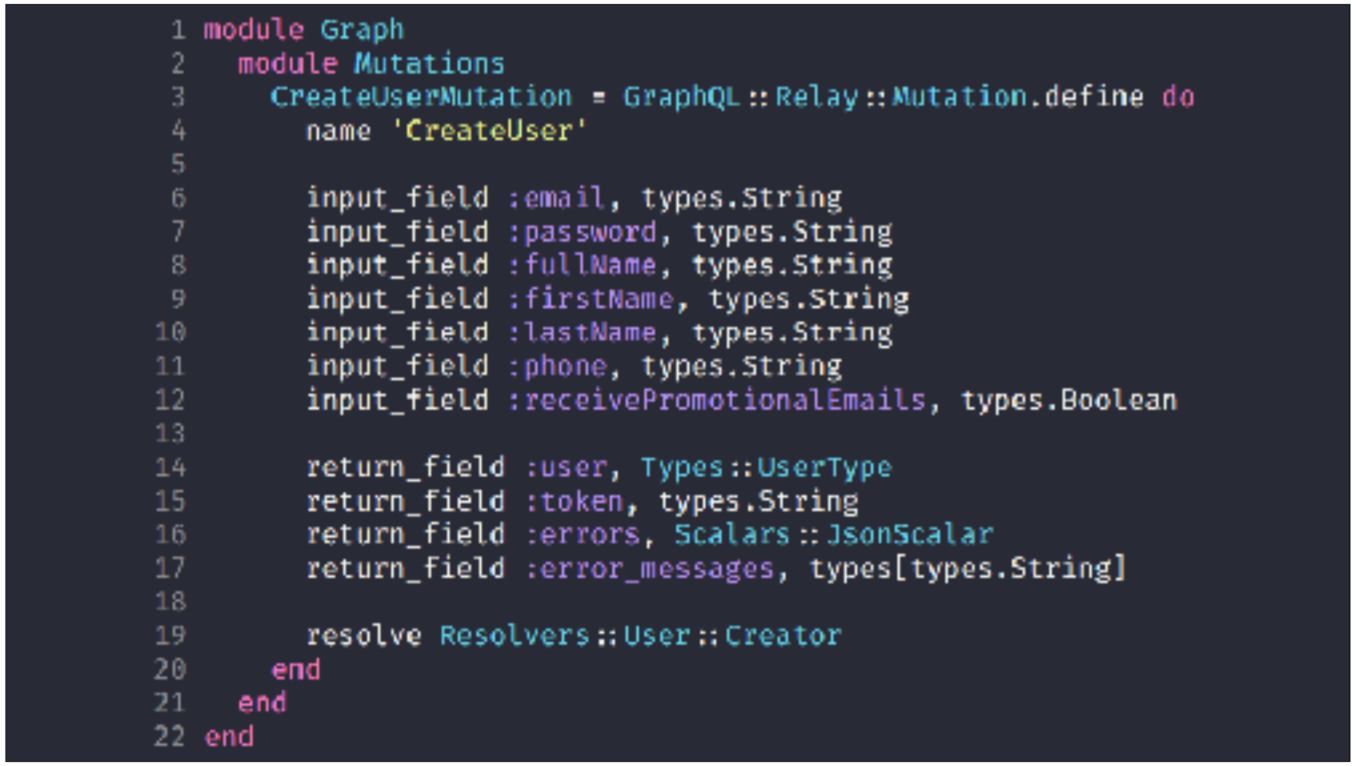
```ruby
module Graph
  module Types
    PostType = GraphQL::ObjectType.define do
      name 'Post'
      description 'A blog post'
      field :id, !types.ID
      field :title, !types.String
      field :body, !types.String
      field :author, !AuthorType
      field :comments, types[!CommentType]
      field :comment_count, !types.Int
      field :created_at, !Scalars::DateTimeScalar
    end
  end
end
```

And then you need to specify a type for each resource you have.
You can think of types as serializers. They explain which attributes from the object should be exposed.
You can also depricate fields here, or define inline respolvers.



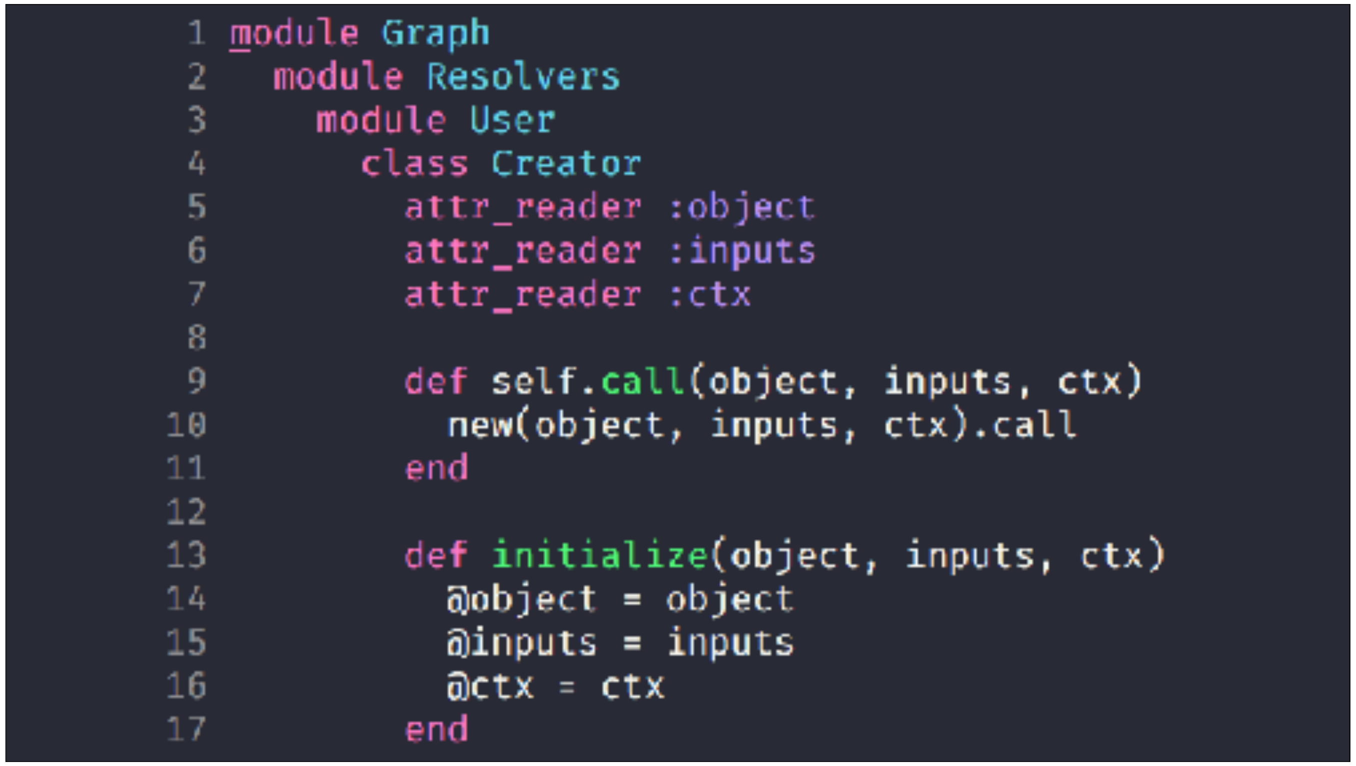
Organization

Touching the subject of resolvers, let’s talk about code organisation.



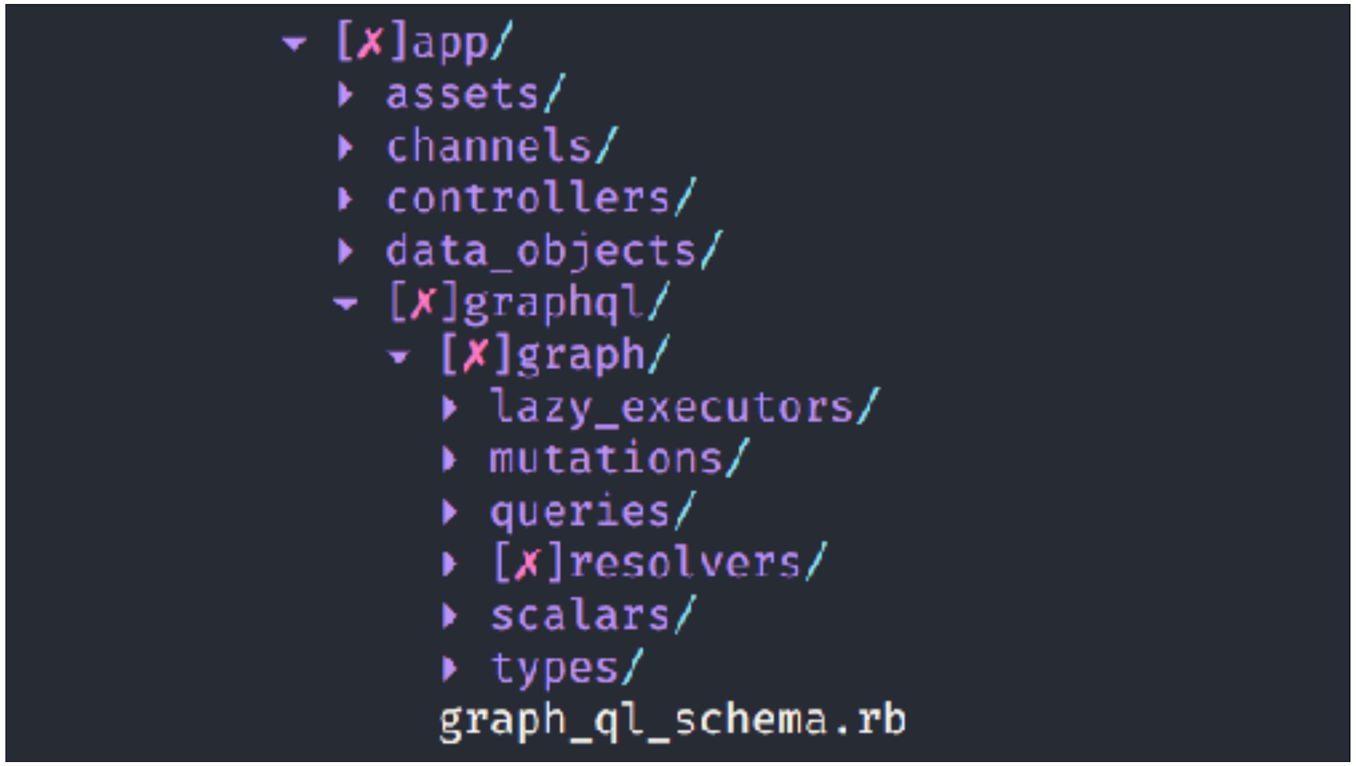
```ruby
module Graph
  module Mutations
    CreateUserMutation = GraphQL::Relay::Mutation.define do
      name 'CreateUser'

      input_field :email, types.String
      input_field :password, types.String
      input_field :fullName, types.String
      input_field :firstName, types.String
      input_field :lastName, types.String
      input_field :phone, types.String
      input_field :receivePromotionalEmails, types.Boolean

      return_field :user, Types::UserType
      return_field :token, types.String
      return_field :errors, Scalars::JsonScalar
      return_field :error_messages, types[types.String]

      resolve Resolvers::User::Creator
    end
  end
end
```

Instead of passing Lambdas as specified in the documentation you can also pass objects or classes.



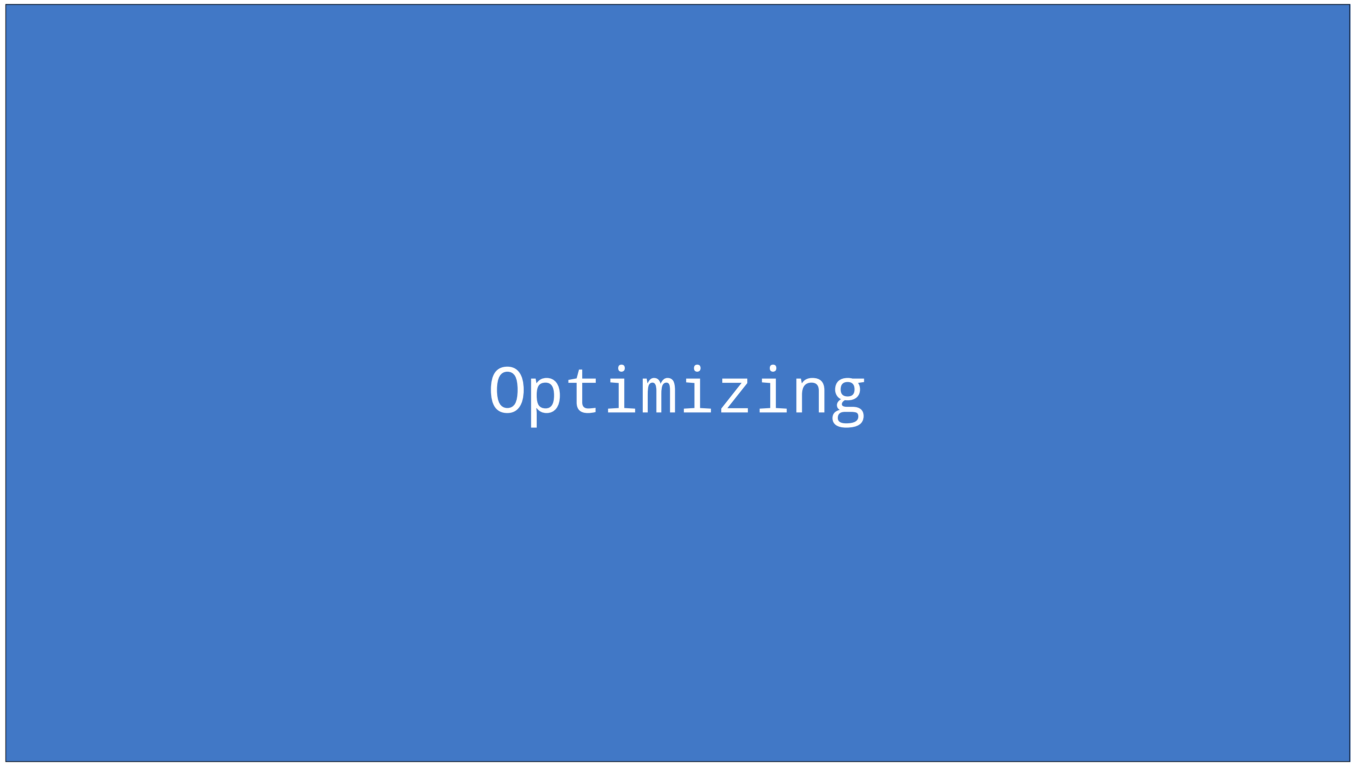
```ruby
module Graph
  module Resolvers
    module User
      class Creator
        attr_reader :object
        attr_reader :inputs
        attr_reader :ctx

        def self.call(object, inputs, ctx)
          new(object, inputs, ctx).call
        end

        def initialize(object, inputs, ctx)
          @object = object
          @inputs = inputs
          @ctx = ctx
        end
```

The passed class has to have a call method and has to accept three parameters - the object, the inputs and a context.



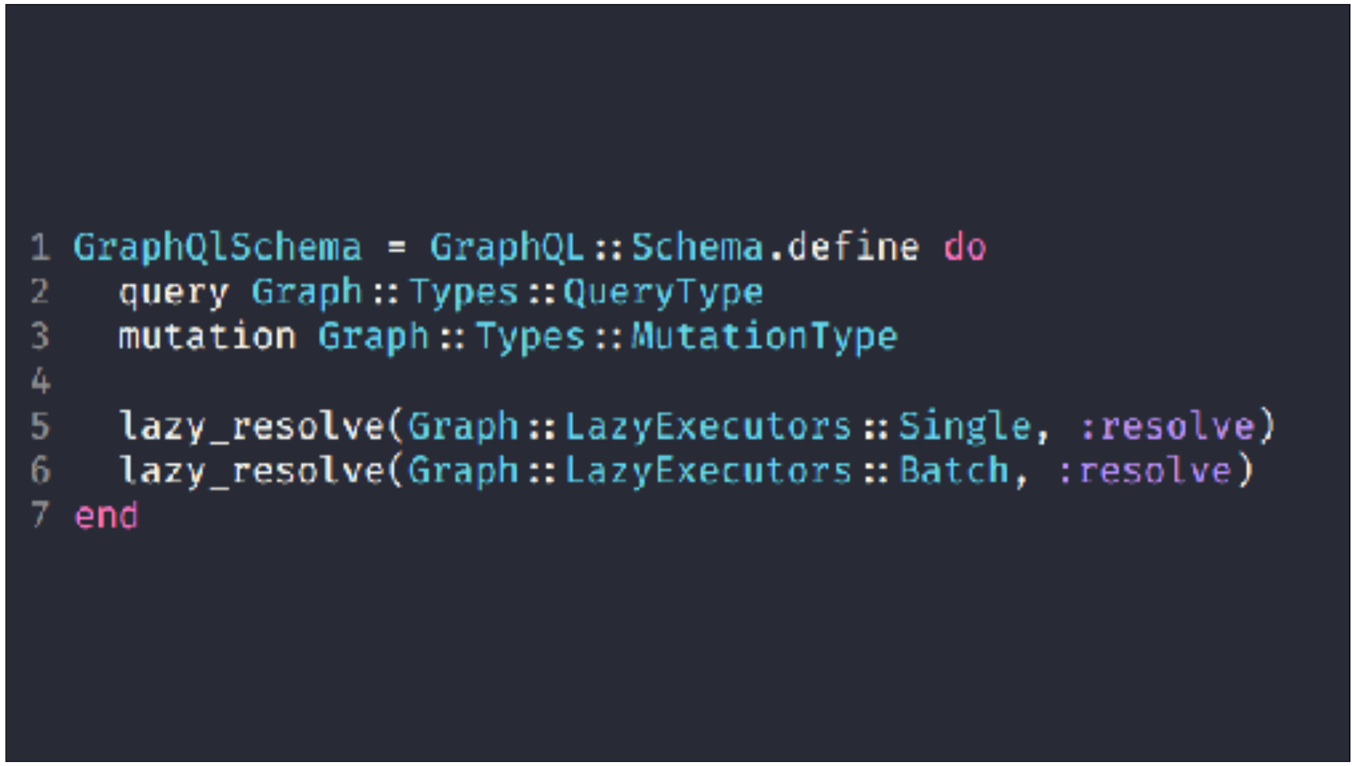
```
▾ [✗]app/
  ▸ assets/
  ▸ channels/
  ▸ controllers/
  ▸ data_objects/
  ▾ [✗]graphql/
    ▾ [✗]graph/
      ▸ lazy_executors/
      ▸ mutations/
      ▸ queries/
      ▸ [✗]resolvers/
      ▸ scalars/
      ▸ types/
      graph_ql_schema.rb
```

I lifted the naming resolvers form the official JS GraphQL implementation.



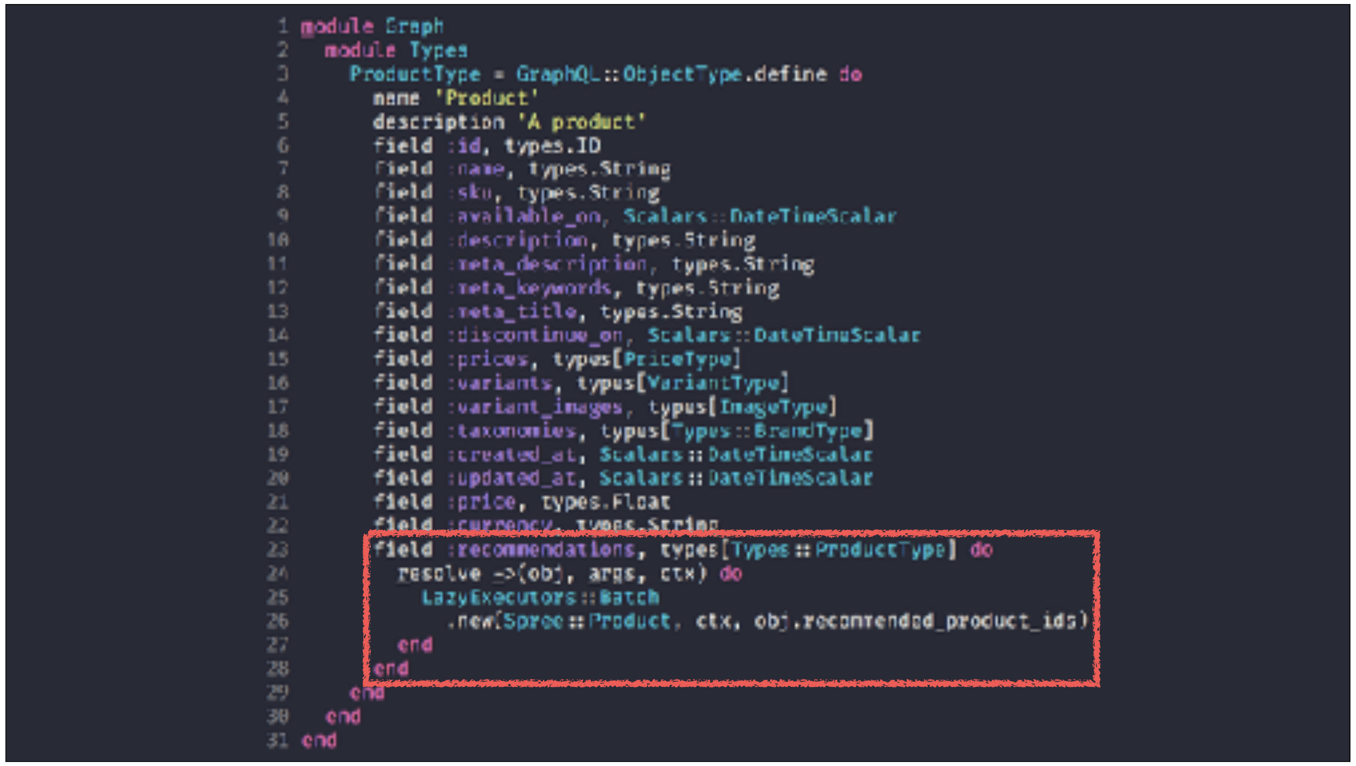
Optimizing

This is an often question I get. How do I optimize this, or how do I resolver N+1 queries.



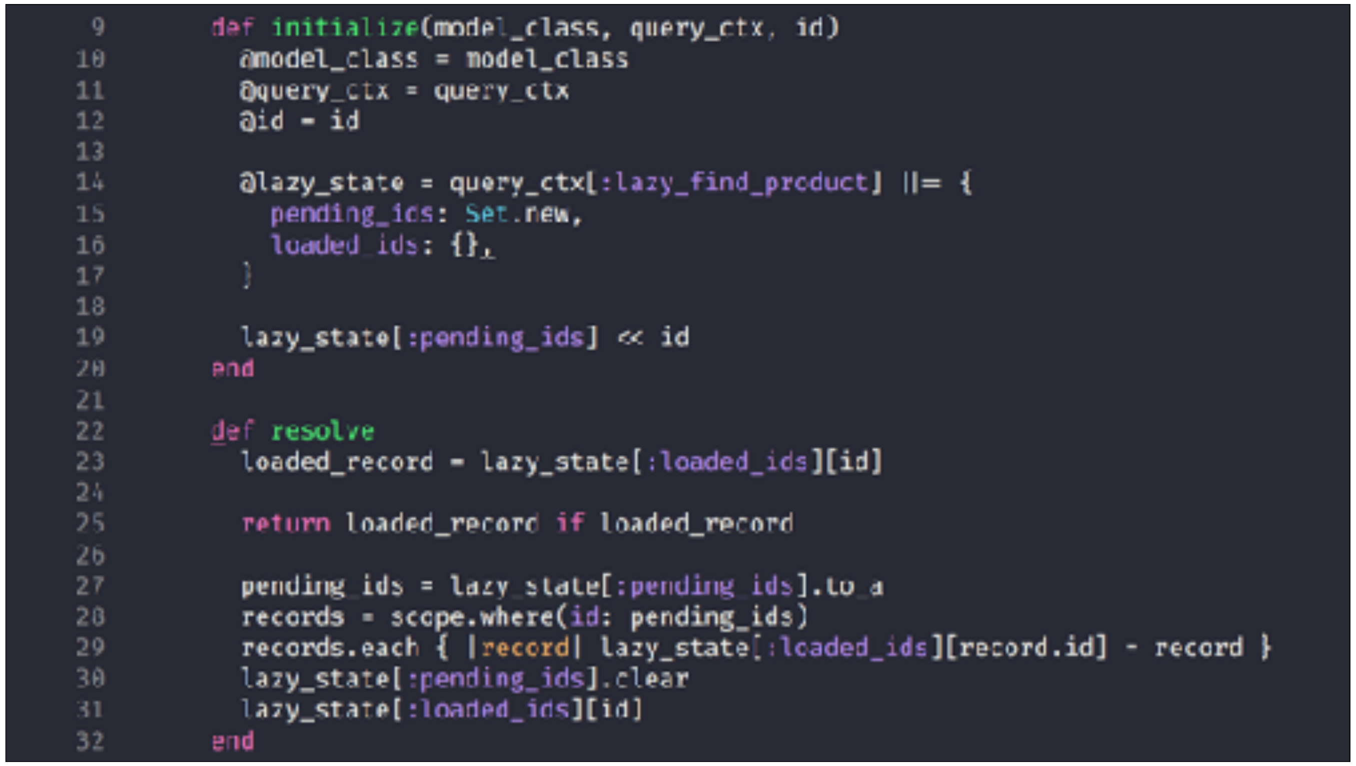
```ruby
GraphQlSchema = GraphQL::Schema.define do
  query Graph::Types::QueryType
  mutation Graph::Types::MutationType

  lazy_resolve(Graph::LazyExecutors::Single, :resolve)
  lazy_resolve(Graph::LazyExecutors::Batch, :resolve)
end
```

The schema allows for definition of lazy resolvers. Basically you tell it which method to call on an object of the specified class if it encounters it during serialization.



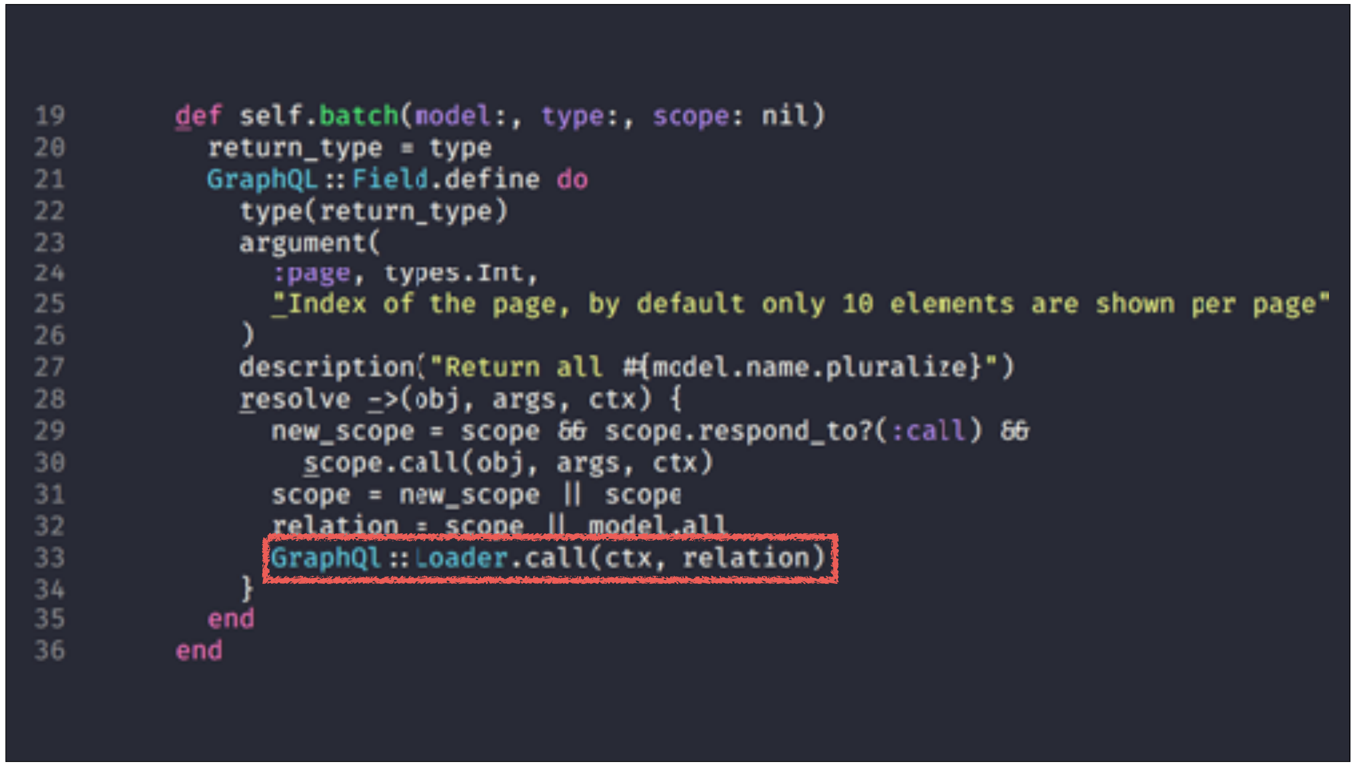
```ruby
module Graph
  module Types
    ProductType = GraphQL::ObjectType.define do
      name 'Product'
      description 'A product'
      field :id, types.ID
      field :name, types.String
      field :sku, types.String
      field :available_on, Scalars::DateTimeScalar
      field :description, types.String
      field :meta_description, types.String
      field :meta_keywords, types.String
      field :meta_title, types.String
      field :discontinue_on, Scalars::DateTimeScalar
      field :prices, types[PriceType]
      field :variants, types[VariantType]
      field :variant_images, types[ImageType]
      field :taxonomies, types[Types::BrandType]
      field :created_at, Scalars::DateTimeScalar
      field :updated_at, Scalars::DateTimeScalar
      field :price, types.Float
      field :currency, types.String
      field :recommendations, types[Types::ProductType] do
        resolve ->(obj, args, ctx) do
          LazyExecutors::Batch
            .new(Spree::Product, ctx, obj.recommended_product_ids)
        end
      end
    end
  end
end
```

So heren is both an exasmple of an inline resolver and a lazy executor.



```ruby
def initialize(model_class, query_ctx, id)
  @model_class = model_class
  @query_ctx = query_ctx
  @id = id

  @lazy_state = query_ctx[:lazy_find_product] ||= {
    pending_ids: Set.new,
    loaded_ids: {},
  }

  lazy_state[:pending_ids] << id
end

def resolve
  loaded_record = lazy_state[:loaded_ids][id]

  return loaded_record if loaded_record

  pending_ids = lazy_state[:pending_ids].to_a
  records = scope.where(id: pending_ids)
  records.each { |record| lazy_state[:loaded_ids][record.id] = record }
  lazy_state[:pending_ids].clear
  lazy_state[:loaded_ids][id]
end
```

This is the code of the executor. When it’s initialized it remembers which id it has to load as well as which ids it has already loaded. When resolve gets called it eather executes a query to fetch all pending ids at once (if the object wasn’t found) or it returns the found object.



```ruby
def self.batch(model:, type:, scope: nil)
  return_type = type
  GraphQL::Field.define do
    type(return_type)
    argument(
      :page, types.Int,
      "Index of the page, by default only 10 elements are shown per page"
    )
    description("Return all #{model.name.pluralize}")
    resolve ->(obj, args, ctx) {
      new_scope = scope && scope.respond_to?(:call) &&
        scope.call(obj, args, ctx)
      scope = new_scope || scope
      relation = scope || model.all
      GraphQl::Loader.call(ctx, relation)
    }
  end
end
```

Or you can roll your own. The previous example only solves problems if you can get the id's without making a query. But if you have a has_many relationship that might not be possible.

I've created a utility that prreloads all data from your GraphQL query to your SQL query. It uses the Rails includes method in the beckgorund. This itn's publically available yet, but will be by next Monday.

Questions?

Any questions?